Group Completion & Monoid Solver in Cubical Agda

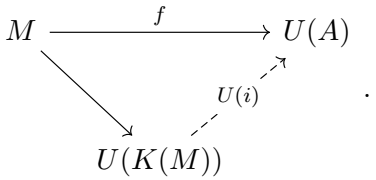
Fabian Masato Endres

February 9, 2023

Contents

Introduction

The first aim of this project was to implement the *Grothendieck group* (or *group completion*, or even *groupification*) of a commutative monoid in Cubical Agda: To a commutative monoid M we want to associate an Abelian group $K(M)$ in an universal manner. Mathematically speaking, this means that we would like to construct a left adjoint $K : \mathbf{CMon} \longrightarrow \mathbf{AbGrp}$ to the forgetful functor $U : \mathbf{AbGrp} \longrightarrow \mathbf{CMon}$. Even though we have not used the language of adjoint functors (or even functors) in Agda, we have provided a proof of the expected universal property the construction $K(M)$ should satisfy. The groupification $K(M)$ of M comes equipped with an universal morphism $M \to U(K(M))$ such that for any morphism $f : M \to U(A)$, where A is an Abelian group, there is an unique induced morphism $i : K(M) \to A$, satisfying



It was initially planned to then use this universal property in Agda, to understand the group completion as the loop space of the Eilenberg-MacLane space of M. However, in the process of completing the first stage of the project, some long and laboursome computations appeared in the proofs. These being mathematically very easy, yet occupying so much space in the code, bothered me aesthetically. Hence instead of moving on to the originally planned second part of the project, I instead implemented a solver with reflection interface for both commutative and non-commutative monoids. The commutative monoid solver was then used to tidy up the code of the implementation of the Grothendieck group a bit.

I tried to supplement this document with enough code to help me bring across my train of thought. However, when the code got a little bit more involved, I realized it is no longer best to explain line by line what my thought process was, as it tends to get too loaded and confusing. Specifically this is the case when we talk about the universal property of the group completion and the reflection interface of the monoid solver. In these cases, I try to give a less rigid overview over the subject. This also applies to the longer proofs of the project - some of them are not (fully) included here, for I intend this document to be an overview over the main points of my work, rather than an in depth dive into the technicalities. Anybody wanting to understand the code line by line is probably best advised to take a look at the code directly. I have included the complete list of this project's pull requests to the Cubical library at the end of the document.

Last but definitely not least, I would like to thank Felix Cherubini for not only introducing me to HoTT, but also to Agda and of course for his supervision on this project! He was

readily available for any question I had and invested quite a lot of time over the course of the project, helping me out immensely.

1 Group Completion of a Monoid

In this first part of the document, we implement the group completion of a commutative monoid and then state and prove its desired universal property.

1.1 The Construction

In classical mathematics, given a commutative monoid $(M, \varepsilon, \_ \cdot \_)$, one can construct its group completion by first introducing a congruence relation on the underlying set of $M \times M$:

$$(a_1, b_1) \sim (a_2, b_2) \quad :\Longleftrightarrow \quad \exists k \in M : \quad k \cdot (a_1 \cdot b_2) = k \cdot (a_2 \cdot b_1).$$

Then one sets $K(M) := M \times M / \sim$. The monoid law of $M \times M$ descends to $K(M)$ and indeed forms an Abelian group structure together with $[(\varepsilon, \varepsilon)]$ as the neutral element and $[(b, a)]$ as the inverse to $[(a, b)]$.

Perhaps unsurprisingly, this construction can be adopted without change in Agda. Indeed, given a commutative monoid `M : CommMonoid ℓ`, we proceed exactly as above, by defining a relation

```
R : ⟨ M² ⟩ → ⟨ M² ⟩ → Type ℓ
R (a₁ , b₁) (a₂ , b₂) = Σ[ k ∈ ⟨ M ⟩ ] k · (a₁ · b₂) ≡ k · (b₁ · a₂)
```

Here I should to mention, that there is no reason I use the Σ-Type instead of its truncated version ∃. Both would have worked out, in the sense that they yield groups satisfying the same universal property. Now we can use the HIT of set quotients already implemented in Cubical Agda to divide out the above relation:

```
M²/R : Type ℓ
M²/R = ⟨ M² ⟩ / R
```

