# Experimentation project report: Translating Haskell programs to Coq programs

Gabe Dijkstra

December 19, 2012

## 1 Introduction

Haskell programmers like to say that *well-typed programs do not go wrong*. However, if our Haskell program type checks, it does not give us any guarantees about termination or that it actually computes the right thing. If we for example have written a function $sort :: [Int] \rightarrow [Int]$ that type checks, we do not know whether it actually returns the sorted version of the input list: $sort = const\ [\,]$ also type checks. The usual Haskell way to go for checking properties like these is to use a tool such as QuickCheck (Claessen and Hughes, 2000) to test the properties on randomly generated input. If we want to go a step further and actually *verify* our Haskell software using a proof assistant, we need to model our Haskell code in the assistant's specification language. We can then formulate the properties and begin proving them.

Choosing Coq for the proof assistant is an attractive option, because its specification language, Gallina, is a functional programming language that in many respects is a lot like Haskell: in most cases we can very easily map our Haskell code to the Gallina equivalent. Manually translating Haskell code into Gallina code, however, quickly becomes a tedious job once the code size grows and more importantly, it is prone to subtle mistakes. Therefore, it would be nice if we had a tool that automates (parts of) this process.

During the verification process, it sometimes so happens that we have to change our Gallina code in order to be able to prove that certain properties hold. Instead of changing the original Haskell code to reflect these changes, new Haskell code can be generated from our Gallina specifications using Coq's extraction mechanism. This gives us another constraint on how we translate our Haskell program: we need do this in such a way that the extracted Haskell code has the same interface (i.e. the types of the definitions are the same) so we can plug the verified module back into the rest of our Haskell code base.

The goal of this experimentation project is to find answers to the following questions:

> "Can we automate the process of translating Haskell code into a Coq script? Can we do this in such a way that if we extract the Haskell code from the Coq script, we get back a module with the same interface and semantics?"

## 2 Method

We implemented our translation tool HsToGallina [1] using Haskell and the UUAGC system. As Haskell parser, we used the `haskell-src-exts` parser. The abstract syntax tree of the Haskell module is then mapped to the corresponding Gallina code. We use Gallina's own constructs to give the definitions, instead of writing a deep embedding of Haskell in Gallina. For the most part, Haskell's type system and syntax coincide with a subset of that of Coq, so we can translate a lot of constructions in a very straightforward manner. However, in many places there are also subtleties and intricacies that we have to take care of, which is the focus of the sections 3–9.

Another aspect of this approach, which is both a positive and a negative one, is that we get Coq's totality checking for free. This is done by checking whether all pattern matches are exhaustive and whether all recursive calls are structurally recursive. These restrictions imply that we cannot map every valid Haskell program to a Gallina counterpart without drastically changing the definition. For definitions with non-exhaustive pattern matches (e.g. $head$) or non-structural recursion (e.g. $quicksort$), we implemented the Bove-Capretta method (section 10).

Apart from non-exhaustive pattern matches and non-structural recursion, Haskell also allows us to work with infinite data structures. In Haskell, we do not distinguish between inductive and coinductive interpretations of data type definitions, e.g. the list type both has finite lists as well as infinite lists (or streams) as its inhabitants. In Coq there is a clear distinction between these two interpretations. Our tool defaults to translating Haskell data types to inductive data types, but we also provide means to translate them to coinductive data types (section 11).

One of the reasons we did not go for a deep embedding of Haskell in Coq is that this makes it a lot easier to produce a Coq script that, if we extract it back to Haskell, produces code that is similar "on the outside" to our original code: the names of the definitions remain unchanged as do the types (albeit up to $\alpha$-equivalence). In section 12 we show how we can make this work. Section 13 shows how we can have support for a fragment of the Haskell Prelude, in such a way that the extracted code has the same references to the same Prelude definitions.

[1] The HsToGallina tool can be found online: `https://github.com/gdijkstra/hs-to-gallina`

## 3 Supported language fragment

Since Haskell is a language with a lot of features, it is unrealistic to expect that we can support every single one of them right away. The language fragment that we currently support is Haskell 98 without the following features:

- modules
- type classes
- **do**-notation
- list comprehensions
- record syntax
- infix notation
- tuple syntax
- guards

Even though Coq does have some notion of type classes, it is very experimental at the moment, therefore we have chosen to disregard type classes for the time being. Since **do**-notation depends on type classes, we also do not support this.

