Engineering Proof by Reflection in Agda

Paul van der Walt and Wouter Swierstra

Department of Computer Science, Utrecht University, The Netherlands
paul@denknerd.org, W.S.Swierstra@uu.nl

Abstract. This paper explores the recent addition to Agda enabling *reflection*, in the style of Lisp [1], MetaML [2], and Template Haskell [3]. It gives a brief introduction to using reflection, and details the intricacies of using reflection to automate certain proofs using *proof by reflection*. It presents a library that can be used for automatically quoting a class of concrete Agda terms to a simple, user-defined inductive data type, alleviating the burden a programmer usually faces when using reflection in a practical setting.

Keywords: dependently-typed programming, reflection, Agda, proof by reflection, metaprogramming

1 Introduction

The dependently typed programming language Agda [4,5] has recently been extended with a *reflection mechanism* for compile time metaprogramming in the style of Lisp [6], MetaML [2], Template Haskell [3], and C++ templates [7]. Agda's reflection mechanisms make it possible to convert a program fragment into its corresponding abstract syntax tree (AST) and vice versa. In tandem with Agda's dependent types, this has promising new programming potential.

This paper addresses the following central questions:

> "What practical issues do we run into when trying to engineer automatic proofs in a dependently typed language with reflection? Are Agda's reflective capabilities sufficient and practically usable, and if not, which improvements might make life easier?"

Contributions. This paper reports on the experience of using Agda's reflection mechanism to automate certain categories of proofs. This is a case study, exemplative of the kind of problems that can be solved using reflection. More specifically this work makes the following contributions:

– We give a brief overview of Agda's reflection mechanism (Sec. 2). Previously, these features were only documented in release notes and comments in Agda's source files.
– We present Autoquote, an Agda library that alleviates much of a programmer's burden when quoting a given abstract syntax tree (Sec. 3).

– We show how to use Agda's reflection mechanism to automate certain categories of proofs (Sec. 4). The idea of *proof by reflection* is certainly not new, but still worth examining in the context of this new technology.

The code presented in this paper compiles using the latest version of Agda (currently 2.3.2). Supporting code, including this paper, is available on GitHub. [1] This paper is also a Literate Agda file, which means the code snippets can be extracted, compiled, and adapted.

1.1 Introducing Agda

Agda is an implementation of Martin-Löf's type theory [8], extended with records and modules. It is developed at the Chalmers University of Technology [4]; in accordance with Curry–Howard isomorphism, it can be viewed as both a functional programming language and a proof assistant for intuitionistic logic. It is comparable to Coq, which is based on Coquand's calculus of constructions [9]. For an excellent tutorial on dependently typed programming using Agda, the reader is referred to Norell's work [5].

Since version 2.2.8, Agda includes a reflection API [10], which allows the conversion of parts of a program's code into an abstract syntax tree, a data structure in Agda itself, that can be inspected or modified like any other data structure. The idea of reflection is old: in the 1980s Lisp included a similar feature, then already called *quoting* and *unquoting*, which allowed run time modification of a program's code, for example by the program itself. This has given rise to powerful techniques for code reuse and abstraction.

This paper explores how such a reflection mechanism can be used in a *dependently typed* language such as Agda.

2 Using Reflection

Before going into too much detail about how reflection can make our life easier and what new programming techniques are possible, we look at Agda's reflection API in some detail. It should be a good starting point for someone familiar with Agda, or at the very least dependently typed programming in general.

The Keywords. There are several new keywords that can be used to quote and unquote Term values: quote, quoteTerm, quoteGoal, and unquote. The quote keyword allows the user to access the internal representation of any identifier. This internal representation can be used to query the type or definition of the identifier. We refer to the release notes [10] for a listing of the data structures involved; the most important one is the type Term : Set which represents concrete Agda terms.

The easiest example of quotation uses the quoteTerm keyword to turn a fragment of concrete syntax into a Term value. Note that the quoteTerm keyword

[1] https://github.com/toothbrush/reflection-proofs

reduces like any other function in Agda. As an example, the following unit test type checks:

```
example₀ : quoteTerm (λ (x : Bool) → x) ≡ lam visible (var 0 [])
example₀ = refl
```

Dissecting this, we introduced a lambda abstraction, so we expect the lam constructor. Its one argument is visible (as opposed to hidden), and the body of the lambda abstraction is just a reference to the nearest-bound variable, thus var 0, applied to an empty list of arguments. Variables are referred to by their De Bruijn indices.

