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 and Template Haskell. It gives a brief introduction to using reflection, and details the complexities encountered when automating certain proofs with *proof by reflection*. It presents a library that can be used for automatically quoting a class of concrete Agda terms to a non-dependent, user-defined inductive data type, alleviating some of the burden a programmer 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 [1,2] has recently been extended with a *reflection mechanism* [3] for compile time metaprogramming in the style of Lisp [4], MetaML [5], Template Haskell [6], and C++ templates. 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, illustrative of the kind of problems that can be solved using reflection. More specifically:

– We give a very brief introduction to Agda's reflection mechanism (Sec. 2). Previously, these features were only documented in the release notes and comments in Agda's source files. A detailed tutorial is available elsewhere [3].
– We present Autoquote, an Agda library that does a declaratively-specified translation of a quoted expression to a representation in a user-defined non-dependent datatype (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 technology.

The code presented in this paper compiles using Agda version 2.3.2.[1]

1.1 Introducing Agda

Agda is an implementation of Martin-Löf's type theory [7], extended with records and modules. It is developed at the Chalmers University of Technology [1]; 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 [8]. There are many excellent tutorials on Agda [1,2,9].

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. 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.

2 Using Reflection

We will now introduce the reflection API with some small examples.

The Keywords. There are several keywords that can be used to quote and unquote terms: **quote**, **quoteTerm**, **quoteGoal**, and **unquote**. The **quote** keyword allows the user to access the internal representation of any identifier. This internal representation, a Name value, 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; most important is Term : Set, representing concrete Agda terms.

The simplest example of quotation uses the keyword **quoteTerm** x : Term, where x is a fragment of concrete syntax. Note that **quoteTerm** 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
```

In dissecting this, we find the lam constructor, since we introduced a lambda abstraction. Its one argument is visible (as opposed to implicit), 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.

[1] All supporting code, including this paper in Literate Agda format, is available on GitHub. `https://github.com/toothbrush/reflection-proofs`

Furthermore, **quoteTerm** type checks its argument before returning the Term. Since type checking a term necessitates normalisation, the returned Term is always in normal form, as example₁ demonstrates.

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

The identity function is applied to zero, resulting in just 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 different. We cannot assign **quoteGoal** an informal type, since it is really a syntactic construct that depends on the context. See the following example.

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

The **quoteGoal** keyword binds the variable e to the Term representing the type expected at the position of **quoteGoal**. In this example, the value of e in the hole will be def ℕ [], i.e., the Term representing the type ℕ, which is a definition, hence def.

The **unquote** keyword takes one argument – a Term – and converts it back to a concrete expression. Just as **quoteTerm** and **quoteGoal**, **unquote** type checks and normalises the Term before splicing it into the program text. Note that it is not yet possible to introduce top-level declarations using **unquote**. This is a technical limitation.

The **quote** x : Name keyword returns the representation of an identifier x as a value in the primitive type Name, if x is the name of a definition (function, datatype, record, or a constructor). Unfortunately, we cannot simply pattern match on constructor names. The reason pattern matching on Names is not supported, is that the elimination principle is not clear, since Name is a built-in, non-inductive type. The only mechanism we have to distinguish Names is decidable equality[2], which results in code as presented below – a lot less concise than the pattern matching equivalent would be. Agda does allow matching on Strings (which similarly only expose decidable equality), so the limitation is a technical one, which might be solved in the future.

