Implementing the Decomposition of Soundness Proofs of Abstract Interpreters in Coq

---

Version of January 24, 2021

Jens de Waard

Implementing the Decomposition of Soundness Proofs of Abstract Interpreters in Coq

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jens de Waard
born in Alphen aan den Rijn, the Netherlands

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

© 2021 Jens de Waard.

Implementing the Decomposition of Soundness Proofs of Abstract Interpreters in Coq

Author: Jens de Waard
Student id: 4009215
Email: j.c.deWaard@student.tudelft.nl

Abstract

Abstract interpretation is a way of approximating the semantics of a computer program, in which we derive properties of those programs without actually performing the necessary computations for running the program, though the use of an abstract interpreter. To be able to trust the result of the abstract interpretation, we would to able to prove the soundness of the approximations of the interpreter. Previous work by Keidel et al. has shown that the soundness proofs of an entire abstract interpreter can be simplified by decomposing the interpreter by implementing concrete and abstract interpreters as instantiations of a generic interpreter. The goal of this thesis is to explore and implement mechanical proofs of soundness of such interpreters. To this end, we have used the interactive proof assistant Coq to implement a generic interpeter for a simple imperative language and instantiate it both concrete and abstract versions. The abstract interpreter is automatically proven sound via the use of Coq's automatic proof capabilities and typeclass system. Both the interpreted language and the used abstractions can be expanded to allow for more features. Soundness proofs can then be written for just the new components, those proofs will then be automatically resolved by Coq.

Thesis Committee:

Chair: Prof. dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member: Dr. G. Gousios, Faculty EEMCS, TU Delft
Committee Member: Dr. R. Krebbers, Faculty EEMCS, TU Delft
External Expert: Ir. S. Keidel, JGU Mainz

Preface

This report is the result of an almost two year journey into the heart of software verification. At the onset of that journey, I was only equipped with rudimentary knowledge of unit tests. In the months that followed, I probed the depths of category theory, wrestled with the intricacies of proof assistants and waded deep into theory of abstract intepretation.

I would like to thank my supervisor, Robbert Krebbers, for suggesting the topic and his guidance during the project. I would also like to thank Sven Keidel for his support and his continued willingness to explain his work. Finally I would like to thank my parents for their seemingly endless patience.

Jens de Waard
Delft, the Netherlands
January 24, 2021

Contents

Preface **iii**

Contents **v**

1 Introduction **1**
1.1 Approaches to software verification . . . 1
1.2 Static analysis . . . 1
1.3 Contributions . . . 3
1.4 Outline . . . 3

2 Background **5**
2.1 What is Abstract Interpretation? . . . 5
2.2 Concrete interpretation of an arithmetic language . . . 6
2.3 Abstract interpretation of the language . . . 8
2.4 Definition of Soundness . . . 9
2.5 Additional Coq Commands . . . 11

3 Building The Interpreters **13**
3.1 The Language . . . 13
3.2 The Concrete Interpreter . . . 14
3.3 Monads . . . 17
3.4 The Abstract Interpreter . . . 23
3.5 Extracting the Generic Interpreter . . . 26

4 Proving The Interpreter Sound **31**
4.1 Galois Connections . . . 31
4.2 Soundness . . . 33
4.3 Sound typeclasses . . . 35
4.4 Standard automation tactics . . . 36
4.5 Custom tactics . . . 37

5 Related Work **39**
5.1 Mechanization of Abstract Analysis . . . 39
5.2 Constructive Galois connections . . . 39
5.3 Generic interpreters . . . 40

6 Conclusions and Future Work **43**

Bibliography **45**

Chapter 1

Introduction

1.1 Approaches to software verification

Software engineering is a complex discipline. While developing, it is easy to lose sight of what the code actually does and over time the difference between what a developer intended and what a developer actually created will only grow. Many tools and techniques exist to mitigate this problem and help developers establish faith in their code.

One common approach to software testing is to write small tests that verify a single unit of a program, such as a single function or a class. The program can be run on a diverse set of inputs and the results verified by a testing library.

Such traditional testing approaches run the system under test to find errors. However, often programs can be run on an infinite input space and it is only feasible to write tests that are run on a finite subset of the input space. In contrast, static code analyzers can look at the source code of a program to see if the code will perform as expected by generalizing over values that the program works with. In this thesis, we will consider these static analyzers.

1.2 Static analysis

A static analyzer that identifies all problems in an analyzed program is said to be sound. The ability to report these problems without reporting false negatives is called precision. Different users of static analysis may place different demands on the performance of the analyzer. Facebook is developing a static analyzer called INFER that places an emphasis on precision and fast reporting speed (Calcagno et al. 2015). For Facebook, it is important that the time of their developers is not spent investigating false negatives (i.e. the analyzer reports a non-existing bug) or waiting for the analysis to be complete.

On the other hand, compilers use static analysis to ensure that they can perform certain optimizations. If a compiler performs an optimization such as in-lining the result of an if statement based on the result of an unsound analysis, this can result in miscompilations. Developers tend to assume their compilers work correctly, so bugs in the compiler go unnoticed or are only found after much time has been spent debugging the input program. Therefore, compilers place a much higher emphasis on soundness.

There exist many kinds of static code analysis. The field is broad, as it refers to any kind of tool that inspects source code. One type of static code analysis is taint analysis, which seeks to find security bugs in a program. User input is often a source of security breaches, as a malicious user could input data that is specially crafted to exploit such a breach. A taint analyzer considers all user input as tainted and tracks the flow of tainted data through the program. Tainted data must be sanitized before it can be used in a high security context. An example of such sanitization is clearing illegal character from a submitted username and password to prevent an SQL injection attack. Another use of static analysis is a type checker. Type check-

ers are used in the compilers of many languages and assure that the provided program is well-typed, which means that all operations are used on types that support those operations. Addition is performed on numbers and boolean logic is only performed on booleans, etcetera.

Abstract interpretation is method of doing static analysis by abstracting over the properties of values in a program, such as parity or the sign of numbers. Research into abstract interpretation started in the 1970s by P. Cousot and R. Cousot (1977). They showed how reasoning over abstract properties could be used to perform program analysis, despite the resulting abstractions being imprecise. A few years later, they build upon their work to define a generic framework to capture the program flow using abstract interpretation (P. Cousot and R. Cousot 1979). Cousot then provided a method for designing an abstract interpreter based on a provided concrete interpreter (P. Cousot 1999). Using Galois connections, which model the relationship between concrete and abstract values, they showed that it is possible to calculate a sound abstract interpreter in what they called the calculational approach.

Proving that static analyzers are sound is a daunting task. Writing a proof manually is a long and cumbersome process. The writer of the proof will have to proceed methodically and keep track of the proof state.

It is possible to use a computer to help write these mathematical proofs. Proof assistants are computer programs that help mathematicians develop proofs by tracking the proof state precisely (Harrison, Urban, and Wiedijk 2014). Any proofs mechanized using an assistant can thus be treated with a high degree of trust.

Significant work on abstract interpretation in Coq, one such proof assistant, was done by Jourdan (2016) in their Verasco static analyzer. Using Coq, the Verasco analyzer is proven completely sound, although they do so in a way that is not flexible with regards the the language being analyzed. In their case this is not a problem because they focus specifically on the C programming language.

Verasco implements several abstract domains and covers a large part of the C language. It is a large project and therefore serves to highlight a challenge in mechanizing soundness proofs. The mechanization of such proofs is also labour intensive and seldom done. It is desirable to find ways to make it easier to develop these soundness proofs.

Keidel, Poulsen, and Erdweg (2018) developed a framework for decomposing soundness proofs into smaller, independent pieces. The idea is that by reusing lemmas and reasoning about shared parts between the abstract and concrete interpreters, the resulting proof effort can be shrunk significantly. The abstract and concrete interpreters are instantiations of a generic interpreter; this generic interpreter only uses methods exposed by an programming interface, and the abstract and concrete interpreters provide abstract and concrete implementations respectively of this interface. The problem of proving the soundness of the abstract interpreter then simplifies to proving the soundness of the corresponding methods of the interfaces. This paper served as the starting point for this thesis.

The proofs in Keidel, Poulsen, and Erdweg (2018) are pen-and-paper proofs. By implementing the ideas presented in an interactive proof assistant instead of Haskell, we can mechanize these pen-and-paper proofs thus making abstract interpreters that are proven sound more accessible. We have chosen to use Coq as our interactive proof assistant.

To build our interpreters, we have implement components that model the various side effects a computer program can have[1]. These side effects are encoded using monads, which allow us to write pure programs while still using these impure side effects. By using monads we are able to reason about the soundness of operations on values and soundness of side effects separately, which makes the necessary proofs easier. For each of these side effects we have also implemented corresponding monad transformers that take a monad and create a new monad with the combined side effects of the original monad and the monad transformer.

[1]This implementation can be found at `https://github.com/jensdewaard/thesis-coq-code`

We have implemented a monad for two such side effects: changing a global state and modeling failure. In addition, these monads are also equipped with lemmas proving that each instance is sound. One of the advantages Coq has over Haskell is the ability to require soundness proofs for each implemented monad.

Unfortunately, we did encounter difficulties proving the monadic laws for our transformed monads. Monads transformed by the monad transformer for failures are no longer monads. However, this did not prevent us from proving the soundness of the entire interpreter as soundness does not depend on the monadic laws.

1.3 Contributions

The goals of this thesis are to mechanize the soundness components of an interpreter as designed by Keidel, Poulsen, and Erdweg (2018) in Coq, and to document the challenges posed by implementing said work in a proof assistant. To this end, we list the following contributions.

- Soundness proof of an abstract interpreter of a small imperative language with exceptions and mutable state, but no loops. This shows that all components come together to prove an abstract interpreter sound and works as an example for other, more complex interpreters.
- Mechanization of small components that are individually proven sound and can be composed to build larger structures, namely by applying a sequence of monad transformers. This allows us to build an interpreter that suits the language we wish to analyze
- Automated resolution of the required components to prove soundness. Coq's automation mechanisms ease our proof burden by allowing us to provide abstract and concrete instances of the components. Each of these instances require short proofs, which will be composed by Coq to prove the entire composite structure sound.
- Mechanization of the definition of soundness via Galois connections, using only the concretization function of those connections, and preorders which allow us to define the soundness of join operators.

1.4 Outline

This thesis consists of 6 chapters. We start with the necessary background knowledge on abstract interpretation and how to read Coq in Chapter 2. In Chapter 3 we show how we have implemented the monadic components and required definitions and how we assemble these to build concrete and abstract interpreters. In Chapter 4 we show how we can prove these components sound and how the automation mechanics of Coq can aid us in this. In Chapter 5 we discuss other mechanization efforts, if they encountered the same problems we did and how they solved those problems, it at all. In Chapter 6 we close the thesis with a conclusion.

Chapter 2

Background

This chapter serves to provide the required knowledge for reading this thesis to readers without a background in abstract interpretation and proof mechanization. It consists of the following sections.

- In Section 2.1 we give a brief introduction into abstract interpretation by looking at a short Java program. We explain how abstract interpretation will allow us to be sure that the given program does not crash.
- In Section 2.2 we define a simple arithmetic language and develop a concrete interpreter for this language.
- In Section 2.3 we develop an abstract interpreter of this same language and show how it has to differ from the concrete implementation.
- In Section 2.4 we define what it means for this abstract interpreter to be sound with regards to the concrete interpreter and how we can write a proof of this soundness property.
- Finally, in Section 2.5 we explain the reader how to read the Coq definitions and proofs in the rest of the thesis.

2.1 What is Abstract Interpretation?

There are multiple ways of analyzing a computer program. If we wish to verify that a program performs as expected, we can run it on a set of inputs and verify that the output is what we would expect it to be. If we have written a program to add two integers, we can run that program on an input of 3 and 6 and ascertain that it indeed returns 9. Because we actually run the system under test, we call this dynamic analysis.

Another way to verify that the program works as intended is to look at the source code of the program. A reader can look for the usage of specific functions that often lead to bugs, or programmatic constructs that are misused. Analyzing the code in this way without running the program is called static analysis.

