Verifying Effectful Haskell Programs in Coq

Jan Christiansen
Flensburg University of Applied Sciences
Germany
jan.christiansen@hs-flensburg.de

Sandra Dylus
University of Kiel
Germany
sad@informatik.uni-kiel.de

Niels Bunkenburg
University of Kiel
Germany
nbu@informatik.uni-kiel.de

Abstract

We show how various Haskell language features that are related to ambient effects can be modeled in Coq. For this purpose we build on previous work that demonstrates how to reason about existing Haskell programs by translating them into monadic Coq programs. A model of Haskell programs in Coq that is polymorphic over an arbitrary monad results in non-strictly positive types when transforming recursive data types likes lists. Such non-strictly positive types are not accepted by Coq's termination checker. Therefore, instead of a model that is generic over any monad, the approach we build on uses a specific monad instance, namely the free monad in combination with containers, to model various kinds of effects. This model allows effect-generic proofs.

In this paper we consider ambient effects that may occur in Haskell, namely partiality, errors, and tracing, in detail. We observe that, while proving propositions that hold for all kinds of effects is attractive, not all propositions of interest hold for all kinds of effects. Some propositions fail for certain effects because the usual monadic translation models call-by-name and not call-by-need. Since modeling the evaluation semantics of call-by-need in the presence of effects like partiality is complex and not necessary to prove propositions for a variety of effects, we identify a specific class of effects for which we cannot observe a difference between call-by-name and call-by-need. Using this class of effects we can prove propositions for all effects that do not require a model of sharing.

CCS Concepts • **Theory of computation → Program verification**; • **Software and its engineering → Functional languages**;

Keywords Haskell, monads, free monad, Coq, verification

1 Introduction

In order to reason about existing Haskell programs in an interactive theorem prover like Coq, we translate Haskell code into Coq code. In preceding work [Dylus et al. 2019] we presented a framework to reason about effectful Haskell programs in Coq. In order to model effects like partiality in Coq we follow an approach by Abel et al. [2005] and lift programs monadically. Abel et al. use Agda programs that are parametric over the monad. Their monadic approach is, however, not applicable anymore, because Agda has become more restrictive in order to prevent logical inconsistencies.[1] Our framework [Dylus et al. 2019] models effects using a free monad [Swierstra 2008] whose functor is a container [Abbott et al. 2003] to model programs that can be instantiated with various effects and pass the termination checker. In contrast to the monadic approach by Abel et al. our framework allows not only to reuse function definitions but to prove propositions that hold for all kinds of effects.

In this paper we observe that proving propositions that are effect-generic comes, however, with its own drawbacks. While proving propositions for all effects is attractive, not all propositions of interest hold for all effects. For example, consider the following Haskell functions.

```
and :: Bool → Bool → Bool
and False _ = False
and True b2 = b2

selfAnd :: Bool → Bool
selfAnd b = and b b
```

The function *selfAnd* is the identity on Boolean values when we explicitly consider total programs. Moreover, this observation is also valid if we consider the partial value *undefined* or an error like *error* "*some message*". We can confirm this observation by proving the corresponding proposition using our framework.

We begin by translating the data type *Bool* to Coq.[2]

```
Inductive Bool :=
| False_ : Bool
| True_ : Bool.
```

In general all arguments of constructors are lifted monadically because in a non-strict language like Haskell an effect may "hide" in every argument of a constructor. We cannot observe this step here, because both constructors are nullary.

The Haskell functions *and* and *selfAnd* are translated into the following Coq code.

```
Definition and (fb1 fb2 : Free C Bool) : Free C Bool :=
  fb1 >>= fun b1 ⇒ match b1 with
                   | False_ ⇒ pure False_
                   | True_ ⇒ fb2
                   end.

Definition selfAnd (fb : Free C Bool) : Free C Bool :=
```

[1] A positivity checker for Agda was implemented in 2006 (see https://github.com/agda/agda/commit/eda120f994b0fb31f9248d0a1cf59e9a895aa988).
[2] We use an underscore as suffix for the constructor names of type *Bool* because Coq already provides definitions named *True* and *False*.

and fb fb.

All arguments of functions as well as their results are lifted monadically. Pattern matching is translated into a combination of a monadic bind on the *Free* value and pattern matching. More precisely, the original pattern matching is performed on the variable *b1*, which is bound to a non-monadic value of type *Bool*. The constructor *False_* of type *Bool* is lifted to a value of type *Free C Bool* using *pure*.

Our framework [Dylus et al. 2019] provides an implementation of a free monad using a container representation for the corresponding functor. That is, instead of parametrizing the free monad by a functor, we use the container extension that is isomorphic to the corresponding functor. For now, it is sufficient to know that a type like *Free C a* represents the free monad over a container *C*. In the following we use *C* to represent arbitrary containers, while containers that correspond to specific effects have a subscript name.

Given a container C_{None} that represents the absence of effects, $C_{Partial}$ that represents partiality and a container C_{Error} for errors, we can prove that *selfAnd* is equivalent to the identity function on Boolean values when considering a setting with one of these effects.

Lemma $selfAnd\_identity_{None}$:
  $\forall\, fb : Free\ C_{None}\ Bool,\ selfAnd\ fb = fb.$

Lemma $selfAnd\_identity_{Partial}$:
  $\forall\, fb : Free\ C_{Partial}\ Bool,\ selfAnd\ fb = fb.$

Lemma $selfAnd\_identity_{Error}$:
  $\forall\, fb : Free\ C_{Error}\ Bool,\ selfAnd\ fb = fb.$

We can, however, construct a counterexample if we consider non-determinism as an effect. If we apply *selfAnd* to a choice between the Boolean values *True* and *False*, the result is not a choice between *True* and *False*. Let us assume an effect-specific function *choice* that chooses non-deterministically between two expressions. Then we can construct a counterexample as follows.

Example $selfAnd\_not\_identity\_counterexample_{ND}$:
  selfAnd (*choice* (*pure False_*) (*pure True_*))
  $\neq$ *choice* (*pure False_*) (*pure True_*).

Based on this example, we can prove the more general proposition that there exists a value in the non-deterministic setting, such that *selfAnd* is not the identity function.

Example $selfAnd\_not\_identity_{ND}$:
  $\exists\, fb : Free\ C_{ND}\ Bool,\ selfAnd\ fb \neq fb.$

In summary, we have seen that *selfAnd* is the identity function on Boolean values for effects like partiality and errors. The function *selfAnd* is, however, not the identity function if we consider a more complex effect like non-determinism. Due to this counterexample, we cannot prove the proposition about *selfAnd* for an arbitrary effect, that is, effect-generic. As a consequence, we observe that there are propositions that only hold for a subset of effects.

This paper builds on our framework [Dylus et al. 2019] and discusses its advantages and limitations when reasoning about Haskell programs with ambient effects. This paper makes the following contributions.

- In Section 3 we show how non-strict tracing as provided by the Haskell function *trace* can be modeled. Then we show that the proposition about *selfAnd* does not only fail if we consider an uncommon effect like non-determinism, but that *selfAnd* is also not equivalent to the identity function on Boolean values in the presence of non-strict tracing.
- In Section 4 we show that while *selfAnd* is not the identity if we consider arbitrary effects, we can restrict the considered effects to a subset of effects where *selfAnd* is the identity. More precisely, we identify a class of effects where we cannot distinguish between call-by-name and call-by-need.
- In Section 5 we discuss how we can model Haskell features that are used to express explicit strictness. In particular we define an effect-generic version of the function *seq*, which evaluates its first argument to head normal form and we discuss strict fields.
- In Section 6 we apply these techniques to prove that the state monad does not satisfy the monad laws if we consider partial values. We also show that the monad laws do not only fail in the case of partial values but for all kinds of effects.

