Bachelor thesis
Computing Science

Radboud University

Using Equations to Define Non-Structurally Recursive Functions in Coq

Author:
Kirsten Hagenaars
s1020609
kirsten.hagenaars@student.ru.nl

Supervisor/assessor:
Prof. Dr. Herman Geuvers
h.geuvers@cs.ru.nl

Assessor:
Dr. Robbert Krebbers
mail@robbertkrebbers.nl

January 10, 2021

Abstract

The Equations plugin for the Coq Proof Assitant allows for defining non-structurally recursive functions in Coq. We explore how Equations can be used to define non-structurally recursive functions in Coq. We do so by creating quicksort, mergesort, and binary search using Equations, along with quicksort on vectors instead of on lists. We aim for these implementations to be straightforward and to stay close to their typical Haskell definitions. We also prove the correctness of these implementations.

Furthermore, the LiquidHaskell verifier allows for proving properties of Haskell programs. We explore how LiquidHaskell can verify the correctness of Haskell functions by doing this for quicksort and mergesort.

Contents

Chapter 1

Introduction

The Coq Proof Assistant allows for writing formal specifications and programs, and for constructing proofs to show that programs adhere to the given specifications. More specifically, one can write functional programs in Coq. What sets Coq aside from functional languages such as Haskell is its ability to guide and verify mathematical proofs. For example, one can define a function with its specifications and construct a formal proof showing the function complies with the specifications.

However, there are restrictions for function definitions in Coq, which do not exist in Haskell. Namely, recursive functions must be structurally recursive (see Section 2.2.1). This means that functions over recursively defined structures (such as lists and trees) are only allowed to have recursive calls on immediate substructures of the current arguments. These structures have minimal elements (such as the empty list and a leaf node), which ensures that the recursive function will terminate. Termination is a requirement for any program in Coq. Coq enforces this restriction by means of a simple syntactic check; only functions passing this check are accepted by Coq. An example of a function where the strict requirements of Coq pose a problem is quicksort, which can be recursively defined in a functional language such as Haskell:

```
quicksort :: Ord nat => [nat] -> [nat]
quicksort  [] = []
quicksort (x:xs) =  quicksort (filter (<=x) xs) ++
                    [x] ++ quicksort (filter (>x) xs)
```

In this example, the immediate substructures of the list `x:xs` in the case for `quicksort (x:xs)` are `x` and `xs`. As the recursive calls are done on `filter <=  x xs` and `filter > x xs`, Coq will not accept a function definition like this.

To alleviate this restriction, the Coq plugin called Equations allows for recursive functions that are not structurally recursive. Equations allows for function definitions on inductive types, which includes recursively defined structures. These function definitions look more similar to those in other functional languages compared to plain Coq, due to the support for dependent pattern matching. Aside from enabling definitions of complex functions, it also provides strong proof principles for these functions. This allows for nontrivial proofs about functions defined using Equations. These proof principles are based on existing proof tactics but are specifically constructed to accommodate functions defined using Equations. To ensure termination, Equation-defined recursive functions have to be well-founded (see Section 2.2.2), which means that they should have a base case thatf will be reached. One shows that a function is well-founded by explicitly stating the decreasing argument in the signature of the function. This generates corresponding proof obligations which, once proven, ensure that the function terminates. These obligations need to be proven by the user. However, Equations does help with these proofs and proves some obligations on its own. Simply said, Equations can help us in showing Coq that a function terminates.

We will explore the possibilities of Equations by defining functions that are not structurally recursive and therefore not allowed by plain Coq. We aim to make these functions well-readable by staying as close as possible to the Haskell definitions. The correctness of the resulting functions will be shown as Coq proofs. This project is made using Coq 8.12 and the corresponding version of Equations.

LiquidHaskell is a verifier for Haskell which allows for proving properties of Haskell programs. Instead of translating a Haskell definition to Coq to prove properties about it, LiquidHaskell may enable us to prove these properties without using Coq. We will briefly explore the possibilities of LiquidHaskell by defining some of the same functions we will define using Equations. We will attempt to verify the correctness of the resulting functions using LiquidHaskell. We use Haskell 8.6.5 and the corresponding version of LiquidHaskell.

Chapter 2 provides the necessary information on Coq, along with important mathematical definitions. Chapters 3 and 4 provide a detailed description of the research that has been carried out on Equations. Chapter 5 describes the research carried out on LiquidHaskell. Chapter 6 places this work in the context of related work. Chapter 7 contains the conclusions.

The Coq and Haskell files created can be found here: `https://github.com/KirstenHagenaars/BachelorThesis/tree/main/Code`.

Chapter 2

Preliminaries

2.1 Introduction to Coq

The Coq system [8] is based on the formalism called the Calculus of Inductive Constructions. This means that all expressions in Coq are terms which have a type in the type system pCIC (predicative Calculus of Inductive Constructions). Note that this also holds for Coq proofs. More detailed information on this formalism can be found in the Coq documentation [10]. At its core, Coq is a type-checker. This section serves as a guide to the basics of Coq. Basic constructions in Coq will be introduced in the form of examples.

A Coq program consists of a series of Coq commands. Such a command is terminated by a period. From now on, `this font` is used for user input to the Coq system and this font is used for the systems response.

Coq has an interesting feature called program extraction, which can extract a Coq program to another language. The languages currently supported are Haskell, Ocaml, and Scheme.

2.1.1 Types in Coq

As mentioned before, all expressions are terms that all have a type. We will introduce three of the most important types:

1. The type `Set` is meant for computations. Terms of this type remain during program extraction.
2. The type `Prop` is meant for propositions. Terms of this type are discarded during program extraction, which allows for terms only used in proofs to be thrown away.

3. The type `Type` is the sort of all types (a sort is the type of a type), thus `Set : Type` and `Prop : Type`. Moreover, to avoid inconsistencies such as `Type : Type`, Coq internally keeps track of universe levels like this: $\texttt{Type}_i\texttt{: Type}_{i+1}$. These are not available to the user.

Another important note about types in Coq is about comparing values. When we want to compare values we use `=?`, `<?` and `<=?`. This will return a boolean, which can be used as a condition in an if-statement but can not be used as a proposition.

The notations `=`, `<` and `<=` return propositions, which can not be used as a condition in an if-statement.

2.1.2 Definitions in Coq

Functions can be defined using the `Definition` keyword. See this example which squares a natural number:

```
Definition square (n : nat) := n*n.
```

This line adds the function `square : nat -> nat` to the system. This function can now be used in other commands. For example, using the `Compute` keyword to compute a function:

```
Compute square 2.
```

Coq responds with: $= 4 : \mathsf{nat}$.

For case distinctions on booleans, one can use the `if ... then ... else ...` construct, see this function which returns the negation of a given boolean `b`:

```
Definition not (b : bool) := if b then false else true.
```

Another way to make a case distinction is to use pattern matching, which is done with the `match ... with ... end` construct. See this function which checks whether a given natural number equals 0:

```
Definition is_zero (n : nat) :=
match n with
  0 => true
  | S x => false
end.
```

where `S` returns the successor of a given natural number. More on natural numbers in Coq can be found in Section 2.1.3.

Lambda expressions can be defined using the `fun` keyword. For example, the following function uses a lambda expression to double a given natural number:

```
Definition double := fun (n : nat) =>2*n.
```

2.1.3 Inductive types

Inductive types are types constructed using constructors. Examples of inductive types are natural numbers, lists, and trees. We will take natural numbers as an example to further illustrate inductive types.

In Coq, natural numbers are defined as either 0 or the successor of a natural number, which is captured by the following definition, which introduces `nat` as a new type, of type `Set`:

```
Inductive nat : Set :=
  O : nat
  | S : nat -> nat.
```

This corresponds to Peano's encoding of natural numbers. The functions that produce the terms are called constructors. In the case of natural numbers, `O` and `S` are the constructors. Inductive types are closed under their constructors, so if `x : nat` then `S x : nat`. Case distinction and recursion are suitable computational concepts for inductive types.

Coq derives induction principles for inductive types. These are useful for proving properties of that type by induction. For natural numbers, the induction principle looks like this:

```
nat_ind : forall P : nat -> Prop,
    P 0
  ->
      (forall n : nat, P n -> P (S n))
    ->
      forall n : nat, P n.
```

To show that property `P` holds for all natural numbers, one can apply this principle with the `induction` tactic or with `apply nat_ind` (applying mathematical induction on natural numbers). This will result in 2 goals, `P 0` and `forall n : nat, P n -> P (S n)` (implications are right-associative in Coq).

Another important inductive type aside from the definition of natural numbers is the definition of lists:

```
Inductive list (A : Type) : Type :=
  nil : list A
  | cons : A -> list A -> list A
```

A list is either an empty list (`nil`) or the result of adding an element to an existing list (`cons`). The infix notation for `cons a l` is `a :: l`, where `l` is

a list and `a` is being added to the list. This results in, for example, the list $[1, 2, 3]$ being denoted as `1 :: 2 :: 3 :: nil` in Coq.

2.1.4 Fixpoint definitions

For recursive function definitions, one uses the `Fixpoint` keyword. Here is an example function that appends two lists:

```
Fixpoint app {A} (l l': list A) :=
match l with
  nil => l'
  | a :: tl => a :: app tl l'
end.
```

Since recursive functions typically have a case distinction (a base case and a recursive case), the `match ... with ... end` construct is often used to accommodate this. As mentioned before, recursive functions in Coq must be structurally recursive. This means that recursive calls can only be made on subterms of the initial argument. To enforce this constraint, those subterms need to be obtained by pattern matching, meaning that they need to be explicitly stated before the `=>`. In the case of `app`, the recursive call is done on `tl` which indeed is a subterm of `l` and indeed is obtained through pattern matching. More on structural recursion can be found in Section 2.2.1. Recursive functions may have multiple arguments, in this case, there must be one that satisfies the structural recursion requirement. This argument is referred to as the *decreasing* argument. The decreasing argument can be stated explicitly by using the `{struct l}` syntax in the function signature, where `l` is the decreasing argument. Coq can often figure out the decreasing argument, therefore it is usually omitted in function definitions. Note than when the length of an argument decreases, we will refer to the length of the argument as the decreasing argument instead of the argument itself.

2.1.5 The Option type

Objects of the Option type can either contain a value or be empty. In other words, an object of the Option type is a list with at most one element. It serves the same purpose as Haskell's Maybe type. It is defined as an inductive type:

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

When the object contains a value `a` it is denoted as `Some a`, when it is empty it is denoted as `None`.

An example where the Option type is useful is the `head` function, which

returns the head of a given list. In case this list is empty, there is no head to return. To accommodate this, one can use an Option type as a return type. When the list is empty, `None` is returned, otherwise the head `a` is returned as `Some a`.

```
Definition head {A} (l: list A) : option A :=
match l with
  nil => None
  |a::t => Some a
end.
```

2.1.6 The Sumbool type

A regular disjunction `A \/ B` is an element of `Prop`. The inductive type Sumbool is used for the informative disjunction, which is an element of `Set` instead of `Prop`, meaning that it will not be erased during program extraction. The informative disjunction between logical propositions `A` and `B` is written as `{A}+{B} : Set`. Its extraction is isomorphic to the type of booleans, therefore the value of a Sumbool is decidable. Take a look at the definition of Sumbool:

```
Inductive sumbool (A B : Prop) : Set :=
  left : A -> {A} + {B}
  | right : B -> {A} + {B}
```

An object of type Sumbool contains a `left` when `A` holds and contains a `right` when `B` holds. The constructors take elements of type `A` or `B`, to result in a Sumbool object. Since these are propositions, their elements these are proofs of these propositions. These proofs are typically infered by Coq and not given explicitly by the user. Because Sumbool is an inductive type with constructors `left` and `right`, one can pattern match on a given object of Sumbool to find out/decide whether `A` or `B` holds:

```
Definition id_bool (b : bool): bool :=
match sumbool_of_bool(b) with
 | left _ => true
 | right _ => false
end.
```

As the name suggests, `sumbool_of_bool` creates a Sumbool object from a given boolean. A commonly used alias for this function is `dec`, which will also be used in this project.