When looking at source code, we can often reason about what a program would actually do when operating on specific values. To illustrate, consider the following Java program which includes dividing two positive numbers, implemented as a class called `PosInt`.

```
1 public int myfunc(PosInt x, PosInt y) {
2     if y == 0 {
3         return 9;
```

```
4     } else {
5         return x / y;
6     }
7 }
```

In Java, dividing by zero results in a runtime error that crashes the program. We would like some assurance that the program will never crash. Traditional unit testing would run the program on several possible inputs, but these could never test the complete input space. Manually looking at the above function will tell us it will never divide by 0, but this quickly becomes infeasible as the program grows. Using abstract interpretation, we can proof that the function never crashes.

There is an easy way for us to be certain that the output of our abstract interpreter is an over-approximation of the output of our concrete interpreter: we can always say that the output can take any value. We call the abstract value that encompasses all concrete values Top, often written as $\top$. When doing interval analysis for example, $\top$ is equivalent to an interval of $[-\infty, \infty]$. While this is trivially sound, it is also as imprecise as we can get. When designing an abstract interpreter, one must consider how important precision and soundness are to the desired application.

Because the program will crash when dividing by zero, we should consider whether input variable y is equal to zero or not. In abstract interpretation, we abstract away the specific information about the values (e.g., $y = 0$) and reason only about properties of those values. Such properties can be varied and very specific like "y is 0", or very abstract such as y is a number.

By reasoning about these properties and looking at the source code, we can determine the possible paths the program execution may take. Let us consider the if statement in line 2 of the above Java code. If we track whether values are zero and y has such an IsZero property, we know that the first branch is entered and an exception will be raised on line 3. Likewise, if we were able to track that y has an IsNegative property, or a "$>$ 10" property, we know that program execution will continue with the else statement on line 5. Different abstractions result in different analyses. If the abstraction we use will not allow us to be certain as to whether y is zero (or not), then an abstract analysis would have to consider both branches of the if statement.

2.2 Concrete interpretation of an arithmetic language

In this section, we will define a simple arithmetic language in Coq, as well as a concrete and an abstract interpreter for that language. In our language, we should be able to model the above Java code, so our language will need division, if statements, variables and a way to communicate a crash. The concrete version of our language will operate on natural numbers and standard division defined on natural numbers.

```
Inductive expr :=
  | EVar : string -> expr
  | EVal : nat -> expr
  | EIfZero : expr -> expr -> expr -> expr
  | EDiv : expr -> expr -> expr.
```

The above Coq command creates an inductive type called `expr`. Values of type `expr` come in four forms, called constructors. Our type has the constructors `EVar` for references to variables, `EVal` for number literals in our program, `EIfZero` for comparing an expression to 0 and `EDiv` for multiplying the results of two expressions. With these constructors, we can reconstruct the earlier program.

```
Definition prog :=
  EIfZero (EVar "y")
    (EVal 9)
    (EDiv (EVar "x") (EVar "y")).
```

We can define functions and type definitions with the `Definition` command. The Coq code above states that value `prog` is equal to a combination of the constructors we defined earlier. Together, these constructors form a program that is equivalent to the Java program above. Like our earlier Java program, this program compares y to 0. If this is the case, the program returns 9. If y is not 0, it returns the result of dividing x by y.

```
Fixpoint eval (e : expr) (st : string → nat) : option nat :=
match e with
| EVal n => Some n
| EVar x => Some (st x)
| EIfZero g e1 e2 =>
    match eval g st with
    | Some O => eval e1 st
    | Some (S n) => eval e2 st
    | None => None
    end
| EDiv e1 e2 =>
    match eval e2 st with
    | None => None
    | Some y =>
        match eval e1 st with
        | None => None
        | Some x => div x y
        end
    end
end.
```

We create our evaluator as a `Fixpoint ...`, which is a type of function that can call itself recursively. In addition to taking the expression to be evaluated as an argument, the function also requires a mapping of strings to natural numbers. This mapping serves as the store where we retrieve the values of our variables. The evaluation returns an `option nat` instead of a plain natural number. This indicates that the evaluation can either return a natural number or fail.

When evaluating the `EDiv` expression, the evaluator calls another function `div` that performs the actual division. We assume that if the second operand of this function is zero, the computation fails and returns None. This models the throwing of a runtime exception.

```
Definition sample_map := t_update (t_update (t_empty 0) "x" 9) "y" 3.
Compute (eval (prog) (sample_map)).
(* = 3 : nat. *)
```

We can create maps via the `t_update` and `t_empty` functions. `t_empty` creates an empty map with a default element, and `t_update` adds a new entry to a map. `Compute` has Coq tell us the resulting value of a function. If we create a mapping that maps x to 9 and y to 3 and use that mapping to evaluate our sample program, we indeed get our expected value of 3.

2.3 Abstract interpretation of the language

When performing an abstract analysis, we generally need to create abstract versions of all the operations and values used in the analyzed program. In our case, we need to decide on an abstraction for natural numbers and create the division operation for that abstraction.

In this example, we abstract our natural numbers by defining an interval in which the number may lie. Intervals have a lower and an upper bound, so we will need values to express this in our abstract interpreter.

When dividing two intervals, we obtain the new lower bound of the interval by dividing the minimum bound of the numerator by the maximum bound of the denominator. The inverse is true of the upper bound.

```
Definition interval := (nat*nat).

Definition interval_div (i j : interval) : option interval :=
  let (i_min, i_max) := i in
  let (j_min, j_max) := j in
  match j_min with
  | 0 => None
  | j_min' => (Nat.div i_min j_max, Nat.div i_max j_min)
  end.
```

The above Coq code shows the definition of an interval as a combination of two natural numbers and defines division on intervals. When using `Definition` to define a function, we give the required arguments between the parentheses. For clarity, we also annotate the type of the resulting value. This is `option interval`

Now that we have abstract values, we can define the abstract interpreter that uses these values when interpreting the program defined above. In the first constructor `EVal` we describe how to define literal, concrete values in our language. To be able to evaluate these commands we will need a function to extract an abstract value from a literal.

```
Definition extract (n : nat) : interval := (n,n).
```

The extract function takes a concrete value and turns it into an abstract approximation of that value. Later in Chapter 3 we see how we make this function more generic.

```
Definition join (i j : interval) : interval :=
  let (i_min, i_max) := i in
  let (j_min, j_max) := j in
  (Nat.min i_min j_min, Nat.max i_max j_max).
```

We have defined a function `join` that takes two intervals and returns a new interval with the minimum of the two lower bounds and the maximum of the two upper bounds as new bounds. We use this function to merge the results of the two branches of the if statement in the case when we are unsure about the branch taken by the concrete interpreter.

```
  Fixpoint eval_abstract (e : expr) (st : string -> interval)
    : option interval :=
  match e with
  ....
  | EIfZero g e1 e2 =>
      match (eval' g st) with
      | None => None
```

```
        | Some (0,0) => (eval' e1 st)
        | Some (S _, _) => (eval' e2 st)
        | _ =>
            match (eval' e1 st), (eval' e2 st) with
            | None, _ | _, None => None
            | Some i, Some j => Some (join i j)
            end
        end
    | EDiv e1 e2 =>
        match (eval' e2 st) with
        | None => None
        | Some y =>
            match (eval' e1 st) with
            | None => None
            | Some x => interval_div x y
            end
       end
   end.
```

The most interesting case here is the evaluation of `EIfZero`. The evaluator looks at the value of the guard, the expression being considered in the if statement. If both interval bounds are zero we know that the expression is definitely zero, so the entire if statement evaluates to the value of the first branch. Likewise, if the lower bound is above zero, we know that the concrete number will never be zero and can evaluate to the second branch. If the interval contains zero, we know that the program may take either path and join the results.

We will run the abstract interpreter on an abstract version of our earlier map. In this example, we know that x is equal to 9, but y can be anywhere between 1 and 3. The result of our computation is as we expected, a value between 3 and 9.

```
Definition abstract_map := t_update
     (t_update (t_empty (0,0))
         "y" (1,3))
         "x" (9,9).
Compute (eval' prog abstract_map).
(* = Some(3,9) *)
```

If we change the lower bound of the denominator to zero, the output of the abstract interpreter changes. It becomes `None` to indicate that the program may crash as the denominator may be zero. A more sophisticated abstract interpreter may recognize that the surrounding if statement guards against this, but we did not implement this in this example for simplicity.

Now that we have a concrete and an abstract interpreter, we can look at what it means for the abstract interpreter to be sound with regards to the concrete interpreter. For this we will introduce partially ordered sets, lattices and Galois connections.

2.4 Definition of Soundness

Before we can prove that our abstract interpreter is sound, we should define what soundness is. We say that an interpreter is sound when, if the inputs to the abstract interpreter are valid approximations of the inputs to the concrete interpreter, then the output of the abstract interpreter is a valid approximation of the output of the concrete interpreter.

Recall the Java program in 2.1, in which we branch based on the value of y. The more precise our analysis is, the more restricted the abstract interval for possible values of y. This

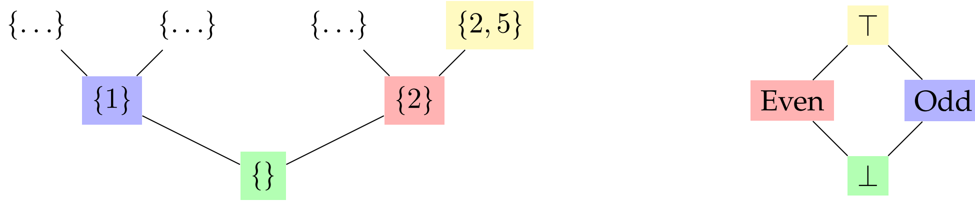

Figure 2.1: Corresponding elements between two partially ordered sets

means that if the compiler can be certain that y is always 0, it can in-line the first branch of the if statement as part of an optimization step. Or if the interval of possible values for a variable that is used to index an array is larged than the array, the compiler may warn the programmer of a possible memory error.

To define the mappings between the concrete and abstract domains, we explore three mathematical definitions; partially ordered sets, lattices and Galois connections. Partially ordered sets are sets equipped with a relation that may hold for two elements of the set. This relation is called the *partial order* and means that one element somehow precedes another. This partial order should be reflexive, transitive and antisymmetric. Partially ordered sets differ from totally ordered sets in that not all elements of the set need to be able to be compared.

For example, natural numbers form a partially ordered set with the less-than-equals operation as the order. In fact, because $\forall x, \forall y, x \leqslant y$ or $y \leqslant x$, they also form a totally ordered set. We are also able to define a partial order for our intervals. Because we use the intervals as a set of possibilities for the concrete value, we can use set inclusion as the order. An interval i is less than or equal to an interval j if all numbers included in i are also included in j.

If for every two elements x and y in a partially ordered set there exists elements z and w such that $x \leqslant z$ and $y \leqslant z$ and such that $w \leqslant x$ and $w \leqslant y$, and z and w are respectively the least and greatest such values, that partially ordered set forms a lattice. It also requires the operations join and meet to result in these bounds.

Now that we know that we can define lattices for both sets of natural numbers and intervals, we can introduce Galois connections. A Galois connection is a connection between two partially ordered sets $(C, \leqslant)$ and $(A, \leqslant)$ consisting of two monotone functions $\gamma : A \to C$ and $\alpha : C \to A$, such that for all $a \in A$ and $c \in C$, $\gamma(a) \leqslant c$ if and only if $a \leqslant \alpha(c)$.

In abstract analysis, these functions are called the concretization and abstract functions. They define the correspondence between the concrete and abstract domains. Because both function are monotone, we have the property that $c \leqslant \gamma(\alpha(c))$. Concrete values that are abstracted and then concretized again should be larger or equal to what they were before. Any abstract interpreter that upholds this property is considered sound.