```
whatever : Term → ...
whatever (con c args) with c ≟-Name quote foo
... | yes p  = { }0   -- foo applied to arguments
... | no ¬p  = { }1   -- not foo, try another Name, etc.
```

2 using the function _≟-Name_

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, the reader is referred to Van der Walt's thesis ([3], Chapter 3).

3 Automatic Quoting

In the previous section, we saw how to recover values of type Term, representing concrete Agda terms. This is a start, but we rarely want to directly manipulate Terms: often it is much more useful to use our own AST for computations. It should be a minor task to write a function to convert a Term into another AST, but this often turns out to become a mess.

When pattern matching is possible, converting elements of one AST to another is a simple task. Unfortunately, Agda functions are required to be total, which means they must have a case for each possible pattern. Since Term covers all quotable terms, it has many alternatives. Furthermore, for Names, we only have decidable equality. This is why such conversion functions tend to become verbose, as in the code snippet of Fig. 1, an excerpt of a 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 a naïve conversion function, from Term into some more specific data type.

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

The Autoquote Library. We will use Expr, presented in Fig. 2, as a running example of a toy AST. It is a simple non-dependent inductive data structure representing terms with Peano-style natural numbers, variables represented using De Bruijn indices, and additions.

We might want to convert an expression, such as $5+x$, to this AST using reflection. In an ideal world, we would just pattern match on concrete constructs such as the _+_ function and return elements like Plus of our AST. The Autoquote library allows just this, exposing an interface which, when provided with such a mapping, automatically quotes expressions that fit. Here, *fitting* is defined as only containing names that are listed in the mapping, or variables with De

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

Fig. 2. The toy expression language Expr. Quoting such terms is now easier.

Bruijn indices, and respecting constructor arities. Trying to convert other terms results in a type error. The user provides a straightforward mapping, such as in Fig. 3, and Autoquote converts Agda terms to values in the AST. Currently only non-dependent inductive types are supported.

```
exprTable : Table Expr
exprTable = (Var, (quote _+_) ↦ Plus ::
                  (quote zero ) ↦ Z    ::
                  (quote suc  ) ↦ S    :: [])
```

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

This table should be interpreted as follows: any variables encountered should be stored in Vars, and the _+_ operator should be mapped to a Plus constructor. A zero, from the Data.Nat standard library, should be treated as our Z constructor, etc. Note that the first item in the table (Var in this case) is special, and should be a constructor for De Bruijn-indexed variables. The rest of the table is an arbitrary list of constructors.

We will not say much about the implementation of this library, since it is not groundbreaking. For more details, we again refer to ([3], Sec. 3.3). Using the library is simple; it exposes a function called doConvert which takes the conversion table, a (hidden, automatically inferred) proof that the conversion is possible, and a Term to convert, and produces an inhabitant of the desired data type, where possible. The implicit proof technique is outlined in Sec. 4.1.

The use of doConvert is illustrated in Fig. 4. The hidden assumption that the conversion is possible causes a type error if an incompatible term is given. The utility of the Autoquote library is clear if you compare this relatively straightforward code to the verbose term2boolexpr snippet in Fig. 1.

Usually, the result from doConvert will require some post-processing – for example, turning all naturals into Fin n values, or scope checking a resulting expression – as we will see in the Boolean tautologies example (Sec. 4.2). However, for now it suffices to say that Autoquote eases converting Terms into other ASTs.

A mechanism like Autoquote is actually an ad-hoc workaround for a more general difficulty in Agda, namely that currently, a watered-down version of pattern matching on data types exposing decidable equality is unreasonably

```
example₃ : {x : ℕ} → doConvert exprTable (quoteTerm (x + 1))
                   ≡                          Plus (Var 0) (S Z)
example₃ = refl
```

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

awkward. If this were possible in general, like it is for String, the Autoquote library would be redundant.

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 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 Chapter 16 of Coq'Art [11] – but instead combines a number of existing methods into a usable package. The following two case studies illustrate proof by reflection and how Agda's reflection mechanism can make the technique more accessible. The first example is a closed example and sets the stage for the second, an open expression type extended to include variables.

4.1 Closed Example: Evenness

To illustrate the concept of proof by reflection, we will follow Chlipala's example of even naturals [12]. Our objective is to be able to automatically prove evenness of certain naturals. To this end, we first write a test function which decides if a natural is even, then prove the soundness of this predicate. This results in a proof generator.

We start by defining the property Even.

```
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: it requires $n/2$ applications of the isEven+2 constructor.

