Modelling Haskell Programs with Free Monads (Extended Abstract)

Sandra Dylus[1] and Jan Christiansen[2]

[1] University of Kiel
sad@informatik.uni-kiel.de
[2] University of Applied Science Flensburg
jan.christiansen@hs-flensburg.de

When we want to prove statements about a Haskell program in an interactive theorem prover like Agda or Coq, we have to transform the Haskell program into an Agda/Coq program. As Agda/Coq programs have to be total and Haskell programs are often not, we have to model partiality explicitly in the target language. In a preceding work Abel et al. (2005) propose such a translation from Haskell to Adga, which totalises potentially partial Haskell programs. A natural way of modelling this partiality is to use the `Maybe` monad. A more general approach is to generalise the monad at play. Indeed, the translation of Abel et al. (2005) uses a monadic lifting of Haskell programs to define partial as well as total Haskell functions in Adga. That is, functions are applied to monadic arguments and yield a monadic result. Since functions yield monadic values, the next step is to lift data type declarations as well. That is, each argument of a data type constructor becomes a monadic value. Monadic values, which are results of the lifted functions, can then be used directly in these lifted data types. All in all, the transformation produces monad generic definitions for total functions; partial functions cannot be defined for any monad, thus, partial functions are explicitly instantiated by `Maybe`.

For the actual proof of properties, the monadic context is then explicitly instantiated with the `Identity` monad for corresponding total Haskell definitions. If any partial functions are involved, we have to instantiate all functions at play with `Maybe`. Therefore, properties have to be explicitly proven in the appropriate context. Properties about functions instantiated with `Identity` cannot be reused when arguing about partial functions instantiated with `Maybe` and vice versa. Moreover, in the work of Abel et al. (2005) all properties of interest are proven for both instantiations, `Maybe` and `Identity`, respectively. As a consequence, even properties that hold independent of the underlying monadic effect are proven for both concrete instantiations.

A first try to implement a recursive Haskell data type – for example lists – in Coq following their approach failed. A recursive data type that is parametrised over a generic monad M fails to compile in Coq because of a *non strictly positive occurrence*. This strict positivity restriction for inductive data type declarations is also implemented in more recent versions of Agda, that is, this problem is not a specific to Coq, but a general restriction regarding state-of-the-art dependently typed languages. The underlying logic of the dependently typed language would become inconsistent if we allow inductive data types that do not comply to strict positivity.

In this work we propose to reanimate the approach of Abel et al. (2005) by using known solutions concerning strict positivity and show how to prove properties of Haskell programs in Coq. Besides the reanimation of the original approach, we argue that due to our modelling, we are now also able to prove monad generic properties without an unreasonable extra effort.

The first idea is to use free monads as introduced by Swierstra (2008) to represent the possible monadic effect instead of a generic monadic parameter M. Free monads capture the essence of a monad, that is, the operations `>>=` and `return`, in a data type that is parametrised over a functor F. This functor has to be strictly positive in order to define the free monad as an inductive data type. Hence, we define the free monad using containers as presented by Abbott et al. (2003). The same idea was realised by Keuchel and Schrijvers (2013) to define fix points of data types for generic programming in Coq.

In order to demonstrate the applicability of our approach we show several exemplary Haskell functions and properties for these functions. These examples demonstrate the benefit of monad-generic proofs: although some of the considered functions are partial, we prove helping lemmas for all monads. This way we can reuse these lemmas when considering statements in a total as well as in a partial setting. However, we will observe that some statements about Haskell functions only hold for all possible effects if the function at hand satisfies additional requirements. These requirements are well-known from other effectful programming languages like functional logic and probabilistic programming languages.

References

M. Abbott, T. Altenkirch, and N. Ghani. *Categories of Containers*, pages 23–38. Springer Berlin Heidelberg, 2003.

A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying haskell programs using constructive type theory. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 62–73, 2005.

S. Keuchel and T. Schrijvers. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, pages 13–24, 2013.

W. Swierstra. Data types à la carte. *Journal of functional programming*, 18(04): pages 423–436, 2008.