Total Haskell is Reasonable Coq

Antal Spector-Zabusky Joachim Breitner Christine Rizkallah Stephanie Weirich
{antals,joachim,criz,sweirich}@cis.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Abstract

We would like to use the Coq proof assistant to mechanically verify properties of Haskell programs. To that end, we present a tool, named `hs-to-coq`, that translates total Haskell programs into Coq programs via a shallow embedding. We apply our tool in three case studies – a lawful `Monad` instance, "Hutton's razor", and an existing data structure library – and prove their correctness. These examples show that this approach is viable: both that `hs-to-coq` applies to existing Haskell code, and that the output it produces is amenable to verification.

1 Introduction

The Haskell programming language is a great tool for producing pure, functional programs. Its type system tracks the use of impure features, such as mutation and IO, and its standard library promotes the use of mathematically-inspired structures that have strong algebraic properties. At the same time, Haskell development is backed by an industrial-strength compiler (the Glasgow Haskell Compiler, GHC) [21], and supported by mature software development tools, such as IDEs and testing environments.

However, Haskell programmers typically *reason* about their code only informally. Most proofs are done on paper, by hand, which is tedious, error-prone, and does not scale.

On the other hand, the Coq proof assistant [22] is a great tool for writing proofs. It allows programmers to reason about total functional programs conveniently, efficiently, and with high confidence. However, Coq lacks GHC's extensive ecosystem for program development.

Therefore, we propose a multimodal approach to the verification of total functional programs: write code in Haskell and prove it correct in Coq. To support this plan, we have developed an automatic translator, called `hs-to-coq`, that allows this approach to scale.

For example, consider the standard `map` function on lists (from the Haskell Prelude), and the corresponding `Functor` instance.

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

instance Functor [] where fmap = map
```

Our tool translates this Haskell program automatically to the analogous Coq definitions. The `map` function becomes the expected fixpoint.

```
Definition map {a} {b} : (a -> b) -> list a -> list b :=
  fix map arg_62__ arg_63__
    := match arg_62__, arg_63__ with
       | _, nil => nil
       | f, cons x xs => cons (f x) (map f xs)
       end.
```

Similarly, the `Functor` type class in Haskell turns into a Coq type class of the same name, and Haskell's `Functor` instance for lists becomes a type class instance on the Coq side.

Once the Haskell definitions have been translated to Coq, users can prove theorems about them. For example, we provide a type class for *lawful* functors:

```
Class FunctorLaws (t : Type -> Type) `{Functor t} :=
  {functor_identity :
     forall a (x : t a), fmap id x = x;
   functor_composition :
     forall a b c (f : a -> b) (g : b -> c) (x : t a),
       fmap g (fmap f x) = fmap (g ∘ f) x}.
```

A `list` instance of the `FunctorLaws` type class is a formal proof that the `list` type, using this definition of `map`, is a lawful functor.

This process makes sense only for *inductive* data types and *total, terminating* functions. This is where the semantics of lazy and strict evaluation, and hence of Haskell and Coq, coincide [9]. However, the payoff is that a successful translation is itself a termination proof, even before other properties have been shown. Furthermore, because Coq programs may be evaluated (within Coq) or compiled (via extraction) these properties apply, not to a formal model of computation, but to actual runnable code.

Our overarching goal is to make it easy for a Haskell programmer to produce Coq versions of their programs that are suitable for verification. The Coq rendition should closely follow the Haskell code – the same names should be used,

CPP'18, January 2018, Los Angeles, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

even within functions; types should be unaltered; abstractions like type classes and modules should be preserved – so that the programmer obtains not just a black-box that happens to do the same thing as the original Haskell program, but a *live* Coq version of the input.

Furthermore, the development environment should include as much as possible of *total* Haskell. In particular, programmers should have access to standard libraries and language features and face few limitations other than totality. Also, because programs often change, the generated Coq must be usable directly, or with *declarative* modifications, so that the proofs can evolve with the program.

Conversely, an additional application of `hs-to-coq` is as a Haskell "rapid prototyping front-end" for Coq. A potential workflow is: (1) implement a program in Haskell first, in order to quickly develop and test it; (2) use `hs-to-coq` to translate it to the Coq world; and (3) extend and verify the Coq output. This framework allows diverse groups of functional programmers and proof engineers to collaborate; focusing on their areas of expertise.

Therefore, in this paper, we describe the design and implementation of the `hs-to-coq` tool and our experiences with its application in several domains. In particular, the contributions of this paper are as follows.

- We describe the use of our methodology and tool in three different examples, showing how it can be used to state and prove the monad laws, replicate textbook equational reasoning, and verify data structure invariants (Section 2).
- We identify several design considerations in the development of the `hs-to-coq` tool itself and discuss our approach to resolving the differences between Haskell and Coq (Section 3).
- We discuss a Coq translation of the Haskell `base` library for working with translated programs that we have developed using `hs-to-coq` (Section 4).

We discuss related work in Section 5 and future directions in Section 6. Our tool, `base` libraries, and the case studies are freely available as open source software.[1]

2 Reasoning about Haskell code in Coq

We present and evaluate our approach to verifying Haskell in three examples, all involving pre-existing Haskell code.

2.1 Algebraic laws

Objective The `Functor` type class is not the only class with laws. Many Haskell programs feature structures that are not only instances of the `Functor` class, but also of `Applicative` and `Monad` as well. All three of these classes come with laws. Library authors are expected to establish that their instances of these classes are lawful (respect the laws). Programmers

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

```
class Applicative m => Monad m where                 # Source
```

The Monad class defines the basic operations over a *monad*, a concept from a branch of mathematics known as *category theory*. From the perspective of a Haskell programmer, however, it is best to think of a monad as an *abstract datatype* of actions. Haskell's do expressions provide a convenient syntax for writing monadic expressions.

Instances of Monad should satisfy the following laws:

- `return a >>= k  =  k a`
- `m >>= return  =  m`
- `m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h`

Furthermore, the Monad and Applicative operations should relate as follows:

- `pure = return`
- `(<*>) = ap`

Figure 1. The documentation of `Monad` lists the three monad laws and the two laws relating it to `Applicative`.

using their libraries may then use these laws to reason about their code.

For example, the documentation for the `Monad` type class, shown in Figure 1, lists the three standard `Monad` laws as well as two more laws that connect the `Monad` methods to those of its superclass `Applicative`. Typically, reasoning about these laws is done on paper, but our tool makes mechanical verification available.

In this first example, we take the open source `successors` library [3] and show that its instances of the `Functor`, `Applicative`, and `Monad` classes are lawful. This library provides a type `Succs` that represents one step in a nondeterministic reduction relation; the type class instances allow us to combine two relations into one that takes a single step from either of the original relations. Figure 2 shows the complete, unmodified code of the library. The source code also contains, as a comment, 80 lines of manual equational reasoning establishing the type class laws.

Experience Figure 3 shows the generated Coq code for the type `Succs` and the `Monad` instance. The first line is the corresponding definition of the `Succs` data type. Because the Haskell program uses the same name for both the type constructor `Succs` and its single data constructor, `hs-to-coq` automatically renames the latter to `Mk_Succs` to avoid this name conflict.

The rest of the figure contains the instance of the `Monad` type class for the `Succs` type. This code imports a Coq version of Haskell's standard library `base` that we have also developed using `hs-to-coq` (see Section 4). The `Monad` type

