Under preparation for CPP 2015

Auto in Agda

Programming proof search

Pepijn Kokke   Wouter Swierstra
Universiteit Utrecht
pepijn.kokke@gmail.com   w.s.swierstra@uu.nl

Abstract

As proofs in type theory become increasingly complex, there is a growing need to provide better proof automation. This paper shows how to implement a Prolog-style resolution procedure in the dependently typed programming language Agda. Connecting this resolution procedure to Agda's reflection mechanism provides a first-class proof search tactic for first-order Agda terms. Furthermore, the same mechanism may be used in tandem with Agda's instance arguments to implement type classes in the style of Haskell. As a result, writing proof automation tactics need not be different from writing any other program.

1. Introduction

Writing proof terms in type theory is hard and often tedious. Interactive proof assistants based on type theory, such as Agda [17] or Coq [9], take very different approaches to facilitating this process.

The Coq proof assistant has two distinct language fragments. Besides the programming language Gallina, there is a separate tactic language for writing and programming proof scripts. Together with several highly customizable tactics, the tactic language Ltac can provide powerful proof automation [7]. Having to introduce a separate tactic language, however, seems at odds with the spirit of type theory, where a single language is used for both proof and computation. Having a separate language for programming proofs has its drawbacks. Programmers need to learn another language to automate proofs. Debugging Ltac programs can be difficult and the resulting proof automation may be inefficient [5].

Agda does not have Coq's segregation of proof and programming language. Instead, programmers are encouraged to automate proofs by writing their own solvers [18]. In combination with Agda's reflection mechanism [1, 25], developers can write powerful automatic decision procedures [2]. Unfortunately, not all proofs are easily automated in this fashion. In that case, the user is forced to interact with the integrated development environment and manually construct a proof term step by step.

This paper tries to combine the best of both worlds by implementing a library for proof search *within* Agda itself. More specifically, this paper makes the several novel contributions.

[Copyright notice will appear here once 'preprint' option is removed.]

- We show how to implement a Prolog interpreter in the style of Stutterheim et al. [23] in Agda (Section 3). Note that, in contrast to Agda, resolving a Prolog query need not terminate. Using coinduction, however, we can write an interpreter for Prolog that is *total*.
- Resolving a Prolog query results in a substitution that, when applied to the goal, produces a solution in the form of a term that can be derived from the given rules. We extend our interpreter to also produce a trace of the applied rules, which allow us to produce a proof term that is a witness to the validity of the resulting substitution (Section **??**).
- We integrate this proof search algorithm with Agda's *reflection* mechanism (Section 4). This enables us to *quote* the type of a lemma we would like to prove, pass this term as the goal of our proof search algorithm, and finally, *unquote* the resulting proof term, thereby proving the desired lemma.
- Finally, we show how we can use our proof search together with Agda's *instance arguments* [10] to implement lightweight type classes in Agda (Section 5). This resolves one of the major restrictions of instance arguments: the lack of a recursive search procedure for their construction.

Although Agda already has built-in proof search functionality [13], we believe that exploring the first-class proof automation defined in this paper is still worthwhile. For the moment, however, we would like to defer discussing the various forms of proof automation until after we have presented our work (Section 6).

All the code described in this paper is freely available from GitHub.[1] It is important to emphasize that all our code is written in the safe fragment of Agda: it does not depend on any postulates or foreign functions; all definitions pass Agda's termination checker; and all metavariables are resolved.

2. Motivation

Before describing the *implementation* of our library, we will provide a brief introduction to Agda's reflection mechanism and illustrate how the proof automation described in this paper may be used.

Reflection in Agda

Agda has a *reflection* mechanism[2] for compile time metaprogramming in the style of Lisp [20], MetaML [24], and Template Haskell [21]. This reflection mechanisms make it possible to convert a program fragment into its corresponding abstract syntax tree

[1] See `https://github.com/pepijnkokke/AutoInAgda`.

[2] Note that Agda's reflection mechanism should not be confused with 'proof by reflection' – the technique of writing a verified decision procedure for some class of problems.

and vice versa. We will introduce Agda's reflection mechanism here with several short examples, based on the explanation in previous work [25]. A more complete overview can be found in the Agda release notes [1] and Van der Walt's thesis [26].

The type Term : Set is the central type provided by the reflection mechanism. It defines an abstract syntax tree for Agda terms. There are several language constructs for quoting and unquoting program fragments. The simplest example of the reflection mechanism is the quotation of a single term. In the definition of idTerm below, we quote the identity function on Boolean values.

```
idTerm : Term
idTerm = quoteTerm (λ (x : Bool) → x)
```

When evaluated, the idTerm yields the following value:

```
lam visible (var 0 [])
```

On the outermost level, the lam constructor produces a lambda abstraction. It has a single argument that is passed explicitly (as opposed to Agda's implicit arguments). The body of the lambda consists of the variable identified by the De Bruijn index 0, applied to an empty list of arguments.

More generally, the **quote** language construct allows users to access the internal representation of an identifier, a value of a built-in type Name. Users can subsequently request the type or definition of such names.