Notably this means, that this quotient comes equipped with a witness `squash/` to it again being a set, which Abelian groups are by definition required to be. Furthermore, the neutral element naturally will have to be

```
0/R : M²/R
0/R = [ ε , ε ]
```

Now, in order to construct the group law and inverse law, we must analyze the relation `R`. Specifically, in order for the multiplication to descent to the set quotient, it has to be congruent:

```
isCongR : ∀ u u' v v' → R u u' → R v v' → R (u · v) (u' · v')
```

Proving this is an easy but somewhat lenghty calculation. Here I will not include the proof of congruence, but I do want to mention it for the following reason. When trying to construct

isCongR, one at some point has to prove (a variant of) the equation

$$(k \cdot s) \cdot ((a \cdot b) \cdot (c \cdot d)) \equiv (k \cdot (a \cdot c)) \cdot (s \cdot (b \cdot d))$$

for terms k, s, a, b, c, d of $\langle$ M $\rangle$. For commutative monoids this is "obviously" true, but even the shortest of proofs will be annoyingly long and require many nested applications of the commutativity and associativity law. Even worse, one does not learn anything from such a proof. Since this is not the only time such a situation came up, we decided to implement a (commutative) monoid solver, having a macro sitting on top of it that automates finding proofs for such equations. We will talk about this in more detail in the second part of this document.

Let us return to the construction of the group structure. Once we have convinced ourselves that R is congruent, it descends to a binary operation

```
_+/_ : M²/R → M²/R → M²/R
```

on the set quotient.

As far as the inverse law is concerned, recall the construction at the beginning of this section: The inverse of $[(a, b)]$ is given by $[(b, a)]$. Thus one gets an inverse law on the set quotient by showing that R is also congruent with respect to the map

```
swap : ⟨ M² ⟩ → ⟨ M² ⟩
swap = λ (a , b) → b , a
```

i.e. we claim and show that

```
h : ∀ u v → R u v → R (swap u) (swap v)
h _ _ (k , p) = k , sym p
```

holds. Now this map descends, just like the group law, to an unary operation

```
-/_ : M²/R → M²/R
```

To actually construct the descended group and inverse law, I used a helper construction already implemented in Cubical Agda (i.e. the *elimination property* of set quotients), c.f. the code of the project.

Now that we have constructed the structure required for a group, it remains to show that it satisfies the desired properties of an Abelian group. That is, we need to show the following four properties

```
assoc/R : (x y z : M²/R) → x +/ ( y +/ z) ≡ (x +/ y) +/ z
```

```
rid/R : (x : M²/R) → x +/ 0/R ≡ x
```

```
rinv/R : (x : M²/R) → x +/ (-/ x) ≡ 0/R
```

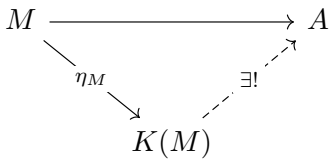
```
comm/R : (x y : M²/R) → x +/ y ≡ y +/ x
```

All of them are easy to prove by again using the elimination property of set quotients to reduce each of them to the corresponding property of the commutative monoid M^2. Except of course in the case of rinv/R, where one cannot use a corresponding property for monoids and hence must use another, but nonetheless easy, approach. Take a glimpse at the code for the here omitted proofs.

Finally, we use this knowledge to actually create the groupification of the given monoid. What I called $K(M)$ in the classical context above, I called Groupification M in the Agda code.

1.2 The Universal Property

We want to express the following mathematical property in Agda: Given a commuative monoid M and an Abelian group A, there is an Abelian group $K(M)$ together with a morphism $\eta_M : M \longrightarrow K(M)$ of monoids, such that there is an unique morphism of groups from $K(M)$ to A making the triangle



commute. Clearly, this is a reformulation of the data given by the adjunction mentioned in the introduction. The unit η_M of the adjunction is given by $m \mapsto [(m, \varepsilon)]$. Hence in Agda, we should set

```
universalHom : CommMonoidHom M (AbGroup→CommMonoid (Groupification M))
fst universalHom = λ m → [ m , ε ]
```

Then it is easily verified, that this morphism satisfies the axioms of a monoid morphism. Next, suppose we are given any morphism $f : M \longrightarrow A$ of monoids. Its unique extension $K(M) \longrightarrow A$ is given by $[(a, b)] \mapsto f(a) - f(b)$, so in Agda we first introduce a helper morphism together with a proof, that it is invariant under the relation R:

```
g = λ (a , b) → f a - f b
proof : (u v : ⟨ M² M ⟩) (r : R M u v) → g u ≡ g v
```

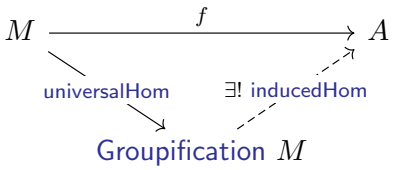
(Ignore the extra parameter M of M^2 and R. This only appears due to the fact that in the code, these were defined within another module that took the monoid as a parameter.)