Currently we only support translating a single module without any imports, except for the (implicit) Prelude import (see section 13). One way to support modules, is to also translate all the modules the current module depends on, but this breaks down as soon as there are dependencies on modules of which we do not have the source code. It also does not fit nicely in the use case where we want to verify a single module of a large project. Instead of translating the modules we depend on, we can generate axioms for all the definitions we import, so we can pretend, in our Coq script, that we have those definitions of the right type. Usually, having only the type of the imported definitions is not enough to be able to prove properties of our own definitions. The user will probably have to define extra axioms that sufficiently characterise the behaviour of the imported definitions.

The other features that we do not support should all be relatively straightforward to implement, but have not been implemented due to time constraints.

## 4 Type signatures

In Haskell we leave out type signatures and let the compiler figure out the type for us. For Coq's type system, type inference is undecidable, so we have to explicitly annotate at least our top-level definitions. Instead of doing the type inference for the Haskell code ourselves, we assume that the user has written explicit type signatures for every top-level definition and use these annotations.

Type signatures for local definitions are ignored when translating to Gallina. These are usually not needed if we already have the type signature for the corresponding top-level definition. Polymorphic local definitions are not supported as the current translation of parametric polymorphism does not play nicely with Gallina's `let`-construct. The user has to lift the local definition to the top-level instead.

## 5 List notation

Our tool supports Haskell's built-in list notation. It is supported both at the type level (e.g. the type $[a]$) and at the term and pattern level (e.g. the terms/patterns $[a, b, c]$, $a : b : [c]$, $[\,]$). For the terms and patterns we translate the infix notation to prefix notation in order to simplify some parts of the code (specifically the implementation algorithm presented in section 10.1.3): we can now assume that every pattern has a unique representation up to $\alpha$-equivalence.

The Haskell list notation is mapped to Coq's list notation as defined in its standard library, so that when we want to interactively prove something about a definition involving lists, Coq uses the more convenient syntax for lists, when pretty printing. It also means that we can more easily map the list functions from the Haskell Prelude to those of the standard library of Coq.

## 6 Data types and type synonyms

Haskell data types can be straightforwardly translated. For example:

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

translates to:

```
Inductive List ( a : Set ) : Set :=
          | Nil : List a
          | Cons : a -> List a -> List a.
```

In the Gallina definition we have to be explicit about the type (or in Haskell terms: *kind*) of the type parameters. Instead of doing actual kind inference, we simply assume all type parameters to have type `Set`, which corresponds to the Haskell kind $*$. Higher-kinded data types, e.g. data types that have type parameters of kind $* \to *$, are therefore not supported.

One important thing to note here is that in Coq names of data constructors cannot coincide with the name of the data type itself, since both can be used in exactly the same places: terms can be used in types and vice versa. Our tool does not check whether there is such an overlap and assumes the names of data constructors and those of type constructors are distinct.

Type synonyms can also be translated easily:

$$\textbf{type}\ \mathit{SillySynonym}\ a\ b\ c = \mathit{Silly}\ b\ c$$

becomes:

```
Definition SillySynonym ( a b c : Set ) : Set := Silly b c.
```

Just as with data types, type parameters that do not have kind $*$ are not supported.

## 6.1 Strictly positive data types

Coq does not allow us to have negative recursive positions in our data types, whereas Haskell does. To illustrate why we do not want this in a system like Coq, we will try to express the following lambda terms using a negative data type in Haskell:

$$\begin{aligned} \omega &= \lambda x.xx \\ \Omega &= \omega\omega \end{aligned}$$

If we perform a $\beta$-reduction on $\Omega$, we will get $\Omega$ again. We can keep on doing this indefinitely: $\Omega$ has no normal form.

Consider the following Haskell data type:

$$\textbf{data}\ \mathit{Term} = \mathit{Lam}\ (\mathit{Term} \to \mathit{Term})$$

Using this data type we can write the following:

$$\begin{aligned} &\mathit{omega} :: \mathit{Term} \to \mathit{Term} \\ &\mathit{omega}\ f = (\textbf{case}\ f\ \textbf{of}\ (\mathit{Lam}\ x) \to x)\ f \\ &\mathit{loop} :: \mathit{Term} \\ &\mathit{loop} = \mathit{omega}\ (\mathit{Lam}\ \mathit{omega}) \end{aligned}$$

We can see the exact same thing happening with *loop* as with $\Omega$: after a couple of reduction steps *loop* reduces to *loop*. Therefore allowing negative data types in Coq means that we can construct terms that have no normal form. Constructions like the above can then be used to define terms of the empty type `False`, which would make the system inconsistent.

