# Implementation of a Monadic Translation of Haskell Code to Coq

Justin Andresen

Bachelor's Thesis
September 2019

Programming Languages and Compiler Construction
Department of Computer Science
Kiel University

Advised by
Michael Hanus
Sandra Dylus

**Selbstständigkeitserklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

# Contents

Chapter 1

# Introduction

There is an always growing demand for software systems to be secure and reliable. While automatic tests help developers to discover and fix bugs during development, testing alone does not suffice to show that a program behaves as intended on all inputs. However, in many applications even small malfunctions can have severe consequences. Thus, the field of software verification is concerned with proving the correctness of programs with regard to their specification using rigorous methods known from mathematics and logic.

Compared to imperative programming languages, the high level of abstraction provided by purely functional programming languages, such as Haskell, makes it easier for their users to reason about code mathematically. Nevertheless, reasoning informally about complex software systems, and code in general, still remains a difficult task that is prone to errors. By contrast, interactive theorem provers, like Coq, allow their users to argue about a program's properties semi-automatically and with high confidence. In a so called specification language provided by the proof assistant, the user first writes a functional program and specifies properties that should be satisfied by the program. The interactive theorem prover assists the user in proving the specified properties. Usually, a verified program can be extracted in another programming langauge – e.g., Haskell – once the correctness of the program has been verified with respect to its specification. This approach is not applicable to existing code, though. Moreover, the formal verification of a program is much more difficult than well established testing and debugging practices. Therefore, it is also favorable to start the development of a new software product in a general purpose programming language rather than Coq or another proof assistant. Only once we have convinced ourselves that the program most likely behaves as intended, we want to attempt a proof of the most important properties. In order to reason about existing and newly developed Haskell programs in Coq, we need to convert Haskell code to the specification language used by Coq.

The `hs-to-coq` compiler developed by Spector-Zabusky et al. (2017) performs such a translation from Haskell to Coq. Their compiler is primarily designed to translate total Haskell programs. The restriction to total Haskell programs is inherited from Coq's requirement that all functions must be total. Unfortunately, most real Haskell programs are not total. In Agda – a proof assistant similar to Coq – the same limitation exists. Abel et al. (2005) presented an approach for the translation from Haskell to Agda that does not require the original Haskell programs to be total. The translation of partial functions is realized through a process known as *monadic translation*. The idea is to model partiality in the target language by the means of the `Maybe` monad explicitly. By performing such a monadic translation, Jessen (2019) implemented a prototype for a compiler from potentially partial Haskell programs to Coq. In this thesis we want to build a compiler for the monadic translation from Haskell to Coq based on the prototype by Jessen (2019). However, in addition to partiality, we would like to enable our model to represent other *ambient effects* (Christiansen et al., 2019) that can occur in Haskell

programs but not in Coq. We implement an approach presented by Dylus et al. (2018) to achieve an *effect-generic* monadic translation.

The remainder of this thesis is structured as follows. We begin in Chapter 2 with the preliminaries of this thesis. This includes among others an introduction to the Coq proof assistant and an explanation of the principles behind the effect-generic monadic translation. In Chapter 3 the translation rules for the conversion from Haskell to Coq are formalized and in Chapter 4 we present the actual implementation of our compiler. The translation rules and their implementation is evaluated by performing a case study in Chapter 5. The last chapter discusses and summarizes our implementation as well as the results of our case study. Furthermore, we provide an outlook on future improvements and extensions of our compiler.

Chapter 2

# Preliminaries

This chapter introduces fundamental concepts that are necessary to comprehend the following chapters. The first section gives a brief introduction to the Coq Proof Assistant based on the electronic textbook *Software Foundations* by Pierce et al. (2019) and highlights key differences between Haskell and Coq that need to be addressed by our compiler. Furthermore, the free monad and how it can be used to model Haskell programs is presented. Since basic knowledge of Haskell is assumed, no introduction to the Haskell programming language is given. However, a subset of Haskell, that we will focus on for the remainder of this thesis is presented in the third section of this chapter. Finally, the notation we are going to employ to describe the translation rules in Chapter 3 is outlined in the last section.

## 2.1 The Proof Assistant Coq

Coq is an interactive theorem prover that has been under development since 1983. Unlike automated theorem provers, Coq does not attempt to prove a proposition autonomously but in an interactive fashion with the help of human guidance. First, the user creates a model of the mathematical objects and algorithms they want to reason about in a functional programming language called *Gallina*. The same language is then used to state propositions about the functional program and construct proofs for these propositions. Coq offers so called *tactics* to automate some otherwise tedious tasks in the proof construction process. The correctness of the final proof is then checked by Coq. Hereinafter, we are going to use the term Coq to refer to both the proof assistant and the programming language. In the following subsections, the most important syntactic constructs of Coq's specification language used throughout this thesis are presented.

### 2.1.1 Types and Data Type Declarations

In contrast to many other programming languages, Coq has a minimal set of build-in data types. Data types like numbers and booleans – which are commonly directly build into a language – can be defined using the same mechanisms as for user-defined types. Custom types are declared using the `Inductive` keyword. The following example demonstrates the declaration of a boolean data type.

```
Inductive bool : Type
  := true
   | false.
```

In Coq, declarations are formed by so called *sentences* which start with a capitalized *command* and end with a period (The Coq Development Team, 2018, p. 30). The `Inductive` sentence, in the example above, declares a data type `bool` with two constructors `true` and `false`. Unlike in Haskell, there are no capitalization rules for identifiers. The colon in the first line annotates the type of `bool`. In this case, `bool` is of the build-in type `Type`. Similarly, we can use a colon to declare the type of the constructors. Both `true` and `false` are constants of the newly declared type `bool`.

```
Inductive bool : Type
  := true  : bool
  |  false : bool.
```

Natural numbers can be represented in Coq by the means of Peano numbers using the following `Inductive` sentence.

```
Inductive nat : Type
  := O           : nat
  |  S (n : nat) : nat.
```

This time, the constructor `S` takes an argument `n` of type `nat`. `(n : nat)` is called a binder for the argument `n`. Multiple arguments of the same type can be combined into a single binder. For example, `(n m : nat)` declares two arguments `n` and `m` of type `nat`. In constructor declarations we can also choose to declare the arguments anonymously. In this case, the constructor `S` must be annotated with a function type as shown below.

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

Even though, they are not directly build into the language, the types `bool` and `nat` are part of Coq's standard library and can be used in every Coq program.

Coq also supports the declaration of polymorphic types. Type arguments are passed like regular arguments, i.e., to declare a polymorphic list data type, we just have to add a binder for a type variable to the `Inductive` sentence.

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

Now `list` is a function that produces a list data type for the given value data type. In Coq, we can print the type of a term using the `Check` command.

```
Check list. (* ==> list : Type -> Type *)
```

However, the type argument `X` has been added to the constructors `nil` and `cons` as well.

```
Check nil.  (* ==> nil  : forall X : Type, list X *)
Check cons. (* ==> cons : forall X : Type, X -> list X -> list X *)
```

We can interpret types with `forall` quantifiers as function types which additionally give a name to their arguments. In other words, `nil` is a function which takes a type as its first argument `X`. The return type of that function depends on the type given as `nil`'s first argument. The term `nil nat` for example is of type `list nat`. Since the return types of functions can depend on their arguments, Coq is called a *dependently typed language*.

Coq also supports type inference. Thus, we do not have to pass the type arguments explicitly if it has been marked as an inferred argument, e.g., using an `Arguments` sentence. Arguments that should be inferred by Coq are simply wrapped in curly braces.

```
Arguments nil {X}.
Arguments cons {X}.
```

Now, the type of `nil` is `list ?X` where `?X` is a type inferred from the respective context. It is also possible to specify that the arguments bound by a binder should be inferred in the declaration itself. The following marks the type argument `X` as implicit for example.

```
Inductive list {X : Type} : Type
  := nil  : list X
  |  cons : X -> list X -> list X.
```

However, now `X` is not only an implicit argument of `nil` and `cons` but also of `list`. Therefore, `list` tries to infer the value type from the context which is not necessarily what we want.

### 2.1.2 Function Declarations and Expressions

In Coq functions are declared using `Definition` sentences. The `Definition` command is followed by the name of the function to declare, binders for its arguments, an annotation of the function's return type and finally a term on the right-hand side of the function declaration. For example, the following `Definition` sentence declares a function `null` that tests whether a list `xs` is empty or not.

```
Definition null {X : Type} (xs : list X) : bool :=
  match xs with
  | nil        => true
  | cons x xs' => false
  end.
```

On the right-hand side of `null` we are using a `match` expression to perform pattern matching. The `match` expression corresponds to a `case` expression in Haskell. However, Coq requires us to list all constructors of the matched data type. Therefore, the following declaration of a function that returns the first element of a list is not allowed in Coq since there is no alternative for the `nil` constructor.

```
Fail Definition head {X : Type} (xs : list X) : X :=
  match xs with
  | cons x xs' => x
  end.
(* ==> The command has indeed failed with message:
       Non exhaustive pattern-matching: no clause found for pattern nil *)
```

The `Fail` command checks that the subsequent declaration contains an error and then continues processing the Coq source file without declaring the function.

Further kinds of Coq expressions known from Haskell include **`if`** expressions, number literals and lambda abstractions. **`if`** expressions look just like in Haskell but since `bool` is not build directly into Coq, they work with any data type which has exactly two constructors (The Coq Development Team, 2018, p. 48). The following example demonstrates, that we can use values of type `nat` for the condition of **`if`** expressions. The first constructor `O` is handled like `true` and the second constructor `S` like `false`.

```
Compute if O then 1 else 2.   (* ==> 1 *)
Compute if S O then 1 else 2. (* ==> 2 *)
```

The literals `1` and `2` denote values of type `nat` and are short for `S O` and `S (S O)`, respectively.

Lambda abstractions are created with the **`fun`** keyword followed by binders for the arguments. The following lambda abstraction computes the successor of a natural number.

```
fun (n : nat) => S n
```

Similar to how we did not have to annotate the types of `x` and `xs'` in the pattern `cons x xs'` above, we do not have to annotate the type of the lambda abstraction's argument either.

```
fun n => S n
```

In this case, Coq infers the type of `n` from the right-hand side of the lambda abstraction.

One crucial difference between functions in Haskell and Coq is, that Coq distinguishes between non-recursive functions – like `null` – and recursive functions. Recursive functions must be declared with the **`Fixpoint`** command instead of the **`Definition`** command. The following function computes the length of a list for example.

```
Fixpoint length {X : Type} (xs : list X) : nat :=
  match xs with
  | nil        => O
  | cons x xs' => S (length xs')
  end.
```

Coq employs an additional check to verify that the function declarations declared by such **`Fixpoint`** sentences terminate on all inputs. More details on the Coq's termination analysis are given in Section 3.4.2.

### 2.1.3 Propositions and Proofs

Since Coq is a proof assistant, it provides a mechanism to denote propositions and their proofs. Propositions are given a name through `Theorem` sentences. For example, the following sentence declares a theorem called `null_length` that states that if a list is empty, its length is zero.

```
Theorem null_length: forall (X : Type) (xs : list X),
                     null xs = true -> length xs = 0.
Proof. (* ... *) Qed.
```

After the theorem has been declared, the user writes a proof script which constructs a proof for the theorem using the tactics provided by Coq. Since we will not write any proofs using Coq in this thesis, the proof script is left blank in the example above. However, one aspect that we have to understand about proofs in Coq is how Coq checks whether the proof constructed by the user actually proves the stated proposition.

Coq makes use of the so called *Curry-Howard isomorphism*, to check the validity of proofs (The Coq Development Team, 2018, p. 1). Basically, propositions are represented by types and values correspond to proofs. A proposition is true if and only if there is a value of the corresponding type. Therefore, Coq can employ its type checker to verify that the proof specified by the user is correct. For example, the propositions `True` and `False` are defined as follows in Coq.

```
Inductive False : Prop := .
Inductive True  : Prop := I : True.
```

The build-in type `Prop` denotes the type of propositions and is a subset of `Type`. Hence, `False` is a type with no constructor and `True` is a type with exactly one value denoted by the constant `I`. A theorem that states that the proposition `True` is true just needs to apply the constructor `I` to create a value of type `True` which serves as evidence, that the proposition holds.

```
Theorem true_is_true: True.
Proof. apply I. Qed.
```

On the other hand, since no value of type `False` exists, the proposition `False` does not hold which is exactly what we would expect.

Of course, the proof scripts of `null_length` and other real theorems and the proofs constructed by them are much more complicated than that. What may be confusing is that the proposition on the right-hand side of `null_length` does not even look like a type anymore since it contains values like `true` and `0`. That values occur within types is a common feature of dependently typed languages. In fact, there is no difference between types and terms in Coq. As we have seen already, types are passed to functions like regular arguments for example.

### 2.1.4 Differences between Haskell and Coq

Let's conclude this brief introduction to Coq by summarizing the key differences between Haskell and Coq. First of all, both Coq and Haskell are functional programming languages and share a lot of similarities as such. Additionally, Coq introduces dependent types which allow it to be used as a proof assistant. However, this extension requires Coq to enforce the following two conditions:

1. all pattern matching must be exhaustive and
2. all functions must provably terminate on all inputs.

In consequence, there is no partiality, no infinite data structures and Coq is not *Turing complete*. Satisfying the termination requirement of Coq presents one of the mayor challenges for the implementation of our compiler. The next section, presents an approach to model partiality in Coq.

## 2.2 Modeling Ambient Effects in Coq

Since Haskell is a purely functional programming language, computations are not allowed to have any implicit side-effects. Therefore, all interactions with the environment need to be performed explicitly within the `IO` monad. However, the evaluation of certain expressions can still implicitly cause an effect in Haskell. For example, the evaluation of a call to `error` or `undefined` causes the program to terminate immediately (Marlow, 2010, p. 16). Similarly, the `trace` function provided by the `Debug.Trace` module from Haskell's `base` library prints a message to the console for debugging purposes. Such *ambient effects* (Christiansen et al., 2019) are not supported by Coq. In this section we present the approach implemented by our compiler to model the ambient effects of translated Haskell programs in Coq.

### 2.2.1 Monadic Translation

Due to the totality requirement of Coq, the function `head` cannot be expressed directly within Coq. When we try to implement `head` in Coq, we get stuck in the case for the empty list as indicated by the three question marks in the left listing below.

```
head :: [a] -> a
head xs = case xs of
  []      -> undefined
  x : xs' -> x
```

```
Definition head (X : Type) (xs : list X) : X
  := match xs with
     | nil        => (* ??? *)
     | cons x xs' => x
     end.
```

However, the same idea used by Haskell to model side-effects using the `IO` monad can be applied to model partiality in Coq. Namely, we can use the `Maybe` monad, to express that the computation

of `head` does not return a value for some inputs. The `Maybe` data type is known as `option` in Coq. The constructors `Nothing` and `Just` from `Maybe` correspond to `option`'s constructors `None` and `Some`, respectively. By lifting the return value of `head` into the `Maybe` monad, we can now implement the function in Coq as well.

```
head :: [a] -> Maybe a
head xs = case xs of
  []      -> Nothing
  x : xs' -> Just x
```

```
Definition head (X : Type) (xs : list X) : option X
  := match xs with
     | nil        => None
     | cons x xs' => Some x
     end.
```

Lifting just the return value of functions is not enough, though. Due to Haskell's lazy evaluation strategy, the evaluation of the argument of a function can be effectful as well. Consider for instance a list of lists `xss` and the expression `head (head xss)` which computes the first element of the first list in `xss`. If `xss` is empty, the inner call to `head` fails. However, Haskell will not terminate until the evaluation of this inner call is requested by the evaluation of the `case` expression in the outer call. To model effects in arguments, the argument type needs to be lifted as well. In consequence, we have to consider the additional case that the given argument represents a runtime error, i.e., is `Nothing`.

```
head :: Maybe [a] -> Maybe a
head mxs = case mxs of
  Nothing        -> Nothing
  Just ([]     ) -> Nothing
  Just (x : xs') -> Just x
```

```
Definition head (X : Type) (mxs : option (list X))
  : option X
  := match mxs with
     | None              => None
     | Some (nil       ) => None
     | Some (cons x xs') => Some x
     end.
```

Since `Maybe` is a monad, we can use the bind operator instead. The bind operator unwraps the value stored within the monad, applies the given operation on the value and handles the absence of a value appropriately.

```
head :: Maybe [a] -> Maybe a
head mxs = mxs >>= \xs ->
  case xs of
    []      -> Nothing
    x : xs' -> Just x
```

```
Definition head (X : Type) (mxs : option (list X))
  : option X
  := mxs >>= (fun xs => match xs with
     | nil        => None
     | cons x xs' => Some x
     end).
```

Similarly, the values inside of the list can be effectful as well. For example, the call `head [undefined]` should return `Nothing` in the lifted variant. However, the return value of `head` is always wrapped by `Just` in the example above. Therefore, we have to remove the `Just` constructor and lift the list's value

type to the `Maybe` monad. While the type `Maybe [Maybe a]` would be sufficient to model a list like `[1, undefined, 2]`, it fails to represent `1 : undefined` since the second argument of `(:)` is not lifted to the `Maybe` monad. In consequence, we have to update the declaration of the list data type to lift all arguments of all constructors.

```
data List a
  = Nil
  | Cons (Maybe a) (Maybe (List a))
```

```
Inductive list (X : Type) : Type
  := nil  : list X
  |  cons : option X
            -> option (list X)
            -> list X.
```

Finally, the `head` function can now be implemented as follows.

```
head :: Maybe (List a) -> Maybe a
head mxs = mxs >>= \xs ->
  case xs of
    Nil          -> Nothing
    Cons mx mxs' -> mx
```

```
Definition head (X : Type) (mxs : option (list X))
  : option X
  := mxs >>= (fun xs => match xs with
     | nil          => None
     | cons mx mxs' => mx
     end).
```

The process of lifting argument and return types of functions as well as the arguments of constructors, as shown above, is known as a *monadic translation* of the program. Abel et al. (2005) present a monadic translation from Haskell programs to Agda. Based on their translation rules, Jessen (2019) developed a prototype for a compiler that monadically translates Haskell programs to Coq.

