Machine assisted proofs in the theory of monads

Thorsten Altenkirch
University of Nottingham
UK
txa@cs.nott.ac.uk

James Chapman
Institute of Cybernetics
Estonia
james@cs.ioc.ee

Tarmo Uustalu
Institute of Cybernetics
Estonia
tarmo@cs.ioc.ee

Abstract

In this paper (and the ongoing development it describes) we formalise some aspects of the theory of monads in Agda. In doing so we give motivation for our work, relate it to previous efforts, explain why it pushes the theory and implementation of type theoretic theorem provers to breaking point, and suggest some future directions.

Introduction

Our aim is to explore the design space of formalising basic category theory in proof assistants and dependently typed programming languages. The ongoing development [4] that accompanies this paper incorporates (at the time of writing) a partial formalisation of: categories; functors; natural transformations; two notions of monads; two notions of adjunctions; Kleisli categories; two notions of Eilenberg-Moore categories; categories of adjunctions; and various constructions relating them.

Basic algebraic structures can be defined as follows: first we define some data (e.g. a set and some operations on it) which provides the basic structure; and second we define some laws which govern how the data (e.g. an operation) behaves. An example of something which fits this pattern is a category:

First the data			
$\mathbf{Obj}$	:	$\mathbf{Set}$	Set of objects
$\mathbf{Hom}$	:	$\mathbf{Obj} \to \mathbf{Obj} \to \mathbf{Set}$	Set of morphisms
$\mathbf{id}$	:	$\forall X : \mathbf{Obj}.\ \mathbf{Hom}\, X\, X$	Identity
$\mathbf{comp}$	:	$\forall X, Y, Z : \mathbf{Obj}.$	Composition
		$\mathbf{Hom}\, Y\, Z \to \mathbf{Hom}\, X\, Y \to \mathbf{Hom}\, X\, Z$	
Then the laws			
$\forall X, Y : \mathbf{Obj}.\ \forall f : \mathbf{Hom}\, X\, Y.\ \mathbf{comp}\, \mathbf{id}\, f \equiv f$			Left identity
$\forall X, Y : \mathbf{Obj}.\ \forall f : \mathbf{Hom}\, X\, Y.\ \mathbf{comp}\, f\, \mathbf{id} \equiv f$			Right identity
$\forall W, X, Y, Z : \mathbf{Obj}.$			Associativity
$\forall f : \mathbf{Hom}\, Y\, Z.\ \forall g : \mathbf{Hom}\, X\, Y.\ \forall h : \mathbf{Hom}\, W\, X.$			
$\mathbf{comp}\, f\, (\mathbf{comp}\, g\, h) \equiv \mathbf{comp}\, (\mathbf{comp}\, f\, g)\, h$			

In Agda [1] we can use dependent records and propositional equations to represent these structures quite naturally. The definition in Agda below is almost the same as the above definition except for some minor details. The sequence of fields is explicitly represented in a record which is itself a `Set` (Agda's notation for a type). Universal quantification is written as `(a : A) → _`. In this definition we use implicit universal quantification `{a : A} → _` as these arguments can often be inferred. However, if later we wish to supply them explicitly, we can, e.g. `id {X}`. Indeed, if we were being really pedantic above we should have provided the relevant objects as arguments to **id** and **comp** in the laws. The last difference is that we must give field names to the laws as we did for the data.

```
record Cat : Set where
  field Obj   : Set
        Hom   : Obj → Obj → Set
        id    : {X : Obj} → Hom X X
        comp  : {X Y Z : Obj} → Hom Y Z → Hom X Y → Hom X Z
        lid   : {X Y : Obj}{f : Hom X Y} → comp id f ≡ f
        rid   : {X Y : Obj}{f : Hom X Y} → comp f id ≡ f
        assoc : {W X Y Z : Obj}{f : Hom Y Z}{g : Hom X Y}{h : Hom W X} →
                comp f (comp g h) ≡ comp (comp f g) h
```

Compare this type of definition with Haskell where we can define the operations of a monoid and define an instance of a monoid (e.g. natural numbers, zero and addition) but we cannot show inside the system that this instance would obey the laws of a monoid. In Agda we can do both: we can write programs that make use of algebraic structure and we can reason about them, and in the process make extra guarantees that the we really have the structures that we say we do. In this paper we focus on the second aspect: reasoning about algebraic structures. Our subject is algebraic structures themselves (those related to monads) and we are using Agda to support our investigation, and development of these structures.

Related work and future directions

The way we have defined a category in Agda seems entirely natural to us: these structures really *are* dependent records and it is natural to represent the laws as (conditional) equations in propositional equality. There is no hope for using definitional equality as they are not definitions; they are laws which are to be obeyed and expect proofs. Previous efforts have centred around using setoids (sets equipped with an equivalence relation) to encode equality [8, 5, 10]. For categories, this means that a homset would be coded as a setoid, with morphism equality as the equivalence relation of the setoid. In our opinion the main problem with this approach is that the setoids clutter up the the development: very soon all one can see are setoids and the category theory becomes buried; we must spend most of our effort showing that operations respect equivalence relations instead of reasoning about category theory.