Our tool does not check for these kind of constraints on data types and defers the error messages about this to Coq.

## 7 Parametric polymorphism and implicit parameters

Coq's type theory does not have the notion of parametric polymorphism. It can be simulated, however, using implicit parameters, e.g.:

$$\begin{aligned}&\mathit{const} :: a \to b \to a\\&\mathit{const}\ x\ \_ = x\end{aligned}$$

can be translated to:

```
Definition const { a b : Set } (x0 : a) (x1 : b) : a :=
             match x0 , x1 with
               | x , _ => x
             end.
```

The curly braces indicate that the parameters `a` and `b` are implicit.[2] These implicit parameters need not be provided when calling the function `const`.

Something we did not mention in section 6, is that we also need implicit parameters for data constructors. If we for example have the following data type:

```
Inductive List ( a : Set ) : Set :=
          | Nil : List a
          | Cons : a -> List a -> List a.
```

then the type of `Cons` is

```
Cons : forall a : Set, a -> List a -> List a
```

This means that every time we call `Cons`, we have to specify the type `a`. Using the contextual implicit parameters option, we can tell Coq to infer these parameters instead.

The simulation of parametric polymorphism as described above is not perfect. There are cases where Coq cannot infer the value of the implicit parameters. Consider the following example:

$$\begin{aligned}&s :: (a \to b \to c) \to (a \to b) \to a \to c\\&s\ p\ q\ r = p\ r\ (q\ r)\\&k :: a \to b \to a\\&k\ x\ \_ = x\\&i :: a \to a\\&i = s\ k\ k\end{aligned}$$

Coq will not be able to infer the type parameter $b$ of the second call to $k$ in the definition of $i$. If we do the type checking by hand, we will notice that we can fill in any type we want in that position, no matter what arguments $i$ gets. GHC solves this problem by filling in the type *GHC.Prim.Any*. Something similar can be done in Coq, by defining a type `Any` as the empty type and manually filling in the parameters it could not figure out by itself:

[2] Here we assume, just as with the type parameters of data types and type synonyms, that all the type variables are of kind $*$.

```
Inductive Any : Set := .

Definition i { a : Set } : a -> a :=
             s k (k (b:=Any)).
```

Of course, it would be better if our tool could automatically figure out which implicit parameters it needs to fill in explicitly, but this means that we would have to implement a type inference mechanism for Haskell, which we refrained from doing.

## 8 Ordering definitions

When writing definitions in Coq, we can only use terms that have been defined previously. In the case of recursive functions, we need to explicitly mark the definition as such using the `Fixpoint` command.

In Haskell, the ordering of our definitions does not matter, so when translating we need to order the definitions according to their dependencies and check whether they are recursive or not. This corresponds to finding all the strongly connected components of the dependency graph in a topological order. If such a strongly connected component consists of more than one definition, we have mutually recursive definitions. Coq supports these constructions by grouping the definitions together with the `with` keyword. We can only group functions with other functions and data types with other data types. Since Haskell does not allow us to write data types and functions that mutually depend on each other, this is not a problem.

### 8.1 Recursion in let-bindings

Local definitions inside **let**s and **where**s, as is the case with top-level definitions, need to be ordered and grouped. For recursive definitions we have the `let fix` construct, but this does not extend to mutually recursive definitions. Our tool therefore does not support mutually recursive local definitions. The user has to lift the mutual recursion to the top-level so that we are be able to translate the program.

## 9 Pattern matching

Haskell allows us to pattern match in a lot of places. In some cases this does not map nicely to Gallina constructs. For example, when writing a lambda expression, we are allowed to immediately pattern match on the argument, e.g. $\lambda(x, y) \to x$. In Gallina we would have to write something like:

```
fun xy => match xy with (x,y) => x end
```

Instead of translating it this way, we assume that the patterns occurring in lambda expressions are variables. Our tool will throw an error if it encounters any other pattern.

Another situation in which we can pattern match are pattern bindings, e.g. $(x, y) = e$. Coq has some support for these bindings if the pattern on the left-hand side happens to be an irrefutable pattern and the definition happens to be inside a `let` construct. For top-level definitions this cannot be done, hence we assume for pattern bindings that the pattern occurring on the left-hand side is a single variable.