Furthermore, quoteTerm type checks and normalises its term before returning the Term, as the following example demonstrates:

```
example₁ : quoteTerm ((λ x → x) 0) ≡ con (quote ℕ.zero) []
example₁ = refl
```

See how the identity function is applied to zero, resulting in only the value zero. The quoted representation of a natural zero is con (quote zero) [], where con means that we are introducing a constructor. The constructor zero takes no arguments, hence the empty list.

The quoteGoal keyword is slightly different. It is best explained using an example:

```
example₂ : ℕ
example₂ = quoteGoal e in { }0
```

The quoteGoal keyword binds the variable e to the Term representing the type of the current goal. In this example, the value of e in the hole will be def ℕ [], i.e., the Term representing the type ℕ.

The unquote keyword converts a Term data type back to concrete syntax. Just as quoteTerm and quoteGoal, it type checks and normalises the Term before it is spliced into the program text.

Unfortunately, we cannot pattern match on constructor names without further ado. This is why we have decidable equality on the following types: Visibility, Relevance, List Args, Arg Types, Arg Terms, Names, Terms, Sorts and Types. Typically, this is useful for deciding which constructor is present in some expression, by comparing to known Names (obtained using quote). Such a comparison is illustrated in the function convert, below.

```
convert : Term → Something
convert (con c args) with c ≟-Name quote foo
... | yes p  = { }0    -- foo applied to arguments
... | no ¬p  = { }1    -- a function other than foo
```

This short introduction should already be enough to start developing simple programs using reflection. For a more detailed description of the reflection API in Agda, including many examples, the inquisitive reader is referred to the chapter in van der Walt's thesis covering this topic [11]. This thesis goes into more detail regarding the data structures involved in Agda's reflection API, and later, gives a detailed account of some real-world applications.

3 Automatic Quoting

In the previous section, we saw how we can get hold of values of type Term, representing concrete Agda terms. This is a great start, but we rarely want to directly manipulate Terms: often it is much more useful to use our own AST for computations. It sounds like a minor task to write a quick function to convert a Term into something more useful. However, it turns out this often becomes a mess.

In a language like Haskell, which has pattern matching, converting elements of one AST to another is usually a simple, if boring, task. Unfortunately for us, though, Agda functions are required to be total, which means they must have a case branch for each possible pattern. Since Term covers all quotable terms, it has many alternatives. Furthermore, we cannot pattern match on the names of the constructors, so we must resort to using decidable equality. This is why such a conversion function is often verbose, as can be seen in the code snippet of Fig. 1, an excerpt of an actual conversion function, used before a better solution was developed.

```
term2boolexpr n (con tf []) pf with tf ≟-Name quote true
term2boolexpr n (con tf []) pf | yes p = Truth
...
term2boolexpr n (def f []) ()
term2boolexpr n (def f (arg v r x :: [])) pf with f ≟-Name quote ¬_
... | yes p = Not (term2boolexpr n x pf)
... | no ¬p with f ≟-Name quote _∧_
...
```

Fig. 1. The gist of the conversion of a Term into some more specific data type.

The (partial) solution to this problem – something which at least mitigates the agony – is presented in this section, in the form of the Autoquote library.

The Autoquote Library. We will examine a toy AST, called Expr, shown in Fig. 2. It is a rather simple inductive data structure representing terms which can contain Peano-style natural numbers, variables (represented by an Agda natural) and additions.

```
data Expr : Set where
  Var  : ℕ                → Expr
  Plus : Expr → Expr → Expr
  S    : Expr             → Expr
  Z    :                    Expr
```

Fig. 2. The toy expression language Expr. We would like support for automatically quoting such terms.