Then again using the elimination property of set quotients, we get an induced morphism

```
inducedHom : AbGroupHom (Groupification M) A
fst inducedHom = elim (λ x → isSetAbGroup A) g proof
```

and it remains to checks that this is indeed a morphism of Abelian groups. This is a somewhat long (but straightforward) calculation, mainly due to the fact, that there is no solver for groups implemented in the Cubical Agda library.

The analogous version of the above triangle for our Agda implementation therefore is



It remains to see, that this triangle commutes and that the induced morphism is uniquely determined by this property. That it actually is a solution, is to claim that

$$\mathsf{solution} : (m : \langle\ M\ \rangle) \to (\mathsf{fst}\ \mathsf{inducedHom})\ (\mathsf{fst}\ \mathsf{universalHom}\ m) \equiv \mathsf{f}\ m$$

and that the solution is unique, is to claim that

$$\begin{aligned}\mathsf{unique} : {} & (\psi : \mathsf{AbGroupHom}\ (\mathsf{Groupification}\ M)\ A) \\ & \to (\psi \mathit{IsSolution} : (m : \langle\ M\ \rangle) \to \psi\ .\mathsf{fst}\ (\mathsf{fst}\ \mathsf{universalHom}\ m) \equiv \mathsf{f}\ m) \\ & \to (u : \langle\ \mathsf{M}^2\ M\ \rangle) \to \psi\ .\mathsf{fst}\ [\ u\ ] \equiv \mathsf{inducedHom}\ .\mathsf{fst}\ [\ u\ ]\end{aligned}$$

Both of these claims are proven in the code of the project, therefore we have verified the desired universal property of the groupification of monoids.

2 Monoid Solver

Let us recall the motivation for having a monoid solver. While constructing the Grothendieck group of a monoid, we encountered the problem of efficiently solving equations of the sort

$$(k \cdot s) \cdot ((a \cdot b) \cdot (c \cdot d)) \equiv (k \cdot (a \cdot c)) \cdot (s \cdot (b \cdot d)).$$

With the help of the solver, we can conveniently automate the proof finding process:

```
lemma : ∀ k s a b c d → (k · s) · ((a · b) · (c · d)) ≡ (k · (a · c)) · (s · (b · d))
lemma = solveCommMonoid M
```

The solver I implemented is not without limitations. It can be the case, that it fails to construct a proof for a true equation, for instance if more than just the monoidal properties are used: Maybe the monoid also has the structure of a group, then the solver would fail to verify $a^{-1} \cdot a \equiv \varepsilon$. Also, only equations where the occuring variables are universally quantified can be solved.

The solver essentially consists of two components - a *naive solver* and a *reflection interface*.

2.1 Naive Solving

We want to verify a given equation in a commutative monoid, say

$$(x \cdot y) \cdot x \equiv x \cdot (x \cdot y) \cdot \varepsilon. \tag{Ex}$$

I will now outline the idea of solving this equation automatically in Agda. In a first step, we look at the left and right hand side separately and represent them as syntax trees. We do this by introducing a type of expressions in n indeterminants:

```
data Expr (M : Type ℓ) (n : ℕ) : Type ℓ where
  |_     : Fin n → Expr M n
  ε⊗     : Expr M n
  _⊗_    : Expr M n → Expr M n → Expr M n
```