## 10 General recursion and partiality

Coq demands that all our definitions be total. This is enforced by checking whether the pattern matches are exhaustive. For recursive definitions we are restricted to structural recursion: there should be at least one argument that decreases structurally in every recursive call, i.e. we pattern match on this argument and call the function recursively on the constituents of the patterns.

Haskell does not enforce these properties: definitions that violate either of these are commonplace. A typical example of a function that has non-exhaustive pattern matches is $head$. Of course, we can rewrite partial functions like these to total ones by using the $Maybe$ data type, but there are cases in which we know that a particular call to $head$ never fails and that the additional overhead of using $Maybe$ is not worth it in terms of readability. We therefore need a way to translate functions like $head$ in such a way that Coq can be convinced that the call is safe and that the extracted code looks a lot like the original Haskell code.

As an example of a program that does terminate, but is not structurally recursive, consider the following Haskell definition:

```
quicksort          :: [Nat] → [Nat]
quicksort []       = []
quicksort (x : xs) = append
                       (quicksort (filter (gt x) xs))
                       (quicksort (filter (le x) xs))
```

where $Nat$ is the natural numbers, $append$ is $(+\!\!+)$, $gt$ and $le$ are the "greater than" and "less than or equal" relations on $Nat$, i.e. they are of type $Nat \to Nat \to Bool$. If we translate this directly to Gallina, we will run into the problem that Coq cannot discover a parameter that structurally decreases every recursive call. It cannot infer that $filter\ (gt\ x)\ xs$ is indeed structurally smaller than $x : xs$.

There are numerous ways of translating a general recursive definition in such a way that Coq does accept it. A popular method is to use well-founded recursion, for example using the `Program` tactic (Sozeau, 2007). This method has the property that we do not need to change the type of our definition. Furthermore, the translation can be done in such a way that the proofs of termination get erased during extraction. This means that the extracted code will both have the same type as the original Haskell code and do not pass around any irrelevant proof objects. However, we did not choose well-founded recursion as it does not allow for definitions with non-exhaustive pattern matches or functions that only terminate for certain inputs.

Another approach is to encapsulate the return value in a non-termination monad, such as Capretta's coinductive delay monad (Bove and Capretta, 2007). These allow for both definitions with non-exhaustive pattern matches as definitions that are not structurally recursive. Unlike well-founded recursion, these approaches need us to change the type of our original definition. Apart from this, the body of the definition needs to be rewritten in monadic style to reflect the change in return type. These changes in type can still be seen in the extracted code, hence it might not be compatible with the other Haskell modules any longer.

The method we chose is the Bove-Capretta method (Bove and Capretta, 2005). Instead of having a general purpose accessability predicate as we have for well-founded recursion, an ad-hoc one is made for the definition we want to translate. The idea is that the function definition can be rewritten to take proofs of this ad-hoc predicate as an extra argument and that we now recurse structurally on these proofs. This means that we do have to change the type of our function, but this can be done in such a way that the proofs get erased again during extraction. The big advantage point of this translation is over well-founded recursion that it also allows for partial functions.

All the good stuff does come with a price: every time we call a function (in our Coq script) translated with the Bove-Capretta method, we need to provide a proof of our ad-hoc accessability predicate. This technique does not magically prove termination for our programs, it just makes it possible to prove termination.

## 10.1 Bove-Capretta method

Suppose $f :: \sigma_0 \to \ldots \to \sigma_n \to \tau$ is a Haskell function defined by the following equations:

$$\begin{array}{lll} f\ p_{00} & \cdots\ p_{0n} = & e_0 \\ \vdots & \ddots & \vdots \\ f\ p_{m0} & \cdots\ p_{mn} = & e_m \end{array}$$

We want to make a special purpose predicate on the input (i.e. an inductive data type that is indexed by the function parameters) that only has inhabitants whenever the function terminates on the given input. If we add this predicate as an argument to our function and pass the right values along recursively, we will see that we are recursing structurally on the proofs of our predicate, that essentially encode the call graphs for the corresponding input.