In our work, we focus our attention on Galois connections between sets of concrete values and abstraction of those sets. In the above example, this means we deal with sets of natural numbers and intervals. Our abstract evaluation is sound if the output of the abstract evaluator over-approximates the output of the concrete interpreter. In terms of the above functions, that means $eval \leqslant \gamma \cdot eval_abstract$.

The γ-function we require for this is different for every Galois connection, and we have a different Galois connection between every concrete type and every possible abstraction. In the Coq code below, we define a γ-function to convert intervals into sets of natural numbers.

```
Definition gamma_interval (i : interval) (n : nat) : Prop :=
  let (i_min, i_max) := i in
  i_min <= n /\ n <= i_max.
```

In 2.1, we show two Hasse diagrams of partially ordered sets. The left Hasse diagram contains a few elements of the partially ordered set of sets of natural numbers. On the right is a Hasse diagram portraying a partially ordered set of parities, abstractions of whether a number is even or odd. Matching colours indicate that the left element is mapped to the right element via the γ-function. The $\top$ and $\bot$ symbols denote Top and Bottom, which correspond to all elements and no elements respectively.

The added value of Coq over other functional programming languages such as Haskell comes from the ability to take definitions and prove properties about them. We start a proof with the `Lemma` command. In the below code, we define a lemma called `eval_sound` which states that for all stores, if all corresponding values in those stores are sound, then our evaluator is sound for all possible expressions. We can indeed proof this in Coq using the above definitions. The entire proof is omitted here, as even for such a simple evaluator the proof becomes lengthy. Proofs are delimited by the `Proof.` and `Qed.` commands. In between these commands come a series of *tactics*, which are special commands that help us complete the proof.

```
Lemma eval_sound : ∀ st st',
  (∀ x, gamma_interval (st' x) (st x)) ->
  (∀ e, gamma_interval (eval_abstract e st') (eval e st)).
```

As we add more features to our interpreter, such as exceptions and mutable state, the required proofs grow even longer and more complex. In the rest of the thesis, we show how we have implemented the work of Keidel, Poulsen, and Erdweg (2018) to lessen the proof burden and what challenges we faced in doing so.

2.5 Additional Coq Commands

We have used Coq to write our definitions and prove the soundness of the abstract interpreter (*A Short Introduction to Coq* n.d.). In this section, we will give some examples of Coq code along with an explanation of what is meant by the code. Readers familiar with Coq can skip this section. Other readers may wish to read it to help understand later code listings.

```
Definition interval_div_lambda : interval -> interval -> interval :=
    λ i, λ j,
         let (i_min, i_max) := i in
         let (j_min, j_max) := j in
         (Nat.div i_min j_max, Nat.div i_max j_min).
```

Here, we have defined interval division using lambda notation. The type signature tells us that `interval_div_lambda` is a function that takes an interval and returns a function that takes an interval and returns a function.

```
Theorem eval_sound : ∀ st st',
  (∀ x, gamma_interval (st' x) (st x)) ->
  (∀ e, gamma_interval (eval_abstract e st') (eval e st)).
```

Aside from `Lemma`, there are several other commands we can use to state a property or law. These keywords `Theorem`, `Lemma`, `Corollary`, `Remark`, `Fact` and `Proposition` are all equivalent. Any semantic difference is up to the developer. In our work, we will mostly use the `Lemma` keyword. We shall use the `Theorem` keyword for the final proof stating the soundness of the interpreters. On occasion, we will use the `Corollary` command to state a lemma that follows trivially from the preceding lemma.

```
Notation "℘ A" := (A -> Prop) (at level 0, only parsing).
Infix "/" := interval_div (at level 40).
```

To keep our Coq code close to the mathematical theory behind it, we introduce a lot of notations. The `Notation` command allows us to introduce those notations to Coq. The new notation is between the quotes, with its definition following after the `:=`. The term between the second pair of brackets is technical information about how Coq should parse the notation, such as left or right associativity. The `Infix` command is a shorthand for introducing a infix notation for a binary function.

```
Class Galois (A A' : Type) : Type :=  γ : A -> ℘ A'.
```

In this thesis we utilize the typeclass mechanism of Coq. Typeclasses are a way to define methods and lemmas for types that implement that typeclass. In the above example, we define a Galois typeclass that takes two types. It states that implementations of this typeclass should define a function γ of type $A' \rightarrow \wp\, A$.

```
Instance galois_parity_nat : Galois interval nat := gamma_interval.
```

To create an instance of a typeclass, we need to define the required functions and lemmas. Here, we have defined an inductive relation called `gamma_interval` that relates instances of `interval` with sets of natural numbers. We then use that definition in creating an `Instance` of the Galois typeclass.

```
  Context {M : Type -> Type} {RO : return_op M}
    {BM : bind_op M} {MF : MonadFail M}.
```

To simplify definition and lemmas, it is possible to use the `Context` command to define variables that are required on all following definitions. This improves readability of the Coq code.

Chapter 3

Building The Interpreters

In this chapter we will describe how the mechanizations are implemented in Coq and the theory behind them. The goal is to have a definition of a generic interpreter at the end of the chapter, as well as concrete and abstract instantiations of this generic interpreter.

- First, in Section 3.1 we describe the syntax of a small imperative language without loops and define a concrete interpreter of this language. In contrast with the arithmetic language defined in Chapter 2, this language will have mutable state and exceptions.
- Then, in Section 3.2 we show a concrete interpreter for this language. We build the interpreter using monads, explain what monads are in Section 3.3 and show how we can decompose the interpreter for our language due to its monadic structure.
- In Section 3.4 After defining a concrete interpreter for our language, we define an abstract interpreter for abstract interval analysis that will operate on this same language. We highlight the differences between these two interpreters, as well as the similarities.
- In Section 3.5, based on the similarities noted in the previous section, we define a generic interpreter. We show how we can instantiate the generic interpreter and obtain our previously defined concrete and abstract interpreters.

After obtaining two interpreters based on a generic interpreter we will show how to prove the abstract interpreter sound with regards to the concrete interpreter in Chapter 4.

3.1 The Language

Programs consist of statements, expressions and values. These programs are evaluated by interpreters. In this subsection, we shall describe the syntax of a small imperative language without loops.

```
Definition cvalue : Type := (nat + bool)%type.
```

Our language has two types of values. Natural numbers, represented by the Coq type `nat` and boolean values, represented by `bool`. We define concrete values as the sum of these two types. A sum of two types is a type itself and means that values of that type can be of the type of either its components.

To be able to use this sum type, we create a typeclass to defines how to convert the component types into the composite type and vice versa. This implementation was inspired by Liang, Hudak, and Jones (1995).

```
Inductive expr : Type :=
  | EVal : cvalue -> expr
  | EVar : string -> expr
  | EPlus : expr -> expr -> expr
  | EMult : expr -> expr -> expr
  | EEq : expr -> expr -> expr
  | ELe : expr -> expr -> expr
  | ENot : expr -> expr
  | EAnd : expr -> expr -> expr.

Inductive com : Type :=
  | CSkip : com
  | CSeq : com -> com -> com
  | CAss : string -> expr -> com
  | CIf  : expr -> com -> com -> com
  | CTryCatch : com -> com -> com
  | CThrow : com.


Class SubType (sub : Type) (super : Type) : Type := {
  inject : sub -> super;
  project : super -> option sub
}.
```

The SubType typeclass specifies a relation between two types that can be converted into one another. The inject method takes a subtype and turns it into the super type. The project type performs the inverse operation. However, because a value of the super type is not necessarily an instance of the subtype, we include the possibility of failure. As an example, we show the instance of the SubType class that converts between `nat` and `(nat + bool)`.

```
Instance subtype_l : ∀ {A B}, SubType A (A + B) := {
  inject := inl;
  project := λ s, match s with | inl x => Some x | _ => None end
}.
```

In the above instance, `inl` is the constructor that takes a value `a : A` and returns a value of type `(A+B)`. Coq will automatically use the `inject` function to turn a natural number into the composite type `cvalue` using its typeclass inference mechanism. The `project` function can turn values of `(A+B)` into values of `option A`.

Expressions are operations on values that result in a new value. They are implemented as an inductive type called `expr`.

Expressions can exist of concrete values (‘5’, ‘true’), names of variables saved in a store (see Section 3.3.4) and operations on these values (+, *, ==, <=, negation and `&&`).

Statements control the flow of the program. In this case, we have implemented a skip statement that effectively does nothing, an assignment statement that assigns the result of an expression to a variable, an if statement, a try-catch statement and a throw statement that acts as an exception.

3.2 The Concrete Interpreter

As shown in Section 3.1 the syntax of the language allows the two types of values, booleans and natural numbers, to be used interchangeably while still having a valid syntax. However,

most operations will be meaningless if used on the wrong operators. For example, "1 + true" is valid syntax, but it is not a valid program.

To ensure that the correct types are used when evaluating operators, the interpreter has an `ensure_type` function. Using the `project` function from the `SubType` class, we can see if the provided value is of the desired type.

```
Definition ensure_type (subType : Type)
  {M : Type -> Type} {MM : Monad M} {MF : MonadFail M}
  {valType : Type}
  {ST : SubType subType valType}
  (n : valType) : M subType :=
  match project n with
  | Some x => returnM x
  | None => fail
  end.
```

We explain the meaning of Monad and MonadFail in Section 3.3. For now, the important part is that the ensure function is a function that is capable of failing. This allows us to reuse the function with whatever monad we desire, as long as it has the capability for failing.

When the ensure function is called with a type other than the desired type, it returns the fail value of its monad M. This failure is propagated throughout the rest of the program. This allows us to model the interpreter crashing if the program is not well-typed.

```
Fixpoint eval_expr {M} {MF : MonadFail} {MS : MonadState M}
    (e : expr) : M cvalue :=
  match e with
  | EVal x => returnM (inject x)
  | EVar x => st << get ;
      returnM (st x)
  | EPlus e1 e2 =>
      v1 << (eval_expr e1) ;
      v2 << (eval_expr e2) ;
      n1 << (ensure_type nat v1) ;
      n2 << (ensure_type nat v2) ;
      returnM (inject (n1 + n2))
  | EMult e1 e2 =>
      v1 << (eval_expr e1) ;
      v2 << (eval_expr e2) ;
      n1 << (ensure_type nat v1) ;
      n2 << (ensure_type nat v2) ;
      returnM (inject (n1 * n2))
  | EEq e1 e2 =>
      v1 << (eval_expr e1) ;
      v2 << (eval_expr e2) ;
      n1 << (ensure_type nat v1) ;
      n2 << (ensure_type nat v2) ;
      returnM (inject (Nat.eqb n1 n2))
  | ELe e1 e2 =>
      v1 << (eval_expr e1) ;
      v2 << (eval_expr e2) ;
      n1 << (ensure_type nat v1) ;
      n2 << (ensure_type nat v2) ;
```

```
      returnM (inject (Nat.leb n1 n2))
  | ENot e =>
      v << (eval_expr e) ;
      b << (ensure_type bool v) ;
      returnM (inject (negb b))
  | EAnd e1 e2 =>
      v1 << (eval_expr e1) ;
      v2 << (eval_expr e2) ;
      b1 << (ensure_type bool v1) ;
      b2 << (ensure_type bool v2) ;
      returnM (inject (andb b1 b2))
  end.
```

The concrete interpreter follows the same structure for each kind of operator. The smaller expressions are evaluated first via recursive calls to the evaluation function. Their returned values are ensured by the interpreter to be of the proper types, so that we may recognizing ill-typed programs. Then the built-in operator corresponding to the expression is called on those values. In addition to the `MonadFail` constraint, the monad used in this interpreter also needs to be an instance of `MonadState`. This constraint requires a monad to have ways of retrieving and setting variables from and to a global state.

```
Fixpoint ceval {M} {MM : Monad M} {ME : MonadExcept M}
    {MS : MonadState M} {MF : MonadFail M}
    (c : com) : M unit :=
  match c with
  | CSkip => returnM tt
  | c1 ;; c2 =>
      (ceval c1) ;; (ceval c2)
  | x ::= a =>
      n << (eval_expr a) ;
      st << get ;
      put (t_update st x n)
  | CIf b c1 c2 =>
      v << (eval_expr b) ;
      b' << (ensure_type bool v) ;
      if b' (ceval c1) (ceval c2)
  | try c1 catch c2 =>
      catch (ceval c1) (ceval c2)
  | CFail => fail
  end.
```

The `ceval` function acts as the concrete interpreter of our language. It is parametrized on a type M, which is required to be an instance of several typeclasses. Each of these typeclasses ensures that our type M has a certain capability. We could have added these capabilities directly into the definition of our interpreter. However, by decomposing these capabilities into monads, we can prove properties about those monads directly and prove them for all interpreters built with those monads. The monad used here also has a `MonadExcept` requirement, which gives it the ability to handle exceptions via the catch method.