We might want to convert an expression, such as $5 + x$, to this AST using reflection. In an ideal world, we would just provide a mapping from concrete constructs such as the _+_ function to elements like Plus of our AST, and get a conversion function for free. The Autoquote library does just this, exposing an interface which, when provided with such a mapping, automatically quotes expressions that fit. Here, *fitting* is defined as only having names that are listed in the mapping, or variables. Other terms are rejected. The user provides an elegant mapping, such as in Fig. 3, and Autoquote automatically converts Agda terms to elements of the AST.

```
exprTable : Table Expr
exprTable = (Var,
              (quote _+_)  # 2 ↦ Plus ::
              (quote ℕ.zero) # 0 ↦ Z    ::
              (quote ℕ.suc)  # 1 ↦ S    :: [])
```

Fig. 3. The mapping table for converting to the imaginary Expr AST.

This should be interpreted as follows: any variables encountered should be stored as Vars, and the _+_ operator should be mapped to a Plus constructor. In each case we are required to manually specify the arity of the constructor, or how many arguments it expects. A zero, from the Data.Nat standard library, should be treated as our Z constructor, and a suc translates to S. These constructors expect 0 and 1 argument, respectively.

We will not say much about the implementation of this library, since it is not groundbreaking. For more details, we again refer to van der Walt's thesis [11]. Using the library is simple; it exposes a function called doConvert which takes the conversion table, a (hidden) proof that the conversion is possible, and a Term to convert, and produces an inhabitant of the desired data type.

The use of doConvert is illustrated in Fig. 4. The hidden assumption that the conversion is possible causes a compile time failure if an incompatible term is given. To convince yourself of the utility of the Autoquote library, compare this relatively elegant code to the verbose term2boolexpr function in Fig. 1.

```
something : {x : ℕ} → doConvert exprTable (quoteTerm (x + 1))
                      ≡                               Plus (Var 0) (S Z)
something = refl
```

Fig. 4. An example of Autoquote in use. See Fig. 3 for the definition of exprTable, a typical Name-to-constructor mapping.

In most cases, the result from doConvert will require some post-processing, as we will see later in the Boolean tautologies example (Sec. 4.2), but for now it suffices to say that Autoquote removes a lot of the burden of converting Terms into other ASTs.

4 Proof by Reflection

The idea behind proof by reflection is simple: given that type theory is both a programming language and a proof system, it is possible to define functions that compute proofs. Reflection is an overloaded word in this context, since in programming language technology reflection is the capability of converting some piece of concrete code into an abstract syntax tree that can be manipulated in the same system. Reflection in the proof technical sense is the method of mechanically constructing a proof of a theorem by inspecting its shape. The proof by reflection technique we describe here is not new – see for example the chapter in Coq'Art [12] – but instead combines a number of existing methods into a usable package. Here we will see two case studies illustrating proof by reflection and how Agda's reflection mechanism can make the technique more convenient.

4.1 Simple Example: Evenness

To illustrate the concept of proof by reflection, we will follow Chlipala's example of even naturals [13]. We start by defining the property Even below. There are two constructors: the first constructor says that zero is even; the second constructor states that if n is even, then so is $2 + n$.

```
data Even   : ℕ → Set where
  isEven0   :                      Even 0
  isEven+2  : {n : ℕ} → Even n → Even (2 + n)
```

Using these rules to produce the proof that some large number n is even is tedious: the proof that $2 \times n$ is even requires n applications of the isEven+2 constructor. For example, here is the proof that 6 is even:

```
isEven6 : Even 6
isEven6 = isEven+2 (isEven+2 (isEven+2 isEven0))
```

To automate this, we will show how to *compute* the proof required. We start by defining a predicate even? that returns the unit type (top) when its input is even and the empty type (bottom) otherwise. In this context, ⊤ and ⊥ can be seen as the analogues of true and false, since there exists a proof that some number is even, if it is 0 or $2 + n$, for even n. Otherwise, no proof exists. The idea of "there exists" is perfectly modelled by the unit and empty types, since the unit type has one inhabitant, the empty type none.

```
even? : ℕ → Set
even? 0              = ⊤
even? 1              = ⊥
even? (suc (suc n))  = even? n
```

Next we need to show that the even? function is *sound*. To do so, we prove that when even? n returns ⊤, the type Even n is inhabited. Since we are working in a constructive logic, the only way to show this is to give some witness. This is done in the function soundnessEven. What we are actually doing here is giving a recipe for constructing proof trees, such as the one we manually defined for isEven6.

```
soundnessEven : {n : ℕ} → even? n → Even n
soundnessEven {0}           tt  = isEven0
soundnessEven {1}           ()
soundnessEven {suc (suc n)} s   = isEven+2 (soundnessEven s)
```

Note that in the case branch for 1, we do not need to provide a right-hand side of the function definition. The assumption, even? 1, is uninhabited, and we discharge this branch using Agda's absurd pattern, `()`.