What does it mean for this function $f$ to terminate? How can we express this as a predicate depending on our input $x_0$ ... $x_n$? We inspect all the equations of our definition and can derive the constructors of our predicate as follows: looking at the $i$-th equation, we can tell that $f$ terminates for input $p_{i0}$ ... $p_{in}$ (as terms instead of patterns) if it terminates for all the recursive calls to $f$ in $e_i$. The constructor for the $i$-th equation consists of a context induced by the patterns (i.e. all the pattern variables and their types) and a termination proof for every recursive call.

Currently we support a very small fragment of Haskell for the right-hand sides of the equations of the function definition: only variables and applications are allowed. It is also required that all recursive calls be fully-applied. Support for guards and case statements can be added, but since this increases the complexity of our tool without really improving on the expressiveness of the language fragment we allow, we did not implement this.

### 10.1.1 Generating the inductive data type

Given a Haskell function $f :: \sigma_0 \to \ldots \to \sigma_n \to \tau$, we need to generate a predicate `f_acc` parametrised by (the translations of) the free type variables of $\sigma_0$, …, $\sigma_n$ and indexed by $\sigma_0$, …, $\sigma_n$. The parameters and indices can be determined directly from the type of the function.

In order to generate the constructor corresponding to the $i$-th equation, the context induced by the left-hand side (the patterns) needs to be determined and every recursive call occurring in the right-hand side has to be found. The context consists of all the variables that occur in the patterns of the equation along with their types. This is done by first annotating the patterns, namely by associating to each pattern variable its type. To be able to do this, the *specification* of every constructor used in the patterns must be known, i.e. the types of all its arguments. Once we have annotated the patterns, we can easily collect the pattern variables along with their types. Note that only the specifications of the data constructors defined in the module that is currently being translated are available, and the specifications of the list constructors.

After we have determined the context for the constructor for the $i$-th equation, we need to find all the recursive calls in the right-hand side $e_i$. Every (fully-applied) recursive call $f\ a_0\ \ldots\ a_n$ translates to a field `f_acc a_0 ... a_n` of the constructor we are generating, where every `a_i` is the translation of $a_i$.

Note that we run into problems in the case of nested recursion, i.e. when some $a_i$ again contains a call to $f$. This means that we have to refer to the function `f` in our definition of the predicate `f_acc`, which in turn is used to define `f`. To be able to write such definitions that have a mutual dependency between data types and functions, we need the so-called induction-recursion scheme (Dybjer, 2000).

### 10.1.2 Generating the new function definition

The Bove-Capretta translation of our Haskell function $f :: \sigma_0 \to \ldots \to \sigma_n \to \tau$ takes a proof of the accessability predicate `f_acc` as an extra argument. The type of the Coq definition `f` becomes `forall (x0 : s_0) ... (xn : s_n), f_acc x0 ... xn -> t`. One caveat of this new dependent type is that since we have enabled the contextual implicit parameters option, Coq will make the `s_0, `…, `s_n` implicit arguments since they can be inferred from a proof of `f_acc x0` . . . `xn`. We actually only need the contextual implicit arguments for data constructors, so we can safely disable this for function definitions, as we already denote which parameters are implicit.

Now that we have adapted the type of our function, we need still need to reflect these changes in the body of the function. One way of doing this is to pattern match only on the `f_acc x0 ... xn` value. This then introduces the same context as we would get from the original pattern matches and gives us the appropriate arguments for the recursive calls. It is easy to see that directly pattern matching on the proofs of the predicate gives us a structurally recursive definition. However, we want `f_acc x0 ... xn` to be of sort `Prop`, so it gets erased during extraction, hence we cannot take this approach. Instead of pattern matching directly on the proofs, we generate (and proof) theorems that essentially do this for us, which is the subject of section 10.1.4. Using these theorems, we can pattern match on our original input and call the appropriate generated theorems for the recursive calls.

If the pattern matches of our original function definition were not exhaustive, we still need to add the missing patterns. We can prove that these missing patterns never occur, given a proof of `f_acc x0 ... xn`. Given such an impossibility proof, we can then use `False_rec` to write a term of the correct type for the right-hand side of the match.

### 10.1.3 Calculating the missing patterns

In order to make the pattern matches of the given function definition exhaustive, we need to determine the missing patterns. The algorithm we implemented is part of an algorithm for compiling pattern matching (Augustsson, 1985). Note that throughout the presentation of the algorithm below, we assume that all patterns are well-typed and *linear*: every pattern variable occurs only once. Being valid Haskell code implies that the patterns satisfy these conditions, hence our tool does not check this.