Note that except for the representation of `undefined` as `Nothing`, the definitions above are completely independent of the `Maybe` and `option` data types, i.e., we can simply swap the `Maybe` monad for another monad to model other effects. The implementation by Jessen (2019) supports the `Identity` monad, for example, for the translation of total programs. In Haskell we could now generalize our definitions to work with any monad by adding an additional type parameter. For instance, the listing below shows the list data type lifted to an arbitrary monad with type constructor `m`.

```
data List m a = Nil | Cons (m a) (m (List m a))
```

In Coq this generalization would allow us to proof properties of functions that hold regardless of the effect. However, it is not possible to simply add this type parameter to the list data type in Coq.

```
Fail Inductive list (M : Type -> Type) (X : Type) : Type
  := nil  : list M X
  |  cons : M X -> M (list M X) -> list M X.
```

As indicated by the `Fail` command, the definition above is rejected by Coq. Coq rejects the definition since it can be used to implement potentially non-terminating functions (Dylus et al., 2018, p. 6). If

the declaration was allowed, the type argument `M` could be instantiated with the following type, for example.

```
Definition NatFun (A : Type) : Type := A -> nat.
```

When `M` is instantiated with `NatFun`, the second argument of the `cons` constructor is a function of type `list NatFun nat -> nat`. We can now implement the following function which invokes the function inside of the list with the list itself.

```
Definition applyFun (xs : list NatFun nat) : nat
  := match xs with
     | nil          => 0
     | cons mx mxs' => mxs' xs.
```

Even though `applyFun` itself is not recursive, the evaluation of an expression like `applyFun (cons 0 applyFun)` does not terminate. Since Coq requires functions to terminate on all inputs, the definition of `list` with a type argument for `M` is not allowed. Therefore, Dylus et al. (2018) model monads using a combination the free monad and a container representation for functors. In the next two subsections, we give a brief introduction to the free monad and containers.

### 2.2.2 Free Monad

The free monad is a data type that turns any functor into a monad (Dylus et al., 2018, p. 7). In Haskell the free monad can be defined as follows.

```
data Free f a = Pure a | Impure (f (Free f a))
```

For any functor $f$, `Free` $f$ is a monad, i.e., the following definitions fulfill the monad laws.

```
return :: Functor f => a -> Free f a
return = Pure

(>>=) :: Functor f => Free f a -> (a -> Free f b) -> Free f b
(Pure x)    >>= f = f x
(Impure fx) >>= f = fmap (>>= f) fx
```

The free monad can be used to model, among others, the `Maybe` and `Identity` monads. If we instantiate the functor $f$ with a data type `Zero` that does not have any constructor, we cannot create a value of type `Free Zero a` using the `Impure` constructor (in a total setting). Therefore, only the `Pure` constructor remains and the data type behaves like `Identity`.

```
data Zero a {- polymorphic type without constructors -}
data Identity a = Identity a
```

Similarly, a data type `One`, with a single inhabitant, can be used to model the `Maybe` monad.

```
data One a = One
data Maybe a = Just a | Nothing
```

The `Pure` constructor corresponds to `Just` and `Impure One` corresponds to `Nothing`.

In the context of our monadic translation, the `Pure` constructor corresponds to an effect-free head normal form and values constructed with `Impure` are interpreted as the monad's effects (Christiansen et al., 2019, p. 127). In case of the `Identity` monad, there are no effects and in case of the `Maybe` monad, there is exactly one effect: the absence of a value.

Unfortunately, if we try to define `Free` in Coq, we run into the same problems as with `list` in the previous subsection.

```
Fail Inductive Free (F : Type -> Type) (A : Type) : Type
  := pure   : A -> Free F A
   | impure : F (Free F A) -> Free F A.
```

To eliminate the possibility of defining non-terminating functions using `Free`, the type argument `F` is replaced by a container data structure that models the original functor but does not permit types like `NatFun` which were used to implement non-terminating functions.

### 2.2.3 Containers

Containers provide an abstraction for modeling data types that store values (Dylus et al., 2018, p. 8). A list, for example, can be characterized through its length and a function that maps indices to values within the list. Similarly, a tree can be modeled by a function that labels its leafs. This time, the values are not identified by a single index but by a more complex *position* within the tree (e.g., the path to the leaf from the root). Just as the length of the list determines which indices are valid positions within the list, the exact *shape* of the tree determines the valid positions of leafs.

In general, a container is characterized by its *shape* and a function that maps *positions* within the shape to values stored by the container. We will denote the data type of the container's shape as *Shape*. The position data type *Pos* `s` depends on the concrete choice of a shape `s` : *Shape*. In case of lists, the shape is a natural number (i.e., the length of the list) and the position type is an interval of valid indices.

$$\begin{aligned} \mathit{Shape} &= \mathbb{N} \\ \mathit{Pos}(n) &= [0; n) \end{aligned}$$

We want to use containers to model functors that are passed into the free monad. For modeling the `Identity` and `Maybe` monads, we introduced the functors `Zero` and `One`, respectively. Since there are no constructors in case of the data type `Zero`, no values of type `Zero` $\tau$ exist for any type $\tau$. Hence, values of type `Zero` $\tau$ neither have a shape nor any positions where values of type $\tau$ could be stored.

```
Inductive Void : Type := (* non-polymorphic type without constructors *).
Definition Shape_Zero := Void.
Definition Pos_Zero (s : Shape_Zero) := Void.
```

The data type `One`, on the other hand, has exactly one constructor without any arguments. Thus, all values of type `One` $\tau$ have the same shape. In the following listing we are using the `unit` data type as defined in Coq's standard library. Since, the constructor of `One` has no arguments, there are again no positions inside of values of type `One` $\tau$.

```
Inductive unit : Type := tt : unit.
Definition Shape_One := unit.
Definition Pos_One (s : Shape_One) := Void.
```

Finally, we need a way to incorporate containers into the definition of `Free`. For this purpose, we replace the type argument `F` by two new type arguments for *Shape* and *Pos*, respectively. In the `impure` constructor instead of a data structure of type `F (Free F A)` we need the concrete shape `s` that characterizes the container and a function `pf` that maps positions to values inside of the container. In this case, the values inside of the container are free monads themselves.

```
Inductive Free (Shape : Type) (Pos : Shape -> Type) (A : Type) : Type
  := pure   : A -> Free Shape Pos A
   | impure : forall (s : Shape) (pf : Pos s -> Free Shape Pos A), Free Shape Pos A
```

This time, Coq does not reject the definition anymore. Therefore, we will use this definition of `Free` for the effect-generic monadic translation implemented by our compiler.

## 2.3 Notation

This section introduces notational conventions used throughout this thesis.

### 2.3.1 Notation for Translation Rules

The translation rules presented in Chapter 3 are based on the work by Abel et al. (2005). We are also going to adopt their notation and write

$$H^{\dagger} = G$$

to express that the Haskell language construct $H$ (e.g., a type, expression or declaration) should be converted to the corresponding Gallina language construct $G$ (e.g., a term or sentence).

### 2.3.2 Naming conventions

When formalizing the assumptions and translation rules, we make extensive use of meta-variables for identifiers, expressions and types. For the sake of readability, those meta-variables will not always

be introduced explicitly. Instead, we rely on naming conventions outlined in this section to determine what the meta-variables stand for. If there are multiple meta-variables of the same kind, we use a subscripted index.

The symbols $\tau$, $T$ and $\alpha$ are used as identifiers for type expressions, constructors and variables, respectively. Instead of $\tau$, we also use $\kappa$ for type expressions in some places. Data type and type synonym declarations will usually be named $D$ or $S$. Analogously $e$, $C$ and $x$ are used for expressions, constructors and variables. Alternatively, $y$ is used for variables as well. Function declarations are called $f$ by convention. We use $\circ$ for infix operators. A lower case $c$ is used for the name of a data constructor in Coq. The lower case $c$ should differ from the corresponding capital $C$ only in that its first letter is converted to lower case. E.g. if $C_i = \texttt{Foo}$ for some index $i$, then $c_i$ will be `foo`.

We are also using the meta-variables *Shape* and *Pos* for the identifiers of the arguments of the free monad. They are passed explicitly as parameters to generated Coq sentences.

### 2.3.3 Notation for renamed identifiers

Not all Haskell identifiers are valid Coq identifiers and need to be renamed if necessary. For example `with` could be used in Haskell as the name for a function or variable, but not in Coq as `with` is a keyword in Coq.

Similarly, Haskell allows types and constructors to have the same name because their namespaces are separated. Since Coq is a dependently typed language, types in Coq can contain values. Therefore, Coq constructors can conflict with types of the same name and need to be renamed as well.

Details on how identifiers are renamed will be given in Section 4.6.2. For the time being, we will simply write $I'$ for the renamed version of a Haskell identifier $I$. For short, we write $e'$ or $\tau'$ for an expression $e$ or type expressions $\tau$ in which all identifiers have been renamed appropriately.

## 2.4 Assumptions

Since it would be error-prone and infeasible to support the complete language specification of Haskell, we make assumptions about input modules that simplify the translation to Coq. Effectively, we specify a subset of Haskell that we are going to focus on for the remainder of this thesis. The language supported by our compiler is based on the Haskell 2010 Language Report (Marlow, 2010). As a general requirement, we specify that all input modules should be valid Haskell modules, i.e., the GHC should be able to compile the module successfully. However, just a small selection of expressions and declarations is actually permitted. There is neither support for language extensions nor type classes.

### 2.4.1 Data Type Declarations

For the definition of custom data types, we allow the usage of **`data`** declarations. No **`newtype`** declarations are supported. Furthermore, the constructors of the data type do not support record syntax.

Hence, a declaration of a data type $D$ with $n$ type arguments $\alpha_1$ through $\alpha_n$ and $m$ constructors $C_1$ through $C_m$ has the following form.

```
data D α1 ... αn =
    C1 τ1,1 ... τ1,p1
  | C2 τ2,1 ... τ2,p2
  | ...
  | Cm τm,1 ... τm,pm
```

Since type classes are not allowed, there is no `deriving` clause.

Constructors can be written in infix notation, but we do not allow the usage of symbolic names. For example, the left definition of a "rose tree" below is not allowed and must be declared as show to the right.

```
{- INALID -}
data Rose a = a :> [Rose a]
```

```
{- VALID -}
data Rose a = a `Rose` [Rose a]
```

### 2.4.2 Type Synonym Declarations

In addition to data type declarations, user-defined types can be introduced using `type` declarations. A declaration of a type synonym $S$ with $n$ type arguments $\alpha_1$ through $\alpha_n$ has the following form.

```
type S α1 ... αn = τ
```

### 2.4.3 Function Declarations

In Haskell functions are usually defined by performing pattern matching over their arguments. For example, boolean negation can be defined by two rules, each matching one of `Bool`'s constructors.

```
not True  = False
not False = True
```

For simplicity, we restrict pattern matching to the right-hand side of function declarations (see also Section 2.4.5). Therefore, all function declarations consist of exactly one rule and the arguments on the left-hand side are variable patterns.

```
f x1 ... xn = e
```

In the example above, `not` would have to be defined as follows.

```
not x = case x of
  True  -> False
  False -> True
```

Guards and local declarations, i.e., `where` clauses, are not permitted.

Functions can be written in infix notation, but we do not allow the definition of custom operators. For example, the left definition of list concatenation below is not allowed and must be declared as show to the right.

```
{- INVALID -}
(++) :: [a] -> [a] -> [a]
xs ++ ys = {- ... -}
```

```
{- VALID -}
append :: [a] -> [a] -> [a]
xs `append` ys = {- ... -}
```

In addition to *function bindings* – that were covered above, the Haskell Report defines so called *pattern bindings* (Marlow, 2010, p. 53). Pattern bindings are declarations that bind variables in patterns to values. For example, the following pattern binding binds the value `42` and `True` to the variables `x` and `y`, respectively.

```
(x, y) = (42, True)
```

Since we allow only explicit pattern matching on the right-hand side of function declarations, all pattern bindings have the following form, i.e., coincide with nullary function declarations.

$x$ `=` $e$

### 2.4.4 Type Signatures and Kinds

As performing type inference is beyond the scope of this thesis, we assume Haskell programs to be correctly typed. Nevertheless, type information is needed for the translation to Coq. Consequently, we require explicit type signatures for all function declarations to be present.

Similar to how type inference checks the types of functions and expressions, kind inference is a mechanism to check the validity of type expressions (Marlow, 2010, p. 37). A *kind* can be thought of as the type of a type. The kind of nullary type constructor is denoted $*$. A type constructor that takes a type argument of kind $\kappa_1$ and produces a type of kind $\kappa_2$ is denoted $\kappa_1$ `->` $\kappa_2$. Just like type inference, kind inference is beyond the scope of this thesis as well. However, the kinds of type variables are not explicitly annotated. Therefore, we assume all type variables to be of kind $*$. Consequently, all $n$-ary type constructors are of kind $\underbrace{* \texttt{ -> } \ldots \texttt{ -> } *}_{n\text{-times}} \texttt{ -> } *$.

For example, the following type synonym that introduces an alias for type constructor application is not allowed because the kind of the type variable `t1` is $* \texttt{ -> } *$.

```
{- INVALID -}
type App t1 t2 = t1 t2
```

This is not really a restriction, as in real applications the usage of type variables as type constructors is usually only useful in conjunction with type classes.

### 2.4.5 Pattern Matching

As mentioned above, pattern matching must be performed explicitly on the right-hand side of function declarations using `case` expressions. We further restrict patterns in `case` expressions to shallow constructor patterns, i.e., all patterns have the form $C\ x_1\ \ldots\ x_n$, where $C$ is an $n$-ary constructor and $x_1$ through $x_n$ are variable patterns. As we neither allow guards nor `where` clauses, `case` expressions have the following form.

```
case e of
  C1 x1,1 ... x1,p1 -> e1
  C2 x2,1 ... x2,p2 -> e2
  ...
  Cm xm,1 ... xm,pm -> em
```

Moreover, we assume pattern matching to be *exhaustive*. If $e$ is of type $D\ \tau_1\ \ldots\ \tau_n$, then $D$ must be a data type with exactly $m$ constructors.

```
data D α1 ... αn =
    C1 τ1,1 ... τ1,p1
  | C2 τ2,1 ... τ2,p2
  | ...
  | Cm τm,1 ... τm,pm
```

Runtime errors that occur due to non-exhaustive patterns must be modeled explicitly by invoking one of the predefined functions `undefined` or `error`.

```
head :: [a] -> a
head xs = case xs of
  []     -> error "head: empty list"
  x : xs' -> x
```

### 2.4.6 Expressions

In addition to `case` expressions, `if` expressions and lambda abstractions are supported. In case of lambda abstractions, the same restrictions regarding pattern matching apply as to function declarations, i.e., their arguments must be variable patterns. Therefore, lambda abstractions have the form `\`$x_1\ \ldots\ x_n$ `->` $e$.

There is no support for local declarations using `let` expressions.

### 2.4.7 Predefined operations and data types

Haskell's *Prelude* offers a rich set of predefined data types and operations. We do not aim to recreate the entire Prelude but still want to expose some commonly used functionalities. Table 2.1 lists all predefined data types and the corresponding constructors supported by our compiler.

**Table 2.1.** Predefined data types and their constructors.

| Name | Type | Constructors |
|---|---|---|
| unit | `()` | `()` |
| pairs | $(\tau_1, \tau_2)$ | `(,)` |
| lists | $[\tau]$ | `[]` and `(:)` |
| booleans | `Bool` | `True` and `False` |
| integers | `Integer` | |

We are especially interested in data types with their own syntax, namely, lists, pairs and the unit type. That these data types have their own notation can be seen as a strong indicator for their importance in actual Haskell code. Internally, we will not handle these types any different from user-defined types and constructors, i.e., the following identities hold.

$$[\tau] = [] \ \tau \qquad \forall \tau \text{ type expression}$$
$$(\tau_1, \tau_2) = (,) \ \tau_1 \ \tau_2 \qquad \forall \tau_1, \tau_2 \text{ type expression}$$

In addition to the constructors, there is a more concise notation for lists and pairs.

$$[e_1, e_2, \ldots, e_n] = e_1 : (e_2 : (\ldots : (e_n : []) \ldots)) \qquad \forall e_1, e_2, \ldots, e_n \text{ expression} \tag{2.4.1}$$
$$(e_1, e_2) = (,) \ e_1 \ e_2 \qquad \forall e_1, e_2 \text{ expression} \tag{2.4.2}$$

Furthermore, Table 2.1 lists `Bool` and `Integer` as predefined data types. In theory `Bool` and its constructors `True` and `False` could be defined by the user. However, `Bool` is important for the translation of `if` expressions. Due to the lack of type classes, `Integer` will be used as the type for all numeric literals, i.e., there are no fixed-precision integers of type `Int`. Decimal, hexadecimal and octal notation can be used.

The following commonly used operations for `Bool` and `Integer` are build into the compiler.

- **Arithmetic operations:** addition `(+)`, subtraction `(-)`, multiplication `(*)`, exponentiation `(^)` and negation (using the `negate` function or unary minus operator).
- **Boolean operations:** conjunction `(&&)` and disjunction `(||)`
- **Comparison of integers:** `(<=)`, `(<)`, `(==)`, `(/=)`, `(>=)`, `(>)`

Floating point numbers, other literals such as `String`s and tuples with more than two elements are not supported yet. Invocations of the `error` function are the only exception. Even though strings are otherwise not officially supported, the argument of `error` is allowed to be a string literal.

Apart from the predefined data types, constructors and operations mentioned in this section, all Haskell modules must be self contained. In particular, `import` declarations are not supported.

Chapter 3

# Translation Rules

This chapter deals with the specification and explanation of the translation rules that govern the generation of Coq code from Haskell modules. The translation rules lay the theoretical basis for the implementation of our compiler presented in Chapter 4. The first two sections cover the translation of type expressions and declarations while the next two sections addresses the translation of expressions and function declarations. The translation rules are based for the most part on work by Abel et al. (2005). In the final section, an extension to the translation is presented that allows for the generation of templates for Coq theorems from QuickCheck properties.

## 3.1 Type Expressions

A computation in Haskell does not necessarily produce a value but can also result in an effect, for example due to partiality. Coq on the other hand does not support such implicit effects. As discussed before, we bypass this restriction by modeling effects using the free monad in Coq. This fact needs to be reflected on type-level as well. Therefore, every Haskell type $\tau$ is lifted into the free monad during the translation.

$$\tau^\dagger = \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ \tau'$$

However, the translation rule above is incomplete when we take higher-order functions into account. Consider for example the following function declaration.

```
map :: (a -> b) -> [a] -> [b]
map f xs = {- ... -}
```

The parameter `xs` is of type $[a]$ and does not pose any problem. Assuming `List` is the Coq type constructor corresponding to `[]` (i.e., $[]' = \mathit{List}$), we can translate the type of `xs` as follows.

$$[a]^\dagger = \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ (\texttt{List}\ a)$$

Lifting the type of `xs` to the free monad allows an application of `map` to pass an effectful computation as the second argument. If we define the constructors of `List` appropriately (see Section 3.2), the list items can also have effects themselves. We do not have to lift the type argument $a$ explicitly.