```
module Control.Applicative.Successors where

data Succs a = Succs a [a] deriving (Show, Eq)

getCurrent :: Succs t -> t
getCurrent (Succs x _) = x

getSuccs :: Succs t -> [t]
getSuccs (Succs _ xs) = xs

instance Functor Succs where
  fmap f (Succs x xs) = Succs (f x) (map f xs)

instance Applicative Succs where
  pure x = Succs x []
  Succs f fs <*> Succs x xs
     = Succs (f x) (map ($x) fs ++ map f xs)

instance Monad Succs where
  Succs x xs >>= f
     = Succs y (map (getCurrent . f) xs ++ ys)
    where Succs y ys = f x
```

Figure 2. The successors library

class from that library, shown below, is a direct translation of GHC's implementation of the base libraries.

```
Class Monad m `{Applicative m} := {
  op_zgzg__ : forall {a} {b}, m a -> m b -> m b;
  op_zgzgze__ : forall {a} {b}, m a -> (a -> m b) -> m b;
  return_ : forall {a}, a -> m a} .

Infix ">>" := (op_zgzg__) (at level 99) .
Notation "'_ >> _'" := (op_zgzg__) .
Infix ">>=" := (op_zgzgze__) (at level 99) .
Notation "'_ >>= _'" := (op_zgzgze__) .
```

As in Haskell, the `Monad` class includes the `return` and `>>=` methods, which form the mathematical definition of a monad, as well as an additional sequencing method `>>`. Again due to restrictions on naming, the Coq version uses alternative names for all three of these methods. As `return` is a keyword, the tool replaces it by `return_`. Furthermore, Coq does not support variables with symbolic names, so the bind and sequencing operators are replaced by names starting with `op_` (such as `op_zgzgze__`, the translation of `>>=`). These names are systematically derived using GHC's "Z-encoding".

Note that our version of the `Monad` type class does not include the infamous method `fail::Monad m=>String->m a`. For many monads, including `Succs`, a function with this type signature is impossible to implement in Coq – this method is frequently partial.[2] As a result, we have instructed `hs-to-coq` to skip this method when translating the `Monad` class and its instances.

[2]In fact, this is considered to be a problem in Haskell as well, so the method is currently being moved into its own class, `MonadFail`; we translate this class (in the module `Control.Monad.Fail`) as well, for monads that have total definitions of this operation.

```
Inductive Succs a : Type :=
  Mk_Succs : a -> list a -> Succs a .

(* Instances for Functor and Applicative omitted. *)

Local Definition instance_Monad_Succs_op_zgzgze__
  : forall {a} {b}, Succs a -> (a -> Succs b) -> Succs b
  := fun {a} {b} => fun arg_4__ arg_5__ =>
    match arg_4__, arg_5__ with
     | Mk_Succs x xs, f => match f x with
       | Mk_Succs y ys => Mk_Succs y
         (app (map (compose getCurrent f) xs) ys)
       end
    end .

Local Definition instance_Monad_Succs_return_
  : forall {a}, a -> Succs a := fun {a} => pure .

Local Definition instance_Monad_Succs_op_zgzg__
  : forall {a} {b}, Succs a -> Succs b -> Succs b
  := fun {a} {b} => op_ztzg__ .

Instance instance_Monad_Succs : Monad Succs := {
  op_zgzg__ := fun {a} {b} =>
    instance_Monad_Succs_op_zgzg__;
  op_zgzgze__ := fun {a} {b} =>
    instance_Monad_Succs_op_zgzgze__;
  return_ := fun {a} =>
    instance_Monad_Succs_return_} .
```

Figure 3. Excerpt of the Coq code produced from Figure 2. (To fit the available width, module prefixes are omitted and lines are manually re-wrapped.)

The instance of the `Monad` class in Figure 3 includes definitions for all three members of the class. The first definition is translated from the `>>=` method of the input file; `hs-to-coq` supplies the other two components from the default definitions in the `Monad` class.

Our base library also includes an additional type class formalizing the laws for the `Monad` class, shown in Figure 4. These laws directly correspond to the documentation in Figure 1. Using this definition (and similar ones for `FunctorLaws` and `ApplicativeLaws`), we can show that the Coq implementation satisfies the requirements of this class. These proofs are straightforward and are analogous to the reasoning found in the handwritten 80 line comment in the library.

Conclusion The proofs about `Succs` demonstrate that we can translate Haskell code that uses type classes and instances using Coq's support for type classes. We can then use Coq to perform reasoning that was previously done manually, and we can support this further by capturing the requirements of type classes in additional type classes.

```
Class MonadLaws (t : Type -> Type)
    `{!Functor t, !Applicative t, !Monad t,
      !FunctorLaws t, !ApplicativeLaws t} :=
  {monad_left_id : forall A B (a : A) (k : A -> t B),
      (return_ a >>= k) = (k a);
    monad_right_id : forall A (m : t A),
      (m >>= return_) = m;
    monad_composition : forall A B C
      (m : t A) (k : A -> t B) (h : B -> t C),
      (m >>= (fun x => k x >>= h)) = ((m >>= k) >>= h);
    monad_applicative_pure : forall A (x : A),
      pure x = return_ x;
    monad_applicative_ap : forall A B
      (f : t (A -> B)) (x : t A),
      (f <*> x) = ap f x}.
```

Figure 4. Coq type class capturing the Monad laws.

2.2 Hutton's razor

Objective Our next case study is "Hutton's razor", from *Programming in Haskell* [17]. It includes a small expression language with an interpreter and a simple compiler from this language to a stack machine [17, Section 16.7]. We present our version of his code in Figure 5.

Hutton uses this example to demonstrate how equational reasoning can be used to show compiler correctness. In other words, Hutton shows that executing the output of the compiler with an empty stack produces the same result as evaluating an expression:

```
exec (comp e) [] = Just [eval e]
```

Experience Even in this simple example, the design of the compiler and its correctness proof are subtle. In particular, in Hutton's original presentation, the `exec` function is *partial*: it does not handle stack underflow. This partiality guides Hutton's design; he presents and rejects an initial version of the `comp` function because of this partiality.

Since Coq does not support partial functions, this posed an immediate problem. This is why the code in Figure 5 has been modified: we changed `exec` to return a `Maybe Stack`, not simply a `Stack`, and added the final equation. Once we made this small change and translated the code with `hs-to-coq`, the proof of compiler correctness was easy. In fact, in Coq's interactive mode, users can follow the exact same (small) steps of reasoning for this proof that Hutton provides in his textbook – or use Coq's proof automation to significantly speed up the proof process.

Conclusion We were successfully able to replicate a textbook correctness proof for a Haskell programs, but along the way, we encountered the first significant difference between Coq and Haskell, namely partiality (Section 3.7 provides more details). Since we only set out to translate total code, we needed to update the source code to be total; once we did so, we could translate the textbook proofs to Coq directly.