Sumbool is often used for propositions and their negations. For example, the `le_dec` decision procedure decides whether a given `n : nat` is less than or equal than a given `m : nat`.

```
le_dec : forall n m : nat, {n <= m} + {~ n <= m}
```

2.1.7 Predicate subtyping

The Program tactic commands allow for enriched specifications and for proving that the code complies with these specifications. This is done by the predicate subtyping mechanism, where type checking conditions are generated for types with added specifications. Use the `Set Program Mode` command to enable using predicate subtyping. When you have a type `T : Set` and you want only a specific subtype that adheres to certain specifications you can use `{x : T | P x}` instead, where `P : T -> Prop` is the property all objects of this type should have. In other words, `{x : T | P x}` is the subtype of `T : Set` where `P` holds. An object of such a type is a pair `<x, p>`, where `x` is an element of `T` and `p` is a proof of `P x`. This `p` needs to be given by the user, which is done by proving the conditions generated by the subtyping mechanism mentioned above. Once these proofs are given, the Coq term is complete.

For example, if the return type of a function is `nat` but we know that its value will always be below 4, then we can specify this by using `{x : nat | x < 4}` as the return type. In this case, the property `P x` is `x < 4`. This gives rise to proof obligations to verify that `x < 4` always holds.

This property `P` may also depend on the input to the function. Take this overly elaborate definition of `successor`, which applies the constructor `S` (the successor from the standard library) twice and substracts 1.

```
Definition successor (n : nat) : {m : nat | m = S n} :=
  (S (S n)) - 1.
```

The return type is the subtype `{m : nat | m = S n}`, specifying that the output `m` is indeed the successor of the input `n`. This gives rise to one proof obligation:

```
successor_obligation_1 : forall n : nat, S (S n) - 1 = S n
```

This property is trivial and therefore it is automatically solved by Coq.

Note that `P` may be as elaborate as one wishes, making it suitable for correctness properties and pre-and post-conditions.

Furthermore, note that the generated obligations when using if-statements may be unprovable since the value of the boolean test of the if-statement is not taken into account. This problem does not occur when using decidability combinators introduced in Section 2.1.6 instead. For example:

```
Program Definition id (n : nat) : { x : nat | x = n } :=
  if dec (n =? 0) then 0
```

```
else S (n - 1).
```

2.1.8 Dependent pairs and Sigma types

Dependent pairs are pairs where the type of the second element depends on the value of the first element. For example, take the pair (a, b) where $a \in A$ and $b \in B\ a$, the type of b depends on the value of a. We will discuss three ways of creating such a dependent pair in Coq, which depend on the type of `B`.

1. Using an existential quantifier: `exists (x:A), B x : Prop`. This is only applicable in case `B : A -> Prop`.
2. Using a subset: `{x:A | B x} : Set/Type`. This is also only applicable in case `B : A -> Prop`.
3. Using Sigma types: `sig (x:A), B x : Set/Type`. This is applicable regardless of the type of `B`.

Even though all given options essentially create the same dependent pair, there is a slight difference. When you have a proof of `exists (x:A), B x`, Coq can not give you an `a` for which `B a` holds by using an exists-elimination. This is different from the subset or Sigma type. Thus, the use of an existential quantifier is more restrictive.

2.1.9 Proofs in Coq

Coq allows for interactive proofs of, for example, lemmas or theorems. Such a proof consists of commands called *proof tactics*. When proving, Coq displays information about what the user needs to prove. This information, also known as the goal, consists of the *conclusion* and the *context*. The conclusion is what one needs to prove. The context contains all of the assumed information in the form of hypotheses. Often, there are multiple subgoals. When applying a proof tactic, Coq will update the corresponding subgoal. This can lead to more or fewer subgoals, or it can change the conclusion or a hypothesis. Once there are no more subgoals left, the proof is complete.

A proof tactic, just like any other command, is terminated by a period. To apply a tactic `t` to all subgoals resulting from tactic `u`, use a semicolon to combine the tactics into one command like this: `u; t.`.

See Table 2.1 for an overview of useful tactics to apply on logical connectives either in a hypothesis or in the conclusion. The Coq standard library contains countless lemmas one may apply in a proof. These are already verified and thus make your proofs more concise. Application of a lemma is done by the `apply` keyword followed by the name of the lemma. The tactics `Qed` or `Defined` are applied at the end of a proof. Applying `Qed` makes the

	Hypothesis H	Conclusion
$\forall$	`apply H`	`intros H`

$\exists$	`elim H`
	`case H`
	`destruct H as [x H1]`
		`exists` v

$\neg$	`elim H` `case H`	`intros H`

$\wedge$	`elim H`
	`case H`
	`destruct H as [H1 H2]`
		`split`

$\vee$	`elim H`
	`case H`
	`destruct H as [H1 | H2]`
		`left` or
		`right`

$\rightarrow$	`apply H`	`intros H`
$=$	`rewrite H` `rewrite <- H`	`ring` `reflexivity`
`False`	`elim H` `case H`	

Table 2.1: Tactics for logical connectives [1]

proof term opaque, meaning that it can not be unfolded. Applying `Defined` makes the proof term transparent, allowing the unfolding of the definition.

One important proof tactic is `intro`, which applies a forall-introduction or an implication-introduction. When the tactic is followed by a name, this becomes the name of the new hypothesis resulting from applying the tactic, otherwise, Coq decides the name. The tactic `intros` repeatedly applies the `intro` tactic until it no longer applies.

Another important proof tactic is `induction`, which applies induction on a given inductive type. This will create a subgoal for each constructor of the inductive type. Use `induction n` to apply induction on `n`. One can also follow up the `induction` tactic with a numeral n, this will apply `intro` until it encounters the n-th non-dependent product in the goal. Afterward, it applies induction on the hypothesis which was last introduced to the context.

[1] The content of this table is from Coq in a Hurry [3]

Example of a Coq proof

We will further illustrate Coq proofs using an example. Let's say we want to prove that the `app` function defined previously, which appends two lists, is associative. First, we formalize this as a lemma:

```
Lemma app_associative {A} :
forall (x y z: list A),
  app (app x y) z = app x (app y z).
```

When running this line, Coq launches the proof mode. In case one does not want to prove the lemma yet, one can use the command `Admitted`. This will close the proof mode and simply assumes that the lemma holds. The lemma can then already be applied in other proofs. Table 2.2 contains a full proof walkthrough for this lemma. Every step in the proof corresponds to a row, where each row is divided by a dashed line. The current goal is above the dashed line and the proof tactic that is applied in response to this goal is below the dashed line, along with some explanation.

A : Type ————————————(1/1) forall x y z : list A, app (app x y) z = app x (app y z)
`intros.` This tactic does a forall-introduction for `x`, `y`, and `z`
A : Type x, y, z : list A ————————————(1/1) app (app x y) z = app x (app y z)
`induction x.` This tactic launches a proof by induction on `x`, the current goal is split into a base step and an inductive step
A : Type y, z : list A ————————————(1/2) app (app nil y) z = app nil (app y z) ————————————(2/2)

app (app (a :: x) y) z = app (a :: x) (app y z)

`reflexivity.`

This is the base step. It is easily solved by simplifying both sides of the equation, which is done using this tactic

---

A : Type

a : A

x, y, z : list A

IHx : app (app x y) z = app x (app y z)

————————————————————(1/1)

app (app (a :: x) y) z = app (a :: x) (app y z)

`simpl.`

We have now entered the inductive step. This goal is simplified using the definition of `app`

A : Type

a : A

x, y, z : list A

IHx : app (app x y) z = app x (app y z)

————————————————————(1/1)

a :: app (app x y) z = a :: app x (app y z)

`rewrite IHx.`

This tactic rewrites the goal by applying hypothesis `IHx`, which is the induction hypothesis

---

A : Type

a : A

x, y, z : list A

IHx : app (app x y) z = app x (app y z)

————————————————————(1/1)

a :: app x (app y z) = a :: app x (app y z)

`trivial.`

This goal trivially holds, therefore it is solved using this tactic
No more subgoals.
`Qed.` Now that there are no more subgoals left, one can use the command `Qed` to save the proof. The lemma can now be applied in other proofs using the command `apply app_associative`

Table 2.2: Step-by-step proof of lemma `app_associative`

2.2 Mathematical definitions

2.2.1 Structural and generative recursion

In general, recursion can be divided into two types, structural and generative recursion [7]. The terms 'generatively recursive functions' and 'non-structurally recursive functions' refer to the same set of functions and will be used interchangeably.

Structurally recursive functions work on inductive types and perform their recursive calls on immediate substructures of those types. An example of such a function is one that returns the sum of the values of a list:

```
Fixpoint sum l :=
match l with
  nil => 0
  | hd :: tl => hd + sum tl
end.
```

In this case `l` is a list, which is an inductive type. A list is either an empty list (`nil`) or the result of an element added to a list (`hd :: tl`). Notice how these constructors are used in the `match ... with` statement. This function sums up the values of the list by recursively adding the value of the head (`hd`) to the sum of the tail (`tl`). The recursive call is done on the tail of the list, which is an immediate substructure of a list, making this function structurally recursive.

Generatively recursive functions perform recursive calls not in the immediate substructures, but on some generated values. An example of this is the function that computes the greatest common divisor of two natural numbers:

```
Fixpoint gcd (a b : nat): nat :=
match a with
```

```
  O => b
  | S a' => gcd (b mod (S a')) (S a')
end.
```

Notice that the recursive call is done on `b mod (S a')` and `S a'`. The function is not structurally recursive since the recursive call is not simply done on `a'`. However, we can see that argument `a` is decreasing, meaning that the base case will be reached. Since this is a simple case, Coq can figure out that the function terminates and thus accepts it. For more complex functions this is not the case.

We make a distinction between these two types of recursion when investigating termination. For structurally recursive functions we clearly have a decreasing argument since the recursive call is always done on a smaller object. Therefore we know that the base case will be reached, which is `nil` in the `sum` function. This ensures that the function terminates. Proof of termination for such functions is done by structural induction:

Definition 2.2.1 (The Principle of Structural Induction[2])**.**
If

1. proposition P holds for all the minimal structures
2. *if* proposition P holds for the immediate substructures of structure X (the Induction Hypothesis)
   then proposition P also holds for X

then proposition P holds for all such structures.

Structural induction is not suitable for proving termination of generatively recursive functions, since the recursive calls are not done on immediate substructures. Therefore, termination needs to be proven in some other way. One possibility is by proving that the function is well-founded, which we discuss in the following section.

2.2.2 Well-founded relations

Definition 2.2.2 (Well-founded relations)**.** A binary relation R on a class X is *well-founded* if the following holds:

$$\forall S \subseteq X \quad [S \neq \emptyset \rightarrow \exists m \in S \quad [\forall s \in S \quad [\neg sRm]]]$$

In natural language: every non-empty subset S of X contains an m such that sRm does **not** hold for any $s \in S$. Such an m is called a minimal element. So for a relation to be well-founded, every non-empty subset of the

[2]This principle is derived from the Principle of Structural Induction given in Semantics With Applications [14]

corresponding class should contain such a minimal element. An equivalent definition of a well-founded relation R on a class X is that there does not exist an infinitely decreasing R-chain, where a decreasing R-chain looks like this:

$$\ldots (s\ (i-3))\ R\ (s\ (i-2))\ R\ (s\ (i-1))\ R\ (s\ i)$$

where $s : \mathbb{Z} \to X$ and $i \in \mathbb{Z}$. Note that this chain continues infinitely to the left.

