# A Very Short Introduction to Agda's Reflection Mechanism

Xiaoning Bian and Andrew Christopher Hawk

August 14, 2024

This code is self contained and checked by Agda version 2.6.4.3.

## 1 Introduction

Suppose we have a term (program) $t$. We can evaluate it to get, say, $t'$, which is what we normally do about a term. We can also manipulate its syntactic expression $s$, rewrite $s$ to $s'$ somehow, and then somehow turn $s'$ back to some program $t''$. The second way is called "reflection", since we manipulate $t$ by reflecting on $t$ using $s$ and $s'$ to get $t''$.

There are two kinds of reflection:

1. The kind that is supported by Agda's builtin keywords ("quote", "unquote", ...), which reflects on an Agda term by manipulating its Agda-AST representation. No parsing is needed here, as we can just quote the terms to get the Agda-AST representation.
2. The kind that doesn't need keyword support, which reflects on an Agda term by manipulating a user-defined AST that is somehow "isomorphic" to the term's structure, i.e., captures the main structure of the term. As an example, the structure of the term $x + y * z$" with $x\ y\ z$ being variables, can be captured by a syntax tree. The term-to-AST parsing is manual, i.e., we have to manually type "$x :+ y :* z$" to build the AST for the term.

Both share "reflection", i.e., both manipulate the term (program) by AST and do not directly execute the term.

Both 1) and 2) are used in Agda programming. Combing both seems good, and, actually, many have done so — a quick GitHub search for "agda reflection" should yield a couple of interesting repos.

"Combining both" actually refers to the following sequence of steps:

1. Use 1) to get Agda-AST automatically.
2. Translate the resulting Agda-AST representation to a user-defined AST.
3. Finally, do the proof under 2).

This process prevents 2)'s manual typing part. We will, however, see that there are complications in this combining.

We only explain the first kind. For the second kind, there is a very short and nice explanation in Section 4 of: `https://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Documentation?action=download&upname=AgdaOverview2009.pdf`

```
{-# OPTIONS --without-K --safe --cubical-compatible #-}
module ShortIntroReflection where
```

# 2 Basic Types and Associated Functions

We begin by defining the Booleans and natural numbers, as well as some associated functions, e.g., addition and the equality check function.

## 2.1 The Boolean Type

```
data Bool : Set where
  true : Bool
  false : Bool
```

## 2.2 The Natural Number Type

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ

infixr 7 _+_
_+_ : ℕ → ℕ → ℕ
zero + y = y
succ x + y = succ (x + y)

infix 6 _=?_
_=?_ : ℕ → ℕ → Bool
zero =? zero = true
zero =? succ y = false
succ x =? zero = false
succ x =? succ y = x =? y
```

## 2.3 Equality

We *also* define equality, which is an equivalence relation and a congruence. In addition, we define some functions which operate on values of this new equality type.

```
infix 6 _==_
data _==_ {A : Set} (a : A) : A → Set where
  refl : a == a
```

```
sym : ∀ {A : Set} {a b : A} → a == b → b == a
sym refl = refl

trans : ∀ {A : Set} {a b c : A} → a == b → b == c → a == c
trans refl refl = refl

cong : ∀ {A B : Set} {a a' : A} →
       (f : A → B) → a == a' → f a == f a'
cong f refl = refl

cong2 : ∀ {A B C : Set} {a a' : A} {b b' : B} →
        (f : A → B → C) → a == a' → b == b' → f a b == f a' b'
cong2 f refl refl = refl
```

### 2.4 Example of Application

A lemma says $n + 0 = 0$ for all $n$.

```
lemma-n+0 : ∀ {n} → n + zero == n
lemma-n+0 {zero} = refl
lemma-n+0 {(succ n)} with lemma-n+0 {n}
... | ih = cong succ ih
```

## 3 Rewriting

Suppose now, given a term $(n + \mathsf{zero}) : \mathbb{N}$, we want rewrite it to $(n)$. One way to do it manually is:

```
--  n + 0 ≡< lemma-n+0 >
--  n


rewrite-manually : ∀ n → succ (n + zero) == succ n
rewrite-manually n = cong succ lemma-n+0
```

If we have _==_ as the builtin equality, then we can use the "rewrite" keyword. But if we are doing setoid reasoning, then we don't have the "rewrite" keyword anymore.

Manually doing many of these rewritings would be incredibly tedious, so we should really look for a different method. Another way is to translate an expression to, say, a string, and then write a program to replace "`n + 0`" with "`n`". We could then translate the string back to terms and somehow insert "`lemma-n+0`" at the right place. We would probably need some structure that is better than strings to support recording places where "`lemma-n+0`" is needed, though.

So we use an AST such as

```
infixr 8 _:+_
data AST : Set where
```

```
Leaf : ℕ → AST
_:+_ : AST → AST → AST
```

We parse expressions like "`x + y`" in as $x :+ y$ and do rewriting on it. Note that we cannot rewrite directly on $n$ + zero, since _+_ is a fuction, and $n$ + zero is a number without using an AST. Let f be as follows:

```
f : ℕ → AST → AST
f n (Leaf m) = Leaf m
f n t@(Leaf m :+ Leaf zero) with m =? n
... | true = Leaf m
... | false = t
f n (l :+ r) = f n l :+ f n r
```

So f $n$ rewrites all Leaf $n$ :+ Leaf zero to Leaf $n$.

