Experimentation project report: Translating Haskell programs to Coq programs

Gabe Dijkstra

November 20, 2012

1 Introduction

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 an answer to the following question:

> "Can we automate the process of translating Haskell code into a Coq script?"

2 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 aim to support is Haskell 98 without the following features:

- typeclasses
- do-notation and list comprehensions
- record syntax

3 Parametric polymorphism

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

```
const :: a → b → a
const x _ = x
```

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.

4 General recursion

Since Coq only allows structural recursion, not all recursive Haskell definitions can translated directly into Gallina. In order to deal with some of these cases, we can generate Bove-Capretta predicates from the original function definition and rewrite the function definition using this predicate.

With the "bc" pragma we can specify of which definitions we want to generate Bove-Capretta predicates:

```
{-# OPTIONS_Hs2Gallina bc: quicksort #-}
...
```

$$
\begin{array}{l}
\mathit{quicksort} :: [\mathit{Nat}] \to [\mathit{Nat}] \\
\mathit{quicksort}\ [\,] \qquad\ \ = [\,] \\
\mathit{quicksort}\ (x : \mathit{xs}) = \mathit{quicksort}\ (\mathit{filter}\ ({<}x)\ \mathit{xs}) +\!\!+ \\
\qquad\qquad\qquad\quad\ \ \mathit{quicksort}\ (\mathit{filter}\ ({\geqslant}\ x)\ \mathit{xs})
\end{array}
$$

This method does not work well with nested recursion as we will need to simultaneously define the predicate and the function, which is something that cannot be done in Coq.

5 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.

6 Modules

Since one probably will only verify a single Haskell module, pretending all the imported functions have already been verified. We want the tool to facilitate this strategy. For example, generate appropriate axioms for the imported definitions, such that if we extract the Haskell module from the Coq script, we can plug it back in our Haskell software without having to change the resulting code manually.