The parameter `f` is different, though. While it is correct to lift the type of `f`, i.e., $a \to b$, into the free monad such that potential effects in the first argument of `map` can be handled, doing so is not

sufficient.

$$(a \to b)^\dagger = \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ (a \to b)$$

The argument in an application of `f` could have an effect as well and its result is also potentially effectful. In case of `xs`, the `List` constructors took care of lifting the item types. However, we do not provide a type constructor for `->` that does the same for functions. Therefore, we need to lift the argument and return type of `f` explicitly.

$$(a \to b)^\dagger = \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ (\texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ a \to \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ b)$$

In general, the argument and return types of all arbitrarily deeply nested function types in $\tau$ need to be lifted. For this purpose, we introduce another translation operation $\tau*$ as shown in Figure 3.1.

$$\tau^\dagger = \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ \tau* \qquad \forall \tau \text{ type expression} \tag{3.1.1}$$

$$(\tau_1 \to \tau_2)* = {\tau_1}^\dagger \to {\tau_2}^\dagger \qquad \forall \tau_1, \tau_2 \text{ type expression} \tag{3.1.2}$$

$$(\tau_1\ \tau_2)* = {\tau_1}*\ {\tau_2}* \qquad \forall \tau_1, \tau_2 \text{ type expression} \tag{3.1.3}$$

$$T* = T'\ \mathit{Shape}\ \mathit{Pos} \qquad \forall T \text{ type constructor} \tag{3.1.4}$$

$$\alpha* = \alpha' \qquad \forall \alpha \text{ type variable} \tag{3.1.5}$$

**Figure 3.1.** Final translation rules for type expressions.

The * operation recursively lifts the argument and return types of function types contained in $\tau$. Additionally, the parameters *Shape* and *Pos* are passed to type constructors. That is, in the example above, the type of `xs` would be translated as follows.

$$[a]^\dagger = \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ (\texttt{List}\ \mathit{Shape}\ \mathit{Pos}\ a)$$

`List` must be provided with *Shape* and *Pos* to lift the argument types of its constructors. Otherwise, list items would not be allowed to be effectful. The translation of data types and their constructors will be elaborated on in the next section.

## 3.2 Type Declarations

This section covers the translation of type synonyms and data type declarations.

Since the order of declarations matters in Coq, special care needs to be taken to translate only declarations whose dependencies were translated before. We say that a type expression $\tau$ depends on the declaration of a type constructor $T$ (e.g., the name of a type synonym or data type declaration) if $\tau$ contains an application of the type constructor $T$. For short, we also say that $\tau$ depends on or uses $T$. A type synonym now depends on all type constructor used by its right-hand side. Likewise, a data type declaration depends on all type constructors used by the fields of its data constructors.

For simplicity, we will first consider only individual declarations and assume all additional data

types used by our examples to have been declared beforehand. At the end of this section, we describe how to translate multiple type declarations without introducing dependency related issues to the generated Coq code. We also extend the translation rules to support mutually recursive declarations in the final subsection.

### 3.2.1 Type Synonym Declarations

Type synonyms can be used in Haskell to give a name to a more complex type expression. Similarly, we can use a `Definition`-sentence to assign a name to a term in Coq. As Coq does not distinguish types and terms, declarations of type synonyms can be simply translated to `Definition`-sentences as shown in Figure 3.2.

$$
\begin{aligned}
(\texttt{type}\ S\ \alpha_1\ \ldots\ \alpha_n\ \texttt{=}\ \tau)^{\dagger} = \texttt{Definition}\ S'\ &(\mathit{Shape}\ \texttt{:}\ \texttt{Type})\ (\mathit{Pos}\ \texttt{:}\ \mathit{Shape}\ \texttt{->}\ \texttt{Type}) \\
&(\alpha'_1\ \ldots\ \alpha'_n\ \texttt{:}\ \texttt{Type}) \\
&\texttt{:}\ \texttt{Type} \\
&\texttt{:=}\ \tau^{*}.
\end{aligned}
$$

**Figure 3.2.** Translation rule for $n$-ary polymorphic type synonym declarations.

The parameters *Shape* and *Pos* have to be added such that the translated type can use the free monad or other data types. For example, the following type synonym

```
type Queue a = List a
```

needs *Shape* and *Pos* when translated to Coq because they need to be passed to `List`.

```
Definition Queue (Shape : Type) (Pos : Shape -> Type) (a : Type) : Type
               := List Shape Pos a.
```

On the right-hand side of the `Definition`-sentence in Figure 3.2 the * translation is applied instead of $^{\dagger}$. We have to use * because we desire the expansion of type synonyms in Haskell and Coq to be compatible with respect to the $^{\dagger}$ translation. To convince ourselves that the translation rule above fulfills this property let's denote the expansion of type synonyms in Haskell and Coq with $\beta_H$ and $\beta_C$, respectively, and show that for all type synonyms `type` $S\ \alpha_1\ \ldots\ \alpha_n$ `=` $\tau$ and type expressions $\tau_1, \ldots \tau_n$ the following holds true.

$$\beta_H(S\ \tau_1\ \ldots\ \tau_n)^{\dagger} = \beta_C((S\ \tau_1\ \ldots\ \tau_n)^{\dagger})$$

For the instantiation of $S$ and $S'$, we first define substitutions $\sigma$ and $\sigma'$.

$$
\begin{aligned}
\sigma &:= \{\, \alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n \,\} \\
\sigma' &:= \{\, \alpha'_1 \mapsto {\tau_1}^{*}, \ldots, \alpha'_n \mapsto {\tau_n}^{*} \,\}
\end{aligned}
$$

Now we can prove the proposition by applying the translation rules for types presented in the previous section.

$$
\begin{aligned}
\beta_H(S\ \tau_1\ \ldots\ \tau_n)^\dagger &= \sigma(\tau)^\dagger && \text{(expansion of } S\text{)} \\
&= \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ \sigma(\tau)^* && \text{(Equation 3.1.1)} \\
&= \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ \sigma'(\tau^*) && (\star) \\
&= \beta_C(\texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ (S'\ \mathit{Shape}\ \mathit{Pos}\ \tau_1{}^*\ \ldots\ \tau_n{}^*)) && \text{(undo expansion of } S'\text{)} \\
&= \beta_C(\texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ (S\ \tau_1\ \ldots\ \tau_n)^*) && \text{(Equation 3.1.3 and 3.1.4)} \\
&= \beta_C((S\ \tau_1\ \ldots\ \tau_n)^\dagger) && \text{(Equation 3.1.1)}
\end{aligned}
$$

□

The equation marked with $(\star)$ follows from the following property of $\sigma$ and $\sigma'$.

$$\sigma(\alpha_i)^* = \tau_i{}^* = \sigma'(\alpha_i') = \sigma'(\alpha_i{}^*) \qquad \forall i \in \{1, \ldots, n\}$$

We assume without a proof that this property holds for the extensions of $\sigma$ and $\sigma'$ as well.

### 3.2.2 Data Type Declarations

So far we have described how to translate type expressions and assign names to them, but we are not able to actually define our own data types. One of the simplest kinds of data types that can be defined in Haskell are so called *sum types*. A well known example for a sum type is `Bool` which is commonly defined as follows.

```
data Bool = True | False
```

In general, the constructors of a sum type do not have any arguments and are enumerated on the right-hand side of a `data` type declaration. In Coq data types are defined using `Inductive`-sentences in a very similar fashion to `data` declarations in Haskell. The `Bool` type could look like this for example.

```
Inductive Bool : Type
  := true  : Bool
   | false : Bool.
```

Apart from the concrete syntax, the main difference is that the type of the constructors is annotated explicitly. In contrast to Haskell, we follow the Coq convention to choose lower case names for the constructors.

Sum types on their own are usually not very useful. Instead, one wishes to compose existing data types to more interesting data structures. Such types are referred to as *product types*. If there was for instance a type `Double` that represents floating point numbers, we could compose two such numbers to model a complex number.

```
data Complex = Complex Double Double
```

This time the constructor `Complex` is not a constant like `True` but a function that takes two values of type `Double` and produces a `Complex` number.

```
GHCi> :t Complex
Complex :: Double -> Double -> Complex
```

This insight is important, as we need to annotate the full type of the constructor in Coq again. By applying the translation rules for types we would obtain the following Coq type for the constructor.

$$\begin{aligned}(Double \to Double \to Complex)^* &= Double^\dagger \to (Double \to Complex)^\dagger && \text{(Equation 3.1.2)}\\ &= Double^\dagger \to \texttt{Free}\ Shape\ Pos\ (Double \to Complex)^* && \text{(Equation 3.1.1)}\end{aligned}$$

Wrapping the return type of a constructor is not allowed in Coq, though. However, it is not necessary for us to lift the intermediate results of the constructor either. Lifting them would be redundant as (partial) constructor applications never have an effect in Haskell. Therefore, it is sufficient to lift the argument types of the constructor (which could indeed be effectful). For this purpose, we still need to add the parameters *Shape* and *Pos* to the generated sentence.

```
Inductive Complex (Shape : Type) (Pos : Shape -> Type) : Type
  := complex : Free Shape Pos (Double Shape Pos)
            -> Free Shape Pos (Double Shape Pos)
            -> Complex Shape Pos.
```

Haskell allows us to use *parametric polymorphism* to generalize from this example and define pairs of two arbitrary types.

```
data Pair a b = Pair a b
```

Analogously to how we added the parameters *Shape* and *Pos* above, we can also introduce the type variables *a* and *b* in the header of the `Inductive`-sentence.

```
Inductive Pair (Shape : Type) (Pos : Shape -> Type) (a b : Type) : Type
  := pair : Free Shape Pos a
            -> Free Shape Pos b
            -> Pair Shape Pos a b.
```

We wrap the binders of *a* and *b* in regular parentheses, because we want to pass the type arguments explicitly to `Pair` when we instantiate the data type. However, Coq also adds all parameters of the `Inductive`-sentence to the constructors automatically, i.e., `pair` takes four additional parameters.

```
Check pair.
(* ==> pair : forall (Shape : Type) (Pos : Shape -> Type) (a b : Type),
              Free Shape Pos a -> Free Shape Pos b -> Pair Shape Pos a b *)
```

For convenience, we can "hide" those arguments using an `Arguments`-sentence.

```
Arguments pair {Shape} {Pos} {a} {b}.
```

Coq will then try to infer the values of *Shape*, *Pos*, $a$ and $b$ from the context as well as the types of the remaining arguments.

Both Haskell and Coq do not require us to distinguish sum and product types. For example, we can combine both concepts to define a data type for optional values.

```
data Maybe a = Nothing | Just a
```

As shown in Figure 3.3, an arbitrary data type can be translated similar to how we translated product types above. Just as with sum types, additional constructors are separated by a vertical bar.

$$\left(\begin{array}{l} \texttt{data}\ D\ \alpha_1\ \ldots\ \alpha_n \\ \quad = C_1\ \tau_{1,1}\ \ldots\ \tau_{1,p_1} \\ \quad \mid \ldots \\ \quad \mid C_m\ \tau_{m,1}\ \ldots\ \tau_{m,p_m} \end{array}\right)^{\dagger} = \begin{array}{l} \texttt{Inductive}\ D\ (\mathit{Shape}\ \texttt{:}\ \texttt{Type})\ (\mathit{Pos}\ \texttt{:}\ \mathit{Shape}\ \texttt{->}\ \texttt{Type}) \\ \qquad (\alpha'_1\ \ldots\ \alpha'_n\ \texttt{:}\ \texttt{Type})\ \texttt{:}\ \texttt{Type} \\ \quad \texttt{:=}\ c'_1\ \texttt{:}\ {\tau_{1,1}}^{\dagger}\ \texttt{->}\ \ldots\ \texttt{->}\ {\tau_{1,p_1}}^{\dagger}\ \texttt{->}\ (D\ \alpha_1\ \ldots\ \alpha_n)^{*} \\ \quad \mid \ldots \\ \quad \mid c'_m\ \texttt{:}\ {\tau_{m,1}}^{\dagger}\ \texttt{->}\ \ldots\ \texttt{->}\ {\tau_{m,p_m}}^{\dagger}\ \texttt{->}\ (D\ \alpha_1\ \ldots\ \alpha_n)^{*}. \end{array}$$

**Figure 3.3.** Translation rule for $n$-ary polymorphic data type declarations with $m$ constructors. In Coq we are using renamed lower case variations $c'_i$ of the Haskell constructors $C_i$.

If we apply this translation rule to our `Maybe` data type, we obtain the following `Inductive`-sentence for example.

```
Inductive Maybe (Shape : Type) (Pos : Shape -> Type) (a : Type) : Type
  := nothing : Maybe Shape Pos a
   | just    : Free Shape Pos a -> Maybe Shape Pos a.
```

Before we conclude this section, notice that we are using the * translation for the return type of constructors in Figure 3.3. We have to do so because $^{\dagger}$ would lift the type to the free monad but Coq does not allow us to wrap the return type of constructors as we pointed out above already. However, we are usually interested in lifted values. With that end in view, we define a function for each regular constructor that simply applies the constructor and lifts its result using the `pure` constructor of the free monad. We are going to refer to these functions as *smart constructors* hereinafter. The smart constructor for `just` would look as follows.

```
Definition Just (Shape : Type) (Pos : Shape -> Type) {a : Type}
                (x : Free Shape Pos a)
                : Free Shape Pos (Maybe Shape Pos a)
                := pure (just x).
```

This time we are using upper case identifiers again. Unlike with regular constructor, we do not have to generate an `Arguments`-sentence for the smart constructor. Generating `Arguments`-sentence is not necessary as we can mark the type argument as implicit directly in the `Definition`-sentence by wrapping its binder in curly braces. The general generation scheme for smart constructors and `Arguments`-sentences is summarized in Figure 3.4. In case of `Just`, we were able to manually choose appropriate names for the arguments of the smart constructor. However, when we generate those sentences, we need to generate *fresh identifiers* automatically. The generation of fresh identifiers will be covered in Section 4.6.2.

```
Arguments c' {Shape} {Pos} {α1} ... {αn}.

Definition C' (Shape : Type) (Pos : Shape -> Type) {α1 ... αn : Type}
           (x1 : τ1†) ... (xp : τp†)
           : (D α1 ... αn)†
           := pure (C x1 ... xp).
```

**Figure 3.4.** Additional sentences to generate for each data constructor $C\ \tau_1\ \ldots\ \tau_p$ of a data type $D\ \alpha_1\ \ldots\ \alpha_n$. $c'$ refers to the renamed lower case constructor name. $x_1, \ldots, x_p$ are fresh identifiers.

### 3.2.3 Handling Mutually Recursive Type Declarations

In Haskell the order of declarations does not matter. Therefore, `A` is allowed to use `B` in the following example, even though `B` is declared after `A`.

```
data A = A B
data B = B
```

In Coq, on the other hand, the order of declarations is relevant. If we convert the example to Coq, the declaration of `B` must precede the declaration of `A`.

```
Fail Inductive A (Shape : Type) (Pos : Shape -> Type) : Type := a : B† -> A*.
(* ==> The command has indeed failed with message:
       The reference B was not found in the current environment. *)
Inductive B (Shape : Type) (Pos : Shape -> Type) : Type := b : B*.
```

For this reason, we have to analyse the dependencies of all data type declarations in the Haskell module before we translate them to Coq. As a first approach, we can simply sort the declarations according to their dependencies. In the example above, we would obtain the list `[B, A]` of sorted data type declarations. Translating the declarations in that order, yields a valid Coq program.

```
Inductive B (Shape : Type) (Pos : Shape -> Type) : Type := b : B*.
Inductive A (Shape : Type) (Pos : Shape -> Type) : Type := a : B† -> A*.
```

This approach also suffices to translate simple *recursive data types* such as `List`.

```
data List a = Nil | Cons a (List a)
```

A data type is called recursive if values of that type are used to construct new values of the same type. As Coq allows `Inductive`-sentences to use the data type that is declared in the same sentence, we can apply the existing translation rule in such cases directly.

```
Inductive List (Shape : Type) (Pos : Shape -> Type) (a : Type) : Type
  := nil  : List Shape Pos a
   | cons : Free Shape Pos a -> Free Shape Pos (List Shape Pos a) -> List Shape Pos a.
```

However, the approach above fails in case of *mutually recursive data types*.

```
data Tree a = Tree a (Forest a)
data Forest a = Empty | NonEmpty (Tree a) (Forest a)
```

The declarations of `Tree` depends on the declaration of `Forest` and vice versa. Regardless of which data type we choose to translate first, Coq will reject the generated code as the other one will be undefined. As we have seen in case of ordinary recursive data types like `List`, Coq allows us to use the data type defined by an `Inductive`-sentence within that sentence. Thus, we need a mechanism to define both `Tree` and `Forest` in the same `Inductive`-sentence, such that they can use each other. Coq offers the `with` keyword for this purpose. The `with` keyword is used to concatenate multiple `Inductive`-sentences to a single sentence. In the example above, the following would work.

```
Inductive Tree (Shape : Type) (Pos : Shape -> Type) (a : Type) : Type
  := tree : Free Shape Pos a -> Forest Shape Pos a -> Tree Shape Pos a
with Forest (Shape : Type) (Pos : Shape -> Type) (a : Type) : Type
  := empty    : Forest Shape Pos a
  |  nonEmpty : Free Shape Pos (Tree Shape Pos a) -> Forest Shape Pos a.
```

The question that remains is how we can identify mutually recursive data type declarations. For this purpose, we represent the dependencies among the data types as a directed graph. The nodes of the *dependency graph* correspond to data type and type synonym declarations. There is an edge between the nodes of two declarations $D_1$ and $D_2$ if and only if $D_1$ depends on $D_2$. The dependency graphs for the data type declarations covered in this subsection so far are depicted in Figure 3.5.