Do notation

The interpreters make excessive use of the bind function of the monads. Because of this, we introduce a special notation for the function, also borrowed from Haskell.

If we did not include this notation, making use of the Monad would quickly become cumbersome. With it, we are able to write data pipelines in an imperative style. To illustrate, the below code shows the two different styles. As the length of the 'imperative' program increases, the functional notation grows more cumbersome and error prone due to the number of parentheses involved.

```
Example do_notation :=
    v << eval_expr e ;
    b << ensure_bool v ;
    returnM (inject (negb b)).

Example nested_notation :=
    (eval_expr e) >>= (λ v,
        (ensure_bool v) >>= (λ b,
            returnM (inject (negb b)))).
```

3.3 Monads

Many tutorials explaining monads exist and monads are generally regarded as difficult to understand (Petricek 2018). We will briefly try to explain them here, though a complete understanding monads is not necessary to follow later sections. Understanding that we can decompose the program along the lines of the methods provided by the Monad typeclass is sufficient.

Monads are a way to add impure effects to a pure language. Examples of impure effects are modifying a global state, throwing and handling exceptions and perform IO operations. A Computer Science course (*IO and monads* n.d.) at the University of Pennsylvania likens monads to a recipe, a description of steps to take. Such descriptions are values themselves, which makes using monads pure.

In Coq we can implement monads in the form of a typeclass (Sozeau and Oury 2008), as it is done in Haskell (Wadler 1995). Typeclasses are somewhat like interfaces in object-oriented programming languages. A programmer can write functions operating on typeclasses instead of concrete types, and those functions can accept all types that implement the functions defined in that specific typeclass. In this case, all types belonging to the Monad typeclass must implement two functions, *bind*, originally called * in the paper by Wadler, and *return*, originally called *unit*. An implementation of monads as a typeclass in Coq is given below.

```
Class Monad (M : Type → Type) : Type :=
{
  returnM : ∀ {A}, A -> M A;
  bindM : ∀ {A B}, M A -> (A -> M B) -> M B;
}.
Notation "m >>= f" := (bindM m f) (at level 40, left associativity).
```

Because return is a reserved keyword in Coq, we rename both `return` and `bind` into `returnM` and `bindM` respectively. `returnM` takes a value and wraps it in the monad. `bindM` is a function that can apply a function to a monadic value. As is the convention in Haskell, we will use »= as an infix notation for the bindM function.

One of the advantages that Coq typeclasses have over those in Haskell is the ability to add lemmas to the typeclass in addition to the methods. Proper monads have to observe three laws, together called the Monad laws. These laws are left identity, right identity and associativity. These mean that binding `returnM` to a function f is equivalent to just applying

f to the value wrapped by returnM. The second law states that binding to returnM is effectively a no-op, and the last law requires bindM to be associative. In Coq, we can add the following lemmas to a typeclass to ensure that all its instances are valid monads, which is something that Haskell cannot guarantee.

```
Class CorrectMonad (M : Type -> Type) : Type := {
  bind_id_left : ∀ {A B} (f : A -> M B) (a : A),
    bindM (returnM a) f = f a;
  bind_id_right : ∀ {A} (m : M A),
    bindM m returnM = m;
  bind_assoc : ∀ {A B C} (m : M A) (f : A -> M B) (g : B -> M C),
    bindM (bindM m f) g = bindM m (λ a, bindM (f a) g);
}
```

When we will define the monad transformer for the abstract option monad in Section 3.4.2, we shall see that the resulting type does not satisfy all the monadic laws. Because it turns out that the interpreters and their soundness proofs do not depend on these laws, we have not added the laws to our Monad typeclass.

Eventually, we will also see that the bind method of the optionAT monad transformer requires additional constraints on the transformed monad M compared to its return method.

For our interpreter, we require monads that add two different side effects. The first is the option monad, also known as Maybe in the Haskell standard library. The option monad adds the possibility of failure to an operation. The other is the State monad, which allows us to keep track of a global state. In the next few sections we will discuss the implementation of these monads.

3.3.1 Option

```
Inductive option A : Type :=
    | Some : A -> option A
    | None : option A.
```

The first monad we use is the option monad, which is based on the option type from the Coq standard library. Values of the type option A can be of either the form Some a, where a is a value of type A, or None. Here, None represent a failed computation. For example, a monadic version of division nat -> nat -> option nat could return None when dividing by zero, and a Some value with the right answer in other cases.

The return method of option is the same as the Some constructor. The bindM method of the option monad performs a case analysis on the provided value. In the case of Some, it extracts the inner value and applies the given function to it. In the case of None, it returns None. This allows the chaining of multiple operations without explicit error handling between each step.

```
  Definition return_option {A} : option A := Some.

  Definition bind_option {A B}
  (m : option A) (k : A -> option B) : option B :=
    match m with
    | None => None
    | Some a => k a
    end.

  Instance option_monad : Monad option :=
```

```
{
  returnM := return_option;
  bindM := bind_option;
}.
```

Recall that the monadic laws require that `returnM` is a left and right identity for `bindM`, and that `bindM` is associative. If we have proven this for `Some` and `bind_option`, we can combine those three lemmas together with the definition of `bind_option` and `Some` into an instance for the `CorrectMonad` typeclass.

3.3.2 State

When we evaluate a program, we will require a way to keep track of a global state. This state can be as simple as an integer that keeps a count of the number of operations performed, or a mapping of strings to values `V`, as was the case in our interpreter in Chapter 2. Recall that in the `eval` and `eval'` functions we had to pass along the `st` and `st'` values to each recursive call.

The State monad allows us to add the ability for tracking state to a pure program without referring to a global state variable or explicitly passing along such a variable. It is defined as `Definition State (S A : Type) := S -> (A * S).` The State monad wraps a value in a function that keeps track of an additional value of type S. Looking at the types of State and return shows us how we ought to implement the return function for State.

```
Definition return_state {S A} : A → State A :=
    λ a : A, λ st : S, (a, st).
```

The `bindM` method for State passes the state variable along to the next computation. Updated states are passed along and the current state is accessible at each step of the computation operation.

```
Definition bind_state {A B}
  (m : State S A) (f : A -> State S B) : State S B :=
  λ st, let (x, st') := m st in f x st'.
```

The above code shows us how the bind method is implemented. The result of binding a State value to a function is a new function (as all values of State need to be) in which the provided state variable is applied to the State value. The result of this computation, including the new updated state, is then passed to the continuation function `f`.

In theory, this monad should work for any type S that respects the above laws. The monad can be used with an integer that counts something, or a boolean that tracks the current player in a two-player game. In our work we want to use the State monad to track variables, so we use a mapping between strings and values as a type for S.

The State and option monads show us how we would implement the desired properties of our small imperative language in a pure manner. But right now, we will have to choose between the two. We can either have the evaluator utilize the State monad to keep track of state, or have it use the option monad to allow exceptions, but we cannot yet have a runtime in which we can both throw exceptions as well as keep track of a global state. There is a way to combine the functionalities of monads via the use of monad transformers.

3.3.3 Monad Transformers

Monad transformers (Liang, Hudak, and Jones 1995) are a way to add the side effects of a monad to another monad. For both of the monads we defined earlier, we define a corresponding type that takes a monad as an input and returns a new monad instance.

```
Context {M : Type -> Type}.
Definition optionT M A: Type := M (option A).
Definition StateT S M A : Type := S -> M (A*S)%type.
```

Listing 1: Definitions for the monad transformers

We define two transformer monads that correspond to the monads defined above: optionT and StateT. These extend a monad by allowing for failing computations and the ability to retain a global state.

When providing instances for the Monad typeclass for these types, we require the use of the `bindM` and `returnM` methods of the wrapping monad `M`. For example, in the definition of `bind_optionT` below, we see that the implementation unwraps the monad M, exposing the underlying option value. We perform a case analysis on this value in a way that is similar to how we defined `bind_option`, `None` values are propagated and values wrapped with `Some` are threaded through chained bind calls.

```
Definition bind_optionT {A B} (x : optionT M A)
  (f : A -> optionT M B) : optionT M B :=
   x >>= (λ v : option A,
    match v with
    | None => returnM None
    | Some a => f a
    end
  ).

Inductive Identity (A : Type) : Type :=
    identity : A -> Identity A.
```

Aside from State and option, we also implemented the Identity monad. This monad, like the identity function, adds no extra functionality. Its value lies in the ability to use it where we would a monad in cases where we don't want to add extra functionality. To reduce our code base, we implemented State as the StateT transformer applied to the Identity monad. The resulting monad simplifies to our prior definition of State, without us needing to prove soundness of both `State` and `StateT`. Having access to the Identity monad also opens up the possibility of reusing the monad when implementing more monad transformers.

3.3.4 MonadFail, MonadExcept and MonadState

With the use of monad transformers we are able to build a so-called monad stack to encode the side effects we want in a pure way. However, when we write code that reasons about possible side effects, we want to abstract away the specific stack that is used. It is therefore desirable to be able to reason about the abilities provided by a given monad stack.

For this use case, we have ported MonadFail, MonadExcept and MonadState from Haskell (Gibbons and Hinze 2011). The easiest way to explain why we use these typeclasses is that the typeclasses option and State were more concerned with *how* the side effects were implemented, whereas these typeclasses deal more with *what* side effects a type can cause in a monad stack.

```
Class MonadFail M {BM : bind_op M} : Type := {
  fail : ∀ {A : Type}, M A;
}.
```

This is the definition for the `MonadFail` typeclass. The definition tells us that all instances of `MonadFail` have a value called `fail`. Convention holds that when this value is used on the left-hand side of a bind, the resulting value is also `fail`. We shall follow that convention as well in our instances of MonadFail.

The implementation of this typeclass for the option monad is simple. We can use `None` as the `fail` value. The proof for the `fail_left` lemma, which states that `fail` on the left side of a bind results in `fail`, follows readily from the definition of `bind_option`. The same goes for optionT, in which case our fail value is `returnM None`.

However, things get slightly more interesting when looking at `StateT`. `StateT` preserves the ability for failure if the underlying monad `M` is an instance of MonadFail. The `Identity` monad is not an instance of `MonadFail`, so the standard `State` monad is not an instance either.

```
Section fail_stateT.
  Context {M : Type -> Type} {RO : return_op M}
    {BM : bind_op M} {MF : MonadFail M}.
  Context {S : Type}.

  Definition fail_stateT {A} : StateT S M A :=
    λ st, fail >>= λ a, returnM (a, st).

  Lemma fail_stateT_left : ∀ (A B : Type) (f : A -> StateT S M B),
    fail_stateT >>= f = fail_stateT.
  Proof. ... Qed.

  Global Instance monad_fail_stateT : MonadFail (StateT S M) :=
  {
    fail := fail_stateT;
  }.
End fail_stateT.
```

The above code shows how the usage of fail method of the underlying MonadFail is changed to work for the transformed StateT monad instead. In the definition of `fail_stateT`, the `fail` value is that of the underlying monad `M`. This value is bound to the return method of `StateT`, which ”lifts” the fail value to a value of type `StateT S M A`. The proof unfolds the methods defined in terms of StateT and rewrites the goal based on the monadic laws for bind and the fail_left lemma.