```
module Compiler where

data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y

type Stack = [Int]

type Code = [Op]

data Op = PUSH Int | ADD

exec :: Code -> Stack -> Maybe Stack
exec []           s         = Just s
exec (PUSH n : c) s         = exec c (n     : s)
exec (ADD    : c) (m : n : s) = exec c (n + m : s)
exec (ADD    : c) _         = Nothing

comp :: Expr -> Code
comp e = comp' e []

comp' :: Expr -> Code -> Code
comp' (Val n) c = PUSH n : c
comp' (Add x y) c = comp' x (comp' y (ADD : c))
```

Figure 5. Hutton's razor

2.3 Data structure correctness

Objective In the last case study, we apply our tool to self-contained code that lives within a large, existing code base. The Bag module[3] from GHC [21] implements multisets with the following data type declaration.

```
data Bag a
  = EmptyBag
  | UnitBag a
  | TwoBags (Bag a) (Bag a)
      -- INVARIANT: neither branch is empty
  | ListBag [a]
      -- INVARIANT: the list is non-empty
```

The comments in this declaration specify the two invariants that a value of this type must satisfy. Furthermore, at the top of the file, the documentation gives the intended semantics of this type: a `Bag` is "an unordered collection with duplicates". In fact, the current implementation satisfies the stronger property that all operations on `Bag`s preserve the *order* of elements, so we can say that their semantics is given by the function `bagToList :: Bag a -> [a]`, which is defined in the module.

Experience The part of the module that we are interested in is fairly straightforward; in addition to the `Bag` type, it contains a number of basic functions, such as

[3] http://git.haskell.org/ghc.git/blob/ghc-8.0.2-release:/compiler/utils/Bag.hs

```
isEmptyBag :: Bag a -> Bool
unionBags  :: Bag a -> Bag a -> Bag a
```

We formalize the combined invariants as a boolean predicate `well_formed_bag`. Then, for each translated function,[4] we prove up to two theorems:

1. We prove that each function is equivalent, with respect to `bagToList`, to the corresponding list function.
2. If the function returns a Bag, we prove that it preserves the Bag invariants.

Thus, for example, we prove the following three theorems about `isEmptyBag` and `unionBags`:

```
Theorem isEmptyBag_ok {A} (b : Bag A) :
  well_formed_bag b ->
  isEmptyBag b = null (bagToList b).
Theorem unionBags_ok {A} (b1 b2 : Bag A) :
  bagToList (unionBags b1 b2) =
  bagToList b1 ++ bagToList b2.
Theorem unionBags_wf {A} (b1 b2 : Bag A) :
  well_formed_bag b1 -> well_formed_bag b2 ->
  well_formed_bag (unionBags b1 b2).
```

Interestingly, we can see that `isEmptyBag`'s correctness theorem requires that its argument satisfy the Bag invariants, but `unionBags`'s does not.

Verifying Bag The verification effort proceeded just as though we were verifying any data structure library written in Coq. We verified nineteen different functions on Bags, and no proof was longer than eight lines (using the `ssreflect` tactic library [13]).

Along the way, we discovered a minor omission in the documentation of the `foldBag` function. This function has type

```
foldBag :: (r -> r -> r) -> (a -> r) -> r -> Bag a -> r
```

The expression `foldBag t u e` maps `u` over every element of the bag and then, starting with `e`, combines these results from the right using the operator `t`, à la `foldr`.

The documentation for `foldBag` requires that `t` be associative, and says that it is then a "more tail-recursive" version of a commented-out reference implementation which combines the results according the internal structure of the Bag instead of from the right. However, as we discovered when attempting to prove the two implementations equal, the reference implementation is *not* the same as `foldBag` in all cases – they are only the same when `e` is the identity for `t`. This discrepancy is minor, but has been present for over 21 years [25].

[4] We skipped monadic functions such as `mapBagM`, along with three further functions that referred to code we did not translate.

Selectively translating Bag As a part of GHC, the Bag module cannot stand on its own; it imports a number of other modules from GHC, such as `Outputable` and `Util`. However, there is a great deal of code we don't care about in GHC. For example, the `Outputable` module contains infrastructure for pretty printing. For our verification goals, this module is completely irrelevant, so it would be unfortunate if we could not proceed without translating it into Coq. But it would be equally unfortunate if we had to edit the GHC sources to remove code that we were not interested in.

It is for these sorts of reasons that `hs-to-coq` supports declaratively configuring the translation process: it can take as input a file of declarative instructions, called *edits*, that influence the translation process. One such instruction is to skip translating a module:

```
skip module Outputable
```

Similar instructions exist to skip functions, type classes, instances and type class methods; for example, the `Util` module contains a number of utility functions that aren't used by Bag, and so are unnecessary.

Conclusion Because `hs-to-coq`'s translation is configurable, we were able to slice the code of interest out of a large, existing codebase, without having to translate irrelevant parts or change the original source code. Once translated, the code was pleasant and straightforward to work with, and we completed both invariant preservation and semantic correctness proofs. We also saw that specifications are subtle, and edge cases in *documentation* can be caught by such verification.

3 The design and implementation of `hs-to-coq`

The previous section describes `hs-to-coq` in action: it processes a Haskell program, along with a separate files of "edits", which are commands that modify the translation in well-defined ways, and produces verifiable Coq code. Our design goals for `hs-to-coq` include:

1. Produce output resembling the original input;
2. Produce output amenable to interactive proof development;
3. Handle features commonly found in modern Haskell developments; and
4. Apply to source code as is, even if it is part of a larger development.

We have made the somewhat controversial choice to focus on *total* Haskell programs. This choice follows from our first two goals above: total programs require fewer modifications to be accepted by Coq (for example, no need use a monad to model partiality) and provide more assurances (if a translation is successful we know that the code is total). At the same time, reasoning about total functions is simpler than reasoning about partial ones, so we encourage Haskell proof development by concentrating on this domain.

The configurable edits support this design. Example edits include skipping functions that aren't being verified, or renaming a translated type or value to its Coq equivalent for interoperability. By providing this in a separate file, this per-project changes do not need to be applied to the code itself, and do not have to be re-done as the code evolves.

We use the Glasgow Haskell Compiler (GHC), version 8.0.2, as a library [21]. By using its parser, `hs-to-coq` can process most Haskell code as seen in the wild. In fact, our tool adopts the first two stages of GHC. First, the source code passes through the *parser* and an AST is produced. This AST then goes through the *renamer*, which resolves name references and ensures that programs are well scoped. Based on this, the tool generates the Coq output.

Note that `hs-to-coq` generates the Coq output before the *typechecking* and *desugaring* phases. Going after the desugaring, and hence translating GHC's intermediate language Core, would certainly simplify the translation. But the resulting code would look too different from the Haskell source code, and go against our first goal.

Many of the syntactic constructs found in Haskell have direct equivalents in Coq: algebraic data types, function definitions, basic pattern matching, function application, let-bindings, and so on. Translating these constructs is immediate. Other syntactic constructs may not exist in Coq, but are straightforward to desugar: **`where`** clauses become **`match`** or **`let`** expressions, **`do`** notation and list comprehensions turn into explicit function calls, etc.

However, many complex Haskell features do not map so cleanly onto Coq features. In the following we discuss our resolution of these challenging translations in the context of our design goals.

3.1 Module system

Haskell and Coq have wildly different approaches to their module systems, but thankfully they both have one. The largest point of commonality is that in both Haskell and Coq, each source file creates a single module, with its name determined by the file name and the path thereto. The method for handling modules is thus twofold:

- translate each Haskell file into a distinct Coq file; and
- always refer to all names fully qualified to avoid any differences between the module systems.

In each Coq module, we make available (through **`Require`**) all modules that are referenced by any identifiers. We do this instead of translating the Haskell **`import`** statements directly because of one of the differences between Haskell and Coq: Haskell allows a module to *re-export* identifiers that it imported, but GHC's frontend only keeps track of the *original* module's name. So the fully-qualified name we generate refers to something further back in the module tree that must itself be imported.

3.2 Records

In Haskell, data types can be defined as *records*. For example, the definition of the functions `getCurrent` and `getSuccs` in Figure 2 could be omitted if the data type were defined as