The next section presents the necessary concepts of our framework [Dylus et al. 2019] that are used in the remainder of this paper and revisits the propositions about *selfAnd* presenting the corresponding proofs. The interested reader can find a more detailed explanation about the design decisions of the framework in the original paper.

We assume that the reader is familiar with or at least makes sense of the most basic concepts of Coq[3], namely, data type and function definitions as well as Coq's tactic language. We explain the intuition behind proofs, but refrain from explaining all individual steps in detail. Finally, we use lower case type variables to represent type variables in Haskell, and upper case type variables for meta variables.

2 Reasoning about Haskell Programs

In this section we discuss how our framework [Dylus et al. 2019] models Haskell functions in more detail. We guide this introduction with an example to prove that Boolean negation in Haskell is involutive, that is, that the negation function is its own inverse.

In Section 1 we have seen how the Haskell data type *Bool* is translated into a Coq data type. Next, we consider the following definition of Boolean negation in Haskell.

notb :: *Bool* → *Bool*

[3]The presented Coq code compiles using version 8.9.0 and can be found at https://github.com/ichistmeinname/free-proving-code.

Table 1. Overview of effect functors as well as the corresponding containers and free monads

Functor	Container	Shape	Position	Corresp. Monad
Inductive *Zero A* := .	C_{None}	*Void*	*Void*	*Identity*
Inductive *One A* := *one* : *One A*.	$C_{Partial}$	*unit*	*Void*	*Maybe*
Inductive *Const B A* := *const* : *B* ⟶ *Const B A*.	C_{Error}	*B*	*Void*	*Either B*
Inductive *Log A* := *log* : *string* ⟶ *A* ⟶ *Trace A*.	C_{Trace}	*string*	*unit*	*Trace*
Inductive *Pair A* := *pr* : *A* ⟶ *A* ⟶ *Pair A*.	C_{ND}	*unit*	*bool*	*Tree*

```
notb False = True
notb True = False
```

This definition is translated into the following monadic Coq version, where *C* is an arbitrary container.

```
Definition notb (fb : Free C Bool) : Free C Bool :=
  fb >>= fun b ⇒ match b with
                 | False_ ⇒ pure True_
                 | True_ ⇒ pure False_
                 end.
```

The function *pure* is one of *Free*'s two constructors and used to represent an effect-free head normal form.

The definition of *Free* uses a container [Abbott et al. 2003]. A container provides a set of shapes. For every shape, the container provides the set of valid positions. A prominent example for a container is the list data type. In the case of a list, the shape is the length of the list, the valid positions of a shape are the integers from one to the length of the list.

We define a container as a type class *Container* with associated types *Shape* and *Pos*. *Pos* is a dependent type that depends on a value of type *Shape*.

```
Class Container := {
  Shape : Type;
  Pos : Shape ⟶ Type
}.
```

A container extension provides a function mapping valid positions to values in the container. In the case of a list the function maps an index to the corresponding element in the list. We implement a container extension by the following data type.

```
Inductive Ext (C : Container) (A : Type) :=
| ext : ∀ (s : Shape), (Pos s ⟶ A) ⟶ Ext C A.
```

Finally, the data type *Free*, parameterized by a container *C*, provides two constructors called *pure* and *impure*, of which *impure* models an effectful value. More precisely, the *impure* constructor takes a container extension, modeled using *Ext*, as argument.

```
Inductive Free (C : Container) (A : Type) : Type :=
| pure : A ⟶ Free C A
| impure : Ext C (Free C A) ⟶ Free C A.
```

In the case of *Free* the *Shape* of the container characterizes the set of possible effects.

By choosing a specific container we can model effects like partiality or errors and also reason in an imaginary total version of Haskell. Table 1 gives an overview of effect functors as well as the corresponding containers and the monads that are isomorphic to the corresponding instance of *Free*. For the effects we consider the position types are independent of the shape , that is, *Pos* is a constant function. Thus we only state the result type and omit the function. The container C_{None}, for example, models a total version of Haskell, that is, explicitly represents the absence of any effect. Thus, we use the empty data type *Zero* in this case. As this functor has no constructors, the corresponding container representation uses the empty type *Void* as shape and position type.

The following proof shows that *notb* is involutive in a total version of Haskell.

```
Lemma notb_involutive_None :
  ∀ fb : Free C_None Bool, notb (notb fb) = fb.
Proof.
  intros fb.
  destruct fb as [ b | [ s pf] ].
  (*  fb = pure b  *)
  - destruct b; reflexivity.
  (*  fb = impure (ext s pf)  *)
  - destruct s.
Qed.
```

This proof performs a case distinction: It distinguishes whether the value *fb* is an effectful or pure value. Since the set of possible effects is empty in a total setting, destructing the shape *s* in the *impure* case is equivalent to observing that the empty set does not have an inhabitant.

We can as well model partiality by using an appropriate container. For simplicity, we use a type class function *from*, when we want to construct effectful values. The function *from* takes a functor and yields the corresponding container. There is also a function *to* that takes a container and yields the corresponding functor.

When using a free monad, partiality can be modeled using the functor *One* as shown in Table 1 [Swierstra 2008]. *One* is a data type with a single constructor named *one*, which does not take any arguments. By means of this data type we can define a constant *undefined* that models the corresponding Haskell constant.

```
Definition undefined a : Free C_Partial a :=
  impure (from one).
```

The following proof shows that negation is involutive in Haskell if we consider partiality as an effect.

```
Lemma notb_involutive_Partial :
  ∀ fb : Free C_Partial Bool, notb (notb fb) = fb.
Proof.
  intros fb.
  destruct fb as [ b | [ s pf ] ].
  (*  fb = pure b  *)
  - destruct b; reflexivity.
  (*  fb = impure (ext s pf)  *)
  - simpl.
    do 2 f_equal. extensionality p. destruct p.
Qed.
```

We again perform a case distinction to check whether the value *fb* is a pure or effectful value; in this case the effect is the absence of a value. In contrast to the proof in a total setting we now have to actually handle the effect but do not have a value at hand. We remove equal constructors from both sides of the equation and show extensional equality of the position functions by introducing a position variable *p*. Since the position type for $C_{Partial}$ is *Void*, we then conclude the proof by using `destruct` again.

Moreover, we can model other effects like errors as well. For example, we can define a function that models the *error* function in Haskell as follows. Here *const* is the constructor introduced in Table 1.

```
Definition error a (msg : string) : Free C_Error a :=
  impure (from (const msg)).
```

Using the container C_{Error} we can prove that *notb* is also involutive when we consider a language with errors.

```
Lemma notb_involutive_Error :
  ∀ fb : Free C_Error Bool, notb (notb fb) = fb.
```

The proof in the case of errors is the same as in the case of partiality because they are similar effects. They both extend the inhabitants of every type with an exceptional value.

These simple examples demonstrate that our framework [Dylus et al. 2019] enables the reuse of the generated Coq code to prove propositions considering various effects. While it is convenient that we can reuse the Coq code, in contrast to previous work [Abel et al. 2005] our framework also allows proving effect-generic propositions. For example, the following proposition and corresponding proof show that *notb* is involutive no matter which container and effect, respectively, we are considering.

```
Lemma notb_involutive :
  ∀ fb : Free C Bool, notb (notb fb) = fb.
Proof.
  intros fb.
  induction fb as [ b | s pf IH ].
  (*  fb = pure b  *)
  - destruct b; reflexivity.
  (*  fb = impure (ext s pf)  *)
  - simpl.
    do 2 f_equal. extensionality p.
    apply IH.
Qed.
```

This proof is quite similar to the proofs before. The main difference is that we perform an induction instead of a simple case distinction and, therefore, have to apply the induction hypothesis. We have to perform an induction because in general the considered effect might be recursive as, for example, in the case of non-determinism. Because the proof is effect-generic, we even know that *notb* is involutive when we consider an effect like non-strict tracing.