To automate this, we will show how to *compute* the proof required. We define a predicate even? that returns the unit type (top) when its input is even and the empty type (bottom) otherwise. In this context, $\top$ and $\bot$ 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.

```
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 if and only if 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 a witness. This is done in the function soundnessEven. Note that we are actually giving a recipe for constructing proof trees.

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

In the case of $n = 1$, 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, soundnessEven builds it. Note that the value of n is inferred. The only argument we must to provide to soundnessEven is 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 term for arbitrary even numbers, without having to explicitly write down the proof tree. Note that it is not possible to give a term with type Even 27, or any other uneven number, since the parameter even? n is equal to ⊥, which is uninhabited. Providing tt anyway will produce a type error stating that the types ⊤ and ⊥ cannot be unified.

Implicit Proofs. 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 just write soundnessEven as the proof, letting Agda fill in the missing proof. This trick works because Agda supports eta expansion for record types. Concretely, Agda will automatically fill in implicit arguments of the unit type. Here, the type system is doing more work than for general data types; for records eta expansion is safe, since recursion is not allowed. This trick will be used from here on to ameliorate our proof generators' interfaces.

Friendlier Errors. It is possible to generate a descriptive "error" of sorts, by replacing the ⊥ with an empty type that has a friendly 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 dire than a type error in a visible argument, but no spurious proofs are being generated.

Limitations. This is a very simple, closed example. In particular, it would not work in the presence of quantifications, for example to define a lemma like Even x → Even (x + 100). Why this is the case, and how it could be solved, is discussed at the end of Sec. 4.2.

The next step will be to use the same approach for a problem involving variables.

4.2 Open Example: Boolean Tautologies

We will now apply the same steps as above to a different problem, clarifying the relationship to the previous example at each step. This example of proof by reflection will be lifting a predicate that checks if a Boolean expression with indexed variables is a tautology under all possible assignments, to a proof generator.

Take as an example the following proposition.

$$(p_1 \vee q_1) \wedge (p_2 \vee q_2) \Rightarrow (q_1 \vee p_1) \wedge (q_2 \vee p_2) \qquad (1)$$

If we squint, we see that (1) is a tautology, but explicitly proving this in Agda would be rather tedious. Assuming we want to check if the formula always holds by trying all possible variable assignments, this would require 2^n pattern matching cases, where n is the number of variables.

To automate this process, we start by defining an inductive data type to represent Boolean expressions with at most n free variables (see 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. Modelling Boolean expressions with n free variables.

We use the type Fin n to ensure that variables (represented by Atomic and identified by their De Bruijn index) are in scope. If we want to evaluate the expression, we will need some way to map variables to values. For this we use Env n: a vector of n Boolean values.

Now we can define an interpretation function, which tells us if an expression is true or not, given 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 ⊢ And be be₁ ⟧  =  ⟦ env ⊢ be ⟧ ∧ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Atomic n      ⟧  =  lookup n env
...
```

Recall our test 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, mapping Booleans to types (see Fig. 6). As before we decorate the empty type, this time with a string, to give more informative error messages.

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

Fig. 6. Empty type Error, facilitating clearer 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 (1).

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

Fig. 7. The term exampletheorem : Set encodes (1).