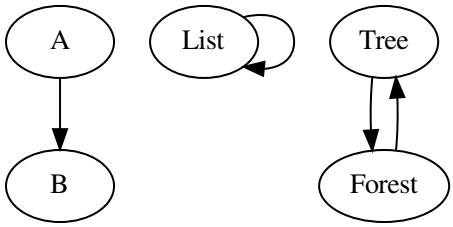



**Figure 3.5.** Dependency graphs for the data types `A` and `B`, `List` as well as `Tree` and `Forest` as defined above.

If there is an edge between $D_1$ and $D_2$, $D_2$ needs to be declared before or in the same sentence as $D_1$. The same is true if there is a path from $D_1$ to $D_2$. Hence, both must be declared in the same sentence if a path from $D_2$ to $D_1$ exists as well. In our concrete examples that means that `B` needs to be declared before `A` because there is an edge from `A` to `B`. It would also be okay to declare them in the same sentence. `Tree` and `Forest` on the other hand definitely need to be declared in the same sentence, as the corresponding nodes are reachable from each other.

In graph theory a set of nodes with the property that all nodes within the set are reachable from all other nodes within the set is referred to as a *strongly connected component* of the graph. In our case that means that all declarations whose nodes form a strongly connected component of the dependency graph need to be declared in the same sentence. Likewise, if two declarations depend on each other – and therefore need to be declared in the same sentence – they are also necessarily in the same strongly connected component. Thus, the identification of mutually recursive data type declarations coincides with the identification of strongly connected components of the dependency graph. There are algorithms to compute strongly connected components in linear time. As a final step, the identified strongly connected components need to be sorted. If the component $SCC_1$ contains at least one declaration that depends on a declaration in $SCC_2$, $SCC_2$ should precede $SCC_1$. This order also is referred to as *reverse topological order* and can be solved in linear time as well. In our case we obtain the following sorted lists of strongly connected components.

$$[\{\texttt{B}\}, \{\texttt{A}\}] \qquad [\{\texttt{List}\}] \qquad [\{\texttt{Tree}, \texttt{Forest}\}]$$

There is one additional caveat to be aware of. Until now, we only considered mutually recursive data type declarations. We did not consider mutually recursive type synonym declarations because type synonym declarations are not allowed to form a cycle on their own. Thus, the following is invalid in Haskell.

```
type A = B
type B = A
```

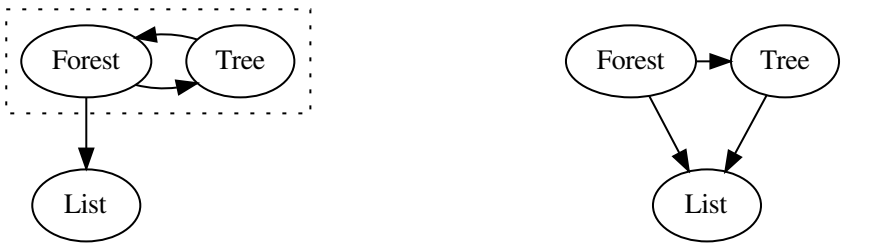
However, it is perfectly fine for a type synonym declaration to be part of a cycle formed by data type declarations. For example, one may notice that the `Forest` data type we defined earlier, can also be expressed in terms of `List`.

```
data Tree a = Tree a (Forest a)
type Forest a = List (Tree a)
```

Again, `Tree` and `Forest` form a strongly connected component of the dependency graph. Unlike before, we cannot use the `with` keyword in this case. `Tree` is translated to an `Inductive`-sentence while `Forest` is translated to a `Definition`-sentence. Unfortunately, `with` can only be used to join two or more `Inductive`-sentences. The solution is to expand type synonyms that occur within such a strongly connected component. After this preprocessing step, we obtain the following Haskell declarations.

```
data Tree a = Tree a (List (Tree a))
type Forest a = List (Tree a)
```

`Forest` still depends on `Tree` and needs to be translated afterwards. However, `Tree` can now be translated independently as can be seen in Figure 3.6.




[{ `List` }, { `Tree`, `Forest` }] [{ `List` }, { `Tree` }, { `Forest` }]

**Figure 3.6.** Dependency graphs for `Tree` and `Forest` before inlining the type synonym `Forest` into the definition of `Tree` (left) and after inlining (right). Strongly connected components are listed in reverse topological order below the graphs.

If there are more than one type synonyms in the strongly connected component, we need to sort them with respect to their dependencies among each other. But as pointed out above, we can be sure in those cases that they do not form a cycle themselves. Therefore, it is fine to translate them individually as we did before.

## 3.3 Expressions

Starting with this section we direct our attention away from type-level language constructs. Instead, we will have a look at the translation of expressions. Nevertheless, we have to keep the translation rules for types in mind. We have to do so because we wish the translation of expressions to be

compatible with the translation of expressions. In other words, when an expression $e$ of type $\tau$ is translated to Coq, the resulting term $e^\dagger$ must be of type $\tau^\dagger$. When designing the translation rules for expressions, we have to make sure that this *invariant* is preserved at all times.

$$e :: \tau \implies e^\dagger : \tau^\dagger \qquad \text{(invariant)}$$

### 3.3.1 Variables

In the simplest case, the expression is just a variable $x$. As mentioned before, when translating $x$ to Coq, we may have to rename $x$ to avoid name conflicts.

$$x^\dagger = x'$$

If we design the translation rules that are concerned with binding variables correctly, the invariant holds automatically. For example, when we perform pattern matching on a list (as defined in the previous section) the pattern `cons x xs` binds the variables `x` and `xs` to values of type $\mathtt{a}^\dagger$ and $(\mathtt{List\ a})^\dagger$, respectively. However, as we will see later, there are some rare cases in which we need unlifted variables, i.e., variables of type $\tau*$. In this case, we have to manually lift $x'$ to restore the invariant. Thus, the final translation rule for variables are as shown in Figure 3.7.

$$x^\dagger = \begin{cases} x' & \text{if } x' : \tau^\dagger \\ \mathtt{pure}\ x' & \text{if } x' : \tau* \end{cases}$$

**Figure 3.7.** The translation of a variable $x$ depends on the type $x'$ has been bound to in Coq.

### 3.3.2 Function Applications

In general, a function application has the form $e_1\ e_2$ in Haskell where $e_1$ is of some function type $\tau_1 \to \tau_2$ and $e_2$ is of type $\tau_1$. The application of $e_1$ to $e_2$ yields a value of type $\tau_2$. If we now translate both expressions to Coq, the invariant guarantees us to obtain lifted values.

$$\begin{aligned} &{e_1}^\dagger : (\tau_1 \to \tau_2)^\dagger \implies {e_1}^\dagger : \mathtt{Free}\ \mathit{Shape}\ \mathit{Pos}\ ({\tau_1}^\dagger \to {\tau_2}^\dagger) \\ &{e_2}^\dagger : {\tau_1}^\dagger \end{aligned}$$

If we had a value $f$ of type ${\tau_1}^\dagger \to {\tau_2}^\dagger$, we could apply $f$ to ${e_2}^\dagger$ to obtain a value of type ${\tau_2}^\dagger$ which would satisfy the invariant. As ${e_1}^\dagger$ is lifted to the free monad, we cannot access the contained function directly but have to use the bind operator to unwrap the function as shown in Figure 3.8.

$$(e_1\ e_2)^\dagger = {e_1}^\dagger \mathtt{>>=} (\mathtt{fun}\ f \Rightarrow f\ e_2)$$

**Figure 3.8.** General translation rule for function applications. More specific rules will be introduced for the application of constructors and defined functions below.

Usually, functions do not take just a single argument, though. For example, consider the application of a function $f$ to $n$ arguments.

$$\begin{aligned}(f\ e_1\ \ldots\ e_n)^\dagger = f^\dagger\ &\texttt{>>= (fun } f_1\ \texttt{=>}\ f_1\ {e_1}^\dagger\texttt{)}\\ &\vdots\\ &\texttt{>>= (fun } f_n\ \texttt{=>}\ f_n\ {e_n}^\dagger\texttt{)}\end{aligned}$$

As this example suggests, the rule above will cause the generation of code that is long and difficult to comprehend. Moreover, binding the intermediate results is often superfluous because we know that the partial application of a constructor or defined function will never have an effect in Haskell. For this reason, Abel et al. (2005) suggest an optimization that allows the generation of more readable code if a constructor or defined function is fully applied.

$$\begin{aligned}(C\ e_1\ \ldots\ e_n)^\dagger &= C'\ \mathit{Shape}\ \mathit{Pos}\ {e_1}^\dagger\ \ldots\ {e_n}^\dagger\\ (f\ e_1\ \ldots\ e_n)^\dagger &= f'\ \mathit{Shape}\ \mathit{Pos}\ {e_1}^\dagger\ \ldots\ {e_n}^\dagger\end{aligned}$$

**Figure 3.9.** Translation rule for the full application of a $n$-ary constructor $C$ or defined function $f$.

We actually applied this optimization already during the generation of smart constructors. The constructor `Just` of the `Maybe` data type has the type $a \to \texttt{Maybe}\ a$ for example. According to the invariant, the expression `Just` should now be translated to a Coq term of type

$$(a \to \texttt{Maybe}\ a)^\dagger = \texttt{Free}\ \mathit{Shape}\ \mathit{Pos}\ (a \to \texttt{Maybe}\ a)^*$$

However, the smart constructor `Just` has never been lifted to the free monad. Only its argument and return types are monadic.

```
Check Just.
(* ==> Just : forall (Shape : Type) (Pos : Shape -> Type) (a : Type),
              Free Shape Pos a -> Free Shape Pos (Maybe Shape Pos a) *)
```

Therefore, the application of `Just` does not involve the bind operator as shown in Figure 3.9. We will apply the same optimization when we translate function declarations in the final section of this chapter.

The only drawback is, that the optimization does not allow partial function applications so far. As Abel et al. (2005) point out, this problem can easily be solved by performing $\eta$-abstractions until all function applications are fully applied. An individual $\eta$-abstraction transforms a function $f$ to an equivalent function by wrapping $f$ with a lambda abstraction and and passing the argument to $f$.

$$f \underset{\eta}{\to} \lambda x.f\ x$$

The full process is depicted in Figure 3.10. The right-hand side of both equations requires us to be able to convert lambda abstractions to Coq. But before we examine the translation of lambda abstractions next, let's first look into another kind of function application: operator applications.

$$(C\ e_1\ \ldots\ e_m)^\dagger = (\backslash x_1\ \texttt{->}\ \ldots\ \backslash x_{m-n}\ \texttt{->}\ C\ e_1\ \ldots\ e_m\ x_1\ \ldots\ x_{n-m})^\dagger$$
$$(f\ e_1\ \ldots\ e_m)^\dagger = (\backslash x_1\ \texttt{->}\ \ldots\ \backslash x_{m-n}\ \texttt{->}\ f\ e_1\ \ldots\ e_m\ x_1\ \ldots\ x_{n-m})^\dagger$$

**Figure 3.10.** Translation rule for the partial application of a $n$-ary constructor $C$ or defined function $f$ to $m < n$ arguments.

Most binary operations such as `(+)` or `(==)` as well as the list constructor `(:)` are usually written in infix rather than prefix notation. It is also possible to write custom functions in infix notation by surrounding the function name in grave accents ("backticks" or "backquotes"). For example, instead of `div` $x$ $y$ one can write $x$ `` `div` `` $y$. When used properly, this notation can often increase the readability of programs. Similarly, there is a unary minus prefix operator that can be used as *syntactic sugar* for the application of the `negate` function from Haskell's Prelude. As Figure 3.11 demonstrates, the translation of operator applications is straightforward.

$$(e_1 \circ e_2)^\dagger = (\texttt{(}\circ\texttt{)}\ e_1\ e_2)^\dagger$$
$$(\texttt{-}e)^\dagger = (\texttt{negate}\ e)^\dagger$$

**Figure 3.11.** Translation rules for operator applications based on the identities listed by the Haskell Report (Marlow, 2010, p. 18).

However, in Haskell it is also possible to apply operators only partially. So called *sections* are written as `(`$e_1$ $\circ$`)` ("left section") or `(`$\circ$ $e_2$`)` ("right section"). A left section coincides with a regular partial application of the operator. A right section on the other hand cannot be expressed in this way. Therefore, we need to introduce the missing left argument in a similar way to an $\eta$-abstraction.

$$\begin{aligned}(e_1 \circ)^\dagger &= (\texttt{(}\circ\texttt{)}\ e_1)^\dagger \\ &\underset{\eta}{\rightarrow} (\backslash x_2\ \texttt{->}\ \texttt{(}\circ\texttt{)}\ e_1\ x_2)^\dagger \\ (\circ\ e_2)^\dagger &= (\backslash x_1\ \texttt{->}\ \texttt{(}\circ\texttt{)}\ x_1\ e_2)^\dagger\end{aligned}$$

**Figure 3.12.** Translation rules for left and right sections based on the identities listed by the Haskell Report (Marlow, 2010, p. 19).

### 3.3.3 Lambda Abstractions

Let's first consider a lambda abstraction `\`$x$ `->` $e$ with a single argument. Such an unary lambda abstraction can be represented in Coq by the term `fun` $x'$ `=>` $e^\dagger$. However, we still need to wrap the

function with the `pure` constructor of the free monad to make the invariant hold.

$$(\backslash x \texttt{ -> } e)^\dagger = \texttt{pure (fun } x' \texttt{ => } e^\dagger\texttt{)}$$

If we now try to extend this rule to lambda abstractions $\backslash x_1 \ \ldots \ x_n$ `->` $e$ with multiple arguments, we cannot simply add more binders to the Coq function. The following would be valid Coq, but violates our invariant.

$$\texttt{pure (fun } x'_1 \ \ldots \ x'_n \texttt{ => } e^\dagger\texttt{)}$$

The term above has the type

$$\texttt{Free } \mathit{Shape}\ \mathit{Pos}\ ({\tau_1}^\dagger \to \ldots \to {\tau_n}^\dagger \to \tau^\dagger)$$

but should have the type

$$\texttt{Free } \mathit{Shape}\ \mathit{Pos}\ ({\tau_1}^\dagger \to \texttt{Free } \mathit{Shape}\ \mathit{Pos}\ (\ldots \to \texttt{Free } \mathit{Shape}\ \mathit{Pos}\ ({\tau_n}^\dagger \to \tau^\dagger)\ldots))$$

according to our invariant. Therefore, we have to wrap all intermediate results with `pure`. We can do so by rewriting a lambda abstraction with multiple arguments to multiple nested unary lambda abstractions as shown in Figure 3.13.

$$(\backslash x \texttt{ -> } e)^\dagger = \texttt{pure (fun } x' \texttt{ => } e^\dagger\texttt{)}$$
$$(\backslash x_1 \ \ldots \ x_n \texttt{ -> } e)^\dagger = (\backslash x_1 \texttt{ -> } \ldots \texttt{ -> } \backslash x_n \texttt{ -> } e)^\dagger \qquad (\text{if } n > 0)$$

**Figure 3.13.** Translation rules for lambda abstractions.

### 3.3.4 `case` and `if` Expressions

In this subsection we cover both the translation of `case` and `if` expressions. Since both kinds of expressions behave very similar, their translation rules are based on the same key ideas. In fact, the Haskell Language Report defines the semantics of `if` expressions in terms of `case` expressions (Marlow, 2010, p. 20).

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 = \texttt{case } e_1 \texttt{ of \{ True -> } e_2 \texttt{ ; False -> } e_3 \texttt{ \}}$$

For simplicity, let's first focus on the translation of `if` expressions.

Unlike before, we cannot base our translation rule for `if` expressions on the invariant. If the expression `if` $e_1$ `then` $e_2$ `else` $e_3$ has type $\tau$, both $e_2$ and $e_3$ must also have type $\tau$. If we naïvely translated the expression by just translating $e_1$ through $e_3$, the resulting term

$$\texttt{if } {e_1}^\dagger \texttt{ then } {e_2}^\dagger \texttt{ else } {e_3}^\dagger$$

satisfies our invariant because ${e_2}^\dagger$ and ${e_3}^\dagger$ both have the type $\tau^\dagger$ and therefore the entire term has that type. Interestingly, if we applied this naïve translation rule, the generated code works. However, the generated code does not behave as intended.

```
Compute (if True†      then e1† else e2†). (* ==> e1† *)
Compute (if False†     then e1† else e2†). (* ==> e1† *)
Compute (if undefined† then e1† else e2†). (* ==> e2† *)
```

This odd behaviour results from the flexibility of `if` expressions in Coq. Coq does not have a build-in boolean type. Therefore, the condition of `if` expressions can be of any type with exactly two constructors (The Coq Development Team, 2018, p. 48). If the condition matches the first constructor, the `then` branch is selected and the `else` branch otherwise. Since the type of $e_1{}^\dagger$ is `Free` *Shape Pos* `(Bool` *Shape Pos*`)` and `Free` has exactly two constructors `pure` and `impure`, $e_1{}^\dagger$ can be used in `if` expressions. The semantics of Coq's `if` expressions fit the observation in the listing above. `True` and `False` are values without an effect. Therefore, their underlying representation uses the `pure` constructor which is the first constructor of `Free`. `undefined` on the other hand does have an effect. The usage of the `impure` constructor for the representation of effects leads to the `else` branch being selected if the condition is `undefined`.