As mentioned in the introduction, proving propositions that are effect-generic comes with its own drawbacks. Although we can define and prove propositions that hold for all effects, not every proposition of interest holds for all effects. Hence, we reconsider the Haskell functions *and* as well as *selfAnd* and their Coq translation defined in Section 1. In the case of partiality or errors *selfAnd* is the identity for Boolean values. We can formalize that *selfAnd* is the identity if we consider partiality as follows. As the proof in the case of errors is the same, we leave it out.

```
Lemma selfAnd_identity_Partial :
  ∀ fb : Free C_Partial Bool, selfAnd fb = fb.
Proof.
  intros fb.
  destruct fb as [ b | [ s pf ] ].
  (*  fb = pure b  *)
  - simpl. destruct b; reflexivity.
  (*  fb = impure (ext s pf)  *)
  - unfold selfAnd; simpl.
    do 2 f_equal. extensionality p. destruct p.
Qed.
```

In order to demonstrate that this proposition holds when considering effects like partiality and errors but does not hold for an arbitrary effect, we again consider the effect of non-determinism. In order to model non-determinism we define the function that represents a non-deterministic choice as follows.

```
Definition choice a (x y : Free C_ND a) : Free C_ND a :=
  impure (from (pr x y)).
```

Table 1 contains the definition of C_{ND} and the pair constructor *pr*, which allows us to represent non-determinism as a binary tree monad.

In Section 1 we have shown that *selfAnd* is not the identity if we consider a choice between *False_* and *True_*. We formalize this counterexample as follows.

```
Example selfAnd_not_identity_ND :
  ∃ (fb : Free C_ND Bool), selfAnd fb ≠ fb.
Proof.
  ∃ (choice (pure False_) (pure True_)).
  intros Heq. dependent destruction Heq.
  apply equal_f with (x0 := false) in x. inversion x.
Qed.
```

In the first line of the proof we provide the counterexample. Afterwards we have to show that it is actually a counterexample. Due to the definition of ≠ we need to construct something of type *False*. We transform the hypothesis *Heq* until we have an equation with different constructors on both sides. To finish the proof, we observe this obvious inequality with `inversion`. We primarily present the proof here because we will later see other proofs that are quite similar.

3 Call-by-Name vs. Call-by-Need

As $selfAnd\_not\_identity_{ND}$ demonstrates *selfAnd* is not equivalent to the identity function if we consider non-determinism. However, Haskell does not provide non-determinism as an ambient effect. Nevertheless, the proposition does not only fail in the case of non-determinism, but also in the case of non-strict tracing as provided by the Haskell function *trace*.

In order to demonstrate this behavior in Coq we first have to define the tracing effect as illustrated in Table 1. The tracing effect is modeled by a constructor that captures the *string* that is printed on the console. When we instantiate *Free* with C_{Trace}, the second argument of the constructor *log* has the type *Free* C_{Trace} τ for some τ. That is, in the functor representation logging multiple messages is modeled by a stack of values of type *Log*.

By means of C_{Trace} we can now define a function that models Haskell's *trace* function. Note that for simplicity the *string* that is logged to the console is effect-free. If we want to model the function *trace* accurately, we would have to use an argument of type *Free* C_{Trace} (*List* C_{Trace} *Char*) where the data type *List* is lifted as demonstrated before [Dylus et al. 2019] and *Char* is some Coq data type modeling the character data type in Haskell.

```
Definition trace a (msg : string) (fx : Free C_Trace a) :=
  impure (from (log msg fx)).
```

The function *selfAnd* is not the identity function, if we consider the effect of non-strict tracing, as the following example demonstrates.

```
Example selfAnd_not_identity_Trace :
  ∃ (fb : Free C_Trace Bool), selfAnd fb ≠ fb.
Proof.
  ∃ (trace "debug" (pure True_)).
  intros Heq. dependent destruction Heq.
  apply equal_f with (x0 := tt) in x. inversion x.
Qed.
```

In order to illustrate why the proposition fails in the presence of *trace* we consider the following example.

```
Example trace_twice :
  selfAnd (trace "debug" (pure True_))
  = trace "debug" (trace "debug" (pure True_)).
```

Applying *selfAnd* to the result of applying *trace* to the pure value *True_* is equivalent to performing the tracing effect twice and yielding the pure result *True_*. The effect is performed for the first time because the function *and* pattern matches on its first argument and a second time because *and* yields its second argument.

When we evaluate the corresponding Haskell expression, the message "debug" is only logged once on the console. Because *selfAnd* shares its argument, the effect is only evaluated once. The model we are considering here, however, models call-by-name and not call-by-need. That is, in our model

```
let b = trace "debug" True_ in and b b
```

and

```
and (trace "debug" True_) (trace "debug" True_)
```

behave the same while they do not in Haskell.

The function *selfAnd* shares an effectful value, namely the argument *fb*, by using it twice. Therefore, we may not share a value of type *Free C Bool* for an arbitrary container *C*. We can, however, not observe a difference between call-by-name and call-by-need if we only share effect-free values of, for example, type *Bool*. This observation is illustrated by applying a transformation known as *let-to-case* as discussed by Santos [1995] to the definition of *and*. More precisely, we can replace an expression of the form `let` $x = e$ `in` e' by `case` e `of` $v \rightarrow e'$ if e' is strict in x. In the definition of *selfAnd*, *and* is strict in its first argument. Therefore, the following definition is equivalent to *selfAnd*.

```
selfAnd' :: Bool → Bool → Bool
selfAnd' b = case b of
               b' → and b' b'
```

This definition is translated into the following Coq definition.

```
Definition selfAnd' (fb : Free C Bool) : Free C Bool :=
  fb >>= fun b ⇒ and (pure b) (pure b).
```

The function *selfAnd'* only shares the effect-free value *b*. As it does not share effectful values, we can prove that it is equivalent to the identity if we consider an arbitrary effect.

```
Lemma selfAnd'_identity :
  ∀ fb : Free C Bool, selfAnd' fb = fb.
```

Applying the *let-to-case* transformation only illustrates the problem and does not propose a solution to modeling call-by-need correctly. First, when altering the definition of *selfAnd*, we would need some form of formal semantics to verify that *selfAnd* and *selfAnd'* behave equivalently in a call-by-need setting. Second, we cannot apply this kind of transformation in all cases. For example, the effect we are sharing may be buried in a data structure.

Consider the following Haskell program as an example of effectful components.

```
data Dummy = D { value : Bool }

shareDummy :: Bool → Bool
shareDummy b = let d = D b in and (value d) (value d)
```

As the expression *and* (*value d*) (*value d*) is strict in *d* we can use the following equivalent definition.

```
shareDummy' :: Bool → Bool → Bool
shareDummy' b =
  case D b of
       d → and (value d) (value d)
```

We model the Haskell data type *Dummy* in Coq as follows. Note that the argument is lifted monadically, because in a non-strict language like Haskell an effect may "hide" in the arguments of a constructor.

```
Inductive Dummy :=
| D : Free C Bool → Dummy.
```

```
Definition value (fd : Free C Dummy) : Free C Bool :=
  fd >>= fun d ⇒ match d with
                 | D fb ⇒ fb
                 end.
```

Please note that ambient monadic effects in Haskell behave differently than the corresponding encoding of effects via monads in Haskell, as shown by the necessity to lift arguments of constructors monadically in the translation from Haskell to Coq. That is, in the case of an ambient effect the effect might not be performed because of non-strictness. With this respect effects that are modeled via a monad in Haskell are more strict. For example, Fischer et al. [2009] discusses the difference between this form of non-strict effect and the corresponding monad in Haskell in the case of non-determinism.