If we need a proof that some arbitrary n is even, we only need to call soundnessEven. Note that the value of n is inferred. The only argument we must to provide to soundnessEven is a proof that even? n is inhabited. For any closed term, such as the numbers 28 or 8772, this proof obligation reduces to ⊤, which is proven by its single constructor, tt.

```
isEven8772 : Even 8772
isEven8772 = soundnessEven tt
```

Now we can easily get a proof that arbitrarily large numbers are even, without having to explicitly write down a large proof tree. Note that it is not possible to write something with type Even 27, or any other uneven number, since the parameter even? n is equal to ⊥, thus tt would not be accepted where it is in the Even 28 example. This will produce a type error stating that the types ⊤ and ⊥ cannot be unified.

Since the type ⊤ is a simple record type, Agda can infer the tt argument. This means we can turn the assumption even? n into an implicit argument, so a user could get away with writing just soundnessEven as the proof, letting Agda fill in the missing proof. To achieve this, we modify the soundnessEven function slightly, making all its arguments implicit. This trick works because Agda supports eta expansion for record types. More specifically, Agda will automatically fill in implicit arguments of the unit type. Here, the type system is doing work for us which is not done for general data types; for records eta expansion is safe, since recursion is not allowed. This trick is implemented in the code on GitHub, and will be used from here on to make our proofs easier to use.

Note that it is possible to generate a user-friendly "error" of sorts, by replacing the ⊥ with an empty type having a descriptive name:

```
data IsOdd : ℕ → Set where
```

This makes the soundness proof a little less straightforward, but in return the type error generated if an odd number is used becomes more informative. When a user tries to use the soundnessEven lemma to generate a proof of the statement Even 7, Agda will complain about a missing implicit argument of type IsOdd 7. An unsolved implicit argument is marked yellow in Agda, which looks less scary than a type error in a visible argument, but rest assured that no spurious proofs are being generated. This concludes the example of proving that certain naturals are even using proof by reflection. The next step will be to use the same approach for a more involved and realistic problem.

4.2 Second Example: Boolean Tautologies

We will now apply the same technique as above to a more interesting problem, making the relationship to the previous example clear at each step. The next example of proof by reflection will be a decision procedure that checks if a Boolean expression is a tautology. We will follow the same recipe as for even naturals, with one further addition. In the previous example, the input of our decision procedure even? and the problem domain were both natural numbers.

As we shall see, this need not always be the case: more complex structures and properties may be used. Take as an example the following Boolean formula.

$$(p_1 \vee q_1) \wedge (p_2 \vee q_2) \Rightarrow (q_1 \vee p_1) \wedge (q_2 \vee p_2) \tag{1}$$

It is trivial to see that this is a tautology, but directly proving this in Agda can be rather tedious. It is even worse if we want to check if the formula always holds by trying all possible variable assignments, since this requires 2^n cases, where n is the number of variables.

To automate this process, we will follow a similar approach to the one given in the previous section. We start by defining an inductive data type to represent Boolean expressions with at most n free variables in Fig. 5.

```
data BoolExpr (n : ℕ) : Set where
  Truth Falsehood :                                BoolExpr n
  And Or Imp      : BoolExpr n → BoolExpr n → BoolExpr n
  Not             : BoolExpr n              → BoolExpr n
  Atomic          : Fin n                   → BoolExpr n
```

Fig. 5. Inductive definition of Boolean expressions with n free variables.

There is nothing surprising about this definition; we use the type Fin n to ensure that variables (represented by Atomic) are always in scope. If we want to evaluate the expression, however, we will need some way to map variables to values. Enter Env n: a vector of n Boolean values. It has fixed size n since a BoolExpr n has at most n free variables.