Ultimately, we want to evaluate such an expression to produce a term in our monoid. Thus we need a way to remember which terms the constant expressions $|(j)$ represented in the first place. For instance in the above example we should have $n = 2$ and $|(1)$, $|(2)$ could evaluate to x, y respectively. Clearly $\varepsilon\otimes$ should evaluate to the unit ε of the monoid. In order to remember this assignment, we store the in (Ex) ocurring variables in an *environment vector* of the form

```
Env : ℕ → Type ℓ
Env n = Vec ⟨ M ⟩ n
```

i.e. the type of vectors of length n. Constructing such an environment is automatically done by the reflection interface, which we have yet to discuss. In our example, the environment vector is the vector v with x and y as entries (in that order). Now we can evaluate any expression with respect to a given environment:

```
⟦_⟧ : ∀{n} → Expr ⟨ M ⟩ n → Env n → ⟨ M ⟩
⟦ ε⊗ ⟧       v = ε
⟦ | i ⟧      v = lookup i v
⟦ e₁ ⊗ e₂ ⟧  v = ⟦ e₁ ⟧ v · ⟦ e₂ ⟧ v
```

We now manipulate expressions by *normalizing* them. The idea is based on the observation, that an equation in a commutative monoid is true, as soon as every variable in it occurs with the same multiplicity on both the left and right hand side. Guided by this, the normalization of an expression should capture exactly these multiplicities. Hence we define the normal form and normalization of an expression in the following way.

```
NormalForm : ℕ → Type _
NormalForm n = Vec ℕ n


normalize : {n : ℕ} → Expr ⟨ M ⟩ n → NormalForm n
normalize (| i)      = e[ i ]
normalize ε⊗         = emptyForm
normalize (e₁ ⊗ e₂)  = (normalize e₁) ⊞ (normalize e₂)
```

Here e[i] is the i-th unit vector, emptyForm is the zero vector and $\boxplus$ denotes the componentwise addition of vectors[1]. For instance, the normalization of the syntax tree $\big(\ |(1) \otimes |(2)\ \big) \otimes |(1)$ is the vector $2 :: 1 :: []$.

We can also evaluate a normal form with respect to a given environment vector. Informally, I like to think about this evaluation as a multiplication of the normal form vector with the transpose of the environment vector. The formal implementation is the following.

```
eval : {n : ℕ} → NormalForm n → Env n → ⟨ M ⟩
eval []       v       = ε
eval (x :: xs) (v :: vs) = iter x (λ w → v · w) (eval xs vs)
```

For instance, the evaluation of the normal form $2 :: 1 :: []$ with respect to the environment vector $x :: y :: []$ is $x \cdot (x \cdot y) \cdot \varepsilon$.

Therefore, we now have two ways of evaluating an expression. We either evaluate with the function ⟦ ⟧ as is, exactly reproducing the term it was representing, or we first normalize it and then evaluate it with eval. With a little bit of effort one can prove that *evaluation is invariant under normalization*, i.e. that these two ways of evaluation produce the exact same term in our monoid:

```
isEqualToNormalform : {n : ℕ}
                    → (e : Expr ⟨ M ⟩ n)
                    → (v : Env n)
                    → eval (normalize e) v ≡ ⟦ e ⟧ v
```

We can use this to solve equations like (Ex) by representing the left and right hand side by expressions, then checking if the evaluations of the respective normalizations match

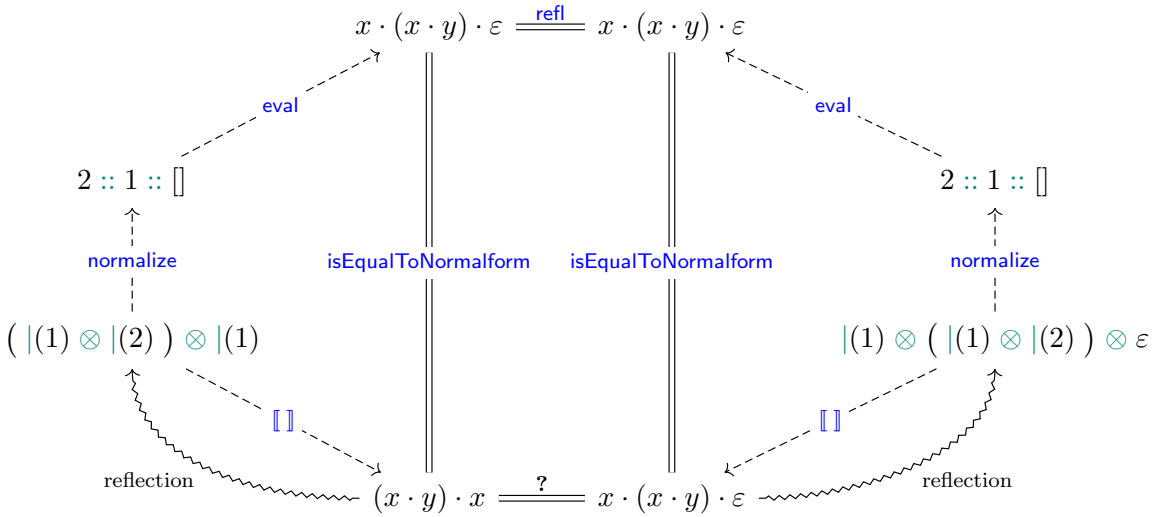
[1]For a formal definition of these notions take a direct look at the Agda code!