The problem we want to solve is: given the set of original patterns[3] $\{\vec{p_0}, \ldots, \vec{p_n}\}$, we want to determine whether this set covers all possible input values, and if this is not the case: find the patterns we need to add to this set such that it does become a covering set.

The main idea of the algorithm is that we want to check whether the current set of *actual patterns* covers the given current *ideal pattern*. This is done by repeatedly splitting the ideal pattern on the most general pattern and recursively invoking the algorithm on these new ideal patterns with the appropriate new actual patterns. The invariant that we have to maintain is that every actual pattern $\vec{p_i}$ can be unified with the ideal pattern $\vec{q}$.

We start of by calculating the initial ideal pattern, i.e. the pattern that covers all possible patterns. For the initial ideal pattern we choose $\vec{q}$ to be $q_0 \ \ldots \ q_n$ where

[3]We also use the word *pattern* for *multipattern*: $n$-tuples of patterns. Arrows above letters indicate that we are talking about multipatterns.

every $q_i$ is a pattern variable of type $\sigma_i$. The fact that $\vec{q}$ covers $\{\vec{p_0}, \ldots, \vec{p_n}\}$ is established by substitutions[4] $s_i = \vec{q} \mapsto \vec{p_i}$.

The first substitution $s_0$ is inspected to see whether it is a injective renaming of pattern variables. In our implementation we do not actually check whether the renaming is injective, since this is implied by the linearity of the patterns and the way we generate our substitutions. If this is the case, then the ideal pattern $\vec{q}$ is $\alpha$-equivalent to the actual pattern $\vec{p_0}$. If it happens that the current set of actual patterns has size $> 1$, then we have overlapping patterns, which is also something we do not allow.

If the substitution $s_0$ does not simply rename variables, there exists a pattern variable $v$ in the ideal pattern $\vec{q}$ that gets mapped to a constructor pattern. The ideal pattern gets split into several new patterns by mapping $v$ to all the possible constructor patterns. The algorithm is then recursively invoked on these ideal patterns with the refined actual patterns, i.e. the current actual patterns that can be unified with the new ideal pattern, in order to maintain the invariant. The results of these invocations are then concatenated and returned.

Since we need to know the types of every pattern variable, we need to again work with annotated patterns. Apart from that, we also need to know into which constructor patterns a pattern variable can be split, given its type. For this we need a mapping from type constructors to their data constructors. These are looked up using the name of the type constructor, currently disregarding type synonyms, so even though type synonyms are syntactically supported, they will result in errors when we try to apply the Bove-Capretta method to functions that make use of type synonyms in their type signatures. A solution would be to also keep track of the type synonyms in our tool and performing the right substitutions on the types when needed.

### 10.1.4 Inversion theorems and their proofs

As mentioned in section 10.1.2, we cannot pattern match on proofs of the accessability predicate to access the constituents that we need to do the recursive calls. Instead, we generate inversion theorems that give us the appropriate values: the theorems select the field corresponding to the recursive call of the constructor corresponding to the current match. Apart from the theorem, a proof is also generated.

For the missing patterns we also need theorems telling us that these can never occur when we have a proof of our accessability predicate. For a missing pattern $p_0 \ \ldots \ p_n$ of our function $f : \sigma_0 \to \ldots \to \sigma_n \to \tau$, we need to generate a theorem of the form:

```
forall ctx (x0 : s_0) ... (xn : s_n),
  f_acc x0 .. xn  -> (x0 = p0) -> ... -> (xn = pn) -> Logic.False.
```

where `ctx` is the context induced by the pattern $p_0 \ \ldots \ p_n$. Proofs of these theorems are also automatically generated by the tool.

[4]This slight abuse of notation is justified by the fact that $\vec{q}$ consists solely of pattern variables.