Now we can define an evaluation function, which tells us if a given Boolean expression is true or not, under some assignment of variables. It does this by evaluating the formula's AST, filling in for Atomic values the concrete values which are looked up in the environment. For example, And is evaluated to the Boolean function _∧_, and its two arguments in turn are recursively interpreted.

```
⟦_⊢_⟧ : ∀ {n : ℕ} (e : Env n) → BoolExpr n → Bool
⟦ env ⊢ Truth        ⟧ = true
⟦ env ⊢ Falsehood    ⟧ = false
⟦ env ⊢ And be be₁ ⟧ =   ⟦ env ⊢ be ⟧ ∧  ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Or  be be₁ ⟧ =   ⟦ env ⊢ be ⟧ ∨  ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Not be       ⟧ = ¬ ⟦ env ⊢ be ⟧
⟦ env ⊢ Imp be be₁ ⟧ =   ⟦ env ⊢ be ⟧ ⇒ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Atomic n     ⟧ = lookup n env
```

Recall our decision function even? in the previous section. It returned ⊤ if the proposition was valid, ⊥ otherwise. Looking at ⟦_⊢_⟧, we see that we should just translate true to the unit type and false to the empty type, to get the analogue of the even? function. We therefore define a function P in Fig. 6, mapping Booleans to types. To give more useful error messages, we decorate the empty type with a string.

```
data Error (e : String) : Set where
P : Bool → Set
P true  = ⊤
P false = Error "Argument expression does not evaluate to true."
```

Fig. 6. Helper type Error, enabling clearer type errors.

Now that we have these helper functions, it is easy to define what it means to be a tautology. We quantify over a few Boolean variables, and wrap the formula in the function P. If the resulting type is inhabited, the argument to P is a tautology, i.e., for each assignment of the free variables the entire equation still evaluates to true. An example encoding of such a theorem is Fig. 7 – notice how similar it looks to the version expressed in mathematical notation, in equation 1.

```
exampletheorem : Set
exampletheorem = (p₁ q₁ p₂ q₂ : Bool) →
  P ((p₁ ∨ q₁) ∧ (p₂ ∨ q₂) ⇒ (q₁ ∨ p₁) ∧ (q₂ ∨ p₂))
```

Fig. 7. Encoding of an example tautology.

Here a complication arises, though. We are quantifying over a list of Boolean values *outside* of the function P, so proving P to be sound will not suffice. We just defined an evaluation function ⟦_⊢_⟧ to take an environment, an expression, and return a Boolean. In Fig. 7, though, we effectively quantified over all possible environments. We are going to need a way to lift the function P over arbitrary environments.

The function forallsAcc, in Fig. 8, performs this lifting. This function represents the real analogue of even? in this situation: it returns a type which is only inhabited if the argument Boolean expression is true under all variable assignments. This is done by generating a full binary tree of ⊤ or ⊥ types, depending on the result of ⟦_⊢_⟧ under each assignment. This corresponds precisely to the expression being a tautology if and only if the tree is inhabited. The function foralls simply bootstraps forallsAcc with an empty environment – it is omitted for brevity.

The Diff argument is unfortunately needed for bookkeeping, to prove that forallsAcc will eventually produce a tree with depth equal to the number of free variables in an expression, and can be ignored.

```
forallsAcc : {n m : ℕ} → BoolExpr m → Env n → Diff n m → Set
forallsAcc b acc (Base  ) = P ⟦ acc ⊢ b ⟧
forallsAcc b acc (Step y) =
  forallsAcc b (true :: acc) y × forallsAcc b (false :: acc) y
```

Fig. 8. The function forallsAcc, which decides if a proposition is a tautology. Compare to the even? function in Sec. 4.1.

Soundness. Since we now finally know our real decision function foralls, we can set about proving its soundness. Following the soundnessEven example, we want a function something like this.

```
soundness : {n : ℕ} → (b : BoolExpr n) → foralls b → ...
```

But what should the return type of the soundness lemma be? We would like to prove that the argument b is a tautology, and hence, the soundness function should return something of the form $(b_1 \dots b_n : \text{Bool}) \rightarrow P\ B$, where B is an expression in the image of the interpretation ⟦_⊢_⟧. For instance, the statement exampletheorem is a proposition of this form.