Dual to quotation, the **unquote** mechanism allows users to splice in a Term, replacing it with a its concrete syntax. For example, we could give a convoluted definition of the K combinator as follows:

```
const : ∀ {A B} → A → B → A
const = unquote (lam visible (lam visible (var 1 [])))
```

The language construct **unquote** is followed by a value of type Term. In this example, we manually construct a Term representing the K combinator and splice it in the definition of const.

The final piece of the reflection mechanism that we will use is the **quoteGoal** construct. The usage of **quoteGoal** is best illustrated with an example:

```
goalInHole : ℕ
goalInHole = quoteGoal g in { }0
```

In this example, the construct **quoteGoal** g binds the Term representing the *type* of the current goal, ℕ, to the variable g. When completing this definition by filling in the hole labeled 0, we may now refer to the variable g. This variable is bound to to def ℕ [], the Term representing the type ℕ.

Using proof automation

To illustrate the usage of our proof automation, we begin by defining a predicate Even on natural numbers as follows:

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

Next we may want to prove properties of this definition:

```
even+ : Even n → Even m → Even (n + m)
even+  isEven0       e2 = e2
even+ (isEven+2 e1) e2 = isEven+2 (even+ e1 e2)
```

Note that we omit universally quantified implicit arguments from the typeset version of this paper, in accordance with convention used by Haskell [19] and Idris [3].

As shown by Van der Walt and Swierstra [25], it is easy to decide the Even property for closed terms using proof by reflection. The interesting terms, however, are seldom closed. For instance, if we would like to use the even+ lemma in the proof below, we need to call it explicitly.

```
simple : Even n → Even (n + 2)
simple e = even+ e (isEven+2 isEven0)
```

Manually constructing explicit proof objects in this fashion is not easy. The proof is brittle. We cannot easily reuse it to prove similar statements such as Even (n + 4). If we need to reformulate our statement slightly, proving Even (2 + n) instead, we need to rewrite our proof. Proof automation can make propositions more robust against such changes.

Coq's proof search tactics, such as auto, can be customized with a *hint database*, a collection of related lemmas. In our example, auto would be able to prove the simple lemma, provided it the hint database contains at least the constructors of the Even data type and the even+ lemma. In contrast to the construction of explicit proof terms, changes to the theorem statement need not break the proof. This paper shows how to implement a similar tactic as an ordinary function in Agda.

Before we can use our auto function, we need to construct a hint database:

```
hints : HintDB
hints =
  [] « quote isEven0 « quote isEven+2 « quote even+
```

To construct such a database, we **quote** any terms that we wish to include in it and pass them to the hintdb function. We defer any discussion about the hintdb function to Section **??**. Note, however, that unlike Coq, the hint data base is a *first-class* value that can be manipulated, inspected, or passed as an argument to a function.

We now give an alternative proof of the simple lemma, using this hint database:

```
simple : Even n → Even (n + 2)
simple = quoteGoal g in unquote (auto 5 hints g)
```

The central ingredient is a *function* auto with the following type:

```
auto : (depth : ℕ) → HintDB → Term → Term
```

Given a maximum depth, hint database, and goal, it searches for a proof Term that witnesses our goal. If this term can be found, it is spliced back into our program using the **unquote** statement.

Of course, such invocations of the auto function may fail. What happens if no proof exists? For example, trying to prove Even n → Even (n + 3) in this style gives the following error:

```
Exception searchSpaceExhausted !=<
  Even .n -> Even (.n + 3) of type Set
```

When no proof can be found, the auto function generates a dummy term whose type explains the reason why the search has failed. In this example, the search space has been exhausted. Unquoting this term, then gives the type error message above. It is up to the programmer to fix this, either by providing a manual proof or diagnosing why no proof could be found.

The remainder of this paper will explain how this auto function is implemented.

3. Proof search in Agda

The following section describes our implementation of proof search à la Prolog in Agda. This implementation abstracts over two data types for names—one for inference rules and term constructors. These data types will be referred to as RuleName and TermName, and will be instantiated with types (with the same names) in section 4.

Terms and unification

The heart of our proof search implementation is the structurally recursive unification algorithm described by McBride [15]. Here the type of terms is indexed by the number of variables a given term may contain. Doing so enables the formulation of the unification algorithm by structural induction on the number of free variables. For this to work, we will use the following definition of terms[3]:

```
data PsTerm (n : ℕ) : Set where
  var  : Fin n → PsTerm n
  con  : TermName → List (PsTerm n) → PsTerm n
```

In addition to a restricted set of variables, we will allow first-order constants encoded as a name with a list of arguments.

For instance, if we choose to instantiate PsName with the following Arith data type, we can encode numbers and simple arithmetic expressions:

```
data Arith : Set where
  Suc  : Arith
  Zero : Arith
  Add  : Arith
```

The closed term corresponding to the number one could be written as follows:

```
One : PsTerm 0
One = con Suc (con Zero [] :: [])
```

Similarly, we can use the var constructor to represent open terms, such as x + 1. We use the prefix operator # to convert from natural numbers to finite types:

```
AddOne