An example of a well-founded relation is the relation $<$ on natural numbers. For every non-empty subset of $\mathbb{N}$ there is a minimal element. In the case of $\mathbb{N}$, the minimal element is 0, since there does not exist an $n \in \mathbb{N}$ for which $n < 0$ holds. In any other non-empty subset of $\mathbb{N}$ there also exists an element that is the lowest, which is the minimal element. The relation $\leq$ on natural numbers, however, is not well-founded. For example, the set $S = \{0, 1\}$ does not contain a minimal element because $0 \leq 0 \leq 0 \ldots$. This is due to the reflexivity of $\leq$. This problem does not arise for $<$ since this is not reflexive.

Definition 2.2.3 (Reflexive relations)**.** A binary relation R on a class X is *reflexive* if the following holds:

$$\forall x \in X \quad [xRx]$$

So in a reflexive relation, every element relates to itself. It follows that reflexive relations do not have minimal elements and therefore are not well-founded.

Non-reflexive relations are not necessarily well-founded. For example, the relation $<$ on the integers is not reflexive but $\mathbb{Z}$ does not contain a minimal element (no integer is the lowest).

Chapter 3

The Equations plugin

Equations is a tool for Coq originally created by Matthieu Sozeau in 2010 [19]. Together with Cyprien Mangin, Sozeau rewrote Equations in 2019 to make it more powerful [20]. Equations is the current solution for proving that functions are well-founded and for proving properties about these functions. Equations serves as a improvement to the Program package [18] and the Function package [2]. When defining a function, the user has to explicitly state the decreasing argument. This will generate proof obligations to show that this argument is indeed decreasing. Once these have been proven, Coq knows that the function terminates and therefore accepts the definition. Furthermore, Equations supports dependent pattern matching and with- and where-clauses.

In this chapter, we will introduce some of the features of Equations. The true inner workings of Equations are out of scope for this project, but it is important to note that Equations is merely a definitional extension to Coq; it does not extend the core logic of Coq, therefore assurance is not sacrificed.

3.1 Equations definitions

This section serves as a guide to defining functions using the Equations plugin. The function `sum` computes the sum of all elements of a list of natural numbers. Here it is defined using a regular `Fixpoint` definition:

```
Fixpoint sum (l : list nat) : nat :=
match l with
  nil => 0
  | (h::tl) => h + sum tl
end.
```

Here it is defined using Equations:

```
Equations sum (l : list nat) : nat :=
sum nil := 0 ;
sum (h::tl) := h + sum tl.
```

An Equations definition starts with the keyword `Equations`. Notice that the Equations definition uses pattern matching in a similar syntax to Haskell and thus does not need the `match ... with ... end` construct. Similarly to Haskell, one can pattern match on `_` when the value of the corresponding parameter does not matter. To enter proof mode immediately after running the function definition, follow up the `Equations` keyword with a `?` in the regarding function.

Generatively recursive functions can be defined using the `by wf ...` R construct in the function signature, where R is the well-founded relation. On the dots should be the argument for which the well-founded relation holds. See the following example, which computes the greatest common divisor of two given natural numbers:

```
Equations gcd (n m : nat) : nat by wf m lt:=
gcd x 0 := x ;
gcd x y := gcd y (x mod y).
```

In this case the well-founded relation R is $<$ (`lt`) and the corresponding argument is `m`. We refer to `m` as the decreasing argument. The value of `m` will eventually become `0`, which is the minimal element in the relation $<$ (`lt`) of the natural numbers. This makes the function `gcd` well-founded. However, this does need to be proven before the function is defined. Equations generates multiple proof obligations for this, which are automatically solved except for one, which should be manually proven by the user. Once this proof is complete, the function `gcd` is defined successfully.

An Equations definition supports predicate subtypes as return types, we have seen these types in Section 2.1.7. The `Set Program Mode` command must be executed first to enable it, just like for Program definitions. For clarity, we present the `div2` function implemented once using Program and once using Equations. This function divides natural number by two (rounding down) and returns a predicate subtype:

```
Program Fixpoint div2 (n : nat) :
  { x : nat | n = 2 * x \/ n = 2 * x + 1 } :=
match n with
  | S (S p) => S (div2 p)
  | _ => 0
end.


Equations div2 (n : nat) :
```

```
  { x : nat | n = 2 * x \/ n = 2 * x + 1 } :=
div2 (S (S p)) := S (div2 p);
div2 _ := 0.
```

3.2 Properties generated by Equations

When defining a function using Equations, certain properties of this function are derived. These properties are useful in Coq proofs about Equations definitions. We will discuss these properties using the Equations definition for `sum` given in Section 3.1. Firstly, for each of the patterns a lemma is created corresponding to the function definition. These lemmas are essentially rewrite rules, allowing the user to treat the function definition as a rewriting system. In the case of `sum`, these lemmas are called `sum_equation_1` and `sum_equation_2`:

```
sum_equation_1 : sum nil = 0
```

```
sum_equation_2 :
forall (n : nat) (l : list nat),
  sum (n :: l) = n + sum l
```

Notice how these lemmas indeed correspond to the definition of `sum`.

Then Equations generates the inductive type `sum_graph`, which holds lists of natural numbers and their sums. The relation `sum_graph` defines the function `sum` as a relation. It defines the graph of the function `sum : sum l = n` if and only if the type `sum_graph l n` is inhabited.

```
Inductive sum_graph : list nat -> nat -> Set :=
    sum_graph_equation_1 : sum_graph nil 0
  | sum_graph_equation_2 : forall (n : nat) (l : list nat),
                             sum_graph l (sum l)
                           ->
                             sum_graph (n :: l) (n + sum l)
```

The property `sum_graph_rect` is the standard induction principle for the inductively defined relation `sum_graph`:

```
sum_graph_rect :
forall P : list nat -> nat -> Type,
    P nil 0
  ->
      (forall (n : nat) (l : list nat),
          sum_graph l (sum l)
        ->
            P l (sum l)
          ->
```

```
            P (n :: l) (n + sum l))
    ->
       forall (l : list nat) (n : nat),
         sum_graph l n -> P l n
```

The following property shows that `sum` and `sum_graph` correspond correctly, meaning that any list of natural numbers together with its sum forms a `sum_graph`. That is, `sum_graph` is indeed the graph of the function `sum`.

```
sum_graph_correct :
forall l : list nat,
  sum_graph l (sum l)
```

The following eliminator lemma is useful for proving properties of `sum`, where `P` is the property.

```
sum_elim :
forall P : list nat -> nat -> Type,
    P nil 0
  ->
       (forall (n : nat) (l : list nat),
           P l (sum l)
         ->
           P (n :: l) (n + sum l))
    ->
      forall l : list nat,
        P l (sum l)
```

Chapter 4 will present examples of using such an eliminator.

3.3 Proof tactics for Equations

The Equations plugin includes some proof tactics that work well with Equations-defined functions, which we introduce in this section.

The Coq standard library tactic `simpl` can be used to simplify function applications of functions defined using Fixpoint. This tactic does not work with Equations definitions. Fortunately, a tactic with similar results for Equations definitions is provided, namely `simp` $f_1 \dots f_n$, where $f_1 \dots f_n$ are the names of the Equations definitions you want to simplify. This tactic is an alias for `autorewrite with` $f_1 \dots f_n$`; try typeclasses eauto with Below subtermrelation` $f_1 \dots f_n$. The tactic works by applying rules from the internal associated rewrite hint databases of Coq.

Equations also provides the `funelim` f *args* tactic, where f is a function and *args* are its arguments. This tactic eliminates f by creating subgoals according to the case splitting tree of f.

Chapter 4

Applications of Equations

We explore Equations in practice by defining a couple of well-known generatively recursive functions in Coq using Equations. For each function, we also prove its correctness in Coq. This chapter discusses these functions and gives an overview of how the proofs are done, where some trivial lemmas are left out. Refer to the Coq files for the full proofs.

In this chapter, we often speak of lists or vectors being permutations of each other. Note that the binary relation of 'being a permutation of' is symmetric, so if list l_1 is a permutation of list l_2, then the same holds the other way around.

4.1 Sorted lists and Permutations of lists

As preparation for creating quicksort, mergesort, and binary search on lists, we here introduce the definition of a sorted list and the definition of two lists being permutations of each other.

For the definition of a sorted list we use the `Sorted` predicate from the Coq library Coq.Sorting.Sorted. We will first look at the `HdRel` predicate, which is used in the definition of `Sorted`:

```
Variable A : Type.
Variable R : A -> A -> Prop.

Inductive HdRel a : list A -> Prop :=
  | HdRel_nil : HdRel a []
  | HdRel_cons b l :
                      R a b
                    ->
                      HdRel a (b :: l).
```

The function `R` takes two elements of type `A` and returns a proposition, making `R` a binary relation on type `A`. An `a : A` and a `(b :: l) : list A` inhabit `HdRel` if `R a b` holds. In other words, if the binary relation `R` holds on `a` and `b`, then `HdRel a (b :: l)` holds.

The `Sorted` predicate is defined as follows:

```
Inductive Sorted : list A -> Prop :=
  | Sorted_nil : Sorted []
  | Sorted_cons a l :
                       Sorted l
                     ->
                        HdRel a l
                      ->
                        Sorted (a :: l).
```

This defines a sorted list for the following reasons. Firstly, an empty list is always sorted. Secondly, a non-empty list `a :: l` is sorted if `l` is sorted and if `HdRel a l` holds. Thus, a list is sorted according to the binary relation `R` if all pairs of subsequent elements in the list are related according to `R`.

For the definition of two lists being a permutation of each other, we use the `Permutation` type from the Coq library Coq.Sorting.Permutation.

```
Variable A:Type.

Inductive Permutation : list A -> list A -> Prop :=
  | perm_nil: Permutation [] []
```

```
| perm_skip x l l' :
                     Permutation l l'
                   ->
                     Permutation (x::l) (x::l')
| perm_swap x y l : Permutation (y::x::l) (x::y::l)
| perm_trans l l' l'' :
                        Permutation l l'
                      ->
                        Permutation l' l''
                      ->
                        Permutation l l''.
```

Firstly, two empty lists are clearly permutations of each other. Secondly, two lists are permutations of each other when they have the same head and if their tails are permutations of each other. Thirdly, a list is a permutation of the same list but with the first two elements swapped. Lastly, `Permutation` can be seen as a binary relation which is transitive. For both the definition `Sorted` and `Permutation`, there are many lemmas given in their respective libraries.

4.2 Quicksort

Quicksort is a recursive sorting algorithm that can sort a list with an expected running time of $\Theta(n\ log\ n)$ and a worst-case running time of $\Theta(n^2)$, where n denotes the length of the input list [5]. It works by choosing one element from the list, which we refer to as the pivot, and splitting the remainder of the list into those less or equal to the pivot and those greater than the pivot. Consecutively, a recursive call is made on the two resulting lists. The base case is reached when there is an empty list; in this case, quicksort returns an empty list. Recall the Haskell definition:

```
quicksort :: Ord nat => [nat] -> [nat]
quicksort  [] = []
quicksort (x:xs) = quicksort (filter (<=x) xs) ++
                   [x] ++ quicksort (filter (>x) xs)
```

When trying to replicate this definition in Coq using Fixpoint, you run into problems since quicksort is not structurally recursive. This is the case because `(filter (<=x) xs)` and `(filter (>x) xs)` are not immediate sublists of the input list. Notice, however, that the recursive calls are done on smaller lists than the input list, making quicksort well-founded on the length of the input list. This will allow us to define quicksort using Equations. Using the `by wf _ R` syntax, we can tell Equations that the decreasing argument is the length of the list. This results in the following definition:

```
Equations? quicksort (l : list nat) :
  list nat by wf (length l) lt:=
quicksort nil := nil ;
quicksort (h :: t) := (quicksort (filter_lte h t)) ++
                      (h :: (quicksort (filter_gt h t))).
```

where `filter_lte` and `filter_gt` are helper functions for readability:

```
Definition filter_lte (a: nat) (l : list nat) :=
  filter (fun n=> n <=? a) l.

Definition filter_gt (a: nat) (l : list nat) :=
  filter (fun n=> negb(n <=? a)) l.
```

Notice how close this definition is to Haskell's definition; the readability was not compromised.