What we actually wanted is to test whether the value wrapped by the `pure` constructor is `true` or `false`. Similar to how we had to unwrap a function expression before we could apply the actual function, we have to bind the condition this time as well. The bind operation corresponds to the fact that Haskell evaluates the condition to head normal form (Christiansen et al., 2019, p. 132) before selecting the `then` or `else` branch.

$$(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)^\dagger = e_1{}^\dagger\ \texttt{>>=}\ (\texttt{fun}\ (x\ \texttt{:}\ \texttt{Bool}^\dagger)\ \texttt{=>}\ \texttt{if}\ x\ \texttt{then}\ e_2{}^\dagger\ \texttt{else}\ e_3{}^\dagger)$$

**Figure 3.14.** Translation rule for `if` expressions.

With the translation rule shown in Figure 3.14 we get the expected behaviour.

```
Compute (True†      >>= (fun (x : Bool†) => if x then e1† else e2†)). (* ==> e1† *)
Compute (False†     >>= (fun (x : Bool†) => if x then e1† else e2†)). (* ==> e2† *)
Compute (undefined† >>= (fun (x : Bool†) => if x then e1† else e2†)). (* ==> undefined† *)
```

Note that we have to annotate the type of the fresh variable $x$ in Figure 3.14 and in the examples above explicitly. The type has to be annotated due to the flexibility of `if` expressions in Coq. While we know in Haskell that $x$ in `if` $x$ `then` $e_1$ `else` $e_2$ has to be of type `Bool`, $x$ could be of any type with exactly two constructors in Coq. Therefore, Coq could not infer the type of the fresh variable introduced during the translation of an expression like `if undefined then` $e_1$ `else` $e_2$.

```
Fail Compute (undefined† >>= (fun x => if x then e1† else e2†)).
(* ==> The command has indeed failed with message:
       Cannot infer a type for this expression. *)
```

We can apply the main idea from the translation of `if` expressions to the translation of `case` expressions. Instead of performing pattern matching on the free monad, we want to inspect the actual value. Thus, we have to add the bind operator. This time we do not have to annotate the type explicitly because Coq should be able to infer the type from the constructors in the patterns. Aside from that, the differences between pattern matching in Haskell and Coq shown in Figure 3.15 are mostly of syntactical nature. The observed similarity stems from our initial assumptions. If we allowed guards or nested patterns for example, the translation of `case` expressions would be much more difficult.

$$\left(\begin{array}{l}\texttt{case } e \texttt{ of} \\ \quad C_1\ x_{1,1}\ \ldots\ x_{1,p_1}\ \texttt{->}\ e_1 \\ \quad \vdots \\ \quad C_n\ x_{n,1}\ \ldots\ x_{n,p_n}\ \texttt{->}\ e_n\end{array}\right)^{\dagger} = \begin{array}{l}e^{\dagger}\ \texttt{>>=}\ \texttt{(fun } x\ \texttt{=> match } x\ \texttt{with} \\ \quad \mid c'_1\ x'_{1,1}\ \ldots\ x'_{1,p_1}\ \texttt{=>}\ {e_1}^{\dagger} \\ \quad \mid \vdots \\ \quad \mid c'_n\ x'_{n,1}\ \ldots\ x'_{n,p_n}\ \texttt{=>}\ {e_n}^{\dagger} \\ \texttt{end)}\end{array}$$

**Figure 3.15.** Translation rule for a `case` expression that performs pattern matching on an expression $e$ whose type has $n$ constructors $C_1$ through $C_n$. Note that we match the unwrapped Coq term with the regular constructors $c'_i$, not the smart constructors $C'_i$.

The requirement of Coq that pattern matching must be total is also simply passed down to the user of our compiler. Therefore, pattern matching failures must be handled explicitly using error terms. The translation of said error terms to Coq is described next.

### 3.3.5 Error Terms

So far we have seen no way to introduce actual effects into our programs. As noted earlier, one important effect that we would like to support is partiality. In Haskell partiality usually arises from the usage of incomplete pattern matching. Due to our assumption that pattern matching is total, we have to handle such cases explicitly.

```
head :: [a] -> a
head xs = case xs of []     -> error "head: empty list"
                     x : xs -> x
```

In general, Haskell provides two functions `error` and `undefined` that can be used to deliberately cause runtime errors. If an application of one of those functions is evaluated, the program terminates immediately (Marlow, 2010, p. 16). Additionally, a brief description for the cause of the error can be passed to `error` as demonstrated in the example above already.

```
error :: String -> a
undefined :: a
```

The question is now how we can implement those two functions in Coq. The authors of `hs-to-coq` for example do so by introducing a polymorphic value (or "axiom") `patternFailure` (Spector-Zabusky et al., 2017, p. 10). However, as they point out themselves, this axiom is inherently unsound, i.e., can be used to prove arbitrary propositions.

```
Local Axiom patternFailure : forall {a}, a.

Theorem contradiction: False.
Proof. apply patternFailure. Qed.
```

For this reason, avoiding the introduction of such an axiom has been the very motivation for the monadic transformation performed by our implementation. If we lifted our programs using the `Maybe` monad for example, we do not need the axiom above, but can represent `undefined` with `Nothing`.

```
undefined :: Maybe a
undefined = Nothing
```

For the translation of `undefined`, we now just need to model the `Maybe` monad using the free monad, i.e., instantiate *Shape* and *Pos* appropriately. How *Shape* and *Pos* would have to look like in case of `Maybe`, has been shown in Section 2.2.3 already.

We can also use `Nothing` to implement `error` if we are willing to simply discard the error message.

```
error :: String -> Maybe a
error msg = Nothing
```

However, if we want to keep the error message, we need a different monad, i.e., have to instantiate *Shape* and *Pos* differently. This alternative monad would not be limited to the translation of `error` as we can express `undefined` in terms of `error`.

```
undefined :: a
undefined = error "undefined"
```

As we want to leave the choice of the concrete monad to the user, we abstract from the instantiation of *Shape* and *Pos* using a type class. Instances of the type class contain the implementation of `undefined` and `error` for the monad represented by *Shape* and *Pos*.

```
Require Import Coq.Strings.String.

Class Partial (Shape : Type) (Pos : Shape -> Type) :=
  {
    undefined : forall {A : Type}, Free Shape Pos A;
    error     : forall {A : Type}, string -> Free Shape Pos A
  }.
```

The code generated by the rules in Figure 3.16 uses the interface of `Partial`. Therefore, the generated code works with any monad for which such an instance can be defined. For example, we can define a `Partial` instance for the `Maybe` monad by implementing both `error` and `undefined` with `Nothing`. In contrast, no `Partial` instance can be specified for the `Identity` monad.

$$\texttt{undefined}^{\dagger} = \texttt{undefined}$$
$$(\texttt{error}\ \mathit{msg})^{\dagger} = \texttt{error}\ \mathit{msg}$$

**Figure 3.16.** Translation rules for error terms. Both rules require an instance of the `Partial` type class for *Shape* and *Pos* to be available in the current context.

### 3.3.6 Literals

The last kind of expression whose translation we did not cover yet are literals. We support list, pair and integer literals. As mentioned in Section 2.4 already, list and pair literals are just *syntactic sugar* for the application of their constructors. Thus, we can specify their translation rules directly as listed in Figure 3.17.

$$[e_1,\ e_2,\ \ldots,\ e_n]^\dagger = (e_1\ \texttt{:}\ (e_2\ \texttt{:}\ (\ldots\ \texttt{:}\ (e_n\ \texttt{:}\ \texttt{[]})\ldots)))^\dagger$$
$$(e_1,\ e_2)^\dagger = (\texttt{(,)}\ e_1\ e_2)^\dagger$$

**Figure 3.17.** Translation rules for list and pair literals as derived from Equation 2.4.1 and Equation 2.4.2.

However, before we can specify how to translate integer literals we have to find an appropriate representation for the type `Integer` in Coq first. The Haskell to Coq compiler developed by Jessen (2019) does not support `Integer` but translates `Int` with `nat`. The type `nat` is defined in Coq's standard library and represents natural numbers.

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

There are two problems that make `nat` unsuitable for the representation of integers. First of all, `nat` can represent arbitrarily large numbers whereas the size of `Int` is bounded in Haskell. Even more problematic is the fact that the exact upper bound is not specified. Nowadays integers are 64 bits wide on most machines.

```
GHCi> maxBound :: Int
9223372036854775807
```

Haskell only guarantees at least 30 bits of precision for `Int`, though (Marlow, 2010, p. 181). Proofs would therefore be implementation specific. For this reason, we avoid `Int` and use `Integer` instead. Similar to `nat`, `Integer` has no upper bound. The second problem that remains is that `nat` cannot represent negative numbers. While `nat` represents $\mathbb{N}$, `Integer` corresponds to $\mathbb{Z}$. Therefore, the type `Z` from the `ZArith` library is used by `hs-to-coq` instead of `nat` [1].

```
Require Export ZArith.
Definition Integer := Z.
```

We adapt their code slightly such that the definition is compatible with our translation of custom data type and type synonym declarations. To do so, the parameters *Shape* and *Pos* must be added even though `Integer` does not need them.

[1] https://github.com/antalsz/hs-to-coq/blob/0cd052b8162ea53611871d7be3bf186cc38a4a74/hs-to-coq/examples/ghc-base/GHC/Num.v#L4

```
Definition Integer (Shape : Type) (Pos : Shape -> Type) : Type := Z.
```

The actual integer literals can now be translated by using the numeral notation for `Z`. In contrast to Haskell, Coq does not support hexadecimal or octal notation. Thus, we have to convert integer literals to decimal notation first. By default number literals are interpreted as values of type `nat` by Coq. To create a value of type `Z`, we have to append the suffix `%Z`.

```
Check 42.   (* ==> 42   : nat *)
Check 42%Z. (* ==> 42%Z : Z *)
```

Since `Integer` *Shape Pos* is just a synonym for `Z`, `42%Z` is also of type **`Integer`***. To fulfill our invariant, we finally need to lift the value into the free monad.

$$i^{\dagger} = \texttt{pure}\ i'\texttt{\%Z}$$

**Figure 3.18.** Translation of an integer literal $i$. The decimal value of $i$ is denoted $i'$.

## 3.4 Function Declarations

In this section we will cover the translation of the eponymous component for functional programming languages: function declarations.

Just as with type declarations the order of function declarations matters in Coq as well. Thus, we have to perform a dependency analysis again. Although the basic principles of the dependency analysis still apply, adding support for recursive and mutually recursive functions is more involved this time. The additional complexity arises from the fact that Coq strictly distinguishes recursive and non-recursive function declarations. Therefore, it is not sufficient to simply concatenate generated sentences. Instead, we have to apply a completely different translation scheme to recursive function declarations. For this reason, we will focus on non-recursive function declarations first. The translation of recursive functions is covered in the second half of this section. In the last section, there are some remarks regarding the translation of partial functions.

### 3.4.1 Non-Recursive Functions

In order to recall the basic structure of function declarations in Coq, consider the following implementation of boolean negation without monadic lifting.

```
Definition not (b : bool) : bool :=
  match b with
  | true  => false
  | false => true
  end.
```

Unlike in Haskell, there are not multiple rules and pattern matching is performed explicitly on the right-hand side using `match` expressions. The return type of the function as well as the types of its arguments are annotated inside the declaration itself. Due to our assumptions, i.e., that there is only one rule and the parameters are variable patterns, all Haskell function declarations we are concerned with already look remarkably similar to the `Definition` sentences that we need to generate.

```
f :: τ
f x1 ... xn = e
```

The main difference here is that the type signature is independent of the actual function declaration. If we had a type signature of the form `f :: τ1 -> ... -> τn -> τ`, we could apply the translation proposed in Figure 3.19 directly. Note that we implicitly apply the optimization by Abel et al. (2005) again, i.e., the type of the converted function is not $(\tau_1 \to \ldots \to \tau_n \to \tau)^\dagger$ but ${\tau_1}^\dagger \to \ldots \to {\tau_n}^\dagger \to \tau^\dagger$.

$$\left(\begin{array}{l} f \texttt{ :: } \tau_1 \texttt{ -> } \ldots \texttt{ -> } \tau_n \texttt{ -> } \tau \\ f\ x_1\ \ldots\ x_n \texttt{ = } e \end{array}\right)^\dagger = \begin{array}{l} \texttt{Definition } f' \ (Shape : \texttt{Type}) \ (Pos : Shape \texttt{ -> Type}) \\ \{\alpha'_1\ \ldots\ \alpha'_m : \texttt{Type}\} \\ (x'_1 : {\tau_1}^\dagger)\ \ldots\ (x'_n : {\tau_n}^\dagger) \\ : \tau^\dagger \\ := e^\dagger . \end{array}$$

**Figure 3.19.** Translation rule for a $n$-ary polymorphic function declaration with $m$ type variables. The types $\tau_1$ through $\tau_n$ are the types of the variables $x_1$ through $x_n$, respectively.

However, in general the type signature does not necessarily have that format due to the potential usage of type synonyms. Consider for example the following function declaration.

```
type Predicate a = a -> Bool

nonZero :: Predicate Integer
nonZero n = n /= 0
```

To determine the type of `n` as well as the return type of `nonZero`, we need to expand the type synonym `Predicate Integer`. Not all type synonyms need to be expanded, though. Our current approach suffices to determine the argument and return types the following two functions.

```
testZero :: Predicate Integer -> Bool
testZero p = p 0

greaterThan :: Integer -> Predicate Integer
greaterThan n = (> n)
```

The rule shown in Figure 3.20 expands only those type synonyms that need to be expanded in order to determine the type of an argument. In the example above, $^\dagger$ would expand the type of `nonZero` to `Integer -> Bool` but leaves the type of both `testZero` and `greaterThan` unchanged.

$$\left(\begin{array}{l} f \texttt{ :: } \tau_1 \texttt{ -> } \ldots \texttt{ -> } \tau_i \texttt{ -> } S\ \kappa_1\ \ldots\ \kappa_m \\ f\ x_1\ \ldots\ x_n \texttt{ = } e \end{array}\right)^{\dagger} = \left(\begin{array}{l} f \texttt{ :: } \tau_1 \texttt{ -> } \ldots \texttt{ -> } \tau_i \texttt{ -> } \sigma(\kappa) \\ f\ x_1\ \ldots\ x_n \texttt{ = } e \end{array}\right)^{\dagger} \qquad \forall i < n$$

**Figure 3.20.** A demand driven approach to the expansion of type synonyms in $n$-ary function declarations. We assume that there is type synonym declaration `type` $S\ \alpha_1\ \ldots\ \alpha_m$ `=` $\kappa$. For the expansion of $S$, we apply the substitution $\sigma = \{\, \alpha_1 \mapsto \kappa_1, \ldots, \alpha_m \mapsto \kappa_m \,\}$.

### 3.4.2 Recursive Functions

If we tried to apply the translation rule for non-recursive function declarations to the declaration of a recursive function like `append` (as defined in the listing below), Coq would complain that there is no such function when it encounters the recursive call to `append`.

```
append :: [a] -> [a] -> [a]
append xs ys = case xs of
  []     -> ys
  x : xs' -> x : append xs' ys
```

The reason for this error is that `Definition` sentences are not allowed to reference themselves in Coq. If they were allowed to do so, it would be possible to define a non-terminating function. The following is allowed in Haskell for example.

```
loop :: a
loop = loop
```

Similar to `undefined`, `loop` is a polymorphic value. In much the same way as shown for `undefined` in Section 3.3.5, `loop` could therefore be used to prove arbitrary propositions when translated to Coq. Nevertheless, recursion is an important aspect of functional programming languages and most recursive functions also terminate eventually. In order to still allow recursion, Coq needs to ensure that recursive functions terminate on all inputs. To indicate that this check needs to be performed by Coq, the user has to explicitly annotate a recursive function as such. Instead of a `Definition` sentence, a `Fixpoint` sentence is used in this case. The following non-lifted implementation of `append` would be accepted by Coq for example.

```
Fixpoint append {a : Type} (xs : list a) (ys : list a) : list a
  := match xs with
     | nil        => ys
     | cons x xs' => cons x (append xs' ys)
     end.
```

To check that functions like `append` actually terminate, Coq carries out a syntactic analysis of the `Fixpoint` sentence. Namely, Coq must be able to recognize that the function *decreases* on one of its

arguments. We say that a **Fixpoint** sentence **Fixpoint** $f$ (x1 : τ1) ... (x1 : τ1) : τ := e. decreases on $x_i$ if the $i$-th argument of every recursive call to $f$ in $e$ is *structurally smaller* than the original argument $x_i$ (The Coq Development Team, 2018, p. 111). One source of structurally smaller terms is pattern matching. If pattern matching is performed on the decreasing argument or a structurally smaller term, the variables in the patterns are structurally smaller than the decreasing argument themselves. In the example above, `append` is decreasing on its first explicit argument, namely `xs`. It is decreasing on this argument because in the only recursive call `append xs' ys` the first argument, i.e., `xs'`, is bound to a subterm of `xs` by the means of pattern matching.

Although Coq's termination checker uses an even more sophisticated ruleset to determine whether a term is structurally smaller than another, it rejects the lifted version of `append` that the translation rule in Figure 3.19 would produce if we simply swapped **Definition** for **Fixpoint**.

```
Fixpoint append (Shape : Type) (Pos : Shape -> Type) {a : Type}
                (xs : [a]†) (ys : [a]†) : [a]†
  := xs >>= (fun xs_0 =>
       match xs_0 with
       | nil        => ys
       | cons x xs' => Cons Shape Pos x (append xs' ys)
       end).
```

The definition is rejected because this time `xs'` is not a direct subterm of `xs` but of `xs_0`. That the `>>=` operator is implemented in such a way that `xs_0` is always bound to a subterm of `xs` is not checked by Coq. There is no particular theoretical justification for this limitation of the heuristic employed by Coq's termination checker (Chlipala, 2013, p. 62). As the *halting problem* is undecidable, it is natural that there are always situations like this, where the termination checker cannot guess the decreasing argument. In this concrete situation, the problem results from the usage of a *nested inductive type*. Let's first recall the definition of the monadically lifted `List` type from Section 3.2.3.

```
Inductive List (Shape : Type) (Pos : Shape -> Type) (a : Type) : Type
  := nil  : List Shape Pos a
   | cons : Free Shape Pos a -> Free Shape Pos (List Shape Pos a) -> List Shape Pos a.
```

The `List` type is called a nested inductive type because in the second argument of the `cons` constructor the type `List Shape Pos a` occurs *nested* inside the type constructor `Free Shape Pos` (Dylus et al., 2018, p. 29).

As described by Dylus et al. (2018), we can work around the restriction of Coq's termination checker concerning nested inductive types by transform the recursive function such that it resembles the nested structure of the decreasing argument's data type. For this purpose, the original function is splitted into a recursive helper function and a non-recursive main function. The helper function, must be defined such that its decreasing argument is not lifted to the free monad. In case of `append`, the fact that the helper function `append'` expects a non-monadic argument means that `append'` cannot be invoked before the bind operation has been performed. Therefore, the main function for `append` looks as follows if we insert the call to the helper function as soon as possible.

```
Definition append (Shape : Type) (Pos : Shape -> Type) {a : Type}
```

```
                (xs : [a]†) (ys : [a]†) : [a]†
  := xs >>= (fun xs_0 => append' xs_0 ys).
```

The body of the helper function now has to contain the `match` term that we replaced by the call to `append'` above.

```
match xs_0 with
| nil        => ys
| cons x xs' => Cons Shape Pos x (append xs' ys)
end
```

However, because `append'` must be defined before `append`, `append'` cannot reference `append`. We can solve this problem by inlining the definition of `append` into `append'`.

```
Fixpoint append' (Shape : Type) (Pos : Shape -> Type) {a : Type}
                 (xs : [a]*) (ys : [a]†) : [a]†
  := match xs_0 with
     | nil        => ys
     | cons x xs' => Cons Shape Pos x (xs' >>= (fun xs'_0 => append' xs'_0 ys))
     end.
```

This pair of definitions is now accepted by Coq's termination checker. Most importantly, though, the transformation also does not change the behaviour of `append`. Intuitively, the transformed function behaves in the same way because we perform a bind only when the original definition would also have performed a bind. However, proving the correctness of this transformation is beyond the scope of this thesis.