```
data Succs a = Succs {getCurrent :: a
                     ,getSuccs   :: [a]}
```

The type is the same, but naming the fields enables some extra features: (1) unordered value creation, (2) named pattern matching, (3) field accessors, and (4) field updates [20]. In addition, with GHC extensions, it also enables (5) *record wild cards*: a pattern or expression of the form `Succs {..}` binds each field to a variable of the same name.

Coq features support for single-constructor records that can do (1–3), although with minor differences; however, it lacks support for (4–5). More importantly, however, Haskell records are *per-constructor* – a sum type can contain fields for each of its constructors. Coq does not support this at all. Consequently, `hs-to-coq` keeps track of record field names during the translation process. Constructors with record fields are translated as though they had no field names, and the Coq accessor functions are generated separately. During pattern matching or updates – particularly with wild cards – the field names are linked to the appropriate positional field.

3.3 Patterns in function definitions

Haskell function definitions allow the programmer to have patterns as parameters:

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y
```

This code is not allowed in Coq; pattern matching is only performed by the **`match`** expression. Instead, programmers first have to name the parameter, and then perform a separate pattern match:

```
Definition uncurry {a} {b} {c} :
  (a -> b -> c) -> a * b -> c :=
  fun arg_10__ arg_11__ =>
    match arg_10__, arg_11__ with
      | f, pair x y => f x y
    end.
```

This translation extends naturally to functions that are defined using multiple equations, as seen in the `map` function in Section 1.

3.4 Pattern matching with guards

Another pattern-related challenge is posed by *guards*, and translation tools similar to ours have gotten their semantics wrong (see Section 5).

Guards are side conditions that can be attached to a function equation or a **`case`** alternative. If the pattern matches, but the condition is not satisfied, then the next equation is tried. A typical example is the `take` function from the Haskell standard library, where `take n xs` returns the first `n` elements of `xs`:

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x : xs)   = x : take (n − 1) xs
```

The patterns in the first equation match any argument; however, the match only succeeds if `n <= 0` as well. If `n` is positive, that equation is skipped, and pattern matching proceeds to the next two equations.

Guards occur in three variants:

1. A *boolean guard* is an expression `expr` of type `Bool`, as we saw in `take`. It succeeds if `expr` evaluates to `True`.
2. A *pattern guard* is of the form `pat ← expr`. It succeeds if the expression `expr` matches the pattern `pat`, and brings the variables in `pat` into scope just as any pattern match would.
3. A *local declaration* of the form **`let`** `x = e`. This binds `x` to `e`, bringing `x` into scope, and always succeeds.

Each equation can be guarded by a multiple guards, separated by commas, all of which must succeed in turn for this equation to be used.

Coq does not support guards, so `hs-to-coq`'s translation has to eliminate them. Conveniently, the Haskell Report [20] defines the semantics of pattern matching with guards in terms of a sequence of rewrites, at the end of which all guards have been removed and all case expressions are of the following, primitive form:

```
case e of K x1 . . . xN -> e1
          _            -> e2
```

According to these rules, the `take` function defined above would be translated to something like

```
take :: Int -> [a] -> [a]
take n xs = if n <= 0
            then []
            else case xs of
                 [] -> []
                 _ -> case xs of
                      x : xs -> x : take (n − 1) xs
                      _      -> error "No match"
```

Unfortunately, this approach is unsuitable for `hs-to-coq` as the final pattern match in this sequence requires an catch-all case to be complete. This requires an expression of arbitrary type, which exists in Haskell (`error . . .`), but cannot exist in Coq. Additionally, since Coq supports nested patterns (such as `Just (x : xs)`), we want to preserve them when translating Haskell code.

Therefore, we are more careful when translating case expressions with guards, and we keep mutually exclusive patterns within the same **`match`**. This way, the translated `take` function performs a final pattern match on its list argument:

```
Definition take {a} : Int -> list a -> list a :=
fix take arg_10__ arg_11__
  := let j_13__ :=
    match arg_10__, arg_11__ with
      | _, nil => nil
      | n, cons x xs =>
        cons x (take (op_zm__ n (fromInteger 1)) xs)
    end in
    match arg_10__, arg_11__ with
      | n, _ => if op_zlze__ n (fromInteger 0)
        then nil
        else j_13__
    end.
```

The basic idea is to combine multiple equations into a single **`match`** statement, whenever possible. We bind these **`match`** expressions to a name, here `j_13__`, that earlier patterns return upon pattern failure. We cannot inline this definition, as it would move expressions past the pattern of the second **`match`** expression, which can lead to unwanted variable capture.

In general, patterns are translated as follows:

1. We split the alternatives into *mutually exclusive* groups. We consider an alternative a_1 to be exclusive with a_2 if a_1 cannot fall through to a_2. This is the case if
   a. a_1 has no guards, or
   b. an expression matched by the pattern in a_1 will never be matched by the pattern in a_2.
2. Each group turns into a single Coq **`match`** statement which are bound, in reverse order, to a fresh identifier. In this translation, the identifier of the next group is used as the *fall-through target*.
   The last of these groups has nothing to fall-through to. In obviously total Haskell, the fall-through will not be needed. Partial code uses `patternFailure` as discussed in Section 3.7.
   If the patterns of the resulting **`match`** statement are not complete, we add a wild pattern case (using _) that returns the fall-through target of the current group.
3. Each alternative within such a group turns into one branch of the **`match`**. We translate nested patterns directly, as the semantics of patterns in Coq and Haskell coincide on the subset supported by `hs-to-coq`, which excludes incomplete irrefutable patterns, view patterns, and pattern synonyms [27].
   At this point, a **`where`** clause in the Haskell code (which spans multiple guards) gets translated to a **`let`** that spans all guarded right-hand-sides.
4. Each guarded right-hand-side of one alternative gets again bound, in reverse order, to a fresh identifier. The

last guard uses the fall-through target of the whole mutually exclusive group; the other guards use the next guard.

5. The sequence of guards of a guarded right-hand-side are now desugared as follows:
   a. A boolean guard `expr` turns into

   ```
   if expr then ...
           else j
   ```

   b. A pattern guard `pat ← expr` turns into

   ```
   match expr with | pat => ...
                   | _   => j
   ```

   c. A let guard turns into a **`let`** expression scoping over the remaining guards.

   Here, `...` is the translation of the remaining guards or, if none are left, the actual right-hand side expression, and `j` is the current fall-through target.

This algorithm is not optimal in the sense of producing the fewest **`match`** expressions; for example, a more sophisticated notion of mutual exclusivity could allow an alternative a_1 even when it has guards, as long as these guards cannot fail (e.g., pattern guards with complete patterns, **`let`**-guards). This issue has not yet come up in our test cases.

3.5 Type classes and instances

Type classes [33] are one of Haskell's most prominent features, and their success has inspired other languages to implement this feature, including Coq [29]. As shown in the `successors` case study (Section 2.1), we use this familial relation to translate Haskell type classes into Coq type classes.

As can be seen in Figure 3, `hs-to-coq` lifts the method definitions out of the **`Instance`**. While not strictly required there, this lifting is necessary to allow an instance method to refers to another method of the same instance.

Superclasses Superclass constraints are turned into arguments to the generated class, and these arguments are marked *implicit*, so that Coq's type class resolution mechanism finds the right instance. This can be seen in the definition of **`Class`** `Monad` in Section 2.1, where the `Applicative` superclass is an implicit argument to `Monad`.

Default methods Haskell allows a type class to declare methods with a default definition. These definitions are inserted by the compiler into an instance if it omits them. For example, the code of the `successors` library did not give a definition for `Monad`'s method `return`, and so GHC will use the default definition `return = pure`.

Since Coq does not have this feature, `hs-to-coq` has to remember the default method's definition and include it in the **`Instance`** declarations as needed. This is how the method `instance_Monad_Succs_return_` in Figure 3 arose.

Derived instances The Haskell standard provides the ability to *derive* a number of basic type classes (`Eq`, `Ord`, ...): the Haskell compiler can optionally synthesize whole instances of these type classes. GHC extends this mechanism to additional type classes (`Functor`, `Foldable`, ...). To translate derived instances, we simply take the instance declarations synthesized by the compiler and translate them just as we do for user-provided instances.

Self-referential instances Haskell type class instance are in scope even in their own declaration, and idiomatic Haskell code makes good use of that. Consider the standard instance for list equality:

```
instance Eq a => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _      == _      = False
  xs     /= ys     = not (xs == ys)
```

The operator == occurs three times on the right hand side of method definitions, and all three occurrences have to be treated differently:

1. In `x == y`, which compares list elements, we want to use the polymorphic `op_zeze__` method, so that Coq's instance resolution mechanism picks up the instance for `Eq a`.
2. For the first `xs == ys`, where lists are compared, we cannot use the polymorphic method, because the instance `Eq [a]` is not yet in scope. Instead, we want to refer to the very function that we are defining, so we have to turn that function into a fixed point.
3. The second `xs==ys`, in the definition of `/=`, also cannot be the polymorphic method. Instead, we want to refer to the method function for list's equality that we have just defined.

Unfortunately, `hs-to-coq` does not have the necessary type instance resolution information to reliably detect which variant to use. Therefore, we use following heuristic: By default, the polymorphic method is used. But in a method definition that is generated based on a *default method*, the currently defined methods are used. When this heuristic fails, the user can inject the correct definition using **`redefine`** edits.

3.6 Order of declarations

In Haskell, the order of declarations in a source file is irrelevant; functions, types, type classes, and instances can be used before they are defined. Haskell programmers often make use of this feature. Coq, however, requires declarations to precede their uses. In order to appease Coq, `hs-to-coq` detects the dependencies between the sentences of the Coq file – a sentence that uses a name depends on it – and uses this to sort the sentences topologically so that definitions precede uses.

While this works in most cases, due to the desugaring of type class constraints as invisible implicit arguments (Section 3.5), this process does not always yield the correct order. In such cases, the user can declare additional dependencies between definitions by adding an **order** like

```
order instance_Functor_Dual instance_Monad_Dual
```

to the edit file.

3.7 Partial Haskell

Another feature[5] of Haskell is that it permits partial functions and general recursion. We have only discussed verifying *total* Haskell. Nevertheless, as one starts to work on an existing or evolving Haskell code base, making every function total and obviously terminating should not have to be the first step.

Therefore, `hs-to-coq` takes liberties to produce something useful, rather than refusing to translate partial functions. This way, verification can already start and inform further development of the Haskell code. When the design stabilizes, the code can be edited for making totality obvious.

We can classify translation problems are into four categories:

1. Haskell code with genuinely partial pattern matches; for example,

   ```
   head :: [a] -> a
   head (x : _) = x
   ```

   which will crash when passed an empty list.
2. Haskell code with pattern matches that look partial, but are total in a way that Coq's totality checker cannot see. For example, we can define a run-length encoding function in terms of `group :: Eq a => [a] -> [[a]]`:

   ```
   runLengthEncoding :: Eq a => [a] -> [(a, Int)]
   runLengthEncoding =
     map (\(x : xs) -> (x, 1 + length xs)) . group
   ```

   Since the `group` function returns a list of *nonempty* lists, the partial pattern in the lambda will actually always match, but this proof is beyond Coq's automatic reasoning.
3. Haskell code with genuinely infinite recursion, at least when evaluated strictly; for example,

   ```
   repeat :: a -> [a]
   repeat x = x : repeat x
   ```

   produces an infinite list in Haskell, but would diverge in Coq using the inductive definition of lists.
4. Haskell code with recursion that looks infinite, but terminates in a way that Coq's termination checker cannot see. For example, we can implement a `sort` function in terms of the standard functional quicksort-like algorithm:

   ```
   sort :: Ord a => [a] -> [a]
   sort [] = []
   sort (p : xs) = sort lesser ++ [p] ++ sort greater
     where (lesser, greater) = partition (<p) xs
   ```

   This function recurses on two lists that are always smaller than the argument, but not syntactically, so it would be rejected by Coq's termination checker.

Our tool recognizes partial pattern matches, as described in Section 3.4. If these occur, it adds the axiom

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

to the output and completes the pattern match with it, e.g.:

```
Definition head {a} : list a -> a :=
  fun arg_10__ => match arg_10__ with
                  | cons x _ => x
                  | _        => patternFailure
                  end.
```

Naturally, this axiom is glaringly unsound. But it does allow the user to continue translating and proving, and to revisit this issue at a more convenient time – for example, when they are confident that the overall structure of their project has stabilized. In the case of genuinely partial functions, the user might want to change their type to be more precise, as we did in Section 2.2. In the case of only superficially partial code like `runLengthEncoding`, small, local changes to the code may avoid the problem. At any time, the user can use Coq's `Print Assumptions` command to check if any provisional axioms are left.

For non-structural recursion, we follow a similar path. Since `hs-to-coq` itself does not perform termination checking, it translates all recursive definitions to Coq fixpoints, which must be structurally recursive. If this causes Coq to reject valid code, the user can use an edit of the form **`nonterminating`** `sort` to instruct `hs-to-coq` to use the following axiom to implement the recursion:

```
Local Axiom unsafeFix : forall {a}, (a -> a) -> a.
```

Again, this axiom is unsound, but allows the programmer to proceed. In fact, after including the computation axiom

```
Axiom unroll_unsafeFix : forall a (f : a -> a),
  unsafeFix f = f (unsafeFix f).
```

in the file with the proofs, we were able to verify the correctness of the `sort` function above.

Eventually, though, the user will have to address this issue in order to consider their proofs complete. They have many options: They can apply the common Coq idiom of adding "fuel": an additional argument that is structurally decreasing in each iteration. They can replace the definition with one written in Coq, perhaps using the more advanced commands **`Function`** or **`Program Fixpoint`** [2], which allow explicit termination proofs using measures or well-founded

[5] Some might prefer quotes around this word.

relations. Or, of course, they can refactor the code to avoid the problematic functions at all.

Thus, the intended workflow around partiality and general recursion is to begin with axioms in place, which is not an unusual approach to proof development, and eliminate them at the end as necessary. For example, the correctness theorem about Hutton's razor in Section 2.2 goes through even *before* changing the `exec` function to avoid the partial pattern match! The reason is that the correctness theorem happens to only make a statement about programs and stacks that do *not* trigger the pattern match failure.

3.8 Infinite data structures

As a consequence of Haskell's lazy evaluation, Haskell data types are inherently *coinductive*. For example, a value of type `[Int]` can be an infinite list. This raises the question of whether we should be making use of Coq's support for coinductive constructions, and using **`CoInductive`** instead of **`Inductive`** in the translation of Haskell data types. The two solutions have real tradeoffs: with corecursion, we would gain the ability to translate corecursive functions such as `repeat` (mentioned in Section 3.7) using **`cofix`**, but at the price of our present ability to translate recursive functions such as `filter` and `length`.

We conjecture, based on our experience as Haskell programmers, that there is a lot of Haskell code that works largely with finite values. Moreover, many idioms that do use infinite data structures (e.g., `zipWith [0..]`) can be rewritten to work only with finite values. And reasoning about coinduction and corecursion is much trickier than reasoning about induction and recursion, especially in Coq.

3.9 Unsupported language features

There are language constructs that `hs-to-coq` simply does not yet support, such as mutually recursive definitions, incomplete irrefutable patterns, and a number of language extensions. If any of these features is used in a definition, then `hs-to-coq` creates an axiom with the name and type of the problematic definition so that it does not hold up the translation of code using this function. A code comment next to the axiom explains the nature of the failure. As we encounter these missing features, we extend `hs-to-coq` to support them.

4 GHC's `base` library

The case studies in Section 2 build upon a Coq version of GHC's `base` library [8] that we are developing as part of this project. This section discusses the design questions raised by constructing such a library. This process also stress-tests `hs-to-coq` itself.

Primitive types and operations
`GHC.Prim`, `GHC.Tuple`, `GHC.Num`, `GHC.Char`, `GHC.Base`
Prelude types and classes
`GHC.Real`, `Data.Bool`, `Data.Tuple`, `Data.Maybe`, `Data.Either`, `Data.Void`, `Data.Function`, `GHC.Enum`, `Data.Ord`, `Data.Bits`
List operations
`GHC.List`, `Data.List`, `Data.OldList`
Algebraic structures
`Data.Functor`, `Data.Functor.Const`, `Control.Monad`, `Control.Monad.Fail`, `Data.Monoid`, `Data.Traversable`, `Data.Foldable`, `Control.Category`, `Control.Arrow`

Figure 6. Coq base library modules

4.1 What is in the library?

Our base library consists of a number of different modules as shown in Figure 6. These modules include definitions of primitive types (`Int`, `Integer`, `Char`, `Word`) and their primitive operations, and common data types (`[ ]`, `Bool`, `Maybe`, `Either`, `Void`, `Ordering`, tuples) and their operations from the standard prelude. They also include prelude type classes (`Eq`, `Ord`, `Enum`, `Bounded`) as well as classes for algebraic structures (`Monoid`, `Functor`, `Applicative`, `Monad`, `Arrow`, `Category`, `Foldable`, `Traversable`) and data types that assist with these instances.

During the development of this library we faced the design decision of whether we should translate all Haskell code to new Coq definitions, or whether we should connect Haskell types and functions to parts of the Coq standard library. We have chosen to do the latter, mapping basic Haskell types (such as `Bool`, `[ ]`, `Either`, `Maybe`, and `Ordering`) to their Coq counterparts (respectively `bool`, `list`, `sum`, `option`, and `comparison`). This makes the output slightly less familiar to Haskell programmers – users must know how these types and constructors match up. However, it also makes existing Coq proofs about these data structures available.

Support for this mapping in `hs-to-coq` is provided via **`rename`** edits, which allow us to make that decision on a per-type and per-function basis, as the following excerpt of the edit file shows:

```
rename type  GHC.Types.[] = list
rename value GHC.Types.[] = nil
rename value GHC.Types.:  = cons
```

The library also includes (handwritten) modules that specify and prove properties of this code, including type classes that describe *lawful* functors, applicative functors, and monads, as discussed in Section 2.1. We include proofs that the type constructors `list` and `option` are lawful functors, applicative functors, and monads by instantiating these classes.

4.2 How did we develop the library?

Because of the nature of `base`, some modules were more amenable to automatic translation than others. Of those we translated, half were defined via automatic translation from the GHC source (with the assistance of edit instructions), and half required user assistance (usually through modification of the output of the translation).

We were forced to manually define the modules that define primitive types, such as `GHC.Word`, `GHC.Char`, and `GHC.Num`, because they draw heavily on a feature that we do not support: unboxed types. Instead, we translate primitive numeric types to signed and unsigned binary numbers in Coq (`Z` and `N`, respectively). Similarly, we translate `Rational` to Coq's type `Q` of rational numbers. In the case of fixed precision types, we have chosen these mappings for expediency; in future work, we plan to switch these definitions so that we can reason about underflow and overflow.

Moreover, we were limited in some ways by the expressiveness of our tool. As we describe in Section 3.5, we are not able to translate all type class instances to Coq. Furthermore, some modules make heavy use of safe coercions [4], which cannot be expressed in Coq. Finally, limitations in the difference between Haskell's type inference and Coq's type inference prevented the completely automatic translation of `Control.Category` and `Data.Functor.Const`.

On the other hand, we were able to successfully generate several modules in the base library, including the primary file `GHC.Base` and the list libraries `GHC.List` and `GHC.OldList`. Other notable successes include translating the algebraic structure libraries `Data.Monoid`, `Data.Foldable`, `Data.Traversable`, and `Control.Monad`.

4.3 What is skipped?

During the translation process, the edits allow us to **`skip`** definitions found in the Haskell input. Most modules had at least one skipped definition, and under a quarter had more than twenty.

Many of the skipped definitions are due to partiality. For example, we do not translate functions that could trigger pattern match failure, such as `head` or `maximum`, or that could diverge, such as `repeat` or `iterate`.

Some type classes have methods that are often instantiated with partial functions. We also removed such members, such as the `fail` method of the `Monad` class (as mentioned in Section 2.1), the `foldl1`, `foldr1`, `maximum` and `minimum` methods of the `Foldable` class, and the `enumFromThen` and `enumFromThenTo` methods of the `Enum` class. In the last case, this is not *all* of the partial methods of the class; for example, the `pred` and `succ` methods throw errors in instances for bounded types, and the `enumFrom` method diverges for infinite types. To solve this problem, we have chosen to support the `Enum` class only for bounded types. In this case, we modified the `pred` and `succ` methods so that they return the `minBound` and `maxBound` elements, respectively, at the end of their ranges. For `enumFrom`, we use `maxBound` to provide an end point of the enumeration.

Some functions are total, but it is difficult for Coq to determine that they are. For example, the `eftInt` function in the `Enum` module enumerates a list of integers from a starting number `x` to an ending number `y`. This function is not structurally recursive, so we use the **`Program Fixpoint`** extension to provide its termination proof in our redefinition.

Some parts of these modules are skipped because they relate to operations that are out of scope for our tool. We do not translate any definitions or instances related to `IO`. We also do not plan to support reflection, so all type class instances related to `GHC.Generics` are omitted. Similarly, we do not include arrays, so we skip instances related to array types and indexing.

5 Related Work

5.1 Haskell and Coq

Extraction The semantic proximity of Haskell and Coq, which we rely on, is also used in the other direction by Coq's support for code extraction to Haskell [19]. Several projects use this feature to verify Haskell code [7, 18]. It cannot be used to verify pre-existing Haskell code though.

Manual translation The `coq-haskell` library [34] is a hand-written Coq library designed to make it easier for Haskell programmers to work in Coq. In many ways, it serves a similar purpose to our translation of `base` (Section 4). In addition to enabling easier Coq programming, it also provides support for extracting Coq programs to Haskell.

5.2 Haskell and first-order logic

LiquidHaskell LiquidHaskell [31] augments the Haskell programming language with refinement types: all types can be coupled with a predicate that the inhabitants must satisfy. These refinements are then automatically checked by an SMT solver; a successful solution means that all functions are total and conform to these new, richer, specifications. In practice, when proving theorems in Coq, we can take advantage of a mature environment and proof automation techniques; this can allow for faster verification than LiquidHaskell, given a corresponding Coq program [30].

Halo The prototype contract checker `halo` [32] takes a Haskell program, uses GHC to desugar it into the intermediate language Core, and then translates the Core program into a first-order logic formula. It then invokes an SMT solver such as Z3 [10] or Vampire [28] to prove this formula; a successful proof tells us that the original program is crash-free.

5.3 Translating Haskell to higher-order logic

Haskabelle In the Isabelle/HOL ecosystem, `hs-to-coq` has a direct correspondence in Haskabelle [14], which translates total Haskell code into equivalent Isabelle function definitions. Like our tool, it parses Haskell, desugars syntactic constructs, configurably adapts basic types and functions to their counterpart in Isabelle's standard library. It used to be bundled with the Isabelle release, but it has not been updated recently and was dropped from Isabelle.

While Isabelle/HOL is, like Coq, a logic of total functions, all types in HOL are non-empty and inhabited by the polymorphic value `undefined`. Therefore, Haskabelle can translate partial patterns like described in Section 3.4, but without introducing inconsistency by relying on axioms.

Haskabelle supports boolean guards in simple cases, but does not implement fall-through across patterns on guard failure. In particular, the `take` function shown in Section 3.4 would be translated to a function that is `undefined` when `n` > 0.

HOLCF-Prelude A translation of Haskell into a total logic, as performed by `hs-to-coq` and Haskabelle, necessarily hides the finer semantic nuances that arise due to laziness, and does not allow reasoning about partially defined or infinite values. If that is a concern, one might prefer a translation into the Logic of Computable Functions (LCF) [26], where every type is a domain and every function is continuous. LCF is, for example, implemented in Isabelle's HOLCF package [16, 23]. Parts of the Haskell standard library have been manually translated into this setting [5] and used to verify the rewrite rules applied by `HLint`, a Haskell style checker.

seL4 Haskell has been used as a prototyping language for formally verified systems in the past. The seL4 verified microkernel started with a Haskell prototype that was semi-automatically translated to Isabelle/HOL [11]. As in our work, they were restricted to the terminating fragment of Haskell.

The authors found that the availability of the Haskell prototype provided a machine-checkable formal executable specification of the system. They used this prototype to refine their designs via testing, allowing them to make corrections before full verification. In the end, they found that starting with Haskell lead to a "productive, highly iterative development" contributing to a "mature final design in a shorter period of time."