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 the function ⟦_⊢_⟧ to take one environment and one expression. 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 cumulatively generating a full binary tree – the truth table – 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 makes forallsAcc produce a tree with depth equal to the number of free variables in an expression, putting a bound on the recursion.

```
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. Now we finally know our real decision function foralls, we can set about proving its soundness. Following the soundnessEven example, we want a function with a type something like in Fig. 9.

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

Fig. 9. The informal type of soundness, taking an expression and its truth table.

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 $(b_1 \ldots b_n : \text{Bool}) \rightarrow P\ B$, 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 proposition that the expression is a tautology, by giving back the type equal to the theorem under scrutiny. It first introduces n universally quantified Boolean variables. These variables are accumulated in an environment. Finally, when n binders have been introduced, the BoolExpr n 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 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 requires 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. We do not give p explicitly since the only valid values are nested pairs of tt, which can be inferred automatically. This once again exploits the fact that Agda supports eta expansion for record types.

If the module type checks, we know that the representation of the formula corresponds to the concrete expression, soundness gave a valid proof, and that the formula is in fact a tautology. We also have the corresponding proof object at our disposal, as in someTauto (Fig. 10).

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. As before, this would result in an unsolved meta-variable (a type error stating Error and ⊤ cannot be unified). Agda disallows importing modules with unsolved meta-variables, which means such an unfulfilled proof obligation would not be usable elsewhere in a real-life development.

Limitations. Unfortunately, this approach is only possible using variables with a finite type. If we wanted to prove properties about naturals, for example, we would not be able to enumerate all possible values. Also, not all problems are decidable. In the ring solver example [13] a canonical representation is used, but this does not always exist. One way forward would be if a proof search system

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

could be implemented, going beyond simple reflection. By inspecting the shape of the obligation it might be possible to find a lemma which sufficiently reduces the goal to something we can easily generate. This is motivated by the evenness example: we could imagine it being possible to automatically prove lemmas like Even n → Even (n + 100), given a list of usable lemmas. On inspecting the goal and finding the Plus (Var n) 100 term, we might be able to learn that this lemma (which would have a particularly tedious proof) is an instance of Even x → Even y → Even (x + y), which might be an existing library proof. However, this would require a rather advanced way of recognising structures in proof goals, and a reliable proof search for useful lemmas in a database. This would correspond to implementing an analogue of Coq's `auto` tactic in Agda. The Agda synthesizer Agsy already implements such a proof search, but is built directly into the compiler. This is definitely an avenue for future work.

Summary. The only thing we still have to do manually is convert the Agda representation of the formula (p ∧ q ⇒ q, for example) into our abstract syntax (rep). 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.

We would like this transliteration process to be automated. Luckily Autoquote is available for precisely this purpose, and we show this now.

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. We can get rid of the duplication seen in Fig. 10 using Agda's reflection API. Using the **quoteGoal** keyword to inspect the current goal would give us the Agda representation, and passing that to Autoquote, we can convert it to its corresponding BoolExpr.

The conversion between a Term and BoolExpr is achieved in two phases, necessary because Autoquote only supports non-dependent data types, and 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 Nats instead of Fins.

The Autoquote library uses a lookup table, mentioning which constructor represents variables and how names map to constructors. This way only Terms containing variables or the usual operators are accepted. Using the mapping presented in Fig. 11, we can construct a function that, for suitable Terms, gives us a value in BoolInter.

Once we have a BoolInter expression, the second phase is 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 ℕ values with their Fin n counterparts. We can

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

Fig. 11. The mapping table for quoting to BoolInter.

now write a function proveTautology, which uses the automatic quoter and calls soundness on the resulting term. An approximation of proveTautology's type is given here. In summary, it takes a term (as bound in the body of **quoteGoal**), quotes it with Autoquote, passes it to soundness, which returns a term fulfilling the proofGoal type.

```
proveTautology : (t : Term) → let  t' = doConvert boolTable t
                              in … {i : foralls t'} → proofGoal n t'
```

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
```

With that, we have automatically converted propositions in Agda to our own AST, generated a proof of their soundness, and converted that back into a proof term for the concrete formula.

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] provide an informative historical overview. What we are referring to as reflection dates back to work by Smith [15] and was initially presented in Lisp in the 80s. Since then, many developments in the functional, logic as well as object-oriented programming worlds have emerged – systems with varying power and scope [16], [17]. Unfortunately, reflection is often unsafe: in Smalltalk and Objective-C, for example, calling non-existent functions causes exceptions, to name just one pitfall.

These systems have inspired the reflection mechanism introduced in Agda, which is lacking in a number of fundamental capabilities – most notably type awareness of **unquote**, type preservation when using **quoteTerm** and inability to introduce top-level definitions. Nevertheless, it does provide the safety of a strong type system.