Next, we want to generalize the approach outlined above. Since function declaration can be mutually recursive, we have to consider entire strongly connected components of the dependency graph again. In summary, the following steps need to be performed for the translation.

1. Determine the decreasing argument of all function declarations in the strongly connected component.

2. Split the functions into helper and main functions.

3. Inline the definition of the main functions into the helper functions.

4. Translate all helper functions to a single joined `Fixpoint` sentence and the main functions into individual `Definition` sentences.

For the remainder of this section, we are going to sort out the details of those steps. Of particular interest are the first two steps.

**Determining Decreasing Arguments** We have seen a simplified description of the algorithm used by Coq to test whether a function is decreasing on a specific argument earlier. We are going to use the same simplified version of that algorithm. Of course, that means that we can only translate a subset of all recursive functions that Coq would be able to accept. However, reimplementing the entire termination checker of Coq is beyond the scope of this thesis.

To determine the decreasing argument of a function declaration, we have to test for every argument of the function declaration whether the function is decreasing on this argument. This test cannot be performed independently for each function declaration, because the choice of a decreasing argument influences which arguments of another function are decreasing. Consider two mutually recursive binary functions `foo` and `bar`, where `foo`'s implementation looks something like this.

```
foo x y = case x of
  C x' -> case y of
    C y' -> bar x' y'
```

If `bar` is decreasing on its first argument, then `foo` is also decreasing on its first argument. Conversely, if `bar` does not decrease on its first argument, then `x` is also an invalid choice for the decreasing argument of `foo`. Therefore, we have to consider every possible combination of decreasing arguments and test whether all functions in the strongly connected component actually decrease on those arguments.

**Splitting into Helper and Main Functions** Unlike in the example of `append`, we are going to perform the transformation of the original function into a helper and main function before the conversion to Coq. Another way in which the general case differs from the example is that the generation of a single helper function may not suffice. The need for additional helper functions is a consequence of the restriction that the call to a helper function cannot be inserted before the decreasing argument has been bound for the first time. A bind operator is inserted when translating **`case`** or **`if`** expressions or when invoking a function. As the decreasing argument must be inductively defined, the decreasing argument cannot be a function and will therefore never be invoked. Even though it would be possible for the decreasing argument to be bound by an **`if`** expression, decreasing on a boolean value is not plausible in the first place. Therefore, the only way for a decreasing argument to be bound is when pattern matching is performed on the decreasing argument. As there can be multiple such **`case`** expressions, one helper function must be generated for each of them. For example, two helper functions must be generated for the following function: one for the **`case`** expression in the **`then`** branch and one for the other in the **`else`** branch.

```
f :: τ1 -> τ2                    ⇝    f, f0, f1 :: τ1 -> τ2
f x = if ...                          f x = if ... then f0 x else f1 x
  then case x of ...                  f0 x = case x of ...
  else case x of ...                  f1 x = case x of ...
```

If the **`if`** expression would have been a **`case`** expression itself, the generation of the helper functions would further be complicated by the fact that the inner **`case`** expressions could reference variable patterns from the outer **`case`** expression. The same is true when the **`case`** expression is wrapped by a lambda abstraction. For this reason, additional arguments may have to be added to the generated helper functions.

```
f x = case ... of                 ⇝    f x = case ... of
  C1 y    -> case x of ...                C1 y    -> f1 x y
  C2 y z -> case x of ...                 C2 y z -> f2 x y z
                                        f1 x y    = case x of ...
                                        f2 x y z = case x of ...
```

Furthermore, we neither know the return type of the generated helper functions nor the types of the additional arguments. For example, consider a function `g :: κ -> τ`. If a call to `g` wraps the `case` expression, the corresponding helper function must produce a value of type $\kappa$. However, due to a lack of type inference we do not know $\kappa$.

```
f :: τ1 -> τ                      ⇝    f :: τ1 -> τ
f x = g (case x of ...)                f x = g (f1 x)
                                       f1 :: τ1 -> κ
                                       f1 x = case x of ...
```

As suggested by the examples above, formalizing the process of splitting recursive functions is more technical compared to all translation rules we have seen so far. The splitting procedure involves the following steps.

1. Consider a $n$-ary recursive function declaration $f$ that decreases on its $i$-th argument.

   ```
   f :: τ1 -> ... -> τn -> τ
   f x1 ... xn = e
   ```

   If necessary, expand type synonyms in the type signature of $f$ as shown in Figure 3.20.

2. Find a position $p \in Pos(e)$ of a `case` expression for the decreasing argument.

   $$e|_p = \texttt{case}\ x_i\ \texttt{of}\ alts_p$$

   The selected case expression must not be nested inside another `case` expression for the decreasing argument.

   $$\forall q < p \colon e|_q \neq \texttt{case}\ x_i\ \texttt{of}\ alts_q$$

3. Determine the additional parameters $y_1, \ldots, y_m$ that need to be added to the corresponding helper function. A variable must be added as an additional parameter if it is bound by a surrounding `case` expression or lambda abstraction and occurs freely in $e|_p$.

4. Generate a helper function $f_p$ and replace the `case` expression in the original function by a call to the helper function.

   ```
   f :: τ1 -> ... -> τn -> τ     ⇝   f :: τ1 -> ... -> τn -> τ
   f x1 ... xn = e                   f x1 ... xn = e[fp x1 ... xn y1 ... ym]p
                                     fp :: τ1 -> ... -> τn -> κ1 -> ... -> κm -> κ
                                     fp x1 ... xn y1 ... ym = e|p
   ```

   Where $\kappa_1, \ldots, \kappa_m$ denote the unknown types of the additional parameters $y_1, \ldots, y_m$, respectively, and $\kappa$ denotes the unknown return type of the generated helper function.

5. If there are more `case` expressions for the decreasing argument, go to step 2. Otherwise, we are done.

After all function declarations in the strongly connected component have been splitted into helper and main functions, the definitions of the main functions need to be inlined into the helper functions. It does not suffice to inline the main function into the helper functions that were generated from the same original function only.

**Conversion to Coq** In the last step, the generated main and helper functions need to be converted to Coq. Since the main functions are not recursive anymore, they can be translated to `Definition` sentences as shown in Figure 3.19. The order of the main functions does not matter, as all calls to other functions in the strongly connected component must be subterms of the `case` expressions captured by the helper functions.

The helper functions on the other hand remain mutually recursive. Thus, they need to be translated into a single `Fixpoint` sentence. Figure 3.21 shows the translation of a single helper function to a `Fixpoint` sentence. If there are multiple helper functions, the individual `Fixpoint` sentences can be concatenated using the `with` keyword just as with `Inductive` sentences. To make sure that Coq does not guess a different decreasing argument, we explicitly mark the decreasing argument with a `{struct` $x$`}` annotation.

$$
\begin{pmatrix} h\ \texttt{::}\ \tau_1\ \texttt{->}\ \ldots\ \texttt{->}\ \tau_n\ \texttt{->}\ \kappa_1\ \texttt{->}\ \ldots\ \texttt{->}\ \kappa_p\ \texttt{->}\ \kappa \\ h\ x_1\ \ldots\ x_n\ y_1\ \ldots\ y_p\ \texttt{=}\ \texttt{case}\ x_i\ \texttt{of}\ alts \end{pmatrix}^{\dagger}
$$

$$
= \quad \begin{array}{l} \texttt{Fixpoint}\ h'\ \ (Shape\ \texttt{:}\ \texttt{Type})\ (Pos\ \texttt{:}\ Shape\ \texttt{->}\ \texttt{Type})\ \{\alpha'_1\ \ldots\ \alpha'_m\ \texttt{:}\ \texttt{Type}\} \\ \qquad (x'_1\ \texttt{:}\ {\tau_1}^{\dagger})\ \ldots\ (x'_i\ \texttt{:}\ {\tau_i}^{*})\ \ldots\ (x'_n\ \texttt{:}\ {\tau_n}^{\dagger}) \\ \qquad y'_1\ \ldots\ y'_p\ \{\texttt{struct}\ x'_i\}\ \texttt{:=}\ (\texttt{case}\ x_i\ \texttt{of}\ alts)^{\dagger}. \end{array}
$$

**Figure 3.21.** Translation rule for a single helper function $h$ that is decreasing on its $i$-th argument. $y_1, \ldots, y_p$ are the additional parameters of unknown type $\kappa_1, \ldots, \kappa_p$. The return type $\kappa$ of the helper function is also not known and must be inferred by Coq. The decreasing argument $x_i$ is not lifted into the free monad.

### 3.4.3 Partial Functions

The translation of error terms show in Figure 3.16 required an instance of the `Partial` type class for *Shape* and *Pos* to be available in the current context. In Haskell it would suffice to have a `instance` declaration in the current module or import a module that contains such a declaration.

```
instance Partial (Maybe a) where
  undefined  = Nothing
  error _msg = Nothing
```

When `undefined` is used in a place where a value of type `Maybe a` is expected, Haskell would automatically use the implementation of `undefined` from the type class instance above. In Coq, however, type

class instances must be passed as arguments to functions, similar to how type arguments are passed to functions as well. Thus, if a function contains an error term, i.e., invokes `undefined` or `error`, then the function must have an additional argument for the `Partial` type class instance. For example, `head` needs a `Partial` instance when converted to Coq. Therefore, we add another explicit parameter after *Shape* and *Pos*.

```
⎛ head :: [a] -> a                ⎞†
⎜ head xs = case xs of            ⎟
⎜   [] -> undefined               ⎟
⎝   x : xs' -> x                  ⎠
```

=

```
Definition head (Shape : Type) (Pos : Shape -> Type)
  (P : Partial Shape Pos) {a : Type}
  (xs : [a]†) : [a]† := (* ... *).
```

In consequence, every function that uses `head` also needs a `Partial` instance in order to invoke `head`. If we consider `error` and `undefined` to be normal functions, we can use the dependency graph to determine which functions need to be equipped with a `Partial` instance: if there is a directed path from a function to the nodes of the dependency graph that correspond to `error` or `undefined`, a parameter for the `Partial` instances must be added. In case of recursive functions, the main and helper functions need a `Partial` instance if and only if the original function needs a `Partial` instance. Hereinafter, we will refer to functions that take a `Partial` instance as a parameter as *partial functions*.

The only change to the existing translation rules in Figure 3.19 Figure 3.21 is that the parameter `(P : Partial Shape Pos)` is added after *Shape* and *Pos* if the declared function is partial. Additionally, we have to revise the translation rule for the application of defined functions from Figure 3.9. If the function is partial, the type class instance $P$ must be passed after *Shape* and *Pos* as shown in Figure 3.22.

$$(f\ e_1\ \ldots\ e_n)^\dagger = f'\ Shape\ Pos\ P\ e_1{}^\dagger\ \ldots\ e_n{}^\dagger$$

**Figure 3.22.** Translation rule for the full application of a $n$-ary partial function $f$.

Lastly, note that higher-order functions are not required to be partial, even though a function that is passed to them could be partial. For example, `map` does not require a `Partial` instance. The only function invoked by `map` – except for constructors and itself – is the function passed as its first argument.

```
map :: (a -> b) -> [a] -> [b]
map f xs = case xs of
  []      -> []
  x : xs' -> f x : map f xs'
```

If we try to plug a partial function like `head` into `map`, `head` is equipped with the `Partial` instance at call-time already.

$$(\texttt{map head xss})^\dagger = \texttt{map (fun xs => head } Shape\ Pos\ P\ \texttt{xs) xss}$$

Therefore, there is no need for `map` to know about the `Partial` instance.

## 3.5 QuickCheck Properties

In this last section, an extension to the translation rules for function declarations is presented that allows templates for Coq theorems to be generated from QuickCheck properties.

### 3.5.1 Motivation

The goal of the translation to Coq is to prove properties of the original Haskell programs. Before such a property can be proven using Coq, the user has to formulate a suitable proposition first. Due to the overhead involved with our translation, writing down such propositions in Coq directly is a tedious task. However, since we want to state proposition about Haskell code, it is natural to note them down in Haskell. QuickCheck provides us with a mechanism for writing properties that we expect our program to fulfill.

The QuickCheck library is usually used during development of Haskell programs to create automated tests. The programmer writes a regular function – a so called *QuickCheck property* – that returns a boolean value for example. By convention the names of such QuickCheck properties start with the prefix `prop_`. We can think about them as universally quantified propositions. For example, the following QuickCheck property states, that all integers $n$ and $m$ commute under addition.

```
prop_add_comm :: Integer -> Integer -> Bool
prop_add_comm n m = n + m == m + n
```

QuickCheck generates arbitrary values for the function's arguments and passes them to the function. If the function returns `True`, the property is satisfied and QuickCheck generates further examples to test the function with. Otherwise, QuickCheck terminates printing the counter example it found for the property.

Thus, QuickCheck does not only provide a notation for properties that we want to prove in Coq but also offers the user a way to convince themselves that the program probably fulfills the property before they attempt the proof.

### 3.5.2 Assumptions

The extension proposed in this section is optional. To enable the translation of QuickCheck properties, the Haskell module must contain the following `import` declaration.

```
import Test.QuickCheck
```

This `import` declaration imports only a selected subset of QuickCheck's functionalities. Namely, the following operators are supported:

```
⊳ (==>)          :: Testable prop => Bool -> prop -> Property,
```

▷ `(.&&.), (.||.) :: (Testable prop1, Testable prop2) => prop1 -> prop2 -> Property` and

▷ `(===), (=/=) :: a -> a -> Property`.

The operations above are available only inside QuickCheck properties, i.e., functions whose name starts with `prop_`. The semantics of the operations is explained in the next subsection.

The type `Property` is used by QuickCheck to represent more complex properties than `Bool` can represent. In user-defined functions, it is only allowed to be used as the return type of QuickCheck properties. The type class `Testable` is used in the type signatures above to indicate that the corresponding arguments can either be of type `Bool` or `Property`. The `(===)` and `(=/=)` operations usually require an instance of the type `Eq` type class. Since users cannot implement `Eq` instances in our compiler, we allow these two operators to be used with any type and use structural equality for the actual comparison.

A QuickCheck property $p$ has the following form where $\tau$ is one of `Bool` or `Property`.

```
prop_p :: τ1 -> ... -> τn -> τ
prop_p x1 ... xn = e
```

### 3.5.3 Code Generation

QuickCheck properties are translated to `Theorem` sentences as shown in Figure 3.23. The arguments of the QuickCheck property – including type variables used in it's type signature as well as the parameters *Shape*, *Pos* and potentially $P$ – must be quantified universally. For the translation of the right-hand sides of QuickCheck properties, we introduce a new translation operation denoted by $^{\ddagger}$. The corresponding proof is left blank and must be filled in by the user.

$$\left(\begin{array}{l}\texttt{prop\_}p\ \texttt{::}\ \tau_1\ \texttt{->}\ \ldots\ \texttt{->}\ \tau_n\ \texttt{->}\ \tau \\ \texttt{prop\_}p\ x_1\ \ldots\ x_n\ \texttt{=}\ e\end{array}\right)^{\dagger} =$$

```
Theorem prop_p':
  forall (Shape : Type) (Pos : Shape -> Type)
         {α'1 ... α'm : Type} (x'1 : τ1†) ... (x'n : τn†), e‡.
Proof. (* FILL IN HERE *) Admitted.
```

**Figure 3.23.** Translation rule for QuickCheck properties. If the function `prop_`$p$ is partial, an additional binding `(P : Partial Shape Pos)` must be added.

In the simplest case, the right-hand side of the property is an expression of type `Bool`. Such a property holds if the boolean expressions evaluates to `True`.

$$e^{\ddagger} = e^{\dagger} = \texttt{True}^{\dagger} \qquad \text{if } e \texttt{ :: Bool}$$

Since we do not support type classes, `(==)` and `(/=)` can only be used for the comparison of integers. This complicates writing properties that involve equality of other data types. For example, the following is invalid.

```
prop_append_assoc :: [a] -> [a] -> [a] -> Bool
prop_append_assoc xs ys zs = (xs `append` ys) `append` zs == xs `append` (ys `append` zs)
```

For this reason, we translate QuickCheck's `(===)` and `(=/=)` operators to Coq's built-in structural equality.

$$(e_1 \texttt{ === } e_2)^{\ddagger} = {e_1}^{\dagger} \texttt{ = } {e_2}^{\dagger}$$
$$(e_1 \texttt{ =/= } e_2)^{\ddagger} = {e_1}^{\dagger} \texttt{ <> } {e_2}^{\dagger}$$

Furthermore, the precondition operator `(==>)` is translated to an implication in Coq.

$$(e_1 \texttt{ ==> } e_2)^{\ddagger} = {e_1}^{\ddagger} \texttt{ -> } {e_2}^{\ddagger}$$

The operators `(===)`, `(=/=)` and `(==>)` produce values of type `Property` and correspond to Coq propositions formed by `=`, `<>` and `->`. Therefore, we cannot use `(&&)` or `(||)` to combine such values. In Coq `/\` and `\/` can be used to denote the conjunction and disjunction of propositions, respectively. We are using QuickChecks `(.&&.)` and `(.||.)` operators for this purpose.

$$(e_1 \texttt{ .\&\&. } e_2)^{\ddagger} = {e_1}^{\ddagger} \texttt{ /\ } {e_2}^{\ddagger}$$
$$(e_1 \texttt{ .||. } e_2)^{\ddagger} = {e_1}^{\ddagger} \texttt{ \/ } {e_2}^{\ddagger}$$

Chapter 4

# Implementation

In this chapter the actual implementation of the Haskell to Coq compiler is presented. The implementation is based on the compiler developed by Jessen (2019) but most components have been completely rewritten by now. The first section gives an overview of the general architecture of the compiler. Technologies that have been used for the implementation of the compiler and the phases of the compilation process are summarized. In sections two through six, the individual phases of the compilation process are described in more detail. The last two sections address our approach to error handling and how we realize predefined data types and operations.