The Haskell function *shareDummy'* is translated into the following Coq definition.

```
Definition shareDummy' (fb : Free C Bool) : Free C Bool :=
  pure (D fb) >>= fun d ⇒ and (value (pure d)) (value (pure d)).
```

While the outermost level of *d* is effect-free, the argument of the constructor *D* may contain an effect that is shared by using the variable *d* twice. That is, in this example we are still sharing an effectful value. If we apply *shareDummy'* to an effect we are still duplicating the effect as the following example demonstrates.

```
Example shareDummy_trace_twice :
  shareDummy' (trace "debug" (pure True_))
  = trace "debug" (trace "debug" (pure True_)).
```

If we want to model call-by-need correctly, we could try to transfer an approach by Fischer et al. [2009] to Coq. Fischer et al. present a library for modeling the lazy functional logic programming language Curry in Haskell. Curry is lazy and provides non-determinism with call-time choice. Call-time choice means that if we bind a variable to a non-deterministic computation and use the variable multiple times, the variable always performs the same non-deterministic choice. In order to model this behavior correctly Fischer et al. model sharing explicitly. However, as they use an untyped heap using *unsafeCoerce*, it is not obvious how to transfer this approach to Coq. Furthermore, if we model sharing explicitly, we also have to consider it when performing proofs. While modeling sharing explicitly is a direction for future research, here we will instead restrict the effects we are considering to the subset of effects that do not let us observe a difference between call-by-need and call-by-name. This way, we can still prove propositions that are effect-generic over all effects from the corresponding subset. That is, we can still perform a single proof for a proposition and know that it holds for total programs as well as the effects of partiality and errors. On the downside the proposition may fail in the presence of more complex effects like tracing, where we can observe the duplication of effects.

While it would be attractive to be able to model call-by-value in our framework, in order to do so we would need a different transformation. For example, in the case of *and* as defined in Section 1 we would have to evaluate both arguments in order to model call-by-value correctly. As the transformation is quite fundamental, modeling call-by-value is kind of orthogonal to modeling call-by-name/call-by-need.

4 Simple Containers

As mentioned in Section 3 tracing can be modeled by a free monad with the following functor.

```
Inductive Log (A : Type) : Type :=
| log : string → A → Log A.
```

It is common to define tracing as a simplified version of the state monad using a *string* as state, as for example modeled by Liang et al. [1995]. Modeling the trace effect using *Free* C_{Trace}, the construction is isomorphic to the following monad *Trace* as presented by Piróg and Gibbons [2012]. A pure value *x* of type *A* corresponds to *Nil x*, whereas a tracing messages are stacked using the constructor *LCons*.

```
Inductive Trace (A : Type) :=
| Nil : A → Trace A
| LCons : string → Trace A → Trace A.
```

In order to use tracing we have to model the functor *Log* as a container.

```
Global Instance C_Trace : Container :=
  {
    Shape := string;
    Pos := fun _ ⇒ unit;
  }.
```

We encode the message that is logged in the shape of the container; here, the shape is a synonym for the data type *string*. Furthermore, the constructor *log* contains a single argument of type *A*. The corresponding function yields the type *unit* as the position type independent of the shape.

To validate that we have correctly modeled the functor *Log*, we define two functions, namely *fromLog* and *toLog*, that form a bijection between the two representations. The functions *fromLog* and *toLog* implement the type class functions *from* and *to*.

```
Definition toLog A (e : Ext C_Trace A) : Log A :=
  match e with
  | ext str p ⇒ log str (p tt)
  end.
```

```
Definition fromLog A (t : Log A) : Ext C_Trace A :=
  match t with
  | log str x ⇒ ext str (fun p ⇒ x)
  end.
```

The function that maps positions to values in the container is given by a constant function because for every shape there is only a single position. In case of reconstructing a *Log* for a given container extension we apply the mapping function *p* to the only valid position *tt* of type *unit* in order to access the value of type *A*.

The following propositions state that *fromLog* and *toLog* form a bijection.

```
Lemma to_from_Log :
  ∀ (A : Type) (fx : Log A), toLog (fromLog fx) = fx.
```

```
Lemma from_to_Log :
  ∀ (A : Type) (e : Ext C_Trace A), fromLog (toLog e) = e.
```

Now the question arises why we can distinguish call-by-name and call-by-need in the case of tracing but cannot distinguish it for other effects like partiality. In general we can distinguish call-by-name and call-by-need if an effectful computation is shared, because the effect is performed multiple times in the case of call-by-name, while it is only performed once in the case of call-by-need. In the free monad multiple occurrences of an effect are modeled by stacking the corresponding functor. For example, performing a tracing effect twice results in a value like *trace* "debug" (*trace* "debug" (*pure True_*)). Therefore, we cannot distinguish between call-by-name and call-by-need if we cannot stack the effect, that is, if the functor's type parameter *A* is not used in the argument types of its constructors.

Now recall that the functor is represented by a container. The possible positions of the container correspond to occurrences of the functor's type parameter in the argument types of its constructors. Therefore, we cannot stack a container if the position type is empty for every shape. A container is simple if it satisfies the following proposition, which states that for every *Shape* there is no position.

```
Definition Simple (C : Container) : ℙ :=
  ∀ (s : Shape) (p : Pos s), False.
```

If we take a look at the example $selfAnd\_not\_identity_{Trace}$ again, which demonstrates that *selfAnd* is not the identity in the case of tracing, we see that we provide a witness of the position type. In the case of tracing the witness is *tt*, which is the only value of the position type *unit*. This witness demonstrates that the corresponding functor can be stacked. The same holds for $selfAnd\_not\_identity_{ND}$ where we provide the position *false* as a witness, which is one of two possible positions.

Now that we can identify simple effects, where we cannot distinguish between call-by-name and call-by-need, we take a closer look at the effects we considered so far. We show that no effect, partiality, and errors are simple effects. We leave out the proof for $C_{Partial}$ as it is the same as the proof for C_{Error}. Please note that the proofs for C_{None} and C_{Error} differ, because in the case of C_{None} there is not even a shape that we need to consider, while in the case of C_{Error} we show for every shape that there is no position.

```
Lemma None_Simple :
  Simple C_None.
Proof.
  intros s. destruct s.
Qed.
```

```
Lemma Error_Simple :
  Simple C_Error.
Proof.
  intros s p. destruct p.
Qed.
```

In contrast to the above effects tracing and non-determinism are not simple. In order to prove this statement we have to provide a shape and a corresponding position.

```
Lemma Trace_not_Simple :
  ¬ (Simple C_Trace).
Proof.
  intros HSimple.
  specialize (HSimple "debug" tt).
  apply HSimple.
Qed.
```

```
Lemma ND_not_Simple :
  ¬ (Simple C_ND).
Proof.
  intros HSimple.
  specialize (HSimple tt false).
  apply HSimple.
Qed.
```

In both proofs we need to construct something of type *False* due to the definition of ¬; in both cases the provided arguments lead to the assumption *HSimple* : *False*, which we then use to complete the proof.

Besides these counterexamples for simple effects that confirm our observations about the difference between call-by-name and call-by-need, we can also prove positive statements. For example, we can prove that *selfAnd* is equivalent to the identity if we only consider effects that are simple. That is, there is a single proof that subsumes the proofs for no effect, partiality, and error.

```
Lemma selfAnd_identity_Simple :
  Simple C → ∀ (fb : Free C Bool), selfAnd fb = fb.
Proof.
  intros HSimple fb.
  unfold selfAnd.
  destruct fb as [ b | [ s pf ] ]; simpl.
  (*  fb = pure b  *)
  - destruct b; reflexivity.
  (*  fb = impure (ext s pf)  *)
  - do 2 f_equal.
    extensionality p.
    specialize (HSimple s p).
    inversion HSimple.
Qed.
```

In the case of an *impure* value we take advantage of the fact that the container is simple in order to show that the *impure* value cannot contain another value.