### 10.1.5 Examples

Consider the *quicksort* example given above. We can tell the HsToGallina tool to apply the Bove-Capretta translation to that definition using the following pragma:

> {-# OPTIONS_HsToGallina bc: quicksort #-}

The tool will then generate the following (we renamed `quicksort` to `qs` to make it all fit on paper and have left out the proofs of the inversion theorems as they are rather long and not very interesting):

```
...

Inductive qs_acc : List Nat -> Prop :=
  | qs_acc_0 : qs_acc nil
  | qs_acc_1 : forall (x : Nat) (xs : List Nat) ,
                        qs_acc (filter (gt x) xs) ->
                        qs_acc (filter (le x) xs) ->
                        qs_acc (cons x xs).

Theorem qs_acc_inv_1_0 : forall (x0 : List Nat) (x : Nat) (xs : List Nat),
  qs_acc x0 -> (x0 = cons x xs) -> qs_acc (filter (gt x) xs).
...
Defined.

Theorem qs_acc_inv_1_1 : forall (x0 : List Nat) (x : Nat) (xs : List Nat),
  qs_acc x0 -> (x0 = cons x xs) -> qs_acc (filter (le x) xs).
...
Defined.

...

Fixpoint qs (x0 : List Nat) (x1 : qs_acc x0) : List Nat :=
  match x0 as _y0 return (x0 = _y0) -> List Nat with
    | nil => fun _h0 =>
              nil
    | cons x xs => fun _h0 =>
              append (qs (filter (gt x) xs) (qs_acc_inv_1_0 x1 _h0))
                     (qs (filter (le x) xs) (qs_acc_inv_1_1 x1 _h0))
           end (refl_equal x0).

...
```

As we can see above, there are no missing patterns hence no theorems to handle these. There are two recursive calls with two corresponding inversion theorems. The function definition still has the same shape as before the translation, except for the extra arguments being passed around and some dependent pattern matching to make everything work.

As an example of a definition with non-exhaustive patterns, the translation of *head* as outputted by our tool:

```
Inductive head_acc ( a : Set ) : List a -> Prop :=
  | head_acc_0 : forall (x : a) (_xs : List a), head_acc (cons x _xs).

Theorem head_acc_non_0 : forall (a : Set) (x0 : List a),
  head_acc x0 -> (x0 = nil) -> Logic.False.
...
Defined.

Definition head { a : Set } (x0 : List a) (x1 : head_acc x0) : a :=
  match x0 as _y0 return (x0 = _y0) -> a with
    | cons x _xs => fun _h0 => x
    | nil => fun _h0 => False_rec a (head_acc_non_0 x1 _h0)
  end (refl_equal x0).
```

There are no recursive calls, hence no inversion theorems, and one theorem proving that we cannot have *nil* as input.

The definition itself is not too exciting, but when we want to use this function on some input `e : List a`, we still have to give a proof `p : head_acc e`. The `refine` tactic is particularly useful here. Consider for example the function $headReverse\ x\ xs = head\ (reverse\ (x : xs))$: we know that *reverse* preserves the length of a list, so the pattern match in *head* will never fail in this case. The Coq translation of this would become:

```
Definition headReverse {a : Set} (x: a) (xs : List a) : a.
refine (head (reverse (x :: xs)) _).
...
Defined.
```

We have left out the proof as it is rather involved: we have to proof that we can construct a `h` and `t` such that `reverse (x :: xs) = h :: t`, for any `x` and `xs`. If we have such a `h` and `t` we can construct the required proof of the predicate `head_acc (reverse (x :: xs)`.

The extracted Haskell code of this fragment has none of the proofs and looks almost the same as our original code:

$$\begin{aligned}&headReverse :: a1 \to (List\ a1) \to a1\\&headReverse\ x\ xs =\\&\quad head\ (reverse\ ((:)\ x\ xs))\end{aligned}$$

## 11 Coinduction

A limitation of a direct translation is that in Coq there is a distinction between inductive and coinductive data types. If we for example want to work with infinite lists in Coq, we have to make a separate coinductive data type. With the `codata` and `cofix` pragmas, we can indicate that we want a coinductive translation of our top-level definitions. For example, if we want to define an infinite stream of zeroes, we can write something as follows:

```
{-# OPTIONS_HsToGallina codata: Stream #-}
{-# OPTIONS_HsToGallina cofix: zeroes #-}
...
data Stream a = Cons a (Stream a)
zeroes :: Stream Nat
zeroes = Cons 0 zeroes
```

translates to:

```
CoInductive Stream (a :Set) : Set :=
  | Cons : a -> Stream a -> Stream a.

CoFixpoint zeroes : Stream Nat :=
  Cons 0 zeroes.
```

Just as we have restrictions as to what recursive definitions we can specify in Coq, we have similar restrictions for corecursive definitions: every corecursive call should be *guarded* by a constructor. Our tool will not check whether this is the case and will just blindly translate the Haskell definitions.