```
test-f : _==_ (f (succ zero) (Leaf zero :+ Leaf zero :+ Leaf (succ zero) :+ Leaf zero))
              (Leaf zero :+ Leaf zero :+ Leaf (succ zero))
test-f = refl
```

How should we parse an expression to an AST? At least we can do manual parsing for finitely many expressions… For now, suppose we have the following translations, where "AE" denotes the type of arithmetic expressions:

- [_] : AE → AST
- ⟦_⟧ : AST → AE

AST to expression is easy:

```
⟦_⟧ : AST → ℕ
⟦ Leaf x ⟧ = x
⟦ x :+ x₁ ⟧ = ⟦ x ⟧ + ⟦ x₁ ⟧
```

If we have properties about them:

```
-- ⟦_⟧ ∘ [_] = id      (1)
-- ⟦_⟧ ∘ f = ⟦_⟧       (2)
```

Then we see that $e$ equals id $e$, which is equal to (⟦_⟧ ∘ [_]) $e$, which is equal to (⟦_⟧ ∘ $f$ ∘ [_]) $e$.

The last is an expression with all "`n + zero`" replaced by "n". Note that we can prove (2) by induction on the AST.

```
lemma-⟦⟧ : ∀ n a → ⟦ a ⟧ == ⟦ f n a ⟧
lemma-⟦⟧ n (Leaf x) = refl
lemma-⟦⟧ n (Leaf x :+ Leaf zero) with x =? n
... | true = lemma-n+0
```

```
... | false = refl
lemma-[[] n (Leaf x :+ Leaf (succ x₁)) = refl
lemma-[[] n (Leaf x :+ a₁ :+ a₂) with lemma-[[] n (a₁ :+ a₂)
... | ih = cong (x +_) ih
lemma-[[] n ((a :+ a₂) :+ a₁) with lemma-[[] n (a :+ a₂) | lemma-[[] n a₁
... | ih1 | ih2 = cong2 _+_ ih1 ih2
```

But for (1), there seems no hope. Happily, reflection comes to the rescue!

Using reflection, we can access $(x + y)$'s Agda-AST representation, which is again a term but like ((quote _+_) [ quote $x$ , quote $y$ ]):

```
open import Agda.Builtin.Reflection
open import Agda.Builtin.List

-- We use quoteTerm to get the Agda-AST of an expression.
eg-quote : quoteTerm (zero + zero) ==
  def (quote _+_)
      ( let modalityω = modality relevant quantity-ω in
        arg (arg-info visible modalityω) (con (quote zero) [])
      :: arg (arg-info visible modalityω) (con (quote zero) [])
      :: [])
eg-quote = refl
```

The important thing is that if we do unquote to Agda's AST, we sort of get things back like:

```
-- unquote ( (quote _+_) [quote x, quote y] ) = x + y ....


eg-unquote-refl : zero == zero
eg-unquote-refl = unquote (\h → unify h (quoteTerm (refl {a = zero})))
```

unquote does seem to cancel quoteTerm, but *only* in Agda's TC monad.[1] So we can only have (1) within the TC monad. Luckily, one of the side-effects of TC is that it does satisfy some holes when used properly. So we don't really need to get (1) out of TC; rather, we just use it inside and use the side-effect to make a hole satisfied.

Let's see what's unquote doing under the hood:

```
open import Agda.Builtin.Unit

-- Break "eg-unquote-refl" into two parts:
refl0 : Term → TC ⊤
refl0 h = unify h (quoteTerm (refl {ℕ} {zero}))
```

[1] The reader may or may not care to know that "TC" is an abbreviation of "TypeChecking".