An alternative way to look at a simple container is the observation that all functions that map positions to elements coincide. That is, two *impure* values of a simple container coincide as long as the shapes are equal. The following lemma captures this observation.

```
Lemma eq_impure :
  Simple C → ∀ a (s : Shape) (pf1 pf2 : Pos s → Free C a),
    impure (ext s pf1) = impure (ext s pf2).
```

We can use the lemma *eq_impure* to prove the inductive case in proofs like $selfAnd\_identity_{Simple}$.

We can also show that being *Simple* is necessary for *selfAnd* to be equivalent to the identity. This proof is quite similar to the counterexamples of *selfAnd* being the identity before. That is, we have to provide a position as a witness.

```
Lemma selfAnd_identity_Simple:
  (∀ fb, selfAnd fb = fb) → Simple C.
Proof.
  intros Heq s p.
  specialize (Heq (impure (ext s (fun p ⇒ pure True_)))).
  dependent destruction Heq.
  apply equal_f with (x0 := p) in x.
  inversion x.
Qed.
```

In order to prove this proposition we assume that *selfAnd* is equivalent to the identity function. This hypothesis is called *Heq* in the proof. Then we instantiate this hypothesis with a concrete value, namely *impure* (*ext s* (fun *p* ⇒ *pure True_*)). This value is the generalization of the counterexamples used in propositions like $selfAnd\_not\_identity_{Trace}$. When we apply *selfAnd* to this value, the effect is performed at least twice in the case that the corresponding position type has an inhabitant.

5 Modeling Explicit Strictness

As we can use our framework to model partial values, we are also interested in modeling Haskell language features that are associated with explicit strictness. In the remainder of this section, we will take a look at the function *seq* and how to model type definitions with strict fields.

Modeling seq While intuitively the *seq* function is closely connected to partiality, we can provide a single definition of *seq* that works for all kinds of effects. We can define *seq* effect-generically using the bind operator of the *Free* monad, which, intuitively, models evaluation of a value to head normal form. That is, we define *seq* by evaluating its first argument to head normal form and yielding its second argument.

```
Definition seq a b (fx : Free C a) (fy : Free C b) : Free C b :=
 fx >>= fun _ ⇒ fy.
```

In order to illustrate the behavior of the definition, we first show that it satisfies the specification of the *seq* function, which is often specified as follows.

```
seq :: a → b → b
seq ⊥ y = ⊥
seq x y = y, if x ≠ ⊥
```

We can formalize this specification by the following two Coq propositions.

```
Example seq_spec1 :
  ∀ a b (fy : Free C_Partial b),
    seq (@undefined a) fy = undefined.
```

```
Example seq_spec2 :
  ∀ a b (fx : Free C_Partial a) (fy : Free C_Partial b),
    fx ≠ undefined → seq fx fy = fy.
```

In both proofs we use the proposition *eq_impure* in order to show that arbitrary *impure* values are equal in the case of partiality.

As the definition of *seq* does not only work for partiality but for arbitrary effects, we take a look at using *seq* in the case of tracing and non-determinism.

```
Example seq_Trace :
  ∀ a b (x : a) (fy : Free C_Trace b),
    seq (trace "debug" (pure x)) fy = trace "debug" fy.
```

When we apply *seq* to a tracing effect, we perform the effect but replace the pure value by the second argument of *seq*. The behavior is exactly the same in Haskell, when we evaluate the expression *seq* (*trace* "debug" *x*) *y* for any effect-free value *x* and arbitrary value *y*.

We observe a similar behavior for non-determinism. When we *seq* a non-deterministic computation, the non-determinism is pulled to the outside and pure values are replaced by the second argument of *seq*.

```
Example seq_ND :
  ∀ a b (x1 x2 x3 : a) (fy : Free C_ND b),
    seq (choice (pure x1) (choice (pure x2) (pure x3))) fy
    = choice fy (choice fy fy).
```

The same behavior can be observed when we use *seq* in a functional logic language like Curry [Antoy and Hanus 2010], which supports non-strict non-determinism.

Adding *seq* to a programming language comes at a price. First, free theorems, that is, propositions that are derived from the type of a function as popularized by Wadler [1989], only hold with additional side conditions [Johann and Voigtländer 2004]. Besides requiring additional care when reasoning about polymorphic functions we can also use *seq* to distinguish expressions that are not distinguishable without *seq*. Previously, we have modeled a Haskell function of type $\tau_1 \rightarrow \tau_2$ via a Coq function of type $Free\ C\ \tau_1 \rightarrow Free\ C\ \tau_2$ for some container *C* [Dylus et al. 2019]. However, in the presence of *seq* we have to model a Haskell function of type $\tau_1 \rightarrow \tau_2$ via a Coq function of type $Free\ C\ (Free\ C\ \tau_1 \rightarrow Free\ C\ \tau_2)$.

As mentioned before we can use *seq* to distinguish expressions that are not distinguishable without *seq*. More precisely, we can use *seq* to distinguish an *undefined* function from a function that always yields *undefined*. First, we define these two functions as well as a type synonym *Fun* for lifted function types.

```
Definition Fun C A B := Free C A → Free C B.
```

```
Definition fun1_Partial : Free C_Partial (Fun C_Partial Bool Bool) :=
  undefined.
```

```
Definition fun2_Partial : Free C_Partial (Fun C_Partial Bool Bool) :=
  pure (fun _ ⇒ undefined).
```

Second, we add a helper definition to apply a lifted function to its argument as well as an infix notation for a more convenient usage.

```
Definition app a b (ff : Free C (Fun C a b)) (fx : Free C a) :=
  ff >>= fun f ⇒ f fx.
```

```
Infix "$" := app (at level 28, left associativity).
```

First we show that we cannot distinguish *fun1* and *fun2* by simply applying them to the same argument.

```
Example same_result_Partial :
  ∀ (fb : Free C_Partial Bool), fun1_Partial $ fb = fun2_Partial $ fb.
```

While we get the same results when we apply these two functions to an argument, we can distinguish them using *seq*.

```
Example distinguish_seq_Partial :
  seq fun1_Partial (pure True_) ≠ seq fun2_Partial (pure True_).
```

As *seq* evaluates its first argument to head normal form, the left-hand side simplifies to *impure* (*from one*) and the right-hand side to *pure True_*. That is, the proposition states that a value constructed using *pure* is equivalent to a value constructed using *impure*, which is a contradiction.