Equations generates two proof obligations to show the well-foundedness of the definition:

```
length (filter_lte h t) < S (length t)
length (filter_gt h t) < S (length t)
```

Notice how these proof obligations indeed correspond to `length l` being the decreasing argument. Both goals were provable by induction on `t`. The base case was trivial. In the inductive step we made a case distinction on whether `x <= h`, where `x` is the the first element of `t` and `h` is the pivot.

4.2.1 `quicksort_elim`

As seen in Section 3.2, one of the lemmas generated by Equations corresponding to the quicksort definition is the eliminator lemma `quicksort_elim`. For readability, we present this lemma as a derivation rule. The premises of the derivation rule are given on top of each other, above the line, separated by one line of whitespace. The conclusion of the derivation rule is given below the line.

The derivation rule for `quicksort_elim` is defined as:

```
P nil nil

(forall (n : nat) (l : list nat),
    P (filter_lte n l) (quicksort (filter_lte n l))
  ->
      P (filter_gt n l) (quicksort (filter_gt n l))
    ->
      P (n :: l) (quicksort (filter_lte n l) ++
      n :: quicksort (filter_gt n l)))
------------------------------------------------- [quicksort_elim]
forall l : list nat, P l (quicksort l)
```

where `P : list nat -> list nat -> Type`.

This lemma allows for proving a property (`P`) of the quicksort function. This lemma is used multiple times when proving the correctness of the quicksort definition in the following section.

4.2.2 Correctness of quicksort

Now that we have proven the well-foundedness of our quicksort definition and Coq has accepted the definition, we need to prove its correctness. We define the correctness of a sorting algorithm as follows: the resulting list is sorted and the resulting list is a permutation of the input list. As definitions of sorted lists and two lists being permutations of each other, we use the definitions specified in Section 4.1.

Proof of the permutation property

We start by proving that the quicksort implementation returns a permutation of the input list. We start with this since this property comes in handy

when showing that `quicksort` returns a sorted list (but it is not required). For this proof, we have created multiple lemmas. We first go over these lemmas.

```
Lemma perm_app :
forall (l l1 l2 : list nat),
    (exists (l3 l4 : list nat),
        Permutation l3 l1
      /\
        Permutation l4 l2
      /\
        Permutation l (l3 ++ l4))
  ->
    Permutation l (l1 ++ l2).
```

This lemma was proven by applying some lemmas from the standard library.

```
Lemma perm_lte_gt :
forall (l : list nat) (n : nat),
  Permutation l (filter_lte n l ++ filter_gt n l).
```

This lemma was proven by induction on `l`. In the inductive step, a case distinction similar to the one in the well-foundedness proof is used.

```
Theorem quicksort_permutation :
forall (l : list nat),
  Permutation l (quicksort l).
```

The most important step in proving that `quicksort` returns a permutation of the input list was applying `quicksort_elim`. We also used the previously defined lemmas `perm_app` and `perm_lte_gt`.

Proof of the sorted property

Now that we know that `quicksort` returns a permutation of the input list, we want to prove that `quicksort` always returns a sorted list. By a sorted list is meant a list in non-decreasing order. For this proof, we have created multiple lemmas. We first go over these lemmas.

```
Lemma in_gt :
forall (elem n : nat),
    (exists (l : list nat),
      In elem (filter_gt n l))
  ->
    elem > n.

Lemma in_lte :
forall (elem n : nat),
```

```
    (exists (l : list nat),
      In elem (filter_lte n l))
  ->
    elem <= n.
```

Both of these lemmas were proven using basic proof tactics and some lemmas from the standard library.

```
Lemma in_qs :
forall (l : list nat) (elem : nat),
    In elem l
  <->
    In elem (quicksort l).
```

This lemma was easily proven using the fact that we have already shown that `quicksort` returns a permutation of the input list.

```
Lemma sort_app :
forall (l l': list nat) (n : nat),
    Sorted le l
  ->
      Sorted le l'
    ->
        (forall (elem : nat),
            In elem l
          ->
            elem <= n)
      ->
          (forall (elem : nat),
              In elem l'
            ->
              elem > n)
        ->
          Sorted le (app l (n::l')).
```

This lemma is proven by induction on `l`.

```
Theorem quicksort_sorted :
forall (l : list nat),
  Sorted le (quicksort l).
```

We used `quicksort_elim` to show that `quicksort` always returns a sorted list. We also used the previously defined lemmas `in_gt`, `in_lte`, `in_qs`, and `sort_app`.

Conclusion

Now that we have shown that `quicksort` permutes and sorts, we can easily prove its correctness by proving:

```
Theorem quicksort_correct :
forall (l : list nat),
  Sorted le (quicksort l) /\ Permutation l (quicksort l).
```

Applying `quicksort_sorted` and `quicksort_permutation` proves this theorem.

4.3 Mergesort

Mergesort is a recursive sorting algorithm with a time-complexity of $\Theta(n\ log\ n)$, where n denotes the length of the input list [5]. It works by recursively splitting the list, sorting the two resulting lists, and then merging them in such a way that the result is a sorted list. The base case is an empty list or a singleton list, which both trivially are sorted. Our implementation concerns sorting lists of natural numbers in non-decreasing order. We first take a look at a mergesort implementation in Haskell [12]. First, we look at `merge`, which merges two lists. In the application of `merge`, these two lists will always be sorted. The `merge` function merges these sorted lists into one sorted list. It does so by looking at the first elements of both of the lists and comparing them, the smallest of the two will be put in front of the resulting list.

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x <= y    = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
```

Note that there is not one fixed decreasing argument, depending on whether `x<=y` the recursive call is made on either the tail of the first list (`xs`) and the entire second list (`y:ys`) or the entire first list (`x:xs`) and the tail of the second list (`ys`). Because of this, Coq cannot guess the decreasing argument and therefore we need the help of Equations.

The functions `firstHalf` and `secondHalf` return the first and the second half of a given list, respectively. They are just there for readability's sake. Mergesort (`msort`) does a recursive call on the first half and the second half of the list and then merges the resulting lists. Note that the `[a]` case is necessary to ensure termination, otherwise you end up in an infinite loop when the given list contains one element.

```
firstHalf  xs = let { n = length xs } in take (div n 2) xs
secondHalf xs = let { n = length xs } in drop (div n 2) xs

msort :: Ord a => [a] -> [a]
msort [] = []
msort [a] = [a]
msort xs = merge (msort (firstHalf xs)) (msort (secondHalf xs))
```

Note that mergesort is not structurally recursive. The recursive calls are made on the first and the second half of the list which are substructures of the list but are not immediate substructures of the list. This means that we can not define it in Coq as is, but we can when using Equations. We first

define `merge`. The decreasing argument is `length l + length l'`, since this decreases in every recursive call. Aside from specifying the decreasing argument, the definition is just a translation from Haskell.

```
Equations? merge (l l' : list nat) :
  list nat by wf (length l + length l'):=
merge xs nil := xs ;
merge nil ys := ys ;
merge (x::xs) (y::ys) :=
  if x <=? y then x :: merge xs (y::ys)
  else y :: merge (x::xs) ys.
```

This gives rise to one proof obligation to show that `length l + length l'` is indeed the decreasing argument making the function well-founded:

```
S (length xs + length ys) < S (length xs + S (length ys))
```

This trivially holds and is proven using the tactic `lia`.

For the `mergesort` definition we have also defined two helper definitions for readability. These are `first_half` and `second_half`, which return the first half and the second half of a given list, respectively. They make use of the functions `firstn` and `skipn` from the standard library. The function `Nat.div2` divides a given natural number by two. If the length of the list is uneven, the element in the middle will be in the second half of the list since `Nat.div2` rounds downwards. The resulting definition once again is just a translation from Haskell. The decreasing argument is `length l`, since this is divided by 2 for every recursive call and therefore decreases.

```
Definition first_half (l: list nat) :=
  (firstn (Nat.div2 (length l)) l).
Definition second_half (l: list nat) :=
  (skipn (Nat.div2 (length l)) l).

Equations? mergesort (l : list nat) :
  list nat by wf (length l) lt :=
mergesort nil := nil;
mergesort (a::nil) := a::nil;
mergesort l :=
  merge (mergesort (first_half l)) (mergesort (second_half l)).
```

This definition gives rise to two proof obligations, which together show that `length l` is indeed the decreasing argument and the function is thus well-founded. The context for both obligations is the following:

```
n, n0 : nat
l0 : list nat
l := n :: n0 :: l0 : list nat
```

We first take a look at the first proof obligation:

```
S (length (firstn (Nat.div2 (length l0)) (n0 :: l0)))
  < S (S (length l0))
```

This corresponds to the first recursive call, `mergesort (first_half l)`, where the argument should be smaller than its current value. This proof obligation was proven by induction on `length l0`. We now take a look at the second proof obligation:

```
length (second_half l) < S (S (length l0))
```

This, more clearly, corresponds to the second recursive call. This proof obligation was proven by induction on `l0`.

4.3.1 `merge_elim` and `mergesort_elim`

As we have seen in Section 3.2, Equations derives an eliminator for its functions. Such an eliminator can be used to prove properties about these functions. Both the eliminator for `mergesort` and `merge` will be used in showing the correctness of `mergesort` in Section 4.3.2. These eliminators are shown below as derivation rules. Section 4.2.1 describes how such a derivation rule should be read.

The derivation rule for `merge_elim` is defined as:

```
(forall l : list nat, P l nil l)

(forall (n : nat) (l0 : list nat), P nil (n :: l0) (n :: l0))

(forall (n0 : nat) (l : list nat) (n : nat) (l0 : list nat),
    P l (n :: l0) (merge l (n :: l0))
  ->
     P (n0 :: l) l0 (merge (n0 :: l) l0)
   ->
     P (n0 :: l) (n :: l0)
     (if n0 <=? n then n0 :: merge l (n :: l0)
     else n :: merge (n0 :: l) l0))
-------------------------------------------------------- [merge_elim]
forall l l' : list nat, P l l' (merge l l')
```

where `forall P : list nat -> list nat -> list nat -> Type.`

The derivation rule for `mergesort_elim` is defined as:

```
P nil nil

(forall n : nat, P (n :: nil) (n :: nil))

(forall (n n0 : nat) (l : list nat),
    P (first_half (n :: n0 :: l))
    (mergesort (first_half (n :: n0 :: l)))
  ->
      P (second_half (n :: n0 :: l))
      (mergesort (second_half (n :: n0 :: l)))
    ->
      P (n :: n0 :: l)
      (merge (mergesort (first_half (n :: n0 :: l)))
      (mergesort (second_half (n :: n0 :: l)))))
------------------------------------------------- [mergesort_elim]
forall l : list nat, P l (mergesort l)
```

where `P : list nat -> list nat -> Type`.

4.3.2 Correctness of mergesort

As we have done with quicksort in Section 4.2, we prove the correctness of our mergesort definition by proving that the resulting list is always sorted and that the resulting list is always a permutation of the input list. We use the definitions of a sorted list and of two lists being a permutation of each other given in Section 4.1.

Proof of the permutation property

We start by proving that `mergesort` returns a permutation of the input list. We first go over the lemmas created for this proof.

```
Lemma perm_merge :
forall (l1 l2 l3 : list nat),
    Permutation l1 (app l2 l3)
  ->
    Permutation l1 (merge l2 l3).
```

This lemma was proven using `merge_elim` and a few lemmas from the standard library.

```
Lemma perm_app :
forall (l l1 l2 : list nat),
    (exists (l3 l4 : list nat),
        Permutation l3 l1
      /\
```

```
      Permutation l4 l2
    /\
      Permutation l (l3 ++ l4))
->
  Permutation l (l1 ++ l2).
```

This lemma was proven using a few lemmas from the standard library.