Now that we have described monads that can fail, we will shift the focus to monads that can recover from failure. To describe this ability, we have the MonadExcept typeclass.

```
Class MonadExcept M {RO : return_op M} {BM : bind_op M} := {
  throw : ∀ {A}, M A;
  catch : ∀ {A} {JA : Joinable A A} {JAI: JoinableIdem JA},
    M A -> M A -> M A;
}.
```

For all instances of MonadExcept, we define a throw value and a catch method. These are a lot like try-catch functionality in languages such as Java. The throw function is a value of the monad `M`. The catch function should take two values of type `M A`. If the first is throw, it returns the second provided value. If it is not, it returns the first provided value. This behaviour is encoded in the lemmas associated with the typeclass, which should hold for all valid instances of the class.

```
Definition store := total_map cvalue.

Definition t_update {A:Type} (m : total_map A)
                    (x : string) (v : A) :=
  λ x', if beq_string x x' then v else m x'.



Class MonadState (S : Type) (M : Type -> Type) {MM : Monad M} :=
{
  get : M S;
  put : S -> M unit;
}.
```

In the same way that we use typeclasses to easily access the fail effects, we can use a typeclass to access the state that is carried by the (transformed) monad. Any monad implementing the MonadState typeclass should provide access to a get method that sets the current state as the 'return value' of the function, and a put method that updates the carried state with the given value.

```
Section store_stateT.
  Context {S : Type}.
  Context (M : Type -> Type) {MM : Monad M}.

  Definition stateT_get := λ s : S, returnM (s, s).

  Definition stateT_put := λ s : S, λ _ : S, returnM (tt, s).

  Global Instance store_stateT :
  MonadState S (StateT S M) :=
  {
    get := stateT_get;
    put := stateT_put;
  }.
End store_stateT.
```

Stores

The State monad is instantiated with a type S that should be some sort of storage for the state. In our language, we want to be able to use variables from a store that we can read from and write to.

The concrete store is a map[1] from strings to concrete values. The store is implemented as a total map, which is a map that returns a default value when a key is requested that is not present in the map. Values can be added to the map via the `t_update` function.

We could also have used a partial map. Partial maps return an option, instead of the undecorated type. Returning the default None value would be like throwing an exception. The decision matters little for proving soundness, as the soundness of the interpreter will be predicated on the soundness of the starting states.

[1]The *total_map* comes from Map.v from the public Software Foundations course(*Software Foundations 1 : Logical Foundations* n.d.).

3.4 The Abstract Interpreter

3.4.1 Abstract Values

Defining the abstract domains is an important decision when doing abstract analysis. Different abstractions lend themselves to different types of analyses. Included in our project are two different abstractions of natural numbers and an abstraction of booleans.

Parities

The first abstract type we define is `Parity`.

```
Inductive parity : Type := par_even | par_odd | par_top.
```

The parity of a natural number is whether it is even or odd; e.g. the parity of the number 2 is even, and the parity of the number 9 is odd. Operations that we can define for natural numbers can also be defined for parities. Our language can deal with addition and multiplication, so we have defined these operations for parities.

```
Definition parity_plus (p q : parity) : parity :=
  match p with
  | par_even => q
  | par_odd => match q with
               | par_even => par_odd
               | par_odd => par_even
               | par_top => par_top
               end
  | par_top => par_top
  end.

Definition parity_mult (p q : parity) : parity :=
  match p, q with
  | par_even, _ | _, par_even => par_even
  | par_top,  _ | _, par_top => par_top
  | _, _ => par_odd
  end.
```

Intervals

Parities are only one possible abstraction of natural numbers. Another possibility would be to use intervals. In this section, we describe the implementation of intervals using dependent records.

An interval of natural numbers is a valid abstraction of a set of natural numbers if each of the natural numbers in the set falls within the interval. For example, the interval `[1,5]` is an abstract approximation of the set `{2, 4, 5}`. It is also a valid approximation of the set `{1,5}`.

The same operators than can be defined on natural numbers can be defined on intervals. Addition, multiplication and comparisons can all be implemented. Adding the intervals `[1,3]` and `[4, 7]` would result in the interval `[5, 10]`, for example. Whenever we define an interval, we also need to provide a proof that the minimum end of the interval is less than the maximum end. This is represented in the record by the `min_max` field.

```
Record interval := Interval {
  min : nat;
```

```
  max : nat;
  min_max : min <= max;
}.

Definition iplus (i1 i2 : interval) : interval :=
  Interval ((min i1) + (min i2)) ((max i1) + (max i2)) (plus_min_max i1 i2).

Definition interval_mult (i1 i2 : interval) : interval :=
  Interval ((min i1) * (min i2)) ((max i1) * (max i2)) (mult_min_max i1 i2).
```

Unlike parity, our `Interval` type does not have a candidate for a top value that we can implement in Coq. To work around this, we introduce borrow the `top_op` class from Verasco (Jourdan 2016). By providing two constructors, `Top` and `NotTop`, we can add a top value to any type. Furthermore, we can define lemmas about such lifted types once and for all.

```
Class top_op (A:Type) : Type := top : A.
Inductive toplift (A: Type) :=
  | Top : top_op (toplift A)
  | NotTop : A → toplift A.
Notation "t +T" := (toplift t) (at level 39).
```

Abstract Booleans

The abstract booleans are only slightly different from their concrete counterpart. We supplement the normal boolean values (true and false) with a value for top, again using the toplift type.

```
Definition abstr_bool : Type := bool+T.
```

When we use our abstract booleans, we first do a case analysis to determine whether we are dealing with ⊤. If we are not, we can perform whatever operations on booleans we intended to do.

3.4.2 The Interpreter

We again define the interpreter in a monad-agnostic way. We require the used monad to have the same capabilities as the one in the concrete interpreter, namely failing and state.

The abstract state differs from the concrete state in what values, and with those, what type of store is used. The abstract store is almost the same is the concrete store, but returns abstract values instead of concrete values. When concrete values are inserted in the store, they are first converted into their abstract counterpart via the extract function.

```
Definition avalue := (parity+abstr_bool)%type.

Fixpoint eval_expr_abstract {M} {MF : MonadFail M} {MS : MonadState M}
    (e : expr) : M avalue :=
  match e with
  | EVal x => returnM (inject x)
  | EVar x => st << get ;
      returnM (st x)
  | EPlus e1 e2 =>
      v1 << (eval_expr_abstract e1) ;
```

```
      v2 << (eval_expr_abstract e2) ;
      n1 << (ensure_type parity v1) ;
      n2 << (ensure_type parity v2) ;
      returnM (inject (parity_plus n1 n2))
  | EMult e1 e2 =>
      v1 << (eval_expr_abstract e1) ;
      v2 << (eval_expr_abstract e2) ;
      n1 << (ensure_type parity v1) ;
      n2 << (ensure_type parity v2) ;
      returnM (inject (parity_mult n1 n2))
  | EEq e1 e2 =>
      v1 << (eval_expr_abstract e1) ;
      v2 << (eval_expr_abstract e2) ;
      n1 << (ensure_type parity v1) ;
      n2 << (ensure_type parity v2) ;
      returnM (inject (parity_eq n1 n2))
  | ELe e1 e2 =>
      v1 << (eval_expr_abstract e1) ;
      v2 << (eval_expr_abstract e2) ;
      n1 << (ensure_type parity v1) ;
      n2 << (ensure_type parity v2) ;
      returnM (inject top_op)
  | ENot e =>
      v << (eval_expr_abstract e) ;
      b << (ensure_type abstr_bool v) ;
      returnM (inject (neg_ab b))
  | EAnd e1 e2 =>
      v1 << (eval_expr_abstract e1) ;
      v2 << (eval_expr_abstract e2) ;
      b1 << (ensure_type abstr_bool v1) ;
      b2 << (ensure_type abstr_bool v2) ;
      returnM (inject (and_ab b1 b2))
  end.
```

The abstract interpreter for expressions looks much the same as the concrete version. The difference is that the operations on the types are now the abstract versions. For example, where the concrete interpreter called the standard plus operator for natural numbers, the abstract interpreter calls the parity or interval plus function.

The optionA monad

So far, we have seen the fail method of option used to model crashed programs due to type mismatches, for exampling when adding two booleans, or a boolean and a natural number. These kind of failures should occur at the same time in both the abstract and the concrete interpreter. In our language, we also model exceptions, which introduce a kind of uncertainty. Consider the following Java snippet.

```
if (x == 0) {
    throw new Exception();
else {
    x = 3;
}
```

We could try to model such a program with just the `None` value of option. This proves insufficient when this program is interpreted in an abstract way, because an abstract interpreter might be uncertain whether `x` is equal to 0 or not and has to reflect this in the returned value. We introduce the monad optionA (Keidel, Poulsen, and Erdweg 2018) to capture this side effect. In addition to constructors modelling successful and failed operations, this type has a constructor that models uncertainty as to whether the operation has succeeded.

```
Inductive optionA (A : Type) : Type :=
| NoneA : optionA A
| SomeA : A -> optionA A
| SomeOrNoneA : A -> optionA A.
```

As with option, we have a monad transformer that adds this functionality to a monad. We call this transformer optionAT and we define it as

```
Definition optionAT M A := M (optionA A).
```

3.5 Extracting the Generic Interpreter

As observed in the previous section, the abstract and concrete interpreter share many similarities. This leads us to the question of whether we can decompose the interpreters along the generic structure in a way that eases the proof burden. In this section, we look at how we have extracted the generic interpreter and how this generic structure can be instantiated to create abstract and concrete interpreters like those defined earlier.

3.5.1 Value typeclasses

The generic interpreter needs to have a uniform way of handling values, as it can be instantiated with either abstract or concrete values. For every type of concrete value we will require a corresponding interface. In our case, this means that we will require an interface for boolean types and an interface for numerical types.

For flexibility, we have a typeclass for every operation that should be supported on a value. This allows us to add each operation in a modular way. For booleans, these operations are and (&&), negation and an if operator.

```
Class and_op (A B : Type) : Type := and : A -> A -> B.

Class neg_op (A B : Type) : Type := neg : A -> B.

Class if_op  (A B : Type) : Type := when : A -> B -> B -> B.
```

We have a typeclass for each operation that can occur in the source code. Because the typeclasses are separate, we can easily analyze languages with different capabilities by adding or removing instances and constraints on the interpreter. The language we are considering now works with booleans and natural numbers. All value types, both abstract and concrete, need corresponding instances of the typeclasses.

```
Instance and_ab_op : and_op abstr_bool :=
  λ ab1 ab2,
    match ab1, ab2 with
    | Top, _ | _, Top => Top
    | NotTop b1, NotTop b2 => NotTop (andb b1 b2)
    end.
Instance and_op_bool : and_op bool := andb.
```

In the above code, we can see the and instance for our abstract booleans. If either of the arguments is $\top$, we can no longer be sure of the outcome so it turns into $\top$ as well. Otherwise, we use the result of the standard and operation on booleans. This function from the standard library is also used as an instance of the typeclass for standard booleans. All this allows the type inference to automatically grab the correct operations.