As *seq* can be used for arbitrary effects, we observe a similar behavior for tracing and non-determinism. We can define functions that are analogous to $fun1_{Partial}$ and $fun2_{Partial}$ in the case of tracing and non-determinism, respectively. In the following, instead of presenting examples for both effects we only show code for the former, as the definitions for non-determinism are analogous to tracing.

```
Definition fun1_Trace : Free C_Trace (Fun C_Trace Bool Bool) :=
  trace "debug" (pure (fun _ ⇒ pure True_)).
```

```
Definition fun2_Trace : Free C_Trace (Fun C_Trace Bool Bool) :=
  pure (fun _ ⇒ trace "debug" (pure True_)).
```

While we cannot distinguish these functions when we apply them to an argument, we can, again, distinguish them using *seq*.

```
Example same_result_Trace :
  ∀ (fb : Free C_Trace Bool), fun1_Trace $ fb = fun2_Trace $ fb.
```

```
Example distinguish_seq_Trace :
  seq fun1_Trace (pure True_) ≠ seq fun2_Trace (pure True_).
```

As we are also able to reason about effects more generally, we state generalizations of the propositions considered before. The generic versions of the two functions are parametrized over the shape of the effect in order to construct an effectful value using *impure*.

```
Definition fun1 s : Free C (Fun C Bool Bool) :=
  impure (ext s (fun _ ⇒ pure (fun _ ⇒ pure True_))).
```

```
Definition fun2 s : Free C (Fun C Bool Bool) :=
  pure (fun _ ⇒ impure (ext s (fun _ ⇒ (pure True_)))).
```

That is, when we generalize the proofs to arbitrary effects, we have to provide a shape to construct the impure value.

```
Example same_result :
  ∀ s (fb : Free C Bool), fun1 s $ fb = fun2 s $ fb.
```

```
Example distinguish_seq :
  ∀ s, seq (fun1 s) (pure True_) ≠ seq (fun2 s) (pure True_).
```

These propositions demonstrate that as long as we can provide a shape, both statements hold for arbitrary effects. Because we have to provide a shape, we cannot specialize the above propositions to the container C_{None}.

Types with strict fields There are two more Haskell features related to strictness: bang patterns in type declarations and *newtype*s. In order to illustrate their behavior we consider the following basic Haskell type definitions.

```
data Foo1 = Foo1 Int
data Foo2 = Foo2 !Int
newtype Foo3 = Foo3 Int
```

These Haskell data types are modeled by the following Coq data types.

```
Inductive Foo1 :=
| Foo1_ : Free C Int → Foo1.
```

```
Inductive Foo2 :=
| Foo2_ : Int → Foo2.
```

```
Definition Foo3 := Int.
```

Because the data type *Foo2* is strict in its argument due to the bang pattern, we do not lift the argument of *Foo2_*. This translation expresses the invariant that the argument of *Foo2_* is always a pure computation. In the case of *Foo3* there is no constructor at all because in Haskell a *newtype* is semantically equivalent to its argument type. Therefore, we define a type synonym for *Foo3* and all constructors are removed.

In order to illustrate the behavior of *Foo1* and *Foo2* we consider the following example, once for *Foo1* and once for *Foo2*.

```
xFoo1 = case Foo1 undefined of
          Foo1 _ → 1
```

This example is translated into the following Coq code.

```
Definition xFoo1 :=
  pure (Foo1_ undefined) >>= fun x ⇒ match x with
                                     | Foo1_ _ ⇒ pure 1
                                     end.
```

In contrast, when a constructor with a strict field like *Foo2* is applied to an argument, we have to check whether the argument is an effectful computation. In order to model this behavior we define an operator for strict application and use this operator whenever *Foo2_* is applied.

```
Definition bangAppC a b (ff : Free C (a → b)) (fx : Free C a) :=
  ff >>= fun f ⇒ fx >>= fun x ⇒ pure (f x).
```

```
Infix "$$!" := bangAppC (at level 28, left associativity).
```

Please note that the functional argument of *bangAppC* is wrapped monadically. Therefore, in the definition of *xFoo2* we have to use *pure* to lift the constructor to a monadic value. While this example would be simpler without this monadic context, it simplifies working with multiple arguments.

```
Definition xFoo2 :=
  (pure Foo2_ $$! undefined) >>= fun x ⇒ match x with
                                          | Foo2_ _ ⇒ pure 1
                                          end.
```

Note that we could as well use *seq* to evaluate the argument of *Foo2_* to head normal form before applying the constructor. However, in this case, we would have to lift the argument type of *Foo2_* monadically.

6 Case Study

In this section we consider a concrete example of proving propositions about effectful Haskell programs. Jeuring et al. [2012] present a framework for using the random testing tool QuickCheck to test monad laws in Haskell. In particular they consider partiality and show that the standard implementations of the state monad do not satisfy the monad laws. In the remainder of this section, all Haskell definitions, examples and associated laws are taken from Jeuring et al.

We consider the following implementation of the state monad in Haskell. While Jeuring et al. define a type class *MonadState*, which provides the functions *get* and *put*, for simplicity, we refrain from using a type class here.

```
data Pair a b = Pair a b
newtype State s a = S {runS :: s → Pair a s}
get :: State s s
get = S (\s → Pair s s)

put :: s → State s ()
put s = S (const (Pair () s))
```

This Haskell program is modeled in Coq as follows.[4]

```
Inductive Pair a b :=
| Pair_ : Free C a → Free C b → Pair a b.

Definition State s a := Free C s → Free C (Pair a s).

Definition runS s a (fs : Free C (State s a)) := fs.

Definition get s : Free C (State s s) :=
  pure (fun fs ⇒ pure (Pair_ fs fs)).

Definition put s (fs : Free C s) : Free C (State s Unit) :=
  const (pure (Pair_ (pure Unit_) fs)).
```

Note that since *State* is a *newtype*, it is transformed into a type synonym and that all constructors of this type are removed. Furthermore, we monadically lift the function type *s → Pair a s* in the definition of *State*.

The following Coq type class models the *Monad* type class.

```
Class Monad (m : Type → Type) := {
  ret a : Free C a → Free C (m a);
  bind a b : Free C (m a) → Free C (Fun C a (m b)) → Free C (m b)
}.
```

Please note that we refrain from lifting all function arrows in the types of *ret* and *bind* in order to simplify the presentation. That is, we cannot use this definition to model all possible instances of the type class *Monad* in Haskell. For example, the definition *return* = *undefined* would be type-correct in Haskell but cannot be modeled by our Coq type class because this would require *ret* to be of type *Free C* (*Fun C* (*Free C a*) (*Free C* (*m a*))).

When implementing a monad instance for *State*, there are two possible instances that vary with respect to strictness. In contrast to Haskell we can define two instances for the same type in Coq, because instances are associated with a name.

We define the following two monad instances.

```
Instance monad_State_strict s : Monad (State s) := {
  ret a fx := pure (fun fs ⇒ pure (Pair_ fx fs));
  bind a b fm fk := pure (fun fs ⇒ runS fm $ fs >>= fun p ⇒
                          match p with
                          | Pair_ fx fs' ⇒ runS (fk $ fx) $ fs'
                          end)
}.

Instance monad_State_lazy s : Monad (State s) := {
  ret a fx := pure (fun fs ⇒ pure (Pair_ fx fs));
  bind a b fm fk := pure (fun fs ⇒ let fp := runS fm $ fs
                                   in runS (fk $ fst fp) $ snd fp)
}.
```

The Haskell definition of the lazy instance uses a `let` pattern binding. We model this pattern binding using selector functions *fst* and *snd*.