```
Theorem mergesort_permutation :
forall (l : list nat),
  Permutation l (mergesort l).
```

Using the previously shown lemmas in combination with `mergesort_elim`, this theorem was proven. We have now shown that `mergesort` returns a permutation of the input list.

Proof of the sorted property

Now that we have shown that `mergesort` permutes, we will show that it always returns a list sorted in non-decreasing order. We first go over the lemmas defined for this proof. The definition of `HdRel` is given in Section 4.1.

```
Lemma hdrel_merge :
forall (l l' : list nat) (a a' : nat),
    Sorted le (a :: l)
  ->
      Sorted le (a' :: l')
    ->
        a <= a'
      ->
          HdRel le a (merge l (a' :: l'))
        /\
          HdRel le a (merge (a' :: l') l).
```

This was proven by destructing on `l`, which behaves the same as induction but leaves out the induction hypothesis. A case distinction is made by comparing `a'` and the head of `l`.

```
Lemma merge_sorts :
forall (l : list nat),
    Sorted le l
  ->
    forall (l' : list nat),
        Sorted le l'
      ->
        Sorted le (merge l l').
```

This lemma is proven using `induction 1`, Section 2.1.9 describes what this does. The inductive step is proven using `induction 1`. A case distinction is made by comparing the values of the heads of the two lists `l` and `l'`. The lemma created previously, `hdrel_merge`, was also applied.

```
Theorem mergesort_sorted :
forall (l : list nat),
  Sorted le (mergesort l).
```

By first applying `mergesort_elim` and then `merge_sorts`, this theorem was proven. We have now shown that `mergesort` returns a sorted list in non-decreasing order.

Conclusion

Now that we have shown that `mergesort` permutes and sorts, we can easily prove its correctness by proving:

```
Theorem mergesort_correct :
forall (l : list nat),
  Sorted le (mergesort l) /\ Permutation l (mergesort l).
```

Applying `mergesort_sorted` and `mergesort_permutation` proves this theorem.

4.4 Binary Search

The binary search algorithm finds the position of a given target value in a given sorted list. This list should be sorted in non-decreasing order. Its worst-case running time is $\Theta(log\ n)$, where n is the length of the input list [5]. The binary search algorithm searches for the position of the target value by comparing the element in the middle of the list to the target value. If the target value equals the element in the middle of the list, we have found its position. If the target value is lower, we know that its position is in the first half of the list. Furthermore, if the target is higher, we know that its position is in the second half of the list. These conclusions can be drawn due to the list being sorted.

In this implementation, we also cover the case where the given target value is not in the given list. This is done by having `Maybe Int` as the return type of the function, which equals `Nothing` if the target is not in the list and `Just p` otherwise, where `p` is the position of the target value in the list.

We will refer to the part of the list in which the target value could still be as the *relevant* part of the list. In this implementation, we keep track of the relevant part of the list by keeping track of its borders, where `l` denotes the left border position and `r` denotes the right border position. These borders are both included in the relevant part of the list. The first element of the list has position 0.

We first look at a binary search implementation in Haskell:

```
binarysearch :: Ord a => [a] -> a -> Int -> Int -> Maybe Int
binarysearch list t l r
   | l > r = Nothing
   | list !! m  < t = binarysearch list t (m+1) r
   | list !! m  > t = binarysearch list t l (m-1)
   | otherwise = Just m
   where m = (l+r) `div` 2
```

Once `l > r`, the relevant part of the list is empty. This means that the target value is not in the list, therefore we return `Nothing`.

The recursive calls are done with either `l` increased or `r` decreased, making this function not structurally recursive. The function is well-founded on `R-L`, since its value decreases every recursive call. Note that `R-L` corresponds to the length of the relevant list.

When translating this implementation to Coq one encounters a problem. Equations generates proof obligations to show that binary search is well-founded on `R-L`, which look like this:

```
R - (Nat.div2 (L + R) + 1) < R - L
```

```
Nat.div2 (L + R) - 1 - L < R - L
```

The only relevant restriction in the corresponding context is that `(R <? L) = false`, so it can be the case that `R = L`. This would mean that `R - L = 0`. Bearing in mind that we are working with natural numbers, we can not prove that something is less than 0. To solve this problem, we need to mold the definition of binary search in such a way that its termination becomes provable in Coq.

We do this by shifting the right border `r` one spot to the right, meaning that it now points to the first position after the relevant part of the list. The right border is now no longer included in the relevant part of the list but the left border still is. Instead of checking whether `l > r`, we now check whether `l >= r`. This makes the termination of the resulting definition provable because we now have `(R <=? L) = false` in our context instead of `(R <? L) = false`. It follows that `R > L`, therefore `R - L` does not equal 0 and we no longer have the described problem. To ensure that the right border stays one position to the right from the relevant part of the list, we need to change the arguments given to the second recursive call. This recursive call is executed when the target should be to the left of the middle of the list, which means that the right border should become the current middle of the list (`m`) instead of one position to the left of the middle of the list (`m-1`). Note that shifting `r` has an effect on the value of `m`, but this is not a problem due to the nature of the algorithm. After presenting the resulting definition and its proof of termination, we will walk through an execution of binary search including a visualization.

This leads to the following definition of `binarysearch`:

```
Equations? binarysearch (l: list nat) (target L R :nat) :
  option nat by wf (R-L) lt:=
binarysearch l t L R with dec(R <=? L) =>
  | left H => None;
  | right H with dec(t =? nth (Nat.div2 L+R) l (t+1))=>
    | left H => Some (Nat.div2 L+R);
    | right H => if nth ((Nat.div2 L+R)) l (t+1) <? t
                 then binarysearch l t ((Nat.div2 (L+R))+1) R
                 else binarysearch l t L ((Nat.div2 (L+R)))
```

where `nth (Nat.div2 L+R) l (t+1)` returns the `(Nat.div2 L+R)`-th element of `l` and returns `t+1` in case this element is out of bounds. Note that `Nat.div2 (L+R)` corresponds to the middle of the relevant part of the list, previously denoted as `m`.

Since only the second recursive call has been altered, only the second proof obligation has changed:

```
R - (Nat.div2 (L + R) + 1) < R - L
```

```
Nat.div2 (L + R) - L < R - L
```

In each subproof of these obligations, including proofs of newly defined lemmas, many lemmas on natural numbers were used from the standard library. These proofs were more challenging than they seem, which is due to the properties on $<$ on natural numbers being generally weaker than those on $<$ on integers (since no natural number is less than 0).

The first proof obligation was proven using the following lemma:

```
Lemma lte_div2 :
forall (n m : nat),
    n <= m
  ->
    Nat.div2 n <= Nat.div2 m.
```

which was proven using a couple of lemmas from the standard library.

The second proof obligation was proven using the following lemma:

```
Lemma lt_div2_min :
forall (n m : nat),
    n < m
  ->
    Nat.div2 (n + m) - n < m - n.
```

which was proven using this lemma:

```
Lemma lte_plus_div2 :
forall (n m : nat),
    m <= (n + m) / 2
  ->
    m <= n.
```

which also was proven using lemmas from the standard library.

For clarity, we will now walk through an execution of the binary search implementation. See Figure 4.1 for a visualization of the state at every iteration. In the figure, the squares of elements outside of the relevant part of the list are colored grey. The positions L and R denote the left and right border of the relevant part of the list, respectively. Recall that L is included in the relevant part of the list and R is not. The position M is the middle of the relevant part of the list, calculated as $M = \lfloor \frac{L+R}{2} \rfloor$. For this example, we have 4 as our target value and $[1, 3, 4, 6, 8, 9]$ as our list which is sorted in non-decreasing order.

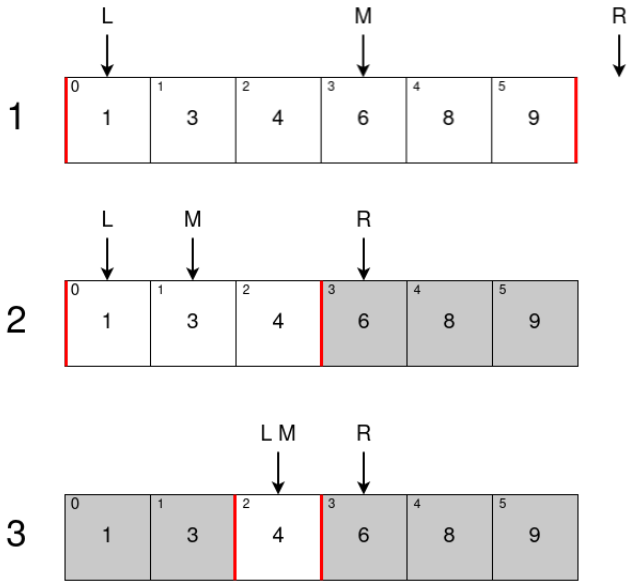

Figure 4.1: Visualization of the binary search algorithm

1. At the start, $L = 0$ and $R = 6$, making $M = \lfloor \frac{0+6}{2} \rfloor = 3$. This position holds the value 6. Since $4 < 6$ we know that the target value 4 is somewhere to the left of position M, therefore we set R to M, so $R = 3$.
2. We now recalculate $M = \lfloor \frac{0+3}{2} \rfloor = 1$. At this position we have value 3. Since $3 < 4$ we know that the target value 4 is to the right of position M, and therefore we set L to $M + 1 = 2$.
3. We, once more, recalculate $M = \lfloor \frac{2+3}{2} \rfloor = 2$. At this position we have the target value 4, which means we return the current value of M (the position of the target value).

4.4.1 Correctness property of binary search

The proof of correctness of the binary search implementation is not executed due to time restrictions. However, we do present the correctness property defined for `binarysearch`.

We define the correctness of `binarysearch` as follows: given a sorted list and a target value that is in the list, `binarysearch` returns a position in the list where the corresponding value equals the target value. Furthermore, given a sorted list and a target value that is not in the list, `binarysearch` always returns `None`.

Bear in mind that there may be duplicates in the list, more specifically, the

target value may be in the list multiple times. In this case, it does not matter which instance of the target value is found, as long as `binarysearch` returns a position at which the value equals the target value. This means that the following correctness property is incorrect: when given a sorted list with the target value at position p, `binarysearch` returns position p.

We have attempted to prove its correctness by defining the correctness property as theorems, just like we did for the other implemented algorithms. Unfortunately, this leads to information loss in the context of the goal, making them unprovable. To show this, one theorem is included in the Coq file. We found that specifying the correctness property in the return type of the `binarysearch` function as a predicate subtype does not lead to information loss. Information on predicate subtypes can be found in Section 2.1.7. This results in the following return type of `binarysearch`:

```
{p : option nat |
    Sorted le l
  ->
      (forall (e:nat),
          p=Some e
        ->
          nth e l (target+1) = target)
    /\
      (p=None -> ~ In target l)}
```

where `nth e l (target+1)` returns the `e`-th value of list `l` and returns `target+1` in case `e` is out of bounds.

This return type will give rise to proof obligations verifying the specified property. Proving these obligations will prove that the correctness property holds for the `binarysearch` function.

4.5 Sorted vectors and Permutations of vectors

As preparation for creating a version of quicksort that operates on vectors, we here introduce the definition of vectors and some definitions about vectors. The definition of vectors we use is from the Vectors.VectorDef library (with adjusted constructor names):

```
Inductive vector (A : Type) : nat -> Type :=
  Vnil : vector A 0
  | Vcons : A -> forall n : nat, vector A n -> vector A (S n)
```

Note that a vector is partially defined by its length, which is `0` for the empty vector `Vnil` and `S n` when an element is added to a vector of length `n`.

We use the definition of `In` created by Sozeau [17], which returns `True` if and only if a given value is an element of a given vector:

```
Equations In {A n} (x : A) (v : vector A n) : Prop :=
  In x Vnil := False;
  In x (Vcons a v) := (x = a) \/ In x v.
```

As a definition of a sorted vector we use a definition created by Sozeau [17]:

```
Inductive Sorted {A : Type} (R : A -> A -> Prop) :
  forall {n}, vector A n -> Prop :=
  | Sorted_nil : Sorted R Vnil
  | Sorted_cons {a n} {v : vector A n} :
      All (R a) v
    ->
        Sorted R v
      ->
        Sorted R (Vcons a v).
```

where `All` is defined as:

```
Inductive All {A : Type} (P : A -> Prop) :
  forall {n}, vector A n -> Prop :=
  | All_nil : All P Vnil
  | All_cons {a n} {v : vector A n} :
      P a
    ->
        All P v
      ->
        All P (Vcons a v).
```

This definition is similar to the one given for lists in Section 4.1. The proposition `All` serves the same purpose for vectors as `HdRel` serves for lists. In the following section we will specifically consider lists sorted in

non-decreasing order, for which we use the following definition also created by Sozeau [17]:

```
Definition sorted {n} (v : vector nat n) :=
  Sorted (fun x y => x <= y) v.
```

We also use a couple of lemmas about `All` and `In` from Sozeau and some created ourselves in the following section.

For the definition of two vectors being a permutation of each other, we have created an inductive definition `VPermutation` which is simply the `Permutation` type from the Coq library Coq.Sorting.Permutation translated to vectors. Basing this definition on the definition for lists allows us to assume its correctness.