3.5.2 A parametric interpreter

```
Fixpoint generic_eval_expr
    {M : Type -> Type}
    {MM : Monad M}
    {MF : MonadFail M} {MS : MonadState (store valType) M}
    ...
```

As was the case with our concrete and abstract interpreter, the generic interpreter is parametrized over a monad that should have the required capabilities. In the case of the generic evaluation of expressions, these capabilities should be failing and the handling of state.

```
    ...
    {valType boolType natType : Type}
    {EC : extract_op cvalue valType}
    {SB : SubType boolType valType}
    {SN : SubType natType valType}
    ...
```

In contrast with the concrete and abstract interpreters, the generic interpreter does not know which type of values it should handle. These will now have to be passed along during instantiation as well. We do impose a few requirements on them, namely the ability to extract the value type from a literal concrete type found in the code. Furthermore, the composite value type should be a supertype of the types used as natural numbers and booleans.

```
    ...
    {PO : plus_op natType natType}
    {MO : mult_op natType natType}
    {EO : eq_op natType boolType}
    {LO : leb_op natType boolType}
    {NO : neg_op boolType}
    {AO : and_op boolType}
    ...
```

Finally, we require that the provided types have implemented the required operations and are therefore suitable to be used as values for natural numbers and booleans.

```
Fixpoint generic_eval_expr
    ...
    (e : expr) : M valType :=
  match e with
  | EVal v => returnM (extract v)
  | EVar x =>
      s  <- get;
      returnM (s x)
  | EPlus e1 e2 =>
```

```
      v1 <- generic_eval_expr e1 ;
      v2 <- generic_eval_expr e2 ;
      n1 <- ensure_type natType v1 ;
      n2 <- ensure_type natType v2 ;
      n  <- returnM (n1 + n2) ;
      returnM (inject n)
  | EMult e1 e2 =>
      v1 <- generic_eval_expr e1 ;
      v2 <- generic_eval_expr e2 ;
      n1 <- ensure_type natType v1 ;
      n2 <- ensure_type natType v2 ;
      n  <- returnM (n1 * n2) ;
      returnM (inject n)
  | EEq e1 e2 =>
      v1 <- generic_eval_expr e1 ;
      v2 <- generic_eval_expr e2 ;
      n1 <- ensure_type natType v1 ;
      n2 <- ensure_type natType v2 ;
      b  <- returnM (n1 = n2) ;
      returnM (inject b)
  | ELe e1 e2 =>
      v1 <- generic_eval_expr e1 ;
      v2 <- generic_eval_expr e2 ;
      n1 <- ensure_type natType v1 ;
      n2 <- ensure_type natType v2 ;
      b  <- returnM (leb n1 n2);
      returnM (inject b)
  | ENot e1 =>
      v1 <- generic_eval_expr e1 ;
      b1 <- ensure_type boolType v1 ;
      b  <- returnM (neg b1);
      returnM (inject b)
  | EAnd e1 e2 =>
      v1 <- generic_eval_expr e1 ;
      v2 <- generic_eval_expr e2 ;
      b1 <- ensure_type boolType v1 ;
      b2 <- ensure_type boolType v2 ;
      b  <- returnM (b1 && b2) ;
      returnM (inject b)
  end.
```

The required instances are added as implicit constraints to the definitions of the generic interpreter. In the definition of the generic interpreter, we have replaced all functions specific to an implementation by functions defined in the typeclasses. The structure of the interpreter remains the same. First we do a case analysis on the expression being evaluated, then we evaluate subexpressions, make certain the result of those subexpressions are valid inputs for our currently evaluated expression, and finally perform the expected operation. By doing this, we can get the type engine of Coq to infer the required instances for us. This greatly simplifies how we can now define the abstract and concrete interpreters. We only need to supply the desired monad and the desired abstraction.

```
Definition ConcreteState := optionT (StateT store option).
```

```
Definition AbstractState :=
  optionAT (StateT abstract_store option).

Definition conrete_interpreter (c : com) : ConcreteState unit :=
  generic_ceval (M:=ConcreteState) (valType:=cvalue)
    (boolType:=bool) (natType:=nat).

Definition abstract_interpreter (c : com) : AbstractState unit :=
  generic_ceval (M:=AbstractState) (valType:=avalue+T)
    (boolType:=abstr_bool) (natType:=parity).
```

This chapter showed how we had defined all necessary components to build our interpreter. In the next chapter, we will discuss how we have implemented our definition of soundness from Chapter 2 in Coq and how we go about proving the soundness of all those components.

Chapter 4

Proving The Interpreter Sound

In this chapter, we describe what it means for the abstract interpreter to be sound with regards to the concrete interpreter and how we define this soundness in Coq. In addition, we show how we utilize the tools Coq provides to automate the soundness proofs.

- In Section 4.1 we describe our mechanization of the Galois connections that form the basis of our definition of soundness.
- In Section 4.2, we will utilize these Galois connections to state the main theorem of this paper, the soundness of the interpreter. We show how we can decompose this soundness proof into many small components that reduce the complexity of the final proof.
- In Section 4.3 we describe how we have implemented a typeclass for each of the components used in the soundness proof. Any instances of these typeclasses are automatically used by Coq to construct the required proof.
- In Section 4.5 we will then describe some of the custom tactics we have written to further ease the proof burden for the various parts of the project. Many proofs regarding, for example, the same typeclass instances follow the same pattern which makes them a excellent candidates for automation.
- Section 4.4 describes how we can use the `auto` and `eauto` tactics to help solve the proofs that are not easily solved via a custom tactic. Proper utilization of these and the `autorewrite` and `autounfold` tactics will make it much easier to add new abstractions of components to the project.

4.1 Galois Connections

In this section we will describe the notion of Galois connections (P. Cousot and R. Cousot 1992). Before we do so, we take a moment to show our mechanization of preordered sets. We define typeclasses for preordered sets and define monotonicity before moving on to define a typeclass for Galois connections. We will also give a few examples of Galois connections to help make the concept more clear.

4.1.1 Preordered sets

The first definition we shall implement is the notion of a preordered set. A preordered set is any set S on which we can define a preorder relation $\sqsubseteq$ such that the preorder relation is reflexive and transitive. That is, $\forall a, a \sqsubseteq a$ (reflexivity) and $\forall a, b, c$, if $a \sqsubseteq b$ and $b \sqsubseteq c$, then

$a \sqsubseteq c$ (transitivity). We refer to such a combination of set S and relation as $(A, \sqsubseteq)$. This gives us the following definition of the preordered set as a Coq type class.

```
Class PreorderedSet (A : Type) : Type :=
{
  preorder : A -> A -> Prop;
  preorder_refl: ∀ x, preorder x x;
  preorder_trans: ∀ x y z,
    preorder x y ->
    preorder y z ->
    preorder x z;
}.
Infix "⊑" := preorder (at level 40).
```

In addition, we introduce the notation $\sqsubseteq$ to denote a preorder. Now that we have defined preorders we can define monotonicity, which is a concept which we require for a proper implementation of Galois connections. A function is monotone if a larger input also results in a larger output.

```
Definition monotone {A B : Type}
  {PA : PreorderedSet A} {PB : PreorderedSet B}
  (f : A -> B) : Prop := ∀ a a', a ⊑ a' -> (f a) ⊑ (f a').
```

4.1.2 Galois connections

Traditionally, Galois connections are defined on two partially ordered sets. However, they can also be defined on preordered sets and for the purposes of proving soundness that is sufficient. The difference between preordered sets and partially ordered sets is that a partial order is antisymmetric in addition to reflexive and transitive. Because we do not require the antisymmetric property, we did not add a lemma for it to the typeclass.

The definition of Galois connections that we have implemented is that of the monotone Galois connection. Given two preordered sets $(A, \sqsubseteq)$ and $(B, \sqsubseteq)$, a Galois connection between those two sets is defined as two functions $\alpha : A \rightarrow B$ and $\gamma : B \rightarrow A$ such that $\alpha\ a \sqsubseteq b$ if and only if $a \sqsubseteq \gamma\ b$.

```
Class Galois (A A' : Type) : Type := γ : A -> ℘ A'.
```

Instancing the Coq type class definition of Galois connections between two types A and A' requires supplying those types, as well as a gamma function $\gamma : A \rightarrow \wp\ A'$.

A difference between the Coq implementation and the mathematical definition is the lack of corresponding α function. This is because we are unable to define this function, due to limitations in Coq. Luckily, it turns out that the α function is not necessary to prove soundness (Jourdan 2016).

Note that the Coq definition does not require A to be a preordered set. This is a design choice that allows us to define the gamma function between an abstract type and a concrete type without having to proof that the abstract type has a preorder. We define the soundness-preserving properties of the preorder separately via the following typeclass.

```
Class PreorderSound (A A' : Type) {PA : PreorderedSet A}
    {GA : Galois A A'} : Prop :=
  preorder_sound : ∀ x y : A,
    x ⊑ y ->
    γ x ⊆ γ y.
```

Any instances of the preorder typeclass should be accompanied by a corresponding instance of the `PreorderSound` typeclass, to verify that the definitions of both preorder and gamma are correct.

For all abstract types, we define an ordering and a gamma connection. For our abstract version of the Maybe monad, we define a lattice with `SomeOrNoneA` as the top element. `NoneA` and `SomeA` correspond to `None` and `Some`. The relationship between preorders and Galois connections can be seen in the code below. For any x that is an approximation of x', all y such that $x \sqsubseteq y$ are also approximations of x'. This is equivalent to the lemma posed by `PreorderSound`.

```
Inductive optionA_le : optionA A -> optionA A -> Prop :=
  | optionA_le_none : optionA_le NoneA NoneA
  | optionA_le_none_justornone : ∀ y,
     optionA_le NoneA (SomeOrNoneA y)
  | optionA_le_just : ∀ x y,
     x ⊑ y ->
     optionA_le (SomeA x) (SomeA y)
  | optionA_le_justornone_r : ∀ x y,
     x ⊑ y ->
     optionA_le (SomeA x) (SomeOrNoneA y)
  | optionA_le_justornone : ∀ x y,
     x ⊑ y ->
     optionA_le (SomeOrNoneA x) (SomeOrNoneA y).

Inductive gamma_optionA {A A'} {GA : Galois A A'} :
    optionA A -> ℘ option A' :=
  | gamma_noneA : gamma_optionA NoneA None
  | gamma_SomeornoneA_none : ∀ a,
     gamma_optionA (SomeOrNoneA a) None
  | gamma_SomeA_Some : ∀ a' a,
     γ a' a ->
     gamma_optionA (SomeA a') (Some a)
  | gamma_Someornone_Some : ∀ a' a,
     γ a' a ->
     gamma_optionA (SomeOrNoneA a') (Some a).
```

4.2 Soundness

The main focus of this thesis is the soundness of the abstract interpreter. In this section, we will define what it means for an abstract function to be sound with regards to a concrete function, and show how we prove the abstract interpreter is sound with regards to the concrete interpreter.

Our definition of sound is dependent on our definition of gamma: an abstract function is sound if its output approximates the output of a concrete function, provided the inputs of the abstract function approximated the inputs of the concrete function.

For example, **if** we have two functions $f : A \to B$ and $f' : A' \to B'$, where there are two Galois connections between A and A', and B and B' respectively, **then** when there is a gamma relation between the input values, there will be a gamma relation between the output values.

This is a bit of an obtuse definition, so we will provide some examples. Let us look at the soundness of the parity plus function. We will show what it means for that function to be sound with regards to standard addition of natural numbers.

```
Definition parity_plus (p q : parity) : parity :=
  match p with
  | par_even => q
  | par_odd => match q with
               | par_even => par_odd
               | par_odd => par_even
               | par_top => par_top
               end
  | par_top => par_top
  end.
```

So in more concrete terms, the function `parity_plus` is sound with regards to `plus` if when, if the inputs are related, the output is also related. In Coq, this notion of relatedness is written as follows

```
Lemma parity_plus_sound : ∀ (p1 p2 : parity) (n1 n2 : nat),
    γ p1 n1 -> γ p2 n2 ->
    γ (parity_plus p1 p2) (plus n1 n2).
```

If `p1` is an abstract representation of `n1`, and `p2` is an abstract representation of `n2`, then `parity_plus p1 p2` should be an abstract representation of `plus n1 n2`.

Due to the automation capabilities of Coq, the actual proof for the soundness of the `parity_plus` function is very short.

```
Proof.
  autounfold with soundness. repeat constructor. intros.
  destruct p1, p2; eauto with soundness.
Qed.
```

The proof is written using case analysis. It considers all possible constructors of the parities and simplifies the result of the `parity_plus` function. Then it uses the lemmas from the standard libraries of adding combinations of even and odd numbers to solve the subgoals.

We have similar proofs for each of the operations on values. We will not list all those proofs here, as they are all short and of similar structure. Examine all possible input values; resolve the functions and ascertain the preservation of the gamma relation in those cases where the hypotheses hold.