```
eg-unquote' : zero == zero
eg-unquote' = unquote refl0
```

We see that what we unquote is refl0, which is of type Term → TC ⊤. We can only unquote things of such type.

Here, eg-unquote' is wanting a value refl, and refl0 *is* more or less a quoted refl… but unify and $h$ are in the way.

Actually, to unquote is to perform the computation m : Term → TC ⊤. (In other words, a TC monad computation with return type ⊤ parameterised by Term). unquote will trigger the following sequence according to the Agda manual:

1. Check if $m$ is of type Term → TC ⊤.
2. Create a fresh hole where the unquote clause is, we normally know the type of the hole, because that is what we want to show. we put unquote where we put a proof term. A hole is also a metavarible, so we say now that Agda has $v : A$ at disposal.
3. Let qv : Term be the quoted representation of $v$, i.e., "qv = unquote $v$".
4. Execute $m$ qv. Mainly excute unify $q$ $v$ $s$. If qv and the other quoted term $s$ (e.g. unquoteTerm (refl {ℕ} {zero})) unify, then $s$ is a proof that fits in the hole, and Agda is accordingly satisfied.

If you want, you can think that Agda replaces the hole with the unquoted $s$, i.e., an editor's AST-unquoted expression or string. But this thinking is just a mnemonic. After all, Agda can only understand AST. The gives a way to hint Agda when Agda is doing typechecking.

Is this reflection stuff still safe? Yes, it is. We didn't change the typechecking algorithm at all; we only provide extra inputs and extra information for the algorithm.

Return to our problem: How do we automatically rewrite when the equation is available only within the TC monad? First, we do the promised translation from Agda-AST to a *quoted* form of our AST. This quote stuff is the complication mentioned at the begining.

```
-- commonly used argument info:
pattern ai = arg-info visible (modality relevant quantity-ω)
```

Note that this function only works for expressions of the form $x + y + z$. It translates the quoted $x + y + z$ to quoted $x :+ y :+ z$.

```
myast-of : Term → Term
myast-of (con (quote zero) []) = con (quote Leaf) (arg ai (con (quote zero) []) :: [])
myast-of (con (quote succ) (arg i a :: [])) with myast-of a
... | con (quote Leaf) args = con (quote Leaf) (arg ai (con (quote succ) args) :: [])
... | _ = unknown
myast-of (def (quote _+_) (arg i1 a1 :: arg i2 a2 :: [])) =
  con (quote _:+_) (arg ai (myast-of a1) :: arg ai (myast-of a2) :: [])
myast-of _ = unknown
```

Let's check that it works as expected:

```
check1 : myast-of (quoteTerm ( zero + zero)) == quoteTerm (Leaf zero :+ Leaf zero)
check1 = refl
```

Note that check1 is a weaker form of equation (1): recall

```
-- ⟦_⟧ ∘ [_] = id (1), but here we have:

-- defining [_] as unquote ∘ myast-of ∘ quoteTerm, we see by varying the
-- equation in check1 and thinking of unquote as the inverse of
-- quoteTerm (only true in TC) that:

-- [ e ] = e' where e' is the corresponding term in our AST, i.e.,
-- such defined [ ] is indeed a translation:

-- [_] : ArithmeticExpression → AST
```

Then we see that (1) should follow, but we currently lack an Agda proof.

We want to automatically rewrite many $n$ + zero while keeping equality, so we need to also able to syntactically manipulate equality. As an example, we can extract the quoted lhs from quoted equality:

```
lhs : Term → Term
lhs (def (quote _==_) ((arg i1 a1) :: (arg i2 a2) :: (arg i3 a3) :: [])) = a2
lhs t = unknown
```

And we have quoted proof of ⟦ lhs ⟧ == ⟦ $f$ $n$ lhs ⟧, i.e., proof of correctness of the rewriting "`f n`" in quoted form. (with all arguments in quoted form).

```
q'⟦⟧ : Term → Term → Term
q'⟦⟧ n lhs = def (quote lemma-⟦⟧) (arg ai n :: arg ai lhs :: [])
```

A parameterised TC computation that try to unify hole with provide solution. We only use Agda buitins, so it's a little mess. Using do-notation will be cleaner. But still easy to see what's there:

1. normalize the type of the hole
2. take out lhs
3. use q'⟦⟧ on lhs to generate a proof sol
4. unify hole and sol.

```
comp : Term → Term → TC ⊤
comp n hole =
  bindTC
    (bindTC
```

```
    (bindTC
      (bindTC
        (inferType hole)
        normalise
      )
      (\hole' → returnTC (myast-of (lhs hole')))
    )
    (\l' → returnTC (q'[[]] n l'))
  )
  (\sol → unify hole sol)
-- Note that if you "unify hole hole", unquote (comp n) will give
-- you a yellow background, indicating an unsolved metavariable,
-- i.e., an unsolved hole.
```

Finally,

```
finally1 : (succ zero + zero) + (succ zero + zero) == succ zero + succ zero
finally1 = unquote (comp (quoteTerm (succ zero)))
```

We can use macro to hide unquote and quoteTerm.

```
macro
  comp' : Term → Term → TC ⊤
  comp' = comp

finally2 : (succ zero + zero) + (succ zero + zero) == succ zero + succ zero
finally2 = comp' (succ zero)
```

Done! Thanks for reading.