```
Inductive VPermutation {A}:
  forall {n m}, (vector A n) -> (vector A m) -> Prop :=
    Vperm_nil : VPermutation Vnil Vnil
  | Vperm_skip {x n m} {v:vector A n} {w:vector A m}:
                  VPermutation v w
                ->
                  VPermutation (Vcons x v) (Vcons x w)
  | Vperm_swap {x y n} {v:vector A n} :
                VPermutation (Vcons y (Vcons x v))
                    (Vcons x (Vcons y v))
  | Vperm_trans {n m o} {u:vector A n} {v:vector A m}
                {w:vector A o}:
                   VPermutation u v
                 ->
                     VPermutation v w
                   ->
                     VPermutation u w.
```

We have created lemmas about `VPermutation` which are lemmas on `Permutation` translated from lists to vectors. Among these are lemmas saying that `VPermutation` is reflexive, symmetric, and transitive. Furthermore, there are a handful of lemmas about `VPermutation` in combination with `app`, which have also been translated from the Coq.Sorting.Permutation library.

4.6 Quicksort on vectors

We will create a version of quicksort that operates on vectors. As we have done for lists, we implement quicksort such that it sorts natural numbers in non-decreasing order. See Section 4.2 for an explanation of the quicksort algorithm and why it is not structurally recursive.

We start by defining the `filter` function on vectors, which does not exist in the Coq standard library unlike its version on lists. The `filter` function should take a vector of natural numbers and a filter condition, which is a function that returns a boolean given a natural number. The `filter` function returns a vector containing only the values from the input vector that satisfy the filter condition. This is challenging to define since vectors are partially defined by their length, which means that we need to know the length of the return vector to specify the return type of the function but we have no way of statically knowing the length of the return vector. This length depends on the values in the input vector and on the filter condition. More specifically, it equals the number of values in the input vector that satisfy the filter condition. We solve this problem by using a dependent pair as a return type. More on dependent pairs can be found in Section 2.1.8. The resulting definition looks like this:

```
Equations? filter {n} (v:vector nat n)(f:nat->bool) :
  sig (p : nat), {w:vector nat p
  | p = len_filtered v f /\ p<=n /\ filtered f w}:=
filter Vnil f := (0, Vnil);
filter (Vcons h t) f with dec(f h) :=
  { | left p0 => (_, Vcons h ((filter t f).2)) ;
    | right q0 => (_, (filter t f).2)}.
```

We first look at the function signature. The `filter` function takes a vector `v` of size `n` and a function `f : nat -> bool`, which is the filter condition. The return type is a dependent pair of `p : nat` and `w : vector nat p`, which is a dependent pair because the type of `w` depends on the value of `p`. This dependent pair is defined as a Sigma type.

To make the return type more specific, we have used predicate subtyping to specify some properties of the return type. These properties will be useful in the proof of correctness of `quicksort` later on. The first property is `p = len_filtered v f`, where `len_filtered` is defined as:

```
Equations len_filtered {n} (v:vector nat n)(f:nat->bool): nat:=
len_filtered Vnil f := 0;
len_filtered (Vcons h t) f :=
match dec(f h) with
  | left p0 => 1+len_filtered t f
```

```
  | right q0 => len_filtered t f
end.
```

Note that `len_filtered` is structurally recursive and therefore could also have been defined as a `Fixpoint` definition.

This function returns the number of values in the vector `v` that satisfy the condition `f`. As said previously, this value should equal the length of the return vector `w` of `filter`, which is `p`. The second property is `p<=n`, which says that the length of the return vector `w` is smaller than or equal to the length of the input vector `v`. The third propery is `filtered f w`, where `filtered` is an inductive type defined as:

```
Inductive filtered (f:nat->bool) :
    forall {n}, (vector nat n) -> Prop :=
  | filtered_nil : filtered f Vnil
  | filtered_cons {h n} {v:vector nat n}:
      f h = true
    ->
        filtered f v
      ->
        filtered f (Vcons h v).
```

This property specifies that all values in the return vector `w` satisfy the filter condition `f`. Note that it is also possible to prove these properties of the return type as lemmas after defining `filter` without using a predicate subtype as return type.

We now look at the body of the `filter` function. We use pattern matching to distinguish between an empty vector `Vnil` and a non-empty vector `(Vcons h t)`. In case of the empty vector, we don't have anything to filter so we return an empty vector. As the return type is a dependent pair of the length of the vector and the vector itself, we return `(0, Vnil)`. In case of a non-empty vector, we have another case distinction to make, which is whether the head of the vector `h` satisfies the filter condition `f`. This is done using `dec(f h)`, see Section 2.1.6 for an explanation of `dec`. When `f h` is true we end up in the `left p0` branch, where we return `(_, Vcons h ((filter t f).2))`. The `_` notation lets Coq fill in the value by itself, which Coq can derive since it is the length of `cons h ((filter t f).2`. This `.2` notation gets the second value of a dependent pair, in this case, `filter t f` returns a dependent pair of the length of the vector and the vector itself, therefore the `.2` retrieves the vector from this pair. When `f h` is false we end up in the `right p0` branch, in this case, `h` does not satisfy the filter condition and therefore we merely recurse on the remainder of the vector `(_, (filter t f).2)`.

Note that `filter` is structurally recursive since it recurses on the tail of the

vector, therefore we do not need to show that it is well-founded.

Since this definition of `filter` uses predicate subtyping, proof obligations are generated to prove that the function indeed adheres to the given properties in the predicate subtype. For each case in the `filter` definition a proof obligation is generated that corresponds to `p = len_filtered v f /\ p<=n /\ filtered f w`. There are 3 such cases, one where `v` is `Vnil` and two where `v` is `Vcons h t`, where there is a case for `f h` being true and one for `f h` being false. These obligations were easily proven, therefore we do not go into more detail.

As we have seen before in Section 4.2, quicksort is well-founded on the length of the input list/vector. This leads to the following definition of `quicksort`:

```
Equations? quicksort {n} (l : vector nat n) :
  vector nat n by wf n lt :=
quicksort Vnil := Vnil ;
quicksort (Vcons h t):= app (quicksort (filter_lte h t).2)
                        (Vcons h (quicksort (filter_gt h t).2))
```

where `filter_lte` and `filter_gt` are defined as:

```
Definition filter_lte {n} (a: nat) (v:vector nat n) :=
  filter v (fun x => x <=? a).

Definition filter_gt {n} (a: nat) (v:vector nat n) :=
  filter v (fun x=> negb(x <=? a))
```

This definition gives rise to three proof obligations. The context for all obligations is the following:

```
l, h : nat
t : vector nat l
```

The first two are for proving that `quicksort` is well-founded on `n`, which is the length of the vector `v`. This corresponds to showing that the length of the vector decreases at each recursive call, resulting in the following proof obligations:

```
(filter_lte h t).1 < S l

(filter_gt h t).1 < S l
```

Remember `filter_lte h t` and `filter_gt h t` are pairs of the length of the vector and the vector itself. Thus, the `.1` retrieves the first element of the pair, which is the length of the vector. These obligations say that the vector resulting from filtering `t` on some condition may not be larger than `t`. Both of these obligations are proven using the property `p<=n` in the return type of the `filter` definition. Since the vector `t` is given as input to `filter`,

its length `l` corresponds to `n` in `filter`. Furthermore, `(filter_lte h t).1` and `(filter_gt h t).1` correspond to `p` in the `filter` definition.

The third proof obligation is for proving that the length of the return vector is `n`, which ensures that the return type of `quicksort` is correct. We have not encountered such proof obligations for lists, since they are not defined by their length. The proof obligation is the following:

```
(filter_lte h t).1 + S (filter_gt h t).1 = S l
```

This obligation is first transformed using the property `p = len_filtered v f` in the return type of the `filter` definition, resulting in the following proof obligation:

```
len_filtered t (fun x : nat => x <=? h) +
S (len_filtered t (fun x : nat => negb (x <=? h))) = S l
```

This is proven using the `len_fixed` lemma, which is defined as:

```
Lemma len_fixed {l}: forall (t: vector nat l) (h:nat),
len_filtered t (fun x : nat => x <=? h)
+ len_filtered t (fun x : nat => negb (x <=? h)) = l.
```

This lemma was proven by induction on `t`.

4.6.1 `quicksort_elim` on vectors

As a result of the `quicksort` definition, Equations has defined the lemma `quicksort_elim`, as we have seen in Section 3.2. This lemma can be used to prove properties of `quicksort`. We present it as a derivation rule on the next page, Section 4.2.1 describes how to read such a derivation rule.

The derivation rule for `quicksort_elim` is defined as:

```
P 0 Vnil Vnil

(forall (n0 h : nat) (t0 : vector nat n0),
    P (filter_lte h t0).1 (proj1_sig (filter_lte h t0).2)
    (quicksort (proj1_sig (filter_lte h t0).2))
  ->
     P (filter_gt h t0).1 (proj1_sig (filter_gt h t0).2)
     (quicksort (proj1_sig (filter_gt h t0).2))
    ->
     P (S n0) (Vcons h t0)
     (eq_rect ((filter_lte h t0).1 + S (filter_gt h t0).1)
     (fun H : nat => vector nat H)
     (app (quicksort (proj1_sig (filter_lte h t0).2))
     (Vcons h (quicksort (proj1_sig (filter_gt h t0).2))))
     (S n0) (quicksort_obligation_3 n0 h t0)))
------------------------------------------------------------------ [quicksort_elim]
forall (n : nat) (l : vector nat n), P n l (quicksort l)
```

where `P : forall n : nat, vector nat n -> vector nat n -> Type`.

4.6.2 Correctness of quicksort on vectors

Coq has now accepted the definition of `quicksort`, we now prove that this definition is correct. We define the correctness of a sorting algorithm on vectors as follows: the resulting vector is a permutation of the input vector and the resulting vector is sorted in non-decreasing order. As definitions of sorted vectors and two vectors being permutations of each other, we use the definitions specified in Section 4.5. The proof of correctness is structured similarly to the proof of correctness of quicksort on lists. Despite this, we will discuss the full proof for completeness.

Proof of the permutation property

As we did in the proof of correctness of quicksort on lists, we start by proving that `quicksort` returns a permutation of the input vector. This property will come in handy when proving that `quicksort` returns a sorted vector. We go over the lemmas created for this proof.