Evaluation. If we look at the taxonomy of reflective systems in programming language technology written up by Sheard [18], we see that we can make a few rough judgements about the metaprogramming facilities Agda currently supports.[3]

- Agda's current reflection API leans more towards analysis than generation,
- 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 keywords such as **quote**, which cannot be represented.

As far as the proof techniques used in Sec. 4 are concerned, Chlipala's work [12] proved an invaluable resource. One motivating example for doing this in Agda was Jedynak's ring solver [13], 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 marginally more refined in terms of the interface presented to the user. We expect that approaches of this kind will become 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, but it is a pity that it is separate. This paper explores an alternative, with metaprograms written directly in the object language. Some people might also appreciate the fact that proof generation in Agda is explicit.

Performance is another possible area of improvement. Introducing reflective proofs requires a lot of compile time computation, and for this approach to scale, Agda would need a more efficient static evaluator than the current call-by-name implementation. The extensive use of proof by reflection in Coq and SSReflect [20], for example for proving the four colour theorem [21], has motivated a lot of recent work on improving Coq's compile time evaluation. We hope that Agda will be similarly improved.

Conclusions. Returning to our research question, repeated here, 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

[3] Of course, having been implemented during a single Agda Implementors' Meeting [19], 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.

reflective capabilities sufficient and practically usable, and if not, which improvements might make life easier?"

This paper 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, some AST conversion can be done automatically. Furthermore, by using the proof by reflection technique, we have shown how to automatically generate a simple class of 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. Unfortunately, these proofs are limited to finite domains, and are still not very scalable or straightforward to implement. In particular, quantifying over variables with infinite domains should not be a great conceptual difficulty, but would necessitate a lot of extra machinery: a smarter goal inspector, and a generalised lemma searching or matching algorithm. Simple pattern matching on Names would also be a useful feature.

It seems conceivable 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, then inspect the shape of the proof obligation, and look at a database of predefined proof recipes to see if one of them might discharge or simplify 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. We would like to thank each of the four anonymous reviewers for taking the time to provide detailed and constructive comments that greatly improved the article.

References

1. 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)
2. 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
3. van der Walt, P.: Reflection in Agda. Master's thesis, Department of Computer Science, Utrecht University, Utrecht, The Netherlands (2012) available online, `http://igitur-archive.library.uu.nl/student-theses/2012-1030-200720/UUindex.html`.
4. Pitman, K.M.: Special forms in Lisp. In: Proceedings of the 1980 ACM conference on LISP and functional programming, ACM (1980) 179–187
5. 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)

6. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. (2002) 1–16
7. 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
8. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3) (1988) 95–120
9. Oury, N., Swierstra, W.: The power of pi. In: Proceedings of the 13th ACM SIGPLAN international conference on Functional programming. ICFP '08, New York, NY, USA, ACM (2008) 39–50
10. Agda developers: Agda release notes, regarding reflection. The Agda Wiki: `http://wiki.portal.chalmers.se/agda/agda.php?n=Main.Version-2-2-8` and `http://wiki.portal.chalmers.se/agda/agda.php?n=Main.Version-2-3-0` (2013) [Online; accessed 9-Feb-2013].
11. 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)
12. Chlipala, A.: Certified programming with dependent types. MIT Press (2011)
13. Jedynak, W.: Agda ring solver using reflection. online, GitHub, `https://github.com/wjzz/Agda-reflection-for-semiring-solver` (2012) [Online; accessed 26-June-2012].
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. 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
16. Stump, A.: Directly reflective meta-programming. Higher-Order and Symbolic Computation **22**(2) (2009) 115–144
17. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
18. Sheard, T.: Staged programming. online, `http://web.cecs.pdx.edu/~sheard/staged.html` [accessed 20-Aug-2012].
19. 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].
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.
21. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In Kapur, D., ed.: ASCM. Volume 5081 of Lecture Notes in Computer Science., Springer (2007) 333