More interesting are the soundness proofs for the interpreters. Because the abstract and concrete interpreters both consist of the same structure, the generic interpreter, the interpreters can be decomposed along the same lines.

As discussed in Section 3.3, we can decompose the program along the lines of the bind methods. If we can proof that the bind methods of our chosen monads are sound and that the functions called by the bind methods are sound, than the entire interpreter is sound.

```
Theorem sound_interpreter : ∀ c,
  γ (generic_ceval (M:=AbstractState) (valType:=avalue+T)
    (boolType:=abstr_bool) (natType:=parity) c)
    (generic_ceval (M:=ConcreteState) (valType:=cvalue)
    (boolType:=bool) (natType:=nat) c).
Proof.
  eapply generic_ceval_sound; typeclasses eauto + eauto with soundness.
Qed.
```

The above theorem is the goal of this thesis, proving that two interpreters, both instantiations of a generic interpreter, are sound. We start by introducing a lemma that turns this theorem into many smaller lemmas.

```
Lemma generic_ceval_sound (M M' : Type -> Type)
  ...
  {GN : Galois natType natType'}
  {GB : Galois boolType boolType'}
  {GM : ∀ A A', Galois A A' -> Galois (M A) (M' A')}
  {MFS : MonadFailSound M M'}
  {EXS : extract_op_sound EX (extract_sum extract_nat extract_bool)}
  {SSB : SubTypeSound SB SB'}
  {SSN : SubTypeSound SN SN'}
  {MCS : catch_op_sound M M'}
  {MTS : throw_op_sound M M'}
  {GS : get_state_sound (S:=store (avalue+T)) (S':=store cvalue) M M'}
  {PS : put_state_sound (S:=store (avalue+T)) (S':=store cvalue) M M'}
  {BS : bind_sound M M'}
  {RS : return_sound M M'}
  {POS : plus_op_sound PO PO'}
  {MOS : mult_op_sound MO MO'}
  {EOS : eq_op_sound EO EO'}
  {LOS : leb_op_sound LO LO'}
  {NOS : neg_op_sound NO NO'}
  {AOS : and_op_sound AO AO'}
  {IOS : if_op_sound IO IO'}
  : ∀ c : com,
  γ
    (generic_ceval (M:=M) (valType:=(avalue+T)) (natType:=natType)
    (boolType:=boolType) c)
    (generic_ceval (M:=M') c).
Proof. ... Qed.
```

Omitted are the constraints covered in the previous chapter. In addition to these constraints, which are placed on both the abstract monad `M` and the concrete monad `M'`, we now also have constraints that require Galois connections between the various values used by the systems, as well as proofs of soundness of each component and all operations. The lemma then goes on to prove that if each separate part is sound, the entire structure is sound.

Applying this lemma to the above theorem, turns that theorem into a series of smaller subgoals. We use Coq's automation mechanisms to automatically solve these subgoals for us by finding the necessary instances of the typeclasses. Any missing instances are left as subgoals to be proven interactively.

How we get Coq to automatically resolve our required instances and solve other simpler goals is covered in the rest of this chapter.

4.3 Sound typeclasses

For every typeclass `T` we use that can be implemented as both an abstract and a concrete version, we also have a `T_sound` typeclass that takes two instances and states that they are sound with regards to one another. We can get Coq to automatically use instances of these typeclasses if they have been defined. This means that proving any interpreter sound boils down to proving its components sound.

Due to the automation tactics of Coq, our soundness proof consists of only a single line. That means we can easily reuse this proof. If we wish to proof another interpreter built from the same components but composed different sound, no change at all is required. If it is necessary to include new components, we would only have to add typeclasses for those components to the constraints of the theorem and proof those components sound. The soundness of the entire structure would follow automatically.

4.4 Standard automation tactics

When developing mathematical proofs it often happens that many cases are trivial to proof. On paper, a writer can just say that the proof is left as an exercise to the reader or remark that it is trivial. It would be helpful if, when developing a proof in a more strict proof assistant such as Coq, we could do so as well.

Luckily, Coq comes with a variety of built-in tactics that allow us to express a similar sentiment. One such tactic is `trivial`. This tactic tries to solve goals of the form `X = X`. Building upon `trivial` is `easy`, which can also solve contradictions by examining the hypotheses in the context of a goal. The most powerful tools at our disposal are the `auto` and `eauto` tactics, which we can customize to work with our specific project.

The automation tactics work with a Hint Database, which is a set of patterns together with a tactic to apply if the goal matches that pattern. Through careful managing of such a hint database, we can solve many of our lemmas automatically. `Hint Extern _ .` is the most powerful way to add hints to the database. It takes a pattern for the goal to have before executing the hint, and allows us to perform any tactics we want when this pattern holds. Via this command we can add the custom tactics defined above to the hint database, upgrading auto to solve proofs requiring case analysis in addition to those merely requiring the application of lemmas.

```
Theorem generic_ceval_sound : ∀ c : com,
  γ
    (generic_ceval (M:=M)
        (valType:=(avalue+T)) (natType:=natType)
        (boolType:=boolType) c)
    (generic_ceval (M:=M') c).
Proof.
    induction c; cbn; eauto with soundness.
Qed.
```

Recall the theorem of the soundness of our interpreter, this time stated without the typeclass requirements. First, we use the `induction` tactic on the command that is being evaluated. The `cbn` tactic simplifies our goal, while the `eauto` tactic uses the soundness hint database to solve each subgoal. The `cbn` tactic turns each goal into something of the form `γ ?a ?a'`. Our soundness database contains every lemma of that form for all our typeclasses, which allows `eauto` to prove the entire interpreter sound in a single line.

```
Definition parity_plus (p q : parity) : parity :=
    match p with
    | par_even => q
    | par_odd => match q with
                 | par_odd => par_even
                 | par_even => par_odd
                 end
```

```
    end.
Hint Unfold parity_plus : soundness.

Lemma parity_plus_sound (p q : parity) (n m : nat) : Prop :=
    γ p n ->
    γ q m ->
    γ (parity_plus p q) (plus n m).
Proof. ... Qed.
```

The automation tactics can do more than just applying lemmas if the goal matches the conclusion of the lemma. The code above contains our definition of `parity_plus`, which defines addition for parities. After defining `parity_plus` we can also proof it sound with regards to the addition on natural numbers. This lemma `parity_plus_sound` can be used as an instance of the `plus_op_sound` typeclass, which is among the typeclasses used to proof the entire interpreter sound. Even the lemma `parity_plus_sound` can be solved using the automation tactics, reducing the proof burden even more.

When trying to prove lemmas regarding our definition, it helps if we can see what those definitions actually mean. In the above code, if we want to proof something about `parity_plus` it helps if we can unfold the definition to see the `match` statements underneath. The `autounfold` tactics helps us to do so. If we add definitions to an unfolding database, `autounfold` will automatically unfold them. We can combine this behaviour with a hint added to the `eauto` database to destruct any symbol occurring in a match statement, effectively automatically performing a case analysis.

The final auto-command we will discuss is the `autorewrite` tactic. When proving the correctness of our definitions, we end up with many lemmas of the form `X = Y`, where the left hand side is noticeably more complicated than the right. This is especially true when proving the monadic laws. The `autorewrite` tactic can take these equalities and simplify our goals. It can recognize when these equalities are applicable better than humans can, so this can make ostensibly daunting proofs much more manageable. Repeated application of the monadic laws turns a monadic equation into a canonical form, so `autorewrite` is especially helpful there. However, the hint database for `autorewrite` must be carefully managed to avoid adding circular rewrite rules. This will result in an infinite loop when calling the tactic.

4.5 Custom tactics

Coq allows us to define our own custom tactics using the `Ltac` command. This is a very flexible approach that allows us to perform a series of tactics using a single command, or try a variety of possible strategies to solve a goal. Custom tactics lend themselves especially well to proving instances of our typeclasses, as those proofs tend to follow the same structure.

```
Ltac unmatch x :=
  match x with
  | context [match ?y with _ => _ end] => unmatch y
  | _ => destruct x eqn:?
  end.

Ltac destr :=
  match goal with
  | [ |- context [match ?x with _ => _ end]] => unmatch x
  | H : match ?x with _ => _ end |- _ => unmatch x
  | [ |- context [let (_, _) := ?x in _]] => destruct x eqn:?
```

```
  | H : (_ , _) = (_, _) |- _ => inv H
  | |- _ /\ _ => split
  | H : _ /\ _ |- _ => destruct H
  end.
```

We start with several generically useful tactics. The first is the unmatch tactic. The tactic takes a single parameter, which is supposed the name of a hypothesis in the context. If the provided hypothesis is of the form `match ?x with _ => _ end)`, we recursively call unmatch on the term on which case analysis is performed. If it is of any other form, we simply destruct the provided term. The unmatch tactic allows us to recursively perform case analysis on match statements in the context.

The `destr` tactic performs a lot of checks to find instances in which we need to perform case analysis. The first is to check if the goal contains match statements, in which case the unmatch tactic is used. In the second case, we resolve let-bindings in the goal. If the context contains a hypothesis regarding the equivalence of the pairs, we call the custom `inv` tactic on that hypothesis. This tactic is a wrapper around the built-in `inversion` tactic that also performs some cleanup. The result is two equality hypotheses for the individual elements of the pair. Because the auto tactics from Coq also don't destruct the logical AND operator, we add a case for those as well. We combine all these into a `simplify` tactic, that can deconstruct a goal into smaller goals via case analysis and functional extensionality.

These tactics help us when we are developing proofs interactively, but we can also develop tactics that can generate the entire proof for us. Many of our proofs are repetitive, as we have to repeatedly write the same tactics with only some minor variations. For example, when we define an instance of the `PreorderedSet` typeclass, we have to provide an ordering and proof that the ordering is both reflexive and transitive, two lemmas that require almost the same proof for every relation.

Chapter 5

Related Work

In this section we'll discuss recent advances made by others in the field of abstract interpretation. We discuss previous work in the mechanization of abstract analysis, defining Galois connections in a constructive way and decomposing interpreters via a generic interpreter.

5.1 Mechanization of Abstract Analysis

A large previous work on the mechanization of abstract analysis in Coq was Verasco, a verified static analyzer for the C programming language (Jourdan 2016). Like in this thesis Verasco only implements the γ function of Galois connections. We were inspired by Verasco's lifting of types to domains that included top and/or bottom. This allows for defining the relation between the top value of an abstract domain and concrete values once and for all.

Like our interpreter, the interpreter used in Verasco is parametric in the abstract domain, meaing the same interpreter can be used for multiple abstractions. However, because Verasco deals with a specific language, there is no need for the flexibility provided by being able to change the monad stack. Therefore Verasco does not use a monad transformer stack. Verasco also does not use a generic interpreter like we do and instead develop the concrete and abstract interpreter separately.

Bertot (2008) implements and proves correct an interpreter that performs interval analysis on a toy language with loops and assignments, but no error handling. They have also implemented their work in Coq. They prove their interpreter correct by implementing Hoare triples, annotating the input program with pre and post conditions for every statement.

Bodin, Jensen, and Schmitt (2015) describe a way to construct correct abstract semantics so that the proof burden is minimal. They base their framework on pretty big step semantics, which has seen little use in the field of static analysis, compared to small step semantics. They used their framework to construct an interpreter for a language that had both loops and exceptions.

5.2 Constructive Galois connections

The work done by Monniaux (1998) shows that the abstraction function α of Galois connections cannot be formally written in constructive logic. Normally, Galois connections are made of two functions, the other being γ. The α function would be responsible for transforming the concrete values into the abstract values that best encompass those concrete values. However, defining the α function in Coq requires the use of non-constructive axioms (Sergey et al. 2013). The work of Pichardie (2005) builds on the work of Monniaux (1998) and shows that we can circumvent this problem by stating the soundness lemmas using the concretization function only. Because of the monotonicity that is required of these functions, any equa-

tions containing α can be rewritten to use only γ. However, there still has been work to circumvent this restriction.