The function proofGoal takes a BoolExpr n as its argument and generates the statement that it is a tautology. That is, it gives back the type equal to the theorem under scrutiny. It does this by first introducing n universally quantified Boolean variables. These variables are accumulated in an environment. Finally, when n binders have been introduced, the BoolExpr is evaluated under this environment.

```
proofGoal  : (n m : ℕ) → Diff n m → BoolExpr m → Env n → Set
proofGoal .m m (Base  ) b acc = P ⟦ acc ⊢ b ⟧
proofGoal  n  m (Step y) b acc =
           (a : Bool) → proofGoal (1 + n) m y b (a :: acc)
```

Now that we can interpret a BoolExpr n as a theorem using proofGoal, and we have a way to decide if something is true for a given environment, we still need to show the soundness of our decision function foralls. That is, we need to be able to show that a formula is true if it holds for every possible assignment of its variables to true or false.

This is done in the function soundness, of which we only provide the type signature. It takes the predicate foralls which is only satisfied when a proposition is a tautology, and gives back a proof which has the type computed by proofGoal. It uses the predicate to safely extract the leaf from foralls corresponding to any given environment resulting from the binders introduced by proofGoal.

```
soundness : {n : ℕ} → (b : BoolExpr n) → {p : foralls b}
                     → proofGoal 0 n (zero-least 0 n) b []
```

Now, we can prove theorems by a call of the form soundness b {p}, where b is the representation of the formula under consideration, and p is the evidence that all branches of the proof tree are true. Agda is convinced that the representation does in fact correspond to the concrete formula, and also that soundness gives a valid proof. We do not even give p explicitly since the only valid values are nested pairs of tt, which can be inferred automatically for closed terms if the type is inhabited. This once again exploits the fact that Agda supports eta expansion for record types.

If the module passes the type checker, we know our formula is both a tautology and that we have the corresponding proof object at our disposal afterwards, as in someTauto (Fig. 9).

```
rep        : BoolExpr 2
rep        = Imp (And (Atomic (suc zero)) (Atomic zero))
                 (Atomic zero)
someTauto  : (p q : Bool) → P (p ∧ q ⇒ q)
someTauto  = soundness rep
```

Fig. 9. An example Boolean formula, along with the transliteration to a proposition and the corresponding proof.

If one were to give as input a formula which is not a tautology, Agda would not be able to infer the proof foralls, since it would be an uninhabited type. This would result in an unsolved meta-variable, which might seem a triviality if you do not read the compiler's output carefully. Luckily Agda disallows importing modules with unsolved meta-variables, which means such a spurious proof would not be usable elsewhere in a real-life development.

The only part we still have to do manually is to convert the concrete Agda representation of the formula (p ∧ q ⇒ q, in this case) into our abstract syntax (rep, in this example). This is unfortunate, as we end up typing out the formula twice. We also have to count the number of variables ourselves and convert them to De Bruijn indices. This is error-prone given how cluttered the abstract representation can get for formulae containing many variables.

It would be desirable for this representation process to be automated. Luckily we have the Autoquote library for precisely this purpose.

4.3 Adding Reflection

It might come as a surprise that in a paper focusing on reflection – in the programming language technology sense – we have not yet presented a convincing use for reflection. This is about to change. We can get rid of the duplication seen in Fig. 9 using Agda's reflection API. There we see the same Boolean formula twice: once in the type signature as an Agda proposition and once expressed as a BoolExpr. Using the quoteGoal keyword to inspect the current goal, and passing that goal to Autoquote, we can convert it to its corresponding BoolExpr automatically.

The conversion between a Term and BoolExpr is achieved in two phases. Since Autoquote only supports simple inductive data types, the first issue we encounter is that BoolExpr n has an argument of type Fin n to its constructor Atomic (see Fig. 5). To work around this, we introduce a simpler, intermediary data structure, to which we will convert from Term. This type, called BoolInter, is not shown here, but the only difference with BoolExpr n is that its variables are represented by naturals instead of Fins.