```
Lemma perm_app {m n o p q}:
forall (v: vector nat m) (v1: vector nat n)
(v2: vector nat o) (v3: vector nat p) (v4: vector nat q),
    VPermutation v3 v1
  ->
     VPermutation v4 v2
    ->
```

```
      VPermutation v (app v3 v4)
    ->
      VPermutation v (app v1 v2).
```

This lemma was proven by using some properties of `VPermutation`.

```
Lemma perm_lte_gt {n} :
forall (v: vector nat n) (a : nat),
  VPermutation v (app (filter_lte a v).2 (filter_gt a v).2).
```

This lemma was proven by induction on `v`, along with quite some properties of `VPermutation`.

```
Theorem quicksort_permutation {n}:
forall (v : vector nat n),
  VPermutation v (quicksort v).
```

We prove that `quicksort_permutation` permutes by applying `quicksort_elim` and the previously defined lemmas `perm_app` and `perm_lte_gt`.

Proof of the sorted property

Now that we have shown that `quicksort` permutes, we also want to prove that it always returns a sorted list. We go over the lemmas created for this proof.

```
Lemma filtered_All {n} :
forall (x : nat) (R : nat -> bool) (v:vector nat n),
    filtered (fun x : nat => R x ) v
  ->
      In x v
    ->
      R x = true.
```

This lemma was proven by induction on `In x v`.

```
Lemma in_lte {n} :
forall (h x: nat) (t: vector nat n),
    In x (proj1_sig (filter_lte h t).2)
  ->
    x <= h.

Lemma in_gt {n} :
forall (h x : nat) (t: vector nat n),
    In x (proj1_sig (filter_gt h t).2)
  ->
    x > h.
```

Both of these lemmas were proven using the property `filtered f w` in the return type of `filter` and the lemma `filtered_All`.

```
Lemma in_qs {n} :
forall (t: vector nat n) (elem : nat),
    In elem t
  <->
    In elem (quicksort t).
```

This lemma was proven using `quicksort_permutation` along with some properties of `VPermutation`.

```
Lemma sort_app {l l'}:
forall (v : vector nat l) (v': vector nat l') (n : nat),
    sorted v
  ->
      sorted v'
    ->
        All (fun y : nat => y > n) v'
      ->
          All (fun y : nat => y <= n) v
        ->
          sorted (app v (Vcons n v')).
```

This lemma was proven by induction on `v`.

```
Theorem quicksort_sorted {n} :
forall (v : vector nat n),
  sorted (quicksort v).
```

We applied `quicksort_elim` to prove this theorem. Furthermore, we have used the previously defined lemmas `in_gt`, `in_lte`, `in_qs`, and `sort_app`.

Conclusion

Now that we have shown that `quicksort` permutes and sorts, we can easily prove its correctness by proving:

```
Theorem quicksort_correct {n} :
forall (v : vector nat n),
    sorted (quicksort v)
  /\
    VPermutation v (quicksort v).
```

Applying `quicksort_sorted` and `quicksort_permutation` proves this theorem.

Chapter 5

The LiquidHaskell verifier

In this chapter, we investigate if, using LiquidHaskell, the correctness of a quicksort and a mergesort implementation in Haskell can be verified. LiquidHaskell is a verifier for Haskell which lets you enforce correctness properties of your program during compile time. It can, for example, check whether a function is total and whether it always terminates. It does so by generating verification conditions and feeding them to an SMT-solver. Such an SMT-solver takes a formula in first-order logic (predicate logic) and returns whether the formula is satisfiable. Note that this is more powerful than a SAT-solver, which can decide whether a formula in propositional logic is satisfiable. Annotations for LiquidHaskell are given in a `{-@ @-}` comment block. These annotations include the properties we want LiquidHaskell to verify. To verify a property of a function, the type signature typically specifies this property and is thus given within such a comment block. Such properties are specified using refinement types, which are subtypes that include some property. These are much like Coq's predicate subtypes which we have discussed in Section 2.1.7.

5.1 Quicksort and mergesort

We explore LiquidHaskell by using it in an attempt to verify the correctness of a quicksort and a mergesort implementation in Haskell. This is done with much help from a blog on LiquidHaskell [21]. We start by defining a refined subtype of lists, the sorted list (`OrdList`). This is defined as a regular list including the constraint `<{\x v -> x <= v}>` , which says that every element in the list is smaller or equal to the subsequent element in the list.

```
{-@ type Ordlist a = [a]<{\x v -> x <= v}> @-}
```

The implementation of quicksort looks almost like a regular quicksort defi-

nition. It takes a list and returns a sorted list. To ensure that LiquidHaskell can verify that a sorted list is returned, the function `app_qs` is created, which holds some properties that a regular append function does not. Specifically, the function `app_qs` takes a pivot element (`piv`) and two sorted lists, where one contains only elements that are less or equal to the pivot element (`v <= piv`) and the other contains only elements that are greater than the pivot element (`v > piv`). These properties are specified in the function signature. LiquidHaskell can verify that if these properties hold for the input of `app_qs` then the function returns a sorted list. Since these properties hold for the application of `app_qs` in `quicksort`, LiquidHaskell can verify that `quicksort` returns a sorted list.

The `/ [len l]` notation tells LiquidHaskell that `quicksort` is well-founded on the length of `l`, where `l` is the input list. This ensures that the function terminates.

```
{-@ app_qs :: piv:a
-> Ordlist {v:a | v <=  piv}
-> Ordlist {v:a | v > piv}
-> Ordlist a
 @-}
app_qs pivot [] ys  = pivot : ys
app_qs pivot (x:xs) ys  = x : app_qs pivot xs ys

{-@ quicksort :: Ord a => l:[a] -> Ordlist a / [len l] @-}
quicksort  [] = []
quicksort (x:xs) = app_qs x
                   (quicksort ([y | y <- xs, y <= x ]))
                   (quicksort ([y | y <- xs, y > x ]))
```

We now look at an implementation of mergesort. The `merge` function takes two sorted lists and returns a sorted list. Moreover, it is well-founded with as decreasing argument `len l + len l'`, ensuring it terminates. The `mergesort` function takes a list and returns a sorted list, and is well-founded with as decreasing argument `len l`. This implementation looks like a regular mergesort definition.

```
{-@ merge :: Ord a => l:Ordlist a -> l':Ordlist a ->
    Ordlist a / [len l + len l'] @-}
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys
```

```
{-@ mergesort :: Ord a => l:[a] -> Ordlist a / [len l] @-}
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort first) (mergesort second)
    where first = take half l
          second = drop half l
          half = length l `div` 2
```

LiquidHaskell manages to verify the annotated properties in the implementations of `quicksort` and `mergesort`, which means that it verifies that both sorting algorithms return sorted lists. This verification that both implementations return sorted lists was significantly easier than proving this property in Coq. Especially for `mergesort` it was only a matter of giving proper function signatures. The `quicksort` implementation did require some creativity to sufficiently help out the SMT-solver, which was done by creating the `app_qs` function and specifying its signature.

This leaves us with having to prove that `quicksort` and `mergesort` return permutations of their input lists. This, unfortunately, is unlikely to be possible using LiquidHaskell. This is due to this permutation property being of second-order logic, where you can quantify over properties/functions instead of merely over values, which is the case in first-order logic. This makes it impossible to solve with an SMT-solver.

5.2 The permutation property

As explained previously, proving that an implementation of a sorting algorithm returns a permutation of the input list is likely to be impossible using LiquidHaskell.

We now look at an attempt of proving the permutation property, which is from an insertion sort case study [11]. This implementation uses sets, from the Data.Set library, to define the refinement type `ListE`, which holds lists containing the same elements as a given set `S`. The function `elems :: (Ord a) => List a -> S.Set a` takes a list and returns a set containing all elements in the list. Keep in mind that sets only keep track of distinct elements, therefore this type ignores duplicates in the list and therefore does not properly enforce the permutation property. This implementation uses a selfmade `List` type, where the empty list is denoted as `Emp` and the `:::` notation is synonymous to Haskells `:` notation.

```
{-@ type ListE a S = {v:List a | elems v = S} @-}

{-@ sortE :: (Ord a) => xs:List a -> ListE a {elems xs} @-}
sortE Emp         = Emp
```

```
sortE (x:::xs)    = insertE x (sortE xs)

{-@ insertE :: (Ord a) => x:a -> xs:List a ->
  ListE a {S.union (S.singleton x) (elems xs)} @-}
insertE x Emp     = x ::: Emp
insertE x (y:::ys)
  | x <= y        = x ::: y ::: ys
  | otherwise     = y ::: insertE x ys
```

Replacing sets with multisets (sets where elements can appear multiple times) in the above implementation would properly specify the permutation property. But, as mentioned before, LiquidHaskell will not be able to verify this.

LiquidHaskell also has a feature to specify laws and prove them step-by-step by writing code. However, this is not interactive as it is in Coq, making it a quite meticulous task to prove even basic laws. Furthermore, the error-reporting for proofs does not seem helpful, so you are truly on your own when using this feature.

Chapter 6

Related Work

We look into other implementations of quicksort and mergesort in Coq. We were unable to find any implementation of binary search in Coq. Afterward, we mention a couple of other ways of proving properties about functional programs aside from using Coq. We hereby want to mention that the Software Foundations series includes an implementation of mergesort using Fixpoint definitions [1] and that Sozeau has implemented quicksort using Program [16]. We will not discuss these implementations.

Throughout this chapter we will speak of the 'typical' implementations of quicksort and mergesort, by these, we refer to their Haskell implementations given in Section 4.2 and 4.3, respectfully.

6.1 Quicksort using the Bove-Capretta approach

W.R.M. Schols has implemented quicksort using Fixpoint definitions [15]. This is done using the Bove-Capretta approach, which is where you define a new inductive type on which the definition is structurally recursive, contrary to the original definition. Schols presents two versions. One of which was suggested by Gabe Dijkstra [6], which is more complicated but allows for easier validation. We will only present the other version.

First, the inductive type essential to the Bove-Capretta approach is defined. The `qs_tree` type contains tree structures created from a given list. In case the list is empty, the tree is just a leaf node. Otherwise, the tree is a parent node containing two child nodes. The lists the child nodes are defined by are precisely the lists that quicksort would recurse on given the list the parent node is defined by.

```
Inductive qs_tree : list nat -> Type :=
  | qs_tree_base : qs_tree nil
```

```
  | qs_tree_step : forall ( x: nat) (xs: list nat),
      qs_tree (filter (fun y => leb y  x) (xs))
    ->
        qs_tree (filter (fun y => negb (leb y x)) (xs))
      ->
        qs_tree (cons x xs).
```

The `qs_helper` function executes the actual sorting of the list. It takes the list to be sorted and its corresponding `qs_tree`. Note that a typical quicksort definition would look like this, minus the use of the `qs_tree`. This function recurses on the immediate substructures of the `qs_tree` (`q`), also refered to as the child nodes, which are `q0` and `q1`. Therefore, this definition is structurally recursive.

```
Fixpoint qs_helper (l : list nat) (q : qs_tree l) {struct q} :
  list nat :=
match q with
  | qs_tree_base => nil
  | qs_tree_step x xs q0 q1 =>
      (qs_helper (filter (fun y : nat => leb y x) xs) q0)
    ++
      (x :: (qs_helper (filter
        (fun y : nat => negb (leb y x)) xs) q1))
end.
```

The following lemma specifies that a `qs_tree` can be defined by any list (or made from any list).

```
Lemma create_qs_tree:
forall (l : list nat),
  (qs_tree l).
```

Then `quickSort` is defined using by passing the list to be sorted to `qs_helper` and the corresponding `qs_tree`.