Sergey et al. (2013) have described a method to create a monadic abstract interpreter from concrete semantics in Haskell. Like in this thesis, they extract access to a store to a monadic typeclass that is then instantiated in a concrete and abstract manner. By bounding the addressable range of their store, the α function becomes computable.

Darais and Horn (2019) describe a method to define constructive Galois connections. Instead of defining the functions $\gamma : A \rightarrow \wp A'$ and $\alpha : \wp A' \rightarrow A$, they define an interpretation function $\mu : A \rightarrow \wp A'$ and an extraction function $\eta : A' \rightarrow A$. When defining Galois connections in terms of μ and η, they are able to derive an abstract interpreter via the calculational approach, a method to calculate the abstract interpreter from the concrete interpreter. This allows them to not only use mechanization to proof existing theorems, but also mechanize existing proofs. Our work, in contrast, requires explicitly constructed concrete and abstract interpreters and the calculational approach offers no benefits there.

The extraction function η by Darais and Horn (2019) is comparable to the extract functions in the IsNat and IsBool typeclasses that we've implemented. If we moved those to the Galois typeclass, we would have obtained something similar to the constructive Galois connections. Because we work with explicit concrete and abstract interpreters derived from a generic intepreter, we've kept our definition of a Galois connection closer to the classical definition, albeit without an α function.

5.3 Generic interpreters

As mentioned in the introduction, the idea for sharing a generic interpreter between the concrete and abstract interpreters comes from the work of Keidel, Poulsen, and Erdweg (2018). In this paper, they propose a methodology for defining abstract and concrete interpreters in a way that allows the soundness proof to be decomposed along the lines of the generic interpreter.

One large difference between their work and ours lies in that we use Monads instead of Arrows to represent effects such as stores and exceptions. Because every monad is an arrow (Hughes 2000), we should be able to implement our work using arrows as well. Lindley, Wadler, and Yallop (2011) have shown that there are more arrows than monads, but monads are more powerful. This means that there are instances of arrows that cannot be implemented as monads. Indeed, some of the moand transformers that we implemented resulted in types that were no longer monads. However, this did not prevent us from proving the soundness of the entire monad stack.

This difference means that we have to decompose our proof along the methods exposed by the monad typeclass instead of the operations on arrows. Fortunately we have analogues to the methods used by Keidel, Poulsen, and Erdweg (2018). The first is the arrow composition method `>>>`, which is like the bindM method of monads. Second, Keidel et al. use the parallel composition method `***`, which takes two arrows `g` and `h` and returns a composed arrow `g *** h`. The composed arrow takes a tuple `(x,y)` and returns `(g x, h y)`. There is no analogue to this method in the standard monad typeclass, but we can simulate the effects with repeated applications of the bindM method.

The decision by Keidel et al. to use arrows was influenced by the fact that arrows and their operations form an algebra. This in turn means that, after defining the interpreters and the required lemmas, the soundness proof for the concrete and abstract interpreter is automatically derived. In contrast, use of monads would mean that the proof of soundness of the interpreters would require manual effort. Fortunately our use of Coq and its automation capabilities means that while providing this proof is still necessary, the proof will automatically be constructed if we have instantiated the proper typeclasses.

One of the problems we ran into was the definition of the bind method on our optionAT monad. It required a join on the value wrapped inside the monad, but an instance of monad cannot impose constraints on the wrapped value. Keidel et al. works around this by forgoing the use of arrow transformers in this paper and defining the entire type of the interpreter from the start. This allows for inlining the joining of the state and preserving soundness. They solve this in their follow-up paper, where Keidel and Erdweg (2019) describe a set of reusable components based on arrow transformers that preserve soundness. In this work, they use a work around they call `joinsecond`, that joins arrows while taking a function that describes how to join the carried values. By providing the join function only when needed, they avoid placing a requirement on the types wrapped by the monad.

Chapter 6

Conclusions and Future Work

We conclude this thesis with a summary of our results and by making a few suggestions about possible future work.

In this thesis we mechanized the work of Keidel, Poulsen, and Erdweg (2018) and implemented components that can be easily proven sound. By combining and rearranging these components one can build interpreters for languages with varying features, such as different ways of handling errors. New feature can be added and proven sound by encapsulating the features in a typeclass, implementing it and adding the typeclass restrictions to the final theorem.

Not all of the effects that we wished to implement for our interpreter could be implemented as monad transformers. Monads transformed by the non-deterministic monad transformer `optionAT` do not obey the monadic law for associativity. Luckily, the soundness proofs do not depend on this property. In fact, while implementing the monadic laws helped us to be certain that we implemented the monads properly, none of them were actually required to proof soundness.

In addition to its bind method not being associative, the `optionAT` monad transformer is not sound when transforming an arbitrary monad. It requires that the monad that is to be transformed can have its side effects merged or joined in some manner. In our work we have implemented the `optionAT` transformer specifically for transforming the `State` monad. Further work is necessary to make explicit this joinable monad requirement for all other monads that share this property with `State`.

In the rest of this chapter, we make three suggestions for possible future work. First, we discuss a possible solution to proving the soundness of the transformation of arbitrary monads by the `optionAT` monad transformer. Secondly, we suggest computations that can be expressed as monads and added to the project. Finally, we discuss possible expansions of the abstract domains handled by this project.

Soundness of transformed monads After transforming a monad with the `optionAT` transformer, it has the ability to express non-determinism, allowing us to model computations with multiple possible execution paths, such as if statements. Abstract implementations of this monad may be uncertain which branch of an if statement is taken. This means that in such cases, they must approximate all possible branches, which requires that we should be able to join the monadic results of these branches.

However, monad transformers should be able to work with any kind of monad as an input, not just those that have such a joining ability. While we can impose requirements on these monads when defining the soundness of the transformed monad, we have not done sone in a generic manner in this thesis, instead opting to define the soundness of a monad transformed by `optionAT` for each of our implemented monads.

We have thought of two possible solutions for this problem. The first is a sort of widen operation on the monad that broadens the side effects of encapsulated by the monad.

The `optionA` monad would widen into the `SomeOrNoneA` constructor, the `State` monad would widen it's carried state, and so on. The second possible solution we explored was an alternative for the bind method. This new bind method would merge the side effects of the "first" monad m with the side effects relevant to performing the continuation function f on the value carried by m. For example, the standard bind method of `State` uses the state `st` from the first monad to perform the computations in f, which results in a new state `st'`. The bind method we require for a sound `optionAT` would have to merge these two states instead of merely retaining `st'`.

Possible monads We used monad to encapsulate the abilities of our imperative language. Via the use of the `option` and `State` monads we were able to model exceptions, errors and variables. Implementing more monads will allow us to model more complex programming languages. The Haskell standard library comes with eight core monads (Figueroa, Leger, and Fukuda 2020): identity, errors, lists, state, reader and writer, RWS and continuations. The error monad is like our `option`, and lists are a more general version of `optionA`, allowing for even more possible results of the computation. Reader and Writer are able to read from and write to a state and RWS is a combination of Reader, Writer and State. Continuations model computations that can be paused and continued. Adding the list and continuation monad to the project would especially increase the usefulness of the project due to the powerful computations they represent. Examples of common language constructs that can be added are functions and loops.

Expansions to the abstract domain The utility of abstract interpretation stems for the abstract domain chosen. Interval analysis can be used to find memory issues that result from array indices that are out of bounds, while abstracting values to just their types allows an abstract interpreter to verify the absence of type errors. Adding more abstract domains to the project will make it possible to write abstract interpreters that offer more feedback to the user. Furthermore, after adding more domains it may be interesting to devise a method of combining the results of two analyses to be able to give a more precise analysis.

Bibliography

A Short Introduction to Coq (n.d.). https://coq.inria.fr/a-short-introduction-to-coq. Accessed: 2019-06-25.

Bertot, Yves (2008). "Structural Abstract Interpretation: A Formal Study Using Coq". In: *lernet*, pp. 153–194. DOI: 10.1007/978-3-642-03153-3_4. URL: http://dx.doi.org/10.1007/978-3-642-03153-3_4.

Bodin, Martin, Thomas Jensen, and Alan Schmitt (2015). "Certified abstract interpretation with pretty-big-step semantics". In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pp. 29–40.

Calcagno, Cristiano et al. (2015). "Moving Fast with Software Verification". In: *NFM*, pp. 3–11. DOI: 10.1007/978-3-319-17524-9_1. URL: http://dx.doi.org/10.1007/978-3-319-17524-9_1.

Cousot, Patrick (1999). "The calculational design of a generic abstract interpreter". In: *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES* 173, pp. 421–506.

Cousot, Patrick and Radhia Cousot (1977). "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *POPL*, pp. 238–252.

— (1979). "Systematic Design of Program Analysis Frameworks". In: *POPL*, pp. 269–282.

— (1992). "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation". In: *plilp*, pp. 269–295.

Darais, David and David Van Horn (2019). "Constructive Galois Connections". In: *JFP* 29. DOI: 10.1017/S0956796819000066. URL: https://doi.org/10.1017/S0956796819000066.

Figueroa, Ismael, Paul Leger, and Hiroaki Fukuda (2020). "Which monads haskell developers use: An exploratory study". In: *Science of Computer Programming*, p. 102523.

Gibbons, Jeremy and Ralf Hinze (2011). "Just do it: simple monadic equational reasoning". In: *ICFP*, pp. 2–14. DOI: 10.1145/2034773.2034777. URL: http://doi.acm.org/10.1145/2034773.2034777.

Harrison, John, Josef Urban, and Freek Wiedijk (2014). "History of Interactive Theorem Proving." In: *Computational Logic*. Vol. 9, pp. 135–214.

Hughes, John (2000). "Generalising monads to arrows". In: *SCP* 37.1-3, pp. 67–111.

IO and monads (n.d.). https://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html. Accessed: 2020-03-17.

Jourdan, Jacques-Henri (2016). "Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié)". PhD thesis. Paris Diderot University, France. URL: https://tel.archives-ouvertes.fr/tel-01327023.

Keidel, Sven and Sebastian Erdweg (2019). "Sound and reusable components for abstract interpretation". In: *PACMPL* 3.OOPSLA. DOI: 10.1145/3360602. URL: https://doi.org/10.1145/3360602.

Keidel, Sven, Casper Bach Poulsen, and Sebastian Erdweg (2018). "Compositional soundness proofs of abstract interpreters". In: *PACMPL* 2.ICFP. DOI: 10.1145/3236767. URL: https://doi.org/10.1145/3236767.

Liang, Sheng, Paul Hudak, and Mark P. Jones (1995). "Monad Transformers and Modular Interpreters". In: *POPL*, pp. 333–343.

Lindley, Sam, Philip Wadler, and Jeremy Yallop (2011). "Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous". In: *ENTCS* 229.5, pp. 97–117. DOI: 10.1016/j.entcs.2011.02.018. URL: http://dx.doi.org/10.1016/j.entcs.2011.02.018.

Monniaux, David (1998). "Réalisation mécanisée d'interpréteurs abstraits". French. Rapport de DEA. Université Paris VII.

Petricek, Tomas (2018). "What we talk about when we talk about monads". In: *Programming* 2.3, p. 12. DOI: 10.22152/programming-journal. URL: https://doi.org/10.22152/programming-journal.org/2018/2/12.

Pichardie, David (2005). "Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés". French. PhD thesis. University Rennes 1.

Sergey, Ilya et al. (2013). "Monadic abstract interpreters". In: *PLDI*, pp. 399–410. DOI: 10.1145/2491956.2491979. URL: http://doi.acm.org/10.1145/2491956.2491979.

Software Foundations 1 : Logical Foundations (n.d.). https://softwarefoundations.cis.upenn.edu/lf-current/Maps.html. Accessed: 2020-03-17.

Sozeau, Matthieu and Nicolas Oury (2008). "First-Class Type Classes". In: *tphol*, pp. 278–293. DOI: 10.1007/978-3-540-71067-7_23. URL: http://dx.doi.org/10.1007/978-3-540-71067-7_23.

Wadler, Philip (1995). "Monads for Functional Programming". In: *afp*, pp. 24–52.