## 12 Extraction

Once we have verified and possibly modified the Gallina code, such that it actually satisfies the properties we wanted to prove, we can translate the code back to Haskell using Coq's extraction mechanism. The translation from the original Haskell module to Gallina specifications has been done in such a way that the extracted code will still have the same types as before (up to $\alpha$-equivalence), the Bove-Capretta proofs are also removed (since they all the predicates live in `Prop`), and the coinductive definitions are extracted to normal Haskell definitions.

Since Coq's type system does not map nicely to Haskell's, it sometimes uses *unsafeCoerce* to convince GHC's type checker. However, Coq produces broken Haskell code when it needs *unsafeCoerce*. Even though this is just a minor syntactic fault that we can fix by applying a very simple `sed` script, it does leave a sour taste.

## 13 Prelude

Now that we have a mapping from a subset of the Haskell syntax to Gallina syntax, we also want to have some support for the Haskell Prelude. We have achieved this by writing our own Coq prelude which implements definitions from the Haskell Prelude as defined in the Haskell Report. Apart from implementing the functions, which sometimes just means picking a definition from the Coq standard library and filling in the correct parameters, we also specify how these definitions should be extracted.

Since we do not support all of Haskell 98, we also cannot support all of the Haskell 98 Prelude. For example, we skipped all the definitions that need type classes such as the numeric operations.

More interesting cases are functions that have non-exhaustive pattern matches, e.g. *head* and *tail*. For these functions we have used the Bove-Capretta method to define them.

So far we have only considered the list functions on finite lists, but sometimes we want to perform a *take* on an infinite list. With the current approach of only translating everything as inductive data types or recursive functions, this is not possible. We could define two versions of *take*: a recursive and a corecursive definition, but this means that whenever we call *take* in our Haskell code, our tool needs to infer whether we want the recursive or the corecursive definition, which gets complicated very quickly. We also do not support functions such as *iterate* and *repeat*, that always produce infinite lists. We can implement similar functions, but they would then work on streams instead of lists.

The function *until* is not supported as its termination behaviour depends non-trivially on the given input. The user therefore has to resort to providing their own special purpose functions and apply the Bove-Capretta translation to those instead.

## 14 Future work

We currently assume that every top-level definitions has an explicit type signature. Ideally, we want to be able to infer this kind of information, but we do not want to go through the trouble of doing the type inference ourselves. One way to solve this is to make use of the GHC API. Apart from type inference, it also provides us with kind inference that we need when dealing with parametric polymorphism.

If we want to support modules, the GHC API can be very helpful here as well to provide us with the necessary information, such as the type signatures of the definitions we want to import. Using these type signatures, we can make axioms in Coq postulating that we have definitions of that type.

Right now, our tool produces invalid Coq code whenever we call a function that has been defined using the Bove-Capretta method, as the translated version expects an extra argument: a proof of the accessability predicate. It would be better if we could do automatically generate scripts in which we apply the `refine` tactic, as we saw in section 10.1.5. We can then put holes wherever we need these Bove-Capretta proofs. The problem with this, however, is that the `refine` tactic does not play nicely with mutually recursive definitions: we cannot wrap the definitions grouped by `with` in a single `refine` command.

## 15 Conclusion

We have seen that it is possible to automatically translate a sizable subset of Haskell 98 to a Coq script. This can also be done in such a way that if we extract the Coq script back to Haskell code, we will get a module in which the definitions have the same types as the corresponding original definitions.

Using the Bove-Capretta method, it is also possible to translate partial Haskell definitions in such a way that we can reason about them in Coq in a natural way, while also preserving the types during extraction. It is possible to automate the menial parts of this translation: the accessability predicate and the new definition can be generated automatically from the Haskell module, leaving only the actual termination proof up to the user.

The result may still need some post-processing in some cases to deal with for example higher-kinded types, but it is viable that these cases can be automated. Our tool currently does not support all of Haskell 98, but we do believe that the current approach can be extended to support things like type classes and modules.

## References

L. Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, pages 368–381. Springer, 1985.

A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

A. Bove and V. Capretta. Computation by prophecy. *Typed Lambda Calculi and Applications*, pages 70–83, 2007.

K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Acm sigplan notices*, volume 35, pages 268–279. ACM, 2000.

P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, pages 525–549, 2000.

M. Sozeau. Subset coercions in coq. *Types for Proofs and Programs*, pages 237–252, 2007.