```
Definition quickSort (ls :list nat) :=
  qs_helper ls (create_qs_tree ls).
```

The Bove-Capretta approach is a quite elegant solution to defining functions that are not structurally recursive. However, using Equations instead of the Bove-Capretta is arguably easier for the creator and easier to understand for a reader.

6.2 Quicksort on vectors by Sozeau

Matthieu Sozeau has implemented Quicksort on vectors using Equations [17]. This implementation was of great help to ours (see Section 4.6) and is

therefore quite similar. The main difference is that Sozeau has created the `pivot` function as a helper function for the quicksort implementation. The definitions of `All`, `Sorted`, and the definition of vectors are given in Section 4.5.

The `filter` function is quite similar to our implementation. The return type is a Sigma type and a predicate subtype, which we have seen in Sections 2.1.8 and 2.1.7, respectively.

```
Equations? filter {n} (v : vector A n) (f : A -> bool) :
  sig (k : nat), { v : vector A k | k <= n /\
  All (fun x => f x = true) v } :=
filter nil f := (0, nil);
filter (cons a v') f with dec (f a) :=
  { | left H => (_, cons a (filter v' f).2);
    | right H => (_, (filter v' f).2) }.
```

The `pivot` function divides the given vector into a vector containing the elements less or equal to the pivot and a vector containing the elements greater than the pivot. The comparison to the pivot is done using the function `f`, which is passed from the `qs` function as `fun x => leb x a` (returns true if $x <= a$). The return type of `pivot` specifies a couple properties useful for proving the correctness of the `qs` function.

```
Equations? pivot {n} (v : vector A n) (f : A -> bool) :
  sig (k : nat) (l : nat) (v' : vector A k), { w : vector A l
  | (k + l = n) % nat /\ forall x, In x v <->
  (if f x then In x v' else In x w) } :=
pivot nil f := (0 , 0 , nil, nil);
pivot (cons a v') f with dec (f a), pivot v' f :=
  { | left H | (k, l, v, w) => (_ , _, cons a v, w);
    | right H | (k, l, v, w) => (_ , _, v, cons a w) }.
```

The `qs` function retrieves the vectors `lower` and `higher` from the `pivot` function, which contain the elements from `l` that are less or equal to `a` (the pivot) and greater than `a`, respectively. The return type of `qs` is a predicate subtype, which we get into below. The `|:|` notation is synonymous to the `::` notation on lists.

```
Equations? qs {n} (l : vector A n) :
  { v : vector A n | sorted v /\ (forall x, In x l <-> In x v)}
  by wf n lt :=
qs nil := nil ;
qs (cons a v) with pivot v (fun x => leb x a) :=
  { | (k, l, lower, higher) =>
  app (qs lower) (a |:| qs higher) }.
```

The return type of `qs` specifies that the returned vector is sorted (`sorted v`). Unfortunately, the proof of this property uses a lemma that was not proven. On top of that, the concerning lemma is false:

```
Lemma Sorted_app {A R n m} (v : vector A n) (w : vector A m) :
    @Sorted A R _ v
  ->
      Sorted R w
    ->
      Sorted R (app v w).
Proof.
Admitted.
```

For clarity, we disprove the `Sorted_app` lemma using a counterexample. Take vectors [2] and $[1, 2]$, which both are sorted in non-decreasing order. Appending these vectors in their given order results in the vector $[2, 1, 2]$, which is not sorted in non-decreasing order. In our implementation of quicksort we created the `sort_app` lemma (see Section 4.6) which serves the same purpose as `Sorted_app` but includes the premises that the values in `v` should be less or equal to a set value and the values in `w` should be greater than that set value. Due to these additional conditions, the lemma does hold.

The return type of `qs` speciefies another property, namely `forall x, In x l <-> In x v`, where `l` is the input vector and `v` is the output vector. Unfortunately, this is a weaker property than `l` and `v` being permutations of each other, since it does not take duplicates into account. As an example, the property holds for vectors [1] and $[1, 1]$, but these vectors are not permutations of each other.

In conclusion, the proof of correctness of the implementation by Sozeau is incomplete. Both the proof that `qs` returns a sorted vector and the proof that the input and the output of `qs` are permutations of each other is incomplete.

6.3 Mergesort using Fixpoint definitions

The Coq.Sorting.Mergesort library [9] contains an implementation of mergesort using Fixpoint definitions. This implementation is very different from a typical mergesort implementation in a functional language, as seen in Section 4.3. This implementation keeps track of a stack of merges. This is done to work around the problem that mergesort is not structurally recursive. This implementation has a time complexity of $O(n\ log\ n)$, as mergesort should, which means that no compromise in time complexity was made. We present the implementation along with some explanation.

The `merge` function merges two given lists. Since the basic merge function is not structurally recursive, as we have seen in Section 4.3, the `merge_aux`

definition uses the `let fix ... in ...` construct as a workaround. This trick works by having `merge_aux` recurse on `l2'`, therefore having `l2'` as decreasing argument. We only do a recursive call to `merge` when `l1` decreases in size, making `l1` the decreasing argument of `merge`.

```
Fixpoint merge l1 l2 :=
let fix merge_aux l2 :=
  match l1, l2 with
    | [], _ => l2
    | _, [] => l1
    | a1::l1', a2::l2' =>
       if a1 <=? a2 then a1 :: merge l1' l2
       else a2 :: merge_aux l2'
  end
in merge_aux l2.
```

The `merge_list_to_stack` function takes the stack and the list for which a merge needs to be added to the stack. The Option type is used in a creative way to perform the correct merges. The insertion and removal of `None` objects on the stack ensure that two consecutive lists of the same size are merged.

```
Fixpoint merge_list_to_stack stack l :=
match stack with
  | [] => [Some l]
  | None :: stack' => Some l :: stack'
  | Some l' :: stack' =>
      None :: merge_list_to_stack stack' (merge l' l)
end.
```

The `merge_stack` function takes the stack and applies all merges in the stack, returning the resulting list.

```
Fixpoint merge_stack stack :=
match stack with
  | [] => []
  | None :: stack' => merge_stack stack'
  | Some l :: stack' => merge l (merge_stack stack')
end.
```

The `iter_merge` function takes the current stack and the list that is to be sorted. For each element in the given list it adds a merge to the stack, which it does by using the `merge_list_to_stack` function. Afterwards, the `merge_stack` function is called with the current stack.

```
Fixpoint iter_merge stack l :=
match l with
```

```
  | [] => merge_stack stack
  | a::l' => iter_merge (merge_list_to_stack stack [a]) l'
end.
```

A list `l` is sorted by calling `sort l`, which calls the `iter_merge` function with an empty stack.

```
Definition sort := iter_merge [].
```

We now describe this implementation in order of execution. When sorting a list using this mergesort implementation we keep track of a stack, which is a list of Option values. The stack is empty at the start. The `iter_merge` function then builds the stack by adding a merge to the stack for each element in the given list, it does so by using the `merge_list_to_stack` function. The merges are pairwise executed by this function. Once a merge is added to the stack for each element in the list, the function `merge_stack` is called with the current stack. The current stack may still hold some pending merges. This function then applies the possibly remaining merge and returns the resulting list, which is sorted.

Mergesort typically cuts the list in half, sorts each half recursively, and merges the resulting list. This implementation does not cut the list in half, but by merging the elements of the list pairwise in the `merge_list_to_stack` function it essentially performs the sorting in the same manner.

This implementation does not look like mergesort at first glance. Aside from the `merge` function, it differs completely from the typical functional implementation of mergesort. This is not the case for the implementation of mergesort using Equations given in Section 4.3, which furthermore is a more straightforward implementation. Creating this implementation seems more challenging than doing so by using Equations since creativity was necessary to come up with a way to implement this generatively recursive function.

6.4 Proving properties of functional programs

Using Coq is not the only way of proving properties of functional programs. We briefly introduce two proof assistants that achieve similar goals to Coq and are also interactive. Moreover, recall from Section 5 that the Haskell verifier LiquidHaskell allows for proving correctness properties of Haskell programs.

The proof assistant Sparkle [13] can verify the correctness of programs written in the lazy functional programming language Clean. The verification of correctness is accomplished by formal reasoning, which is a mathematical process that makes use of the semantics of the respective language.

Then there is the dependently typed programming language and proof assistant Agda [4]. The language has a syntax similar to Haskell. A big difference from Coq is that proofs are written in a functional programming style instead of using proof tactics.

Chapter 7

Conclusions

We have seen how we can use the Equations plugin to define functions that are not structurally recursive in Coq. Recall that this is done by proving that the function is well-founded, which ensures that it terminates. Specifically, we have implemented quicksort, mergesort, and binary search on lists and quicksort on vectors. We have seen that Equations allows for function definitions that have a similar syntax to Haskell, making them reasonably easy to create. However, we have seen that some definitions may have to be altered slightly to be able to prove that they terminate. We have looked into a couple of existing implementations of the functions mentioned and have seen that using Equations typically leads to a more straightforward implementation.

We have seen how LiquidHaskell can verify properties of Haskell programs. This can be less time consuming than creating a Coq proof. However, LiquidHaskell does have its limitations: only properties expressed in up to first-order logic can be proven using the underlying SMT-solver.

7.1 Future work

As we have seen, LiquidHaskell is able to verify the termination of a function given merely the decreasing argument, which we think would be a helpful feature for Equations. Furthermore, it would be interesting to investigate the real-world applicability of using Coq and Equations for verifying Haskell programs. This because translating Haskell programs to Coq may not preserve their semantics and extraction of Coq programs to Haskell results in programs that do not use Haskells own datatypes. As for LiquidHaskell, it would be interesting to investigate whether the permutation property is provable using the feature for theorem proving. We do expect this to be quite challenging since this feature is difficult to work with.

Bibliography

[1] Andrew W. Appel. Software foundations volume 3: Verified functional algorithms. `https://softwarefoundations.cis.upenn.edu/vfa-current/Merge.html`. Accessed: 03-01-2021.

[2] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming*, pages 114–129, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] Yves Bertot. Coq in a Hurry. Lecture, June 2016.

[4] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 73–78. Springer Berlin Heidelberg, 2009.

[5] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Seth Stein. *Introduction to Algorithms*. MIT Press, third edition, 07 2009.

[6] Gabe Dijkstra. Experimentation project report: Translating haskell programs to coq programs. 2012.

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press, 2018.

[8] Inria, CNRS, and contributors. The coq proof assistant. `https://coq.inria.fr/`. Accessed: 18-10-2020.

[9] Inria, CNRS, and contributors. Library coq.sorting.mergesort. `https://coq.inria.fr/stdlib/Coq.Sorting.Mergesort.html`. Accessed: 10-12-2020.

[10] Inria, CNRS, and contributors. Typing rules. https://coq.inria.fr/refman/language/cic.html. Accessed: 10-10-2020.

[11] Ranjit Jhala, Eric Seidel, and Niki Vazou. Programming with refinement types: An introduction to liquidhaskell (case study: Insertion sort). http://ucsd-progsys.github.io/lh-workshop/04-case-study-insertsort.html#/program-1. Accessed: 29-11-2020.

[12] Brian Min. Haskell language - merge sort. https://riptutorial.com/haskell/example/7552/merge-sort. Accessed: 07-11-2020.

[13] Maarten Mol, Marko Eekelen, and Rinus Plasmeijer. Proving properties of lazy functional programs with sparkle. pages 41–86, 06 2007.

[14] H. Riis Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[15] W.R.M. Schols. Capita selecta: Formal verification quicksort. 2012.

[16] Matthieu Sozeau. Library quicksort. https://www.irif.fr/~sozeau/repos/coq/misc/sort/quicksort.html. Accessed: 03-01-2021.

[17] Matthieu Sozeau. Typing rules. https://github.com/mattam82/Coq-Equations/blob/master/examples/quicksort.v. Accessed: 20-12-2020.

[18] Matthieu Sozeau. Programing finger trees in coq. *ACM SIGPLAN Notices*, 42:13–24, 09 2007.

[19] Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 419–434, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[20] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in coq. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[21] Niki Vazou and Ranjit Jhala. Putting things in order (sorting lists). https://ucsd-progsys.github.io/liquidhaskell-blog/2013/07/29/putting-things-in-order.lhs/. Accessed: 27-11-2020.