Our approach is not new; indeed, it is the most obvious. However, it has only recently become feasible and part of the reason to pursue it is to further develop type theory and its implementations to realise our goal: formalising category theory in type theory. Some of the ingredients were provided by McBride and McKinna's internalisation of pattern matching [9] first seen in the prototypic Epigram system [6] and again in the most recent version of Agda where it became useable for larger scale developments. With these advances we can get further than before but there are still some barriers. Firstly, our development is extremely strenuous on the implementation. It is easy to exhaust gigabytes of RAM with relatively small programs. Adding proof irrelevance would speed things up significantly as part of the problem is that the machine wastes time storing, manipulating and comparing equality proofs. Secondly, whilst not necessary for our development of monads, proof irrelevance and quotients are a vital piece of equipment, if one is to have a more general formalisation of category theory. It is our hope that these advances (proof irrelevance and quotients) will be provided by the the next prototype of Epigram, the theory and implementation of which the first and second author are actively engaged in. The future version of Epigram [7] will have Observational Equality [3] (which the second author recently implemented in the latest prototype) and hopes to fulfill this aim: to provide the power of setoid constructions without the technical burden on the user.

Monads

We now return to the technical content of our paper: monads. In the limited space available we will present some constructions related to monads in Agda. We define a monad in the concise Kleisli triple style as an object map `T`, a morphism `η`, and a operation on morphisms `bind`. These operations are governed by three laws. Please note that now when we refer to the field names of other records in our definitions (e.g. `Obj C` below) they act as projections which take as an argument a particular record and return the contents of the appropriate field (in this example the set of objects for the category `C`).

```
record Monad (C : Cat) : Set where
  field T₀      : Obj C → Obj C
        η       : {X : Obj C} → Hom C X (T₀ X)
        bind    : {X Y : Obj C} → Hom C X (T₀ Y) → Hom C (T₀ X) (T₀ Y)
        lidm    : {X : Obj C} → bind (η {X}) ≡ id C {T₀ X}
        ridm    : {X Y : Obj C}{f : Hom C X (T₀ Y)} → comp C (bind f) η ≡ f
        assocm : {X Y Z : Obj C}{f : Hom C Y (T₀ Z)}{g : Hom C X (T₀ Y)} →
                 comp C (bind f) (bind g) ≡ bind (comp C (bind f) g)
```

From this concise definition we can derive that T_0 is a functor, and that `η` and `bind` are natural in `X` and `X,Y` respectively. We can also show that this definition is equivalent to the more long-winded one with multiplication and also that every adjunction gives rise to a monad. We omit these details here for reasons of space. Instead we define the Kleisli category for a monad as a function which takes a monad and returns a category:

```
KleisliCat : {C : Cat} → Monad C → Cat
KleisliCat {C} T = record {Obj   = Obj C;
                           Hom   = λ X Y → Hom C X (T₀ T Y);
```

```
    id    = η T;
    comp  = λ f g → comp C (bind T f) g;
    lid   = lem1 T;
    rid   = ridm T;
    assoc = lem2 T}
```

We can see from this definition that to define the Kleisli category we must provide elements for each field in the category record including the three laws. On the right hand side of the field definitions the field names act as destructors so `Obj C` returns the object field of the category `C`. The `rid` law follows immediately from the second monad law `ridm`. The left identity and associativity we prove as lemmas by simple equational reasoning. First we prove `lem1`:

```
begin
  comp C (bind T (η T)) f
≡⟨ cong (λ X → comp C X f) (lidm T) ⟩
  comp C (id C) f
≡⟨ lid C ⟩
  f
■
```

Note that the first argument to `cong` tells Agda where to apply the second argument. Next we prove `lem2`:

```
begin
  comp C (bind T f) (comp C (bind T g) h)
≡⟨ assoc C ⟩
  comp C (comp C (bind T f) (bind T g)) h
≡⟨ cong (λ X → comp C X h) (assocm T) ⟩
  comp C (bind T (comp C (bind T f) g)) h
■
```

Conclusion

This development has already proven useful in our work on Relative Monads [2] and we are actively expanding it. Algebraic structures such as monads are being used effectively to structure functional programs in Haskell. In a dependently typed functional programming language should we simply use precisely the same approach or should we make use of the extra expressivity of dependent types to more accurately capture and exploit the algebraic structures? It is questions like this that we hope this development will support the investigation of.

References

[1] Agda team. Agda, 2009. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.

[2] T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. Submitted, 2009.

[3] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Proc. of 2007 Wksh. on Programming Languages Meet Program Verification, PLPV 2007*, pages 57–68, New York, 2007. ACM.

[4] J. Chapman. Formalisation of the theory of monads, 2009. `http://www.cs.ioc.ee/~james/repos/AssistedMonads/`.

[5] P. Dybjer and V. Gaspes. Implementing a category of sets in ALF. Technical report, Chalmers University, Gothenberg, 1994.

[6] Epigram team. Epigram, 2009. `http://www.e-pig.org`.

[7] Epigram team. Epigram blog, 2009. `http://sneezy.cs.nott.ac.uk/epilogue`.

[8] G. Huet and A. Saïbi. Constructive category theory. In C. Stirling G. Plotkin and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 239–275. MIT Press, 2000.

[9] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[10] O. Wilander. An E-bicategory of E-categories exemplifying a type-theoretic approach to bicategories. Technical report, University of Uppsala, 2005.