Experimentation project report: Translating Haskell programs to Coq programs

Gabe Dijkstra

November 26, 2012

Abstract?

1 Introduction

Motivate problem some more: Haskell's type system is nice, but not expressive enough for actual verification.

Suppose we want to verify software written in Haskell using a proof assistant like Coq. Before we can begin with the verification process, we need to model our software in the proof assistant's specification language. Manual translation of Haskell code into Coq's Gallina code is a tedious job and more importantly, it is prone to subtle mistakes.

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 will use Haskell along with the UUAG system to implement a translation of a Haskell module into a Coq script. To parse the Haskell file, we will use the `haskell-src-exts` library. We will take the abstract syntax tree with all the sugar instead of translating for example an intermediate language like GHC Core.

Motivation for this? Hopefully more readable code and proofs (we can more or less do the equational reasoning we are used to). More importantly (?): Bove-Capretta and extraction.

Stress that we do not create a deep embedding of Haskell in Coq, but really translate Haskell constructs to the corresponding Gallina constructs (if they exist). A consequence of this approach is that we reason about our Haskell program as if it were total and strict. Coq will complain about missing patterns and non-structural recursion. Ways around this: section 8.1 and section 9.

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 aim to support is Haskell 98 without at least the following features:

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

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

The other features that we do not support are all relatively straightforward to implement.

But have not been implemented due to time constraints. Although infix notation: needs some work with generating names and such and translating priorities.

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 straight-forward manner. However, in many places there are also subtleties and intricacies that we have to take care of, which is the main focus of the following sections.

3 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 ourselves, we assume that the user has written explicit type signatures for every top-level definition and use these annotations.

4 Data types and type synonyms

Haskell data types can be straightforwardly translated. For example:

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

translates to:

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

One important thing to note here is that since 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.

Type synonyms can also be translated easily:

type *SillySynonym a b c* = *Silly b c*

becomes:

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

4.1 List notation

Built-in support for lists: it is supported in patterns and terms, but gets translated to conses and nils, even though we have support for that in Coq. But the extracted code uses conses and nils everywhere...

4.2 Negative data types

Note: negative data types are not allowed. Give examples of where this can go wrong.

4.3 Coinductive types

In Haskell, we do not make a distinction between inductive and coinductive interpretations of data type definitions, e.g. the list type both has finite lists as infinite lists (or streams) as its inhabitants. In Coq there is a clear distinction between these two interpretations. The translation given above reads the data type as an inductive definition, so we are only allowed to give finite inhabitants of that data type.

If we want to deal with infinite data structures, we then need to use coinduction, which we will deal with in section 9.

5 Parametric polymorphism and implicit parameters

Also mention the contextual implicit stuff and data type constructors.

Coq's type theory does not have parametric polymorphism, however, we can simulate this using implicit parameters, e.g.:

$$\begin{array}{l} \mathit{const} :: a \to b \to a \\ \mathit{const}\ x\ \_ = x \end{array}$$

translates to:

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

The parameters `a` and `b` are implicit and usually need not be provided when calling the function `const`. There are however cases where Coq cannot infer the value of such implicit parameters. Consider the following example:

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

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.

This is also reflected in the fact that if we look at the GHC Core output, we will see that GHC fills in GHC.Prim.Any.

Manual solution: add GHC.Prim.Any thing to Coq prelude and use explicit implicit parameter assignments. (How does this look in the extracted Haskell code?)

6 Ordering definitions

Ordering: topological ordering of dependency graph.

(Mutual) recursion: strongly connected components of dependency graph. We need to explicitly group these mutual definitions with the `with` constructor.

Mutual recursion in let bindings does not work.

7 Pattern matching

Pattern bindings do not work as expected. Although we have irrefutable patterns in Coq, we do not use that feature.

8 General recursion and partiality

Explain problem, explain array of solutions, explain why we chose this solution specifically. (Extraction!)

8.1 Bove-Capretta method

explain method

explain how nested recursion leads to induction-recursion and that we cannot easily do this in Coq, in general.

for sake of simplicity, we only consider apps and vars: no case expressions and guards. These can be added.

8.2 Implementation

generate inductive data type

details on Prop versus Set wrt extraction.

generate new function

Detail about contextual implicit arguments for functions that have been generated by this method.

8.2.1 Inversion theorems

Since the predicate is of sort Prop, we cannot pattern match on inhabitants of this predicate, since the type of the result of the function we are transforming is of sort Type. We need inversions theorems to get around this.

Details on the inversion proofs and why they should work.

Note that ltac script would probably be better

8.2.2 Missing patterns

Details of missing pattern implementation.

Details of implementation: what do we restrict ourselves to? Type synonyms stuff again.

8.2.3 Examples

Show examples.

Refine tactic example for how to use stuff. (Find a nice example that I encountered in verification challenge?)

9 Coinduction

Another 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 definitions.

```
{-# OPTIONS_Hs2Gallina codata: Stream #-}
{-# OPTIONS_Hs2Gallina cofix: zeroes #-}
...
data Stream a = Cons a (Stream a)
zeroes :: Stream Nat
zeroes = Cons 0 zeroes
```

would translate 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.

Coinduction stuff as a nice extra. Talk about how we don't check guardedness and no mixing of induction and coinduction.

10 Extraction

Talk about how we want to check whether this is still the same code. Generating QuickCheck tests to compare programs seems rather involved. Unless you make a lot of assumptions. What would be fun to have a mapping from QuickCheck properties to Coq properties and back? Very much future work. Do we have any guarantees that extraction will yield something with the same semantics as the Coq? If so, then we needn't care about the fact that our own translation might not preserve semantics (up to strictness, whatever that means): you verify stuff and you get something that's extracted with the same computational behaviour, that should be good enough.

Coq needs to update its extraction stuff. Produces broken Haskell...

11 Prelude

Show how we support some prelude stuff. Of course we skip all the type class stuff

We also skip the obviously non-terminating stuff like iterate and such.

We have to write our own B-C definitions of partial functions like head and tail, but during extraction they get mapped to the Prelude functions *head* and *tail*.

12 Related work

Verifying Haskell in CTT paper

13 Future work

Modules, better syntax support, refine tactic support, type synonyms, type classes, GADTs, better integration with GHC for finding imports and dependencies and et cetera.

14 Conclusion

Semantics preserved? Nothing has been proved about this. We need to trust extraction mechanism and our Coq prelude stuff.