As a next step we will verify that both of our implementations satisfy certain laws. We start with laws concerning the functions *get* and *put*. The proofs that our implementation satisfies these properties are all straightforward and only rely on rewriting. We prove all laws generically, that is, they hold no matter which effects we consider. The constant *skip* is defined as *return* ().

```
Lemma MSPutPutLaw : ∀ s (fs fs' : Free C s),
    bind (put fs') (pure (fun _ ⇒ put fs)) = put fs.

Lemma MSPutGetLaw : ∀ s (fs fs' : Free C s),
    bind (put fs) (pure (fun _ ⇒ get))
    = bind (put fs) (pure (fun _ ⇒ ret fs)).

Lemma MSGetPutLaw : ∀ s,
    bind (@get _ s) (pure (fun fs ⇒ put fs)) = skip.

Lemma MSGetGetLaw :
  ∀ a s (fk : Free C (Fun C s (Free C s → Free C (State s a)))),
    bind get (pure (fun fs ⇒ bind get (fk $ fs)))
    = bind get (pure (fun fs ⇒ fk $ fs $ fs)).
```

Jeuring et al. also show that both implementations do not satisfy the monad laws if we consider partiality. Here, we only consider the first monad law, but we can also prove that

[4] We use an underscore as suffix for the constructor names of type *Pair* and *Unit* because constructors and types share a namespace in Coq.

the second law fails using similar counterexamples. Because we want to reuse the laws for two instances of the type class *Monad*, we define propositions that take an instance of the type class *Monad* as additional argument.

```
Definition MonadLaw1 m `(Monad m) :=
  ∀ a b (fx : Free C a) (ff : Free C (Fun C a (m b))),
    bind (ret fx) ff = ff $ fx.
```

In order to prove that the first monad law fails if we consider partiality, we explicitly instantiate the monad instance with the container $C_{Partial}$. We construct the following counterexamples.

```
Lemma MonadLaw1_State_lazy_fail_Partial :
  ∀ s, ¬ (MonadLaw1 (monad_State_lazy C_Partial s)).
Proof.
  intros s Heq.
  specialize (Heq _ Unit (pure False_) undefined).
  inversion Heq.
Qed.
Lemma MonadLaw1_State_strict_fail_Partial :
  ∀ s, ¬ (MonadLaw1 (monad_State_strict C_Partial s)).
Proof.
  intros s Heq.
  specialize (Heq _ Unit (pure False_) (const undefined)).
  inversion Heq.
Qed.
```

The first monad law fails for both *State* instances, because the left-hand side is *undefined* while the right-hand side is a function that always yields *undefined*.

While the state monad does not satisfy the first monad law if we consider partiality, monad laws are often verified in a total setting. We can prove that the two implementations satisfy the first monad law in a total setting by using the container C_{None} instead of $C_{Partial}$.

```
Lemma MonadLaw1_State_lazy_None :
  ∀ s, MonadLaw1 (monad_State_lazy C_None s).
Lemma MonadLaw1_State_strict_None :
  ∀ s, MonadLaw1 (monad_State_strict C_None s).
```

While Jeuring et al. are only interested in partiality, we are interested in all kinds of effects. We can, for example, also construct a counterexample if we consider tracing or non-determinism instead of partiality. For convenience, the following proofs use a helper function *constSkip* that constructs a constant function yielding a value of type *State s Unit*.

```
Definition constSkip s : Free C (Fun C Bool (State s Unit)) :=
  const (skip (monad_State_lazy C s)).
Lemma MonadLaw1_State_lazy_fail__Trace :
  ∀ s, ¬ (MonadLaw1 (monad_State_lazy C_Trace s)).
Proof.
  intros s Heq.
  specialize (Heq _ _ (pure False_)
                      (trace "debug" (constSkip C_Trace s))).
  inversion Heq.
Qed.
Lemma MonadLaw1_State_lazy_fail_ND :
  ∀ s, ¬ (MonadLaw1 (monad_State_lazy C_ND s)).
```

In the case of tracing, the-left hand side of the first monad law yields the function _ ⟶ *trace* "debug" () while the right-hand side yields *trace* "debug" (_ ⟶ ()).

Finally, we use the advantages of our modeling and generalize the counterexample to arbitrary effects. Note that the previous three effect-specific counterexamples follow the same idea to contradict the proposition. Based on this observation we construct a generic counterexample by replacing the tracing effect by an *impure* value that always yields *constSkip*.

```
Lemma MonadLaw1_State_lazy_fail :
  ∀ (shape : Shape) s, ¬ (MonadLaw1 (monad_State_lazy C s)).
Proof.
  intros shape s Heq.
  specialize (Heq _ _ (pure False_)
                      (impure (ext shape (fun _ ⇒ constSkip C s)))).
  inversion Heq.
Qed.
```

7 Related Work

There are two main categories of related work. The first category describes how to reason about effectful programs in an interactive theorem prover like Coq using a monadic model. The second category focuses on ideas to reason about effectful Haskell programs using tools like Coq.

7.1 Monadic Modeling of Effects

Abel et al. [2005] propose a translation of Haskell programs into monadic Agda programs to reason about existing Haskell code. In order to model effects Abel et al. translate Haskell programs into monadic Agda programs that are parametrized over a monad *m*. By instantiating the variable *m* with the identity or the maybe monad they use the same Agda program to reason about a Haskell program in a total or a partial setting. Due to Haskell's non-strictness, the translation of data types to Agda lifts the arguments of all constructors monadically as well. When translating recursive data types like lists, however, the resulting lifted data types are not allowed in Agda or Coq anymore. These data type definitions are parametrized by an arbitrary monad and, thus, not allowed as they might be used to define a non-terminating expression. When Abel et al. developed their approach, this restriction was not yet enforced by the Agda compiler.

The work by Abel et al. was the motivation for us to develop a new approach that works in current versions of Agda and Coq [Dylus et al. 2019]. Instead of using a type variable to parametrize the definition over an arbitrary monad, we use a free monad to model various kinds of effects. As the common definition of a free monad parametrized by a functor cannot be defined in Coq either, we model the functor of the free monad by a container [Abbott et al. 2003]. We are not the first to use a free monad with a container to model effects. McBride [2015] presents the *General* monad to reason about possibly non-terminating functions in Agda. The *General* monad is a free monad with a container as functor where the container extension is inlined into the definition of the free monad. That is, our results can as well be applied to the *General* monad. In contrast to the results presented here, McBride is mainly interested in defining arbitrary recursive functions by modeling the recursive structure of a function. Here, we actually leave out all aspects concerned with non-termination or infinite values of recursive data structures.

Koh et al. [2019] verify a networked server implemented in C using *interaction trees*. An *interaction tree* is a freer monad that is used to model the communication of the server. A freer monad as presented by Kiselyov and Ishii [2015] is the left Kan extension of the free monad. That is, by storing a continuation instead of the result of applying this continuation, the freer monad "frees" the type constructor from being a functor. Using Haskell syntax and generalized algebraic data types, *Free* and *Freer* are defined as follows.

```
data Free f a where
  Pure :: a → Free f a
  Impure :: f (Free f a) → Free f a

data Freer f a where
  Pure :: a → Freer f a
  Impure :: f x → (x → Freer f a) → Freer f a
```

Note that the type variable x in the definition of *Freer* is existentially quantified. While the free monad cannot be defined in a theorem prover like Coq due to strict positivity, the definition of a freer monad is accepted. Therefore, Koh et al. [2019] use the freer monad instead of the free monad for their development. Due to the advantage that we do not need the container representation to define *Freer*, we tried to replace the model using the free monad with a freer monad as well. However, because all data structures are lifted monadically, we occasionally need to instantiate the existentially quantified variable x with an instance of *Freer* again. One example is the definition of *undefined* of type *Freer One* (*List One Bool*)[5], because the existentially quantified type variable x is then instantiated with a type that is defined using *Freer* again. Due to Coq's type hierarchy an existentially quantified type variable of a constructor cannot be instantiated with the type itself again.

Atkey and Johann [2015] also investigate how to prove propositions about effectful data types. More precisely, they show how to benefit from describing inductive types interleaved with effects as *initial f-and-m-algebras*. Their description of data types interleaved with effects is similar to the monadic lifting we pursue here. The benefit of using initial f-and-m-algebras comes from the clear separation of monadic effects described by a monad m and the pure data structure represented by a functor f. It would be interesting to investigate if the specialized proof principle that they introduce can be beneficial for our approach using *Free* as well.

[5] *One* is defined as shown in Table 1.

7.2 Reasoning about Effectful Haskell Programs

Besides related work that uses monads to reason about effectful programs there are various approaches for mechanized reasoning about effectful Haskell programs. The first step is the work by Abel et al. [2005] again, as they use a monadic model to reason about partiality in Haskell.