## 4.1 Architecture

The Haskell to Coq compiler presented in this thesis is written in Haskell itself. To manage dependencies and to build the compiler, we are using Cabal[1]. Instructions for how to setup, build and run the compiler can be found in Appendix A.

The compilation process for the translation of a Haskell module to Coq is outlined in Figure 4.1. First, Haskell files are parsed into a data structure known as an *abstract syntax tree* ("AST"). In the second stage, the AST is converted to an intermediate representation that simplifies the analysis and conversion of the Haskell program in the subsequent stages. The converter transforms the simplified Haskell AST to a Coq AST that is printed to the console or output file in the final stage of the compilation process.

Using Haskell for the implementation of our compiler allows us to make use of Haskell's package infrastructure. Especially, the Haskell and Coq parsers, ASTs and pretty printers have not been implemented by ourselves. In the following sections more details will be given regarding the individual stages and used libraries.

## 4.2 Haskell Parser

In the first stage of the compilation process, the input file is parsed into an AST. We have not implemented our own Haskell parser for this purpose but are using the `haskell-src-exts` package[2] instead.

[1] `https://www.haskell.org/cabal/`

[2] `http://hackage.haskell.org/package/haskell-src-exts`

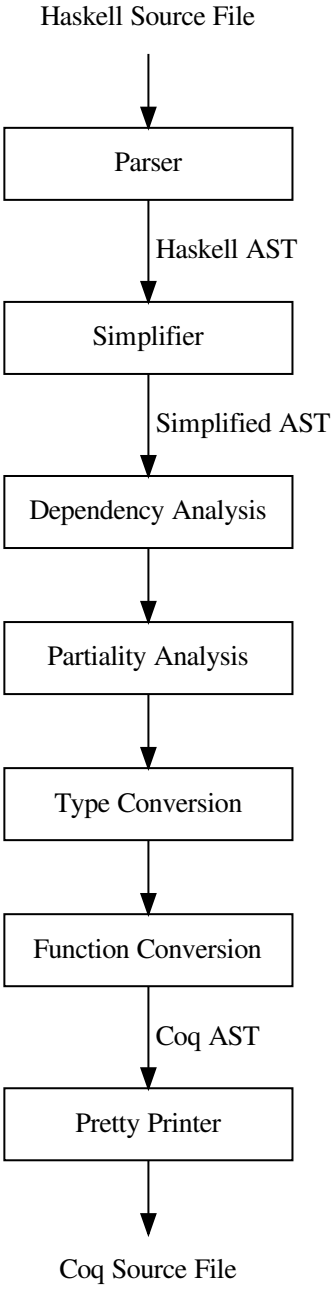



**Figure 4.1.** Stages of the compilation process from Haskell to Coq. Arrows are labeled with the intermediate representation of the data passed between two stages.

`haskell-src-exts` provides a Haskell parser, AST and pretty printer for Haskell and supports a variety of language extensions in addition to standard Haskell. On the one hand, the advantage of using the `haskell-src-exts` is that it allows us to easily add support for further language features in the future. On the other hand, `haskell-src-exts` extensive language support makes it difficult to work directly on its AST.

For example, if we used this data structure to implement the translation of type expressions directly, we would have to consider 19 different constructors of the type that represents type expressions in `haskell-src-exts`. Many of those constructors represent either kinds of type expressions that are part of some language extension or not supported by our compiler yet. Therefore, we would have to ignore a lot of constructors in the implementation of the translation rules. Handling only a subset of the constructors of complex data structures comes at the risk of accidentally missing a case that would have been relevant. Besides the large number of constructors, the hierarchical structure of the AST makes it difficult to reason about source code that uses the AST. In consequence, we do not work directly on the AST from `haskell-src-exts` but use a custom data structure described in the next subsection.

## 4.3 AST Simplification

In order to make the code that is concerned with the translation to Coq more maintainable and readable, we defined our own simplified data structure for the representation of Haskell modules. This simplified AST is tailored to our assumptions in Section 2.4 and models only parts of Haskell, that we actually support.

The Haskell AST generated by `haskell-src-exts` is converted by a component that we call a *simplifier* to our simplified intermediate representation. The simplifier also checks whether only supported features are used by the input module and rewrites syntactic sugar like infix applications in terms of more basic operations.

The simplified AST is used instead of the `haskell-src-exts` AST throughout the compiler to analyse the Haskell program and convert it to Coq. Due to the simplicity of the data structure, all pattern matching performed on the simplified AST can be exhaustive. If we extend the supported subset of Haskell in the future, the exhaustive nature of the pattern matching will help to find parts of the code that need to be adapted.

## 4.4 Dependency Analysis

Since the order of declarations matters in Coq, we have to group and sort type and function declarations according to their dependencies. As has been described in Chapter 3 already, a dependency graph has to be created for this purpose. Declarations whose nodes are in the same strongly connected connected component of the dependency graph, must be converted at the same time and the strongly connected components have to be converted in reverse topological order. Fortunately, there are libraries in Haskell to work with graphs, so we do not have to implement any of those algorithms

on our own. Specifically, we are using the `Data.Graph`[3] module from the `containers` package to handle dependency graphs.

Using the `stronglyConnComp` function from `Data.Graph` one can directly obtain a list of strongly connected components in reverse topological order. To use `stronglyConnComp` we have to represent dependency graphs as a list of triples where each triple represents a node of the dependency graph and has the following components:

- the AST node that corresponds the respective node of the dependency graph,
- the name of the of the type or function declaration and
- a list of names for types and functions used by the declaration.

Thus, the only part of the dependency analysis that we have to implement ourselves is the extraction of used types and functions of the declarations. For simplicity, we consider type and function declarations independently, i.e., we are creating two separate dependency graphs one for type declarations and one for functions. We can ignore dependencies between functions and types as all converted type declarations are placed in front of all function declarations later anyway.

For example, the extraction of a `type` declaration's dependencies is governed by the following rules. In the listing below, $\delta$ denotes a function that computes the set of used identifiers for a given Haskell language construct.

$$\delta(\texttt{type}\ S\ \alpha_1\ \ldots\ \alpha_n\ \texttt{=}\ \tau) = \delta(\tau) \backslash \{\, \alpha_1, \ldots, \alpha_n \,\}$$

$$\begin{aligned} \delta(\tau_1 \to \tau_2) &= \delta(\tau_1) \cup \delta(\tau_2) \\ \delta(\tau_1\ \tau_2) &= \delta(\tau_1) \cup \delta(\tau_2) \\ \delta(T) &= \{\, T \,\} \\ \delta(\alpha) &= \{\, \alpha \,\} \end{aligned}$$

Similar rules are implemented for data type declarations, function declarations and expressions as well.

## 4.5 Partiality Analysis

In addition to sorting function declarations during the dependency analysis, the function dependency graph can also be used to identify partial functions. As discussed in Section 3.4.3, partial functions can be identified by adding two additional nodes to the dependency graph for `undefined` and `error`. However, we cannot simply add an entry for `undefined` and `error` to the list of triples from the previous section since there is no AST node for them that could go into the first components.

Therefore, we first have to look through the list of triples for the names of functions that refer to the identifiers `error` or `undefined`, i.e., identify directly partial functions. We can use `Data.Graph` to find

[3] `http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Graph.html`

the names of all indirectly partial functions, i.e., all nodes from which a path to a directly partial function exists within the dependency graph.

## 4.6 Conversion to Coq

For the conversion to Coq, we need a representation of Coq programs in Haskell. Just like we used an external package for parsing Haskell source code, we are also reusing an existing Coq AST and pretty printer. The pretty printer is a component for converting the Coq AST to a textual representation. Unlike `haskell-src-exts`, there is currently no Coq package on Hackage. However, the developers of `hs-to-coq` implemented a Coq AST and pretty printer already. Thus, we forked their GitHub repository[4] and deleted everything but the Coq language definitions. We refer to the remaining code in the forked repository[5] as the `language-coq` package. Since `hs-to-coq` is released under the MIT license agreement, it is possible to publish the package to Hackage in the future.

With our simplified AST and the `language-coq` package at hand, we can now implement the translation rules from Chapter 3 very easily. The following two subsections provide details on how our compiler handles state and renames identifiers in the conversion process.

### 4.6.1 Environment

During the conversion to Coq, we have to retain a lot of information about the program that we are translating. Amongst others, we have to remember:

- the names we have assigned to the generated Coq sentences, such that we can translate references to them correctly,
- the arity of defined functions, such that we can perform the right number of $\eta$-conversions to partial applications,
- which functions are partial, such that we can pass the `Partial` instance to them and
- which variables have been lifted to the `Free` monad and which haven't.

Since it would be infeasible, to pass all this information as individual parameters, we encapsulate the entire state of the compiler in a single data type **`Environment`**. Additionally, some conversion operations have an effect on the compiler's environment. For example, when translating a function declaration, the function becomes available to subsequently translated function declarations, i.e., an entry must be added to the environment for the declared function. Therefore, every function of the compiler would not only have to be given a parameter of type **`Environemnt`** but can also return an **`Environment`** in addition to the actual result. In consequence, the approach of passing the environment explicitly still causes a lot of overhead. We have solved this issue by using the **`State`** monad. The **`State`** monad allows a state – in our case, the **`Environment`** – to be passed around and manipulated implicitly throughout the computation.

[4] `https://github.com/antalsz/hs-to-coq`

[5] `https://github.com/just95/language-coq`

### 4.6.2 Renaming Identifiers

As mentioned in Section 2.3.3, identifiers may have to be renamed during the translation to Coq. For example, the Coq Reference Manual lists the following reserved keywords which cannot be used as identifiers in Coq (The Coq Development Team, 2018, p. 24).

```
_ as at cofix else end exists exists2 fix for
forall fun if IF in let match mod Prop return
Set then Type using where with
```

Furthermore, words used within vernacular commands, should not be used as identifiers. For example, it is not possible to define a datatype called `Definition` in Coq, i.e., the following is invalid.

```
Inductive Definition : Type := (* ... *).
(* ==> Syntax error: [inductive_definition] expected
       after [finite_token] (in [vernac:gallina]). *)
```

Therefore, we have assembled a list of all Coq keywords and words used within vernacular commands (The Coq Development Team, 2018, pp. 484-487). Whenever there is a declaration whose name occurs within that list, it must be renamed as described below.

Another reason for an identifier to have to be renamed, is that there is a declaration with the same name already. Such name conflicts can occur because – unlike Coq – Haskell has two distinct namespaces for types and values. The following is valid in Haskell but when converted to Coq we have to rename either the type or constructor for example.

```
data Foo = Foo
```

Therefore, we remember the names of all defined types, functions and variables in our environment. If an identifier is translated and occurs already in the environment, it has to be renamed. Additionally, we have to remember the namespace where the identifier has been defined. It is an error if there are two declarations in the same namespace with the same name. For example, it is not allowed to have two type variables with the same name.

```
data Foo a a = Foo
```

Since we want to preserve the original identifier as good as possible, identifiers are renamed by appending a number. The number starts with zero and is increased until the resulting identifier is available. If the original identifier ends with a number, counting continues at that number. Consider for example the following data type declaration.

```
data Definition exists2 = Definition exists2
```

The identifiers that occur within this declaration need to be translated as follows.

- The name of the type constructor is encountered first. As `Definition` is a vernacular command, it has to be renamed to `Definition0`.
- The type variable `exists2` is a keyword and needs to be renamed as well. Since it ends with a number, it is renamed to `exists3` instead of `exists20`.
- The constructor `Definition` again needs to be renamed as it is a vernacular command. However, this time the identifier `Definition0` is taken already. Therefore, the next available identifier is `Definition1`. The capitalized version of the constructor name is used for the smart constructor. The actual constructor of the data type starts with a lower case letter. In case of `Definition`, the data constructor is called `definition` in Coq and does not have to be renamed.
- In the constructor argument, `exists2` is not a declaration but a reference. The compiler looks up the corresponding Coq identifier for `exists2` in the environment. The environment contains an entry that maps the Haskell identifier `exists2` in the type namespace to the Coq identifier `exists3`. Therefore, the type variable is translated to `exists3`.

The code generated by our compiler looks as follows.

```
(* Data type declarations for Definition *)
Inductive Definition0 (Shape : Type) (Pos : Shape -> Type)
                      (exists3 : Type) : Type
  := definition : Free Shape Pos exists3 -> Definition0 Shape Pos exists3.

(* Arguments sentences for Definition *)
Arguments definition {Shape} {Pos} {exists3}.

(* Smart constructors for Definition *)
Definition Definition1 (Shape : Type) (Pos : Shape -> Type)
                       {exists3 : Type} (x_0 : Free Shape Pos exists3)
   : Free Shape Pos (Definition0 Shape Pos exists3) :=
  pure (definition x_0).
```

The identifier `x_0` in the smart constructor declaration is freshly generated by the compiler. Internally fresh identifier consist of a prefix (`x` in the example above), an `@` sign and an increasing number. As the `@` signs are not allowed inside Haskell identifiers, fresh identifiers are guaranteed to never conflict with any user-defined identifiers. For the purpose of generating fresh identifiers, the environment contains an independent counter for each prefix. Since the `@` sign is not allowed within Coq identifiers either, it is replaced by an underscore before renaming the identifier.

## 4.7 Error Reporting

Errors can occur in all stages of the compilation process. For example, there could be:

- an IO error while loading the input file,

- a syntax error when parsing the input file's content,
- an error due to the usage of unsupported Haskell features during the simplification of the AST or
- an error due to the usage of an undefined identifier during the conversion to Coq.

One way to report errors in Haskell is by using the `error` function which prints an error message and terminates the program immediately. However, we do not want our compiler to crash but rather print the error message in a user-friendly manner and terminate gracefully. Furthermore, there are non-fatal kinds of messages, i.e., hints and warnings, that we would like to report to the user. For example, the user should be informed when an identifier is renamed during the conversion to Coq.

Therefore, we model the entire compilation process as a monadic computation based on the `Maybe` and `Writer` monads. Our monad collects reported messages during the computation and cancels the computation if a fatal error occurs. When the computation returns to the compiler's main loop, the collected error messages are printed to the console via the `IO` monad. We are printing messages in a format based on the GHC's message format. For example, the following program

```
1 module Test where
2
3 head :: [a] -> a
4 head (x : xs) = x
```

causes the following error message to be reported.

```
Test.hs:4:6: error:
    Expected variable pattern.
   |
 4 | head (x : xs) = x
   |      ^^^^^^^^
```

## 4.8 Base Library

The code generated by the translation rules presented in Chapter 3 relies on a definition of the free monad to be available in Coq. Furthermore, implementations of predefined data types and operations as described in Section 2.4.7 must be provided. For this reason, our compiler is accompanied by a Coq library called `Base`. The compiler adds a command to the top of every generated Coq file that imports the `Free` and `Prelude` modules of the base library.

```
From Base Require Import Free Prelude.
```

While the `Prelude` module contains predefined data types and operations, the `Free` module exports definitions of the free monad, bind operator and `Partial` type class. Concrete instances for `Identity`, `Maybe` and `Error` are included in the Base library as well but not exported by default.

Similar to how user-defined data types sometimes need to be renamed, the symbols used for predefined data types and operations must be given a valid Coq identifier. For example, the nullary tuple is called `Unit` in the base library while it is simply denoted `()` in Haskell. To reduce the coupling between the compiler and the base library, these names are not hard-coded into the compiler but configurable. For this purpose, there is a `env.toml` configuration file in the root directory of the base library whose format is documented in Appendix B. The separation of the base library from the actual compiler makes it easy to extend the `Prelude` in the future.

One consequence of the separation is, that the user has to explicitly specify the location of the base library using the `--base-libray` command line option (see also Appendix A) if it cannot be found in the directory where Cabal usually places data files. The compiler automatically creates a `_CoqProject` file in the output directory if it does not exists. The `_CoqProject` tells Coq where the `Base` library can be found.

Chapter 5

# Case Study

In this chapter we want to evaluate whether the Coq code generated by our compiler is actually useful to prove properties about the original Haskell program. For our evaluation, we are using an example that has been used in the articles by Abel et al. (2005) and Dylus et al. (2018) before. The goal is to prove the correctness of an optimized queue implementation based on two lists by relating it to a more straightforward implementation of a queue with a single list.

```
type Queue a = [a]

empty :: Queue a
empty = []

isEmpty :: Queue a -> Bool
isEmpty q = null q

front :: Queue a -> a
front (x:q) = x

add :: a -> Queue a -> Queue a
add x q = q ++ [x]
```

```
type QueueI a = ([a], [a])

emptyI :: QueueI a
emptyI = ([], [])

isEmptyI :: QueueI a -> Bool
isEmptyI (f, b) = null f

frontI :: QueueI a -> a
frontI (x : f, b) = x

addI :: a -> QueueI a -> QueueI a
addI x (f, b) = flipQ (f, x : b)

flipQ :: QueueI a -> QueueI a
flipQ ([], b) = (reverse b, [])
flipQ q       = q
```

**Figure 5.1.** The two queue implementations. The implementation on the left uses a single list and the implementation on the right uses two lists.