and finally composing these proofs to obtain a witness to the original equation. Formally speaking, we claim

```
solve : {n : ℕ}
      → (e₁ e₂ : Expr ⟨ M ⟩ n)
      → (v : Env n)
      → (p : eval (normalize e₁) v ≡ eval (normalize e₂) v)
      → ⟦ e₁ ⟧ v ≡ ⟦ e₂ ⟧ v
```

which easily follows from the invariance of evaluation under normalization.

In case of the example (Ex), this process of solving equations might be schematically illustrated as follows:



Note that there is a small imprecision in this diagram, in that the evaluation without normalization ⟦ ⟧ as well as the evaluation of the normalization eval depend on an additional parameter, namely on the environment vector. In this case, this vector is again $x :: y :: []$.

Now that we have talked about solving equations in commutative monoids, let us quickly ponder how to adapt this approach to not necessarily commutative monoids. To make the above approach work, we only have to change what we mean by a normal form of an expression. Clearly, only counting the occurances of every variable $|(i)$ will not suffice, since we now also have to take the order they appear in into account. Hence we should replace the normal forms by the following adjustion.

```
NormalForm : ℕ → Type _
NormalForm n = List (Fin n)
```

Replacing the definition of normal forms and normalization in the above discussion with this, one may reread this section to obtain a naive solver for general monoids. In the project, I have implemented both a solver for commuative and non-commutative monoids - the code being basically the same apart from the adjustment mentioned above. In particular, the reflection interface we will discuss next is exactly the same for both!

2.2 Reflection Interface

One thing we did not explain how to do yet, is how to represent the left and right hand side of an equation in a monoid with syntax trees. Of course, one may do this *naively* by constructing the corresponding expressions manually. In a rather trivial situation this would look like the following

```
module Example (M : CommMonoid ℓ) where
  open CommMonoidStr (snd M)

  _ : ε · ε · ε ≡ ε
  _ = solve M (ε⊗ ⊗ ε⊗ ⊗ ε⊗) ε⊗ [] refl
```

where solve is the function from the previous section. Usually the corresponding syntax trees will be much more difficult than in this silly example, so maybe we can do even better than this naive approach. Indeed, there is an automated solution to the issue: The *reflection interface*. Note, that there already was a ring solver with its very own reflection interface implemented in Cubical Agda, so I did not have to start from scratch and only had to tweak the existing code.

Let us quickly capture the essence of the main routine of the reflection solver. Omitting the definitions of the technical helper functions, the core code is the following.

```
module ReflectionSolver (op unit solver : Name) where


- [ ... ]


solve-macro : Term → Term → TC Unit
solve-macro monoid hole =
  do
    hole′ ← inferType hole >>= normalise
    just (varInfos , equation) ← returnTC (getVarsAndEquation hole′)
      where
        nothing
          → typeError (strErr "Something went wrong when getting the variable names in "
                        :: termErr hole′ :: [])
    {-          The call to the monoid solver will be inside a lamba-expression.
    adjustedMonoid ← returnTC (adjustDeBruijnIndex (length varInfos) monoid)
    just (lhs , rhs) ← returnTC (toMonoidExpression adjustedMonoid (getArgs equation))
      where
        nothing
          → typeError(
              strErr "Error while trying to build ASTs for the equation " ::
              termErr equation :: [])
    let solution = solverCallWithLambdas (length varInfos) varInfos adjustedMonoid lhs rhs
    unify hole solution
```

First of all, the module is parametrized over the names of the monoid operation *op* and the *unit* of the monoid, as well as the naive *solver* we developed in the preceeding section.

These will be supplied to the actual macro by quoting the respective field accessor.