The interactive proof assistant Sparkle [de Mol et al. 2002] was developed to reason about Clean programs and is integrated into the corresponding development environment. As Clean is a non-strict pure functional programming language like Haskell, reasoning about Clean programs is similar to reasoning about Haskell programs. Sparkle explicitly considers *undefined* as a value. For example, when performing a proof about lists by induction, there is an additional case that considers the base case of an undefined value. As Sparkle considers *undefined* it can also be used to reason about explicit strictness [van Eekelen and de Mol 2006].

Farmer et al. [2015] present HERMIT, a toolkit for reasoning about Haskell programs, which is integrated into the GHC pipeline. Conceptually, their approach is similar to the approach of Sparkle but considers Haskell instead of Clean. HERMIT also explicitly considers undefined values. As a case study, Farmer et al. [2015] prove type class laws for various type classes and mechanize a pen-and-paper proof taken from the textbook Pearls of Functional Algorithm Design [Bird 2010].

Finally, we would like to mention an approach that has recently gained new interest. Instead of representing effects explicitly, we can try to keep as much of the structure of the original Haskell program as possible. For example, Dijkstra [2012] presents an experience report of automatically translating Haskell to Coq. In order to model partial function definitions Dijkstra uses a method by Bove and Capretta [2007]. This method extends functions with an additional argument, which proves certain invariants about all function applications. For example, in the concrete case of the partial function *head* on lists we need to provide a proof that the argument is not the empty list. Spector-Zabusky et al. [2018] also automatically translate Haskell to Coq, but focus on total Haskell programs as these can be translated to Coq one-to-one. In the case of partial Haskell programs, they add a polymorphic axiom that is used in the missing cases. As they state, the axiom is highly unsound and is meant to be a temporary solution. This way the user can start proving properties about partial functions right away and totalize these functions later. Breitner et al. [2018] have also presented a case study of applying this approach to prove properties about real world Haskell libraries. In this work they take a slightly different approach with respect to partial functions.

They still use an additional axiom for the missing cases, but prevent Coq from unfolding it. This approach allows them to reason about some simple properties involving partial functions.

8 Future Work

There are three main directions for future work. First of all, in order to apply the techniques presented here we need an automatic transformation from Haskell programs into effect-generic Coq programs. While most of the transformation is straightforward, convincing Coq's termination checker is a challenge. Recursive Haskell functions cannot simply be translated into monadic recursive Coq functions because the termination checker does not accept the resulting definitions. Coq's termination checker has trouble with nested recursive data types [Chlipala 2013] and a type like *Free C* (*List C a*) is nested recursive. One way to convince Coq that the function terminates is to split recursive functions into two parts as we have done before [Dylus et al. 2019]. When using an automatic transformation, we additionally have to think about the correctness of the transformation. Reasoning about programs is pointless if the behavior of the Coq code does not correctly model the behavior of the Haskell code.

The second main direction for future work is modeling call-by-need instead of call-by-name. As mentioned in Section 3 we could generalize the approach by Fischer et al. [2009] to arbitrary effects and finally to Coq. Fischer et al. present a library for modeling the lazy functional logic programming language Curry in Haskell. The transformation to model Curry in Haskell is similar to the transformation from Haskell to Coq. For example, because non-determinism can "hide" in constructor arguments, data types are also lifted monadically. However, as Fischer et al. use an untyped heap using *unsafeCoerce*, it is not obvious how to transfer their approach to Coq.

The third direction for future work is considering inequational propositions. As we can use the free monad to model partiality we can define a "less or equally defined" relation. This relation can be used to model a proposition about the strictness of a function. For example, one of the authors shows that there is a less strict implementation of Peano multiplication compared to the default implementation [Christiansen 2011]. The relation can also be used to mechanize other inequational propositions like, for example, the propositions that occur in the context of inequational free theorems by Johann and Voigtländer [2004]. While modeling the "less or equally defined" relation should be straightforward, we want to generalize this relation to other effects besides partiality. For example, when we replace partiality with error, there is a similar "less or equally defined" relation and similar propositions should hold. The question is whether we can define a generalization of these relations that works for multiple effects.

Acknowledgments

We thank the anonymous reviewers for their helpful remarks.

References

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2003. Categories of Containers. In *Foundations of Software Science and Computation Structures*. Vol. 2620. Springer Berlin Heidelberg, Berlin, Heidelberg, 23–38.

Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. 2005. Verifying Haskell Programs Using Constructive Type Theory. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*. ACM Press, New York, NY, USA, 62–73.

Sergio Antoy and Michael Hanus. 2010. Functional Logic Programming. *Commun. ACM* 53, 4 (2010), 74.

Robert Atkey and Patricia Johann. 2015. Interleaving Data and Effects. *Journal of Functional Programming* 25 (2015).

Richard Bird. 2010. *Pearls of Functional Algorithm Design*. Cambridge University Press.

Ana Bove and Venanzio Capretta. 2007. Computation by Prophecy. In *Typed Lambda Calculi and Applications*. Vol. 4583. Springer Berlin Heidelberg, Berlin, Heidelberg, 70–83.

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-Coq to Real-World Haskell Code. *arXiv:1803.06960 [cs]* (2018). arXiv:cs/1803.06960

Adam Chlipala. 2013. *Certified Programming with Dependent Types*. MIT Press New York.

Jan Christiansen. 2011. Sloth – A Tool for Checking Minimal-Strictness. In *Practical Aspects of Declarative Languages*. Vol. 6539. Springer Berlin Heidelberg, Berlin, Heidelberg, 160–174.

Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. 2002. Theorem Proving for Functional Programmers. In *Implementation of Functional Languages*. Vol. 2312. Springer Berlin Heidelberg, Berlin, Heidelberg, 55–71.

Gabe Dijkstra. 2012. Experimentation Project Report: Translating Haskell Programs to Coq Programs. (2012), 17.

Sandra Dylus, Jan Christiansen, and Finn Teegen. 2019. One Monad to Prove Them All. *The Art, Science, and Engineering of Programming* 3, 3 (2019).

Andrew Farmer, Neil Sculthorpe, and Andy Gill. 2015. Reasoning with the HERMIT: Tool Support for Equational Reasoning on GHC Core Programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell - Haskell 2015*. ACM Press, Vancouver, BC, Canada, 23–34.

Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. 2009. Purely Functional Lazy Non-Deterministic Programming. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, Edinburgh, Scotland, 11–22.

Johan Jeuring, Patrik Jansson, and Cláudio Amaral. 2012. Testing Type Class Laws. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 49–60.

Patricia Johann and Janis Voigtländer. 2004. Free Theorems in the Presence of Seq. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, Venice, Italy, 99–110.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proc. of the Symposium on Haskell*. ACM, New York, NY, USA, 94–105.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2019*. ACM Press, Cascais, Portugal, 234–248.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 333–343.

Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction*. Vol. 9129. Springer International Publishing, Cham, 257–275.

Maciej Piróg and Jeremy Gibbons. 2012. Tracing Monadic Computations and Representing Effects. *Electronic Proceedings in Theoretical Computer Science* 76 (Feb. 2012), 90–111.

André L de M Santos. 1995. Compilation by Transformation in Non-Strict Functional Languages. (1995).

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell Is Reasonable Coq. *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2018* (2018), 14–27. arXiv:1711.09286

Wouter Swierstra. 2008. Data Types à La Carte. *Journal of Functional Programming* 18, 04 (2008).

Marko van Eekelen and Maarten de Mol. 2006. Proof Tool Support for Explicit Strictness. In *Implementation and Application of Functional Languages*. Vol. 4015. Springer Berlin Heidelberg, Berlin, Heidelberg, 37–54.

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. ACM, 347–359.