The two implementations as shown in Figure 5.1 cannot be feed directly into our compiler since they do not comply with our assumptions (see Section 2.4). First, pattern matching on the left-hand side of the function declarations needs to be converted to explicit pattern matching using `case` expressions as shown in Figure 5.2. Furthermore, we have to provide implementations for `null`, `(++)` and `reverse` as they are not yet part of the `Prelude` provided by our compiler. Since we do not permit the declaration of custom operators, we have to rename `(++)` to `append`. Figure 5.3 shows the declarations of these auxiliary functions. The definitions in Figure 5.2 and Figure 5.3 can now be converted to Coq and are also accepted by Coq.

```
type Queue a = [a]

empty :: Queue a
empty = []

isEmpty :: Queue a -> Bool
isEmpty q = null q

front :: Queue a -> a
front q = case q of
  x : q' -> x
  []     -> undefined

add :: a -> Queue a -> Queue a
add x q = q `append` [x]
```

```
type QueueI a = ([a], [a])

emptyI :: QueueI a
emptyI = ([], [])

isEmptyI :: QueueI a -> Bool
isEmptyI q = case q of
  (f, b) -> null f

frontI :: QueueI a -> a
frontI q = case q of
  (f, b) -> case f of
    x : f' -> x
    []     -> undefined

addI :: a -> QueueI a -> QueueI a
addI x a0 = case a0 of
  (f, b) -> flipQ (f, x : b)

flipQ :: QueueI a -> QueueI a
flipQ q = case q of
  (f, b) -> case f of
    []     -> (reverse b, [])
    x : f' -> (x : f', b)
```

**Figure 5.2.** The two queue implementations after transforming pattern matching.

```
null :: [a] -> Bool
null xs = case xs of
  []     -> True
  x : xs -> False

append :: [a] -> [a] -> [a]
append xs ys = case xs of
  []      -> ys
  x : xs' -> x : (append xs' ys)

reverse :: [a] -> [a]
reverse xs = case xs of
  []      -> []
  x : xs' -> reverse xs' `append` [x]
```

**Figure 5.3.** Functions required by the queue example but that are not yet defined in our `Prelude`.

The first property that we want to prove is that all queues constructible using the functions from the implementation with two lists fulfil the following invariant. Again, we need to define the auxiliary function `not` that is usually part of the `Prelude`.

```
not :: Bool -> Bool
not b = case b of
  True  -> False
  False -> True

invariant :: QueueI a -> Bool
invariant qi = case qi of (f,b) -> null b || not (null f)
```

Next, we define two QuickCheck properties: one tests whether the empty queue fulfills the invariant and the other one tests whether a queue constructed by adding an arbitrary element to a queue which fulfills the invariant, preserves the invariant.

```
prop_inv_empty :: Bool
prop_inv_empty = invariant emptyI

prop_inv_add :: a -> QueueI a -> Property
prop_inv_add x q = invariant q ==> invariant (addI x q)
```

If we add a QuickCheck import to the module, our compiler generates the following `Theorem` sentences.

```
Theorem prop_inv_empty : forall (Shape : Type) (Pos : Shape -> Type),
  invariant Shape Pos (emptyI Shape Pos) = True_ Shape Pos.
Proof. (* FILL IN HERE *) Admitted.

Theorem prop_inv_add :
  forall (Shape : Type) (Pos : Shape -> Type) {a : Type}
         (x : Free Shape Pos a) (q : Free Shape Pos (QueueI Shape Pos a)),
  (invariant Shape Pos q = True_ Shape Pos) ->
  (invariant Shape Pos (addI Shape Pos x q) = True_ Shape Pos).
Proof. (* FILL IN HERE *) Admitted.
```

While the definition of `prop_inv_add` is fine, Coq rejects the definition of `prop_inv_empty` with the following error message.

```
Cannot infer the implicit parameter a of invariant whose type is "Type" in environment:
Shape : Type
Pos : Shape -> Type
```

The problem occurs because both the `invariant` and `emptyI` are polymorphic. In case of `prop_inv_add`, Coq can infer the type arguments of `invariant` and `add` from the types of `x` and `q`. However, in case

of `prop_inv_empty` the context contains no type information. In fact, we wanted to prove that the empty queue fulfills the invariant for any value type, but we never quantify the value type in the theorem above. We can fix this issue by adding a binder for a new type variable and passing this type variable explicitly to either `invariant` or `emptyI`. The `@` sign before `emptyI` tells Coq that we want to pass implicit arguments explicitly.

```
Theorem prop_inv_empty : forall (Shape : Type) (Pos : Shape -> Type) (a : Type),
  invariant Shape Pos (@emptyI Shape Pos a) = True_ Shape Pos.
```

Unfortunately, it is not possible to make these changes automatically without performing type inference. Similar problems can occur outside of QuickCheck properties as well. For example, the following definition of `zero` would be rejected by Coq if we applied our current translation.

```
length :: [a] -> Integer
length xs = case xs of
  []      -> 0
  x : xs' -> 1 + length xs'

zero :: Integer
zero = length []
```

Apart from the small change shown above, the prove of `prop_inv_empty` is straightforward since the left-hand side directly evaluates to `True_ Shape Pos`. The complete code for this chapter – including the proof scripts – can be found in the example folder of our compiler[1].

When we attempt the prove of `prop_inv_add`, we find that the property is not true if we consider partial values. If the first component of the input queue is `undefined` and the second empty, the invariant holds for the input due to the lazy evaluation of `(||)`.

```
invariant (undefined, []) = null [] || not (null undefined)
                          = True || not (null undefined)
                          = True
```

Even though the premise holds, the conclusion of the theorem is false in this example. Since `addI` calls `flipQ` and `flipQ` performs pattern matching on the first component, the result of `addI` is `undefined` and not `True`.

```
invariant (addI x (undefined, [])) = invariant (flipQ (undefined, [x]))
                                   = invariant undefined
                                   = undefined
```

The problem is the hidden assumption of the QuickCheck property that the values passed to it are total. Dylus et al. (2018, p. 20) encountered the same problem and proposed to extend the generated QuickCheck property by the premise that the given queue does not contain partial values. For this purpose, we have to manually define inductive properties for the data types in our program that check whether all values are `pure`. Their definition follows the same pattern as the data type declaration in

[1] https://git.informatik.uni-kiel.de/stu203400/haskell-to-coq-compiler/tree/master/example

Coq itself and could be automated in the future. After adding the premise, the theorem looks as follows.

```
Theorem prop_inv_add :
  forall (Shape : Type) (Pos : Shape -> Type)
         {a : Type} (total_a : Free Shape Pos a -> Prop)
         (x : Free Shape Pos a) (q : Free Shape Pos (QueueI Shape Pos a)),
  total_queue Shape Pos total_a q ->
  (invariant Shape Pos q = True_ Shape Pos) ->
  (invariant Shape Pos (addI Shape Pos x q) = True_ Shape Pos).
```

The parameter `total_a` is an arbitrary notion of totality for elements of type `a`. `total_queue` is one of our inductive propositions and ensures that:

- ▷ the queue itself is `pure`,
- ▷ the two lists of the queue are `pure`,
- ▷ all of their sublists are `pure` and
- ▷ the values of the queue are total with respect to `total_a`.

Since `total_a` is universally quantified, the extended theorem does not require the elements of the queue themselves to be total. The inductive propositions add a lot of overhead to our Coq file but the proof of `prop_inv_add` is still simple.

Now that we have shown that the implementation of queues with two lists satisfies the invariant, we want to prove that the implementation is compatible with the single list version. To relate both implementations, we define a function that converts queues from one representation to the other.

```
toQueue :: QueueI a -> Queue a
toQueue qi = case qi of
  (f, b) -> f `append` reverse b
```

We proceed by defining the following QuickCheck properties. In contrast to Dylus et al. (2018), we have to use the `(===)` operator since `(==)` only works with integers in our implementation. The `(===)` operator is translated to Coq's structural equality by our compiler.

```
prop_isEmpty :: QueueI a -> Property
prop_isEmpty qi = invariant qi ==> isEmptyI qi === isEmpty (toQueue qi)

prop_add :: a -> QueueI a -> Property
prop_add x qi = toQueue (addI x qi) === add x (toQueue qi)

prop_front :: QueueI a -> Property
prop_front qi = invariant qi && not (isEmptyI qi) ==> frontI qi === front (toQueue qi)
```

Due to minor differences in the definition of the `Free` monad, slight modifications need to be done to the proofs presented by Dylus et al. (2018). There are further differences in the exact definition of the inductive propositions like `total_queue` which we need for `prop_front` again. Apart from that we can copy their proofs. Since their proof for `prop_add` makes use of induction, we also have to copy and adapt the their induction principles for the free monad (Dylus et al., 2018, pp. 29-31). As proofs by induction are very common, the induction principles have been added to our base library. Only the theorem for `prop_front` requires a `Partial` instance. The other properties are completely effect-generic. While Dylus et al. (2018) were only able to prove `prop_front` for specific monads that model partiality, our approach with the `Partial` type class allows us to extend the proof of `prop_front` to all monads for which a notion of partiality exists.

Chapter 6

# Conclusion

In this final chapter, we conclude the thesis by summarizing our work and providing an outlook for future improvements and extensions to the presented compiler.

## 6.1 Summary and Results

The goal of this thesis was to develop a compiler for the monadic translation for effectful Haskell programs to Coq. First, we have familiarized ourselves with Coq and presented the approach by Dylus et al. (2018) for modeling effectful Haskell code in Coq. Next, we formalized the monadic translation for a subset of Haskell based on the translation from Haskell to Agda by Abel et al. (2005). The resulting translation rules were implemented in Haskell with the help of third party libraries. We extended and adapted an existing prototypical implementation by Jessen (2019) for our implementation.

As demonstrated by our case study in Chapter 5, our compiler generates code which is most often accepted by Coq without user intervention. Minor modifications are sometimes needed due to the lack of type inference. The extension for the translation of QuickCheck properties presented in Section 3.5 helps the user to prove properties by generating templates of theorems from Haskell code. Implicit assumptions of the Haskell code need to be modeled by the user manually in order to prove some properties. Parts of this process could be automated in the future.

In total, we are confident that the compiler presented in this thesis provides a solid basis for future research and the verification of effectful Haskell programs. Nevertheless, our implementation is far from perfect. We present some ideas for future improvements and extensions of our work in the next section.

## 6.2 Future Work

The subset of Haskell supported by our compiler is very restrictive at the moment. To eliminate the limitation on pattern matching, a pattern matching compiler library[1] that is currently being developed by Malte Clement, could be integrated into our compiler in the future. How Haskell features involving strictness, e.g., `newtype` declarations and bang patterns, could be modeled in Coq is described by Christiansen et al. (2019, pp. 132-134). Further language features that we would like to support in the

[1] https://git.informatik.uni-kiel.de/stu204333/placc-thesis

future include type classes, local declarations and type inference. More predefined data types and operations from the Prelude should be added as well. Due to the modular design of our Base library, we expect the extension of the Prelude to be straightforward. The configuration file format developed for the Base library could be reused for the implementation of `import` declarations as well.

Furthermore, improvements need to be made to the translation of recursive functions. Even though the detection of decreasing arguments is very complicated already, its limitations greatly reduce the user's flexibility. In addition, the current translation scheme also does not work very well in conjunction with recursive higher-order functions.

While we have tried to convince ourselves in Chapter 3 that the translation rules are most likely correct, we have not actually proven their correctness in this thesis. Before definitive conclusions can be drawn from proofs about translated programs, it remains to be formally verified that our translation rules model the behaviour of the original program correctly. One aspect of Haskell's semantics which we know is not reflected by the generated Coq code is sharing. Effects are triggered in our implementation through the use of bind operators. If the value is bound twice, the effect is triggered twice. In Haskell sharing prevents a value from being evaluated twice, therefore every effect is triggered at most once. In case of partiality, we cannot distinguish between an effect being triggered once or twice (Christiansen et al., 2019, p. 131). However, we can use the free monad to model a logging effect as provided by the function `trace` in Haskell. In contrast to partiality, we can distinguish an implementation with and without sharing for this effect since the same message may be logged multiple times in the absence of sharing (Christiansen et al., 2019, p. 131). Therefore, the compiler must be adapted to model sharing before it is truly effect-generic.

Appendix A

# Installation and Usage

**Directory Structure** The implementation of the Haskell to Coq compiler presented in this thesis can be found in our Git repository[1]. The code is structured as follows.

- `base`: Contains the Coq base library.
  - `Free`: Contains Coq files for the definition of the free monad.
  - `Prelude` Contains Coq files for predefined data types and operations.
  - `env.toml` Configuration file (see Appendix B) that contains the names of predefined data types and operations.
- `example`: Contains example Haskell and Coq code.
  - `manual`: Manually translated Coq files.
  - `generated`: Coq files generated by the compiler (not included in distributed source).
  - `ExampleQueue.hs` The input module used in the case study.
  - `ExampleQueueTests.v` Proofs and lemmas for QuickCheck properties in `ExampleQueue.hs`.
- `src` Haskell source code.
- `tool` Bash scripts for running and testing the compiler during development.

**Required Software** The Haskell to Coq compiler is written in Haskell and uses Cabal to manage its dependencies. To build the compiler, the GHC[2] and Cabal[3] are required. To use the Coq code generated by our compiler, the Coq proof assistant[4] must be installed. The compiler has been tested with the following software versions on a Debian based operating system.

- GHC, version 8.6.5
- Cabal, version 2.4.1.0
- Coq, version 8.8.2

[1] `https://git.informatik.uni-kiel.de/stu203400/haskell-to-coq-compiler`
[2] `https://www.haskell.org/ghc/`
[3] `https://www.haskell.org/cabal/`
[4] `https://coq.inria.fr/download`

**Compiling the Base Library** In order to use the base library, the Coq files in the base library need to be compiled first. Make sure to compile the base library **before** installing the compiler. We provide a shell script for the compilation of Coq files. To compile the base library with that shell script, run the following command in the root directory of the compiler.

```
./tool/compile-coq.sh base
```

**Installation** First, make sure that the Cabal package lists are up to date by running the following command.

```
cabal new-update
```

To build and install the compiler and its dependencies, change into the compiler's root directory and run the following command.

```
cabal new-install haskell-to-coq-compiler
```

The command above copies the base library and the compiler's executable to Cabal's installation directory and creates a symbolic link to the executable in `~/.cabal/bin`. To test whether the installation was successful, make sure that `~/.cabal/bin` is in your `PATH` environment variable and run the following command.

```
haskell-to-coq-compiler --help
```

**Running without Installation** If you want to run the compiler without installing it on your machine, execute the following command in the root directory of the compiler instead of the `haskell-to-coq-compiler` command.

```
cabal new-run haskell-to-coq-compiler -- [options...] <input-files...>
```

The two dashes are needed to separate the arguments to pass to the compiler from Cabal's arguments. Alternatively, you can use the `./tool/run.sh` bash script.

```
./tool/run.sh [options...] <input-files...>
```

**Usage** To compile a Haskell module, pass the file name of the module to `haskell-to-coq-compiler`. For example, to compile the example module from the case study, run the following command.

```
haskell-to-coq-compiler ./example/ExampleQueue.hs
```

The generated Coq code is printed to the console. To write the generated Coq code into a file, specify the output directory using the `--output` (or `-o`) option. For example, the following command creates a file `example/generated/ExampleQueue.v`.

```
haskell-to-coq-compiler -o ./example/generated ./example/ExampleQueue.hs
```

In addition to the Coq file, a `_CoqProject` file is created in the output directory if it does not exist already. The `_CoqProject` file tells Coq where to find the compiler's base library. Add the `--no-coq-project` command line flag to disable the generation of a `_CoqProject` file.

In order to compile Haskell modules successfully, the compiler needs to know the names of predefined data types and operations. For this purpose, the `base/env.toml` configuration file has to be loaded. If the compiler is installed as described above, it will be able to locate the base library automatically. Otherwise, it may be necessary to tell the compiler where the base library can be found using the `--base-library` (or `-b`) option.

Use the `--help` (or `-h`) option for more details on supported command line options.

Appendix B

# Configuration File Format

As mentioned in Section 4.8, there is a configuration file in the root directory of the base library that configures the names of predefined data types and operations. The configuration file format is Tom's Obvious, Minimal Language ("TOML")[1]. TOML's syntax looks similar to INI files but is standardized in contrast to INI. TOML aims to be more readable and to easier to comprehend that YAML and is superior to JSON as it allows for the usage of comments – which are an essential part of configuration files.

The TOML document in the `env.toml` file must contain three arrays of tables: `types`, `constructors` and `functions`. Each table in these arrays defines a type, constructor or function, respectively. The expected contents of each table are described below.

**Types** The tables in the `types` array must contain the following key/value pairs:

- `haskell-name` (String) the Haskell name of the type constructor.
- `coq-name` (String) the identifier of the corresponding Coq type constructor.
- `arity` (Integer) the number of type arguments expected by the type constructor.

For example, the following entry defines the pair data type.

```
[[types]]
  haskell-name = "(,)"
  coq-name     = "Pair"
  arity        = 2
```

**Constructors** The tables in the `constructors` array must contain the following key/value pairs:

- `haskell-type` (String) the Haskell type of the data constructor.
- `haskell-name` (String) the Haskell name of the data constructor.
- `coq-name` (String) the identifier of the corresponding Coq data constructor.
- `coq-smart-name` (String) the identifier of the corresponding Coq smart constructor.

[1] `https://github.com/toml-lang/toml`

▷ `arity` (Integer) the number of arguments expected by the data constructor.

For example, the following entry defines the data constructor for pairs.

```
[[constructors]]
  haskell-type   = "a -> b -> (a, b)"
  haskell-name   = "(,)"
  coq-name       = "pair_"
  coq-smart-name = "Pair_"
  arity          = 2
```

**Functions** The tables in the `functions` array must contain the following key/value pairs:

▷ `haskell-type` (String) the Haskell type of the function.

▷ `haskell-name` (String) the Haskell name of the function.

▷ `coq-name` (String) the identifier of the corresponding Coq function.

▷ `arity` (Integer) the number of arguments expected by the function.

For example, the following entry defines the equality test for integers.

```
[[functions]]
  haskell-type = "Integer -> Integer -> Bool"
  haskell-name = "=="
  coq-name     = "eqInteger"
  arity        = 2
```

# Bibliography


Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying haskell programs using constructive type theory. pages 62–73, 01 2005. doi: 10.1145/1088348.1088355.

Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN 0262026651, 9780262026659. URL http://adam.chlipala.net/cpdt/cpdt.pdf.

Jan Christiansen, Sandra Dylus, and Niels Bunkenburg. Verifying effectful haskell programs in coq. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, pages 125–138, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6813-1. doi: 10.1145/3331545.3342592. URL http://doi.acm.org/10.1145/3331545.3342592.

Sandra Dylus, Jan Christiansen, and Finn Teegen. One monad to prove them all (functional pearl). *CoRR*, abs/1805.08059, 2018. URL http://arxiv.org/abs/1805.08059.

Benedikt Jessen. Haskell to coq compiler, April 2019. URL https://github.com/beje8442/haskellToCoqCompiler.

Simon Marlow. Haskell 2010 language report. November 2010. URL https://www.haskell.org/definition/haskell2010.pdf.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, January 2019. URL http://www.cis.upenn.edu/~bcpierce/sf. Version 5.6.

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total haskell is reasonable coq. *CoRR*, abs/1711.09286, 2017. URL http://arxiv.org/abs/1711.09286.

The Coq Development Team. *The Coq Reference Manual, Release 8.8.2*, September 2018. URL https://coq.inria.fr/distrib/V8.8.2/files/CoqRefMan.pdf.