The Autoquote library needs a lookup table, mentioning which constructor represents variables and what the arity of the constructors is. This way only Terms containing variables or the usual operators are accepted. Using the mapping presented in Fig. 10, we can construct a function that, for suitable Terms, gives us a value in BoolInter.

```
boolTable : Table BoolInter
boolTable = (Atomic,
      (quote _∧_) # 2 ↦ And         :: (quote _∨_ ) # 2 ↦ Or
   :: (quote ¬_   ) # 1 ↦ Not         :: (quote true  ) # 0 ↦ Truth
   :: (quote false ) # 0 ↦ Falsehood :: (quote _⇒_) # 2 ↦ Imp :: [])
```

Fig. 10. The mapping table for quoting to BoolInter.

Once we have a BoolInter expression, we just need to check that its variables are all in scope (this means that ∀ Atomic $x : x < n$, if we want to convert to a BoolExpr n), and replace all the ℕ values with their Fin n counterparts. We can now write a function called proveTautology, which uses the automatic quoter and calls soundness on the resulting term.

That is all we need to automatically prove that formulae are tautologies. The following snippet illustrates the use of the proveTautology function; we can omit all arguments except e, since they can be inferred.

```
peirce : (p q : Bool) → P (((p ⇒ q) ⇒ p) ⇒ p)
peirce = quoteGoal e in proveTautology e
fave   : exampletheorem   -- defined in Fig. 7
fave   = quoteGoal e in proveTautology e
```

With that, we have automatically converted propositions in the Agda world to our own AST, generated a proof of their soundness, and converted that back into a real proof that Agda trusts.

5 Discussion

Related Work. Our main innovations are novel combinations of existing techniques; as a result, quite a number of subjects are relevant to mention here.

As far as reflection in general goes, Demers and Malenfant [14] wrote an informative historical overview on the topic. What we are referring to as reflection dates back to work by Brian Smith [1] and was initially presented in the Lisp family of languages in the 80s. Since then, many developments in the functional, logic as well as object-oriented programming worlds have been inspired – systems with varying power and scope.

Reflection is becoming more common, to various extents, in industry-standard languages such as Java, Objective-C, as well as theoretically more interesting languages, such as Haskell [15]. Smalltalk, an early object-oriented programming language with advanced reflective features [16], is the predecessor of Objective-C. The disadvantage of Smalltalk, however, is that reflection is extremely unsafe: one can call non-existent functions, to name just one pitfall. Nevertheless, it is surprising that industry programming does not use more of these advanced reflective features which have already been around for a long time.

These systems would seem to be the inspiration for the current reflection mechanism recently introduced in Agda, although it is lacking in a number of fundamental capabilities, most notably type awareness of unquote, and type erasure when using quoteTerm.

Evaluation. If we look at the taxonomy of reflective systems in programming language technology written up by Sheard [17], we see that we can make a few rough judgements about the metaprogramming facilities Agda currently supports. [2]

– Agda's current reflection API leans more towards analysis than generation,

[2] Of course, having been implemented during a single Agda Implementors' Meeting [18], the current implementation is more a proof-of-concept, and is still far from being considered finished, so it would be unfair to judge the current implementation all too harshly. In fact, we hope that this work might motivate the Agda developers to include some more features, to make the system truly useful.

– it supports encoding of terms in an algebraic data type (as opposed to a string, for example),
– it involves manual staging annotations (by using keywords such as quote and unquote),
– it is homogeneous, because the object language is the metalanguage. The object language's representation is a native data type.
– It is only two-stage: we cannot as yet produce an object program which is itself a metaprogram. This is because we rely on built in keywords such as quote, which cannot themselves be represented.

As far as the proof techniques used in the section on proof by reflection (Sec. 4) are concerned, Chlipala's work [13] proved an invaluable resource. One motivating example for doing this in Agda was Wojciech Jedynak's ring solver [19], which is the first example of Agda's reflection API in use that came to our attention. Compared to Jedynak's work, the proof generator presented here is more refined in terms of the interface presented to the user. The expectation is that approaches like these will become more commonplace for proving mundane lemmas in large proofs. The comparison to tactics in a language like Coq is a tempting one, and we see both advantages and disadvantages of each style. Of course, the tactic language in Coq is much more specialised and sophisticated when it comes to generating proofs, but it is a pity that there are two separate languages. This paper explores an alternative, where metaprograms are written directly in the object language. Also, the fact that proof generation in Agda is explicit may be something some people appreciate.