5.4 Haskell and dependently-typed languages

Programmatica/Alfa The Programmatica project [15] included a tool to translate Haskell into the proof editor Alfa. As in our work, their tool only produces valid proofs for total functions over finite data structures. They state: "When the translation falls outside that set, any correctness proofs constructed in Alfa entail only partial correctness, and we leave it to the user to judge the value of such proofs."

The logic of the Alfa proof assistant is based on dependent type theory, but without as many features as Coq. In particular, the Programmatica tool compiles away type classes and nested pattern matching, features retained by `hs-to-coq`.

Agda 1 Dyber, Haiyan, and Takeyama [12] developed a tool for automatically translating Haskell programs to the Agda/Alfa proof assistant. Their solution to the problem of partial pattern matching is to synthesize predicates that describe the domain of functions. They explicitly note the interplay between testing and theorem proving and show how to verify a tautology checker.

Agda 2 Abel et al. [1] translate Haskell expressions into the logic of the Agda 2 proof assistant. Their tool works later in the GHC pipeline than ours; instead of translating Haskell source code, they translate Core expressions. Core is an explicitly typed internal language for Haskell used by GHC, where type classes, pattern matching and many forms of syntactic sugar have been compiled away.

Their translation explicitly handles partiality by introducing a monad for partial computation. Total code is actually polymorphic over the monad in which it should execute, allowing the monad to be instantiated by the identity monad or the `Maybe` monad as necessary. Agda's predicativity also causes issues with the translation of GHC's impredicative, System F-based core language.

5.5 Translating other languages to Coq

Chargueraud's CFML [6] translates OCaml source code to characteristic formulae expressed as Coq axioms. This system has been used to verify many of the functional programs from Okasaki's Purely Functional Data Structures [24].

6 Conclusions and future work

We presented a methodology for verifying Haskell programs, built around translating them into Coq with the `hs-to-coq` tool. We successfully applied this methodology to pre-existing code in multiple case studies, as well as in the ongoing process of providing the `base` Haskell library for these and other examples to build on.

Looking forward, there are always more Haskell features that we can extend the tool to support; we plan to apply this tool to larger real-world software projects and will use that experience to prioritize our next steps. We also would like to develop a Coq tactic library that can help automate reasoning about the patterns found in translated Haskell code as well as extend the proof theory of our `base` library.

Acknowledgments

Thanks to John Wiegley for discussion and support, and to Leonidas Lampropoulos and Jennifer Paykin for their helpful comments.

References

[1] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. 2005. Verifying Haskell Programs Using Constructive Type Theory. In *Haskell Workshop*. ACM, 62–73. DOI: http://dx.doi.org/10.1145/1088348.1088355

[2] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. 2006. Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant. In *FLOPS (LNCS)*, Vol. 3945. Springer, 114–129. DOI: http://dx.doi.org/10.1007/11737414_9

[3] Joachim Breitner. 2017. `successors`: An applicative functor to manage successors. https://hackage.haskell.org/package/successors-0.1. (1 February 2017).

[4] Joachim Breitner, Richard A. Eisenberg, Simon L. Peyton Jones, and Stephanie Weirich. 2014. Safe zero-cost coercions for Haskell. In *ICFP*. ACM, 189–202. DOI: http://dx.doi.org/10.1145/2628136.2628141

[5] Joachim Breitner, Brian Huffman, Neil Mitchell, and Christian Sternagel. 2013. Certified HLints with Isabelle/HOLCF-Prelude. In *Haskell and Rewriting Techniques (HART)*. arXiv:1306.1340

[6] Arthur Charguéraud. 2010. Program verification through characteristic formulae. In *ICFP*. ACM, 321–332. DOI: http://dx.doi.org/10.1145/1932681.1863590

[7] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *SOSP*. ACM, 18–37. DOI: http://dx.doi.org/10.1145/2815400.2815402

[8] Haskell Core Libraries Comittee. 2017. `base`: Basic libraries. https://hackage.haskell.org/package/base-4.9.1.0. (14 January 2017).

[9] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and loose reasoning is morally correct. In *POPL*. ACM, 206–217. DOI: http://dx.doi.org/10.1145/1111037.1111056

[10] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340. DOI: http://dx.doi.org/10.1007/978-3-540-78800-3_24

[11] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. 2006. Running the Manual: An Approach to High-assurance Microkernel Development. In *Haskell Symposium*. ACM, 60–71. DOI: http://dx.doi.org/10.1145/1159842.1159850

[12] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. 2004. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology* 46, 15 (2004), 1011–1025. DOI: http://dx.doi.org/10.1016/j.infsof.2004.07.002

[13] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2016. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. Inria Saclay Ile de France. https://hal.inria.fr/inria-00258384

[14] Florian Haftmann. 2010. From higher-order logic to Haskell: there and back again. In *'10*. ACM, 155–158. DOI: http://dx.doi.org/10.1145/1706356.1706385

[15] Thomas Hallgren, James Hook, Mark P. Jones, and Richard B. Kieburtz. 2004. An overview of the programatica toolset. In *HCSS*.

[16] Brian Huffman. 2012. *HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs*. Ph.D. Dissertation. Portland State University. DOI: http://dx.doi.org/10.15760/etd.113

[17] Graham Hutton. 2016. *Programming in Haskell* (2nd ed.). Cambridge University Press. 241–246 pages. DOI: http://dx.doi.org/10.1017/CBO9780511813672

[18] Adam Megacz Joseph. 2014. Generalized Arrows. (May 2014). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-130.html also see http://www.megacz.com/berkeley/coq-in-ghc/.

[19] Pierre Letouzey. 2002. A New Extraction for Coq. In *TYPES (LNCS)*, Vol. 2646. Springer, 200–219. DOI: http://dx.doi.org/10.1007/3-540-39185-1_12

[20] Simon Marlow (Ed.). 2010. *Haskell 2010 Language Report*.

[21] Simon Marlow and Simon Peyton Jones. 2012. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume 2*. Lulu. http://www.aosabook.org/en/ghc.html

[22] The Coq development team. 2016. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.6.1.

[23] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. 1999. HOLCF = HOL + LCF. *Journal of Functional Programming* 9 (1999), 191–223. DOI: http://dx.doi.org/10.1017/S095679689900341X

[24] Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press. DOI: http://dx.doi.org/10.1017/CBO9780511530104

[25] Will Partain. 1996. GHC commit 6c381e873e. http://git.haskell.org/ghc.git/commit/6c381e873e. (19 March 1996).

[26] Lawrence C. Paulson. 1987. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press. DOI: http://dx.doi.org/10.1017/CBO9780511526602

[27] Matthew Pickering, Gergo Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern synonyms. In *Haskell*. ACM, 80–91. DOI: http://dx.doi.org/10.1145/2976002.2976013

[28] Alexandre Riazanov and Andrei Voronkov. 1999. Vampire. In *CADE-16 (LNCS)*, Vol. 1632. Springer, 292–296. DOI: http://dx.doi.org/10.1007/3-540-48660-7_26

[29] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs (LNCS)*, Vol. 5170. Springer, 278–293. DOI: http://dx.doi.org/10.1007/978-3-540-71067-7_23

[30] Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. In *Haskell Symposium*. ACM, 63–74. DOI: http://dx.doi.org/10.1145/3122955.3122963

[31] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *ICFP*. ACM, 269–282. DOI: http://dx.doi.org/10.1145/2628136.2628161

[32] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. 2013. HALO: Haskell to Logic Through Denotational Semantics. In *POPL*. ACM, 431–442. DOI: http://dx.doi.org/10.1145/2429069.2429121

[33] Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *POPL*. ACM, 60–76. DOI: http://dx.doi.org/10.1145/75277.75283

[34] John Wiegley. 2017. `coq-haskell`: A library for formalizing Haskell types and functions in Coq. https://github.com/jwiegley/coq-haskell. (2017).