Now, when solve-macro is called, (informally and very vaguely speaking) four steps happen in the do-block:

1) **Introduction of *varInfo* and *equation***

   In the from *hole'* inferred list of arguments, we isolate the variables of the equation and group them in a list *varInfo*. We also isolate the *equation* itself. Here I will not give the exact type of *varInfo*, but in the next steps we will see, that it is important to at least remember the deBrujin-index of every variable in *varInfo*.

2) **Introduction of *adjustedMonoid***

   Because the call to the (naive) monoid solver will sit inside a lambda-expression, the deBruijn-indices of the variables of *monoid* need to be shifted by the length of *varInfo*, yielding an *adjustedMonoid*. This is necessary, because the index of a variable will correspond to the respective entry of the environment vector.

3) **Introduction of *lhs* and *rhs***

   First, we define a function buildExpression, taking a syntax tree representing the left or right hand side of an equation and converting it into a syntax tree representing the corresponding monoid expression. The definition is rather enlightening, so let me state it here.

```
module _ (monoid : Term) where

  `ε⊗` : Term
  `ε⊗` = con (quote ε⊗) []

  mutual

    `_⊗_` : List (Arg Term) → Term
    `_⊗_` (harg _ :: xs) = `_⊗_` xs
    `_⊗_` (varg _ :: varg x :: varg y :: []) =
      con
        (quote _⊗_) (varg (buildExpression x) :: varg (buildExpression y) :: [])
    `_⊗_` _ = unknown

    finiteNumberAsTerm : ℕ → Term
    finiteNumberAsTerm ℕ.zero = con (quote fzero) []
    finiteNumberAsTerm (ℕ.suc n) = con (quote fsuc) (varg (finiteNumberAsTerm n) :: [])

    buildExpression : Term → Term
    buildExpression (var index _) = con (quote |) (varg (finiteNumberAsTerm index) :: [])
    buildExpression t@(def n xs) =
      if (n == op)
        then `_⊗_` xs
      else if (n == unit)
        then `ε⊗`
      else
        unknown
```

Here, the names *op* and *unit* are supplied as parameters by the parent module[2] this anonymous module sits in. We then take this definition and put it into a wrapper function

```
toMonoidExpression : Maybe (Term × Term) → Maybe (Term × Term)
toMonoidExpression nothing = nothing
toMonoidExpression (just (lhs , rhs)) = just (buildExpression lhs , buildExpression rhs)
```

We use this in the macro as follows. A function (getArgs *equation*) takes the term *equation* - maybe visualized as of the form $lhs' \equiv rhs'$, and return the pair of terms (lhs', rhs'), to which we apply toMonoidExpression to obtain *lhs* and *rhs*.

4) **Introduction of** *solution*

Note that we are now essentially ready to connect the reflection interface with the naive solver of the previous section. All that is missing, is a function solverCallWithLambdas which will call the naive *solver* and input *lhs* and *rhs* into it.

Now that we have (informally) explained the main routine of the macro, we set

```
macro
  solveMonoid : Term → Term → TC _
  solveMonoid = ReflectionSolver.solve-macro (quote MonoidStr._·_)
                                             (quote MonoidStr.ε)
                                             (quote naiveSolve)

  solveCommMonoid : Term → Term → TC _
  solveCommMonoid = ReflectionSolver.solve-macro (quote CommMonoidStr._·_)
                                                 (quote CommMonoidStr.ε)
                                                 (quote naiveCommSolve)
```

where naiveSolve (resp. naiveCommSolve) is the solve function for monoids (resp. commutative monoids) developed in the previous section. We can test this macro with some examples:

```
module ExamplesMonoid (M : Monoid ℓ) where
  open MonoidStr (snd M)

  _ : ε ≡ ε
  _ = solveMonoid M

  _ : ε · ε · ε ≡ ε
  _ = solveMonoid M

  _ : ∀ x → ε · x ≡ x
  _ = solveMonoid M

  _ : ∀ x y z → (x · y) · z ≡ x · (y · z)
  _ = solveMonoid M

  _ : ∀ x y z → z · (x · y) · ε · z ≡ z · x · (y · z)
  _ = solveMonoid M
```

[2]Which is the module "ReflectionSolver" introduced at the beginning of this section, to be precise.

3 List of Pull Requests

- *Commit 587 in Cubical Agda, Grothendieck Groups of Commutative Monoids*, URL: https://github.com/agda/cubical/pull/587
- *Commit 708 in Cubical Agda, Monoid Solver*, URL: https://github.com/agda/cubical/pull/708