There are performance issues with the current reflection API in Agda. This should not come as a surprise. The extensive usage of proof by reflection in Coq and SSReflect [20] has motivated a lot of recent work on improving Coq's compile time evaluation. We hope that Agda can be extended with similar technology.

Conclusions. Returning to our research question, repeated here to jog the memory, a summary of findings is made.

> "What practical issues do we run into when trying to engineer automatic proofs in a dependently typed language with reflection? Are Agda's reflective capabilities sufficient and practically usable, and if not, which improvements might make life easier?"

This paper has presented two simple applications of proof by reflection, the latter using Agda's reflection API. We have managed to automate generation of a certain class of mundane proofs. This shows that the reflection capabilities recently added to Agda are quite useful for automating tedious tasks. For example, we now need not encode expressions manually: using quoteTerm and Autoquote, the AST conversion can be done automatically. Furthermore, by using the proof by reflection technique, we have shown how to automatically generate proofs, without loss of general applicability. Constraining ourselves to (pairs of) unit types as predicates, we can let Agda infer them, and by tagging an empty type with a string, we can achieve more helpful errors if these predicates are invalid. These simple tools were sufficient to engineer relatively powerful and – more importantly – easily usable proof tools.

It seems conceivable to imagine that in the future, using techniques such as those presented here, a framework for tactics might be within reach. Eventually we might be able to define an embedded language in Agda, in the style of Coq's tactic language, to inspect the shape of the proof that is needed using reflection, and look at a database of predefined proof recipes to see if one of them might discharge the obligation. An advantage of this approach versus the tactic language in Coq, would be that the language of the propositions and tactics is the same.

Acknowledgements. test

References

1. Smith, B.C.: Reflection and semantics in LISP. In: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. POPL '84, New York, NY, USA, ACM (1984) 23–35
2. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. PEPM '97 (1997)
3. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. (2002) 1–16
4. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
5. Norell, U.: Dependently typed programming in Agda. In: Proceedings of the 4th international workshop on Types in language design and implementation. TLDI '09, New York, NY, USA, ACM (2009) 1–2
6. Pitman, K.: Special forms in Lisp. In: ACM Symposium on Lisp and Functional Programming. (1980)
7. Alexandrescu, A.: Modern C++ design. Addison Wesley (2001)
8. Martin-Löf, P.: Constructive mathematics and computer programming. In: Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages, Upper Saddle River, NJ, USA, Prentice-Hall, Inc. (1985) 167–184
9. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3) (1988) 95–120
10. Agda developers: Agda 2.2.8 release notes. The Agda Wiki: `http://wiki.portal.chalmers.se/agda/agda.php?n=Main.Version-2-2-8` (2012) [Online; accessed 6-April-2012].
11. van der Walt, P.: Reflection in Agda. Master's thesis, Department of Computer Science, Utrecht University, the Netherlands (2012)
12. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)

13. Chlipala, A.: Certified programming with dependent types. MIT Press (2011)
14. Demers, F., Malenfant, J.: Reflection in logic, functional and object-oriented programming: a short comparative study. In: Proceedings of the IJCAI. Volume 95. (1995) 29–38
15. Stump, A.: Directly reflective meta-programming. Higher-Order and Symbolic Computation **22**(2) (2009) 115–144
16. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
17. Sheard, T.: Staged programming. online, `http://web.cecs.pdx.edu/~sheard/staged.html` [accessed 20-Aug-2012].
18. Altenkirch, T.: [Agda mailing list] More powerful quoting and reflection? mailing list communication, `https://lists.chalmers.se/pipermail/agda/2012/004127.html` (2012) [online; accessed 14-Sep-2012].
19. Jedynak, W.: Agda ring solver using reflection. online, GitHub, `https://github.com/wjzz/Agda-reflection-for-semiring-solver` (2012) [Online; accessed 26-June-2012].
20. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. Journal of Formalized Reasoning **3**(2) (2010) 95–152 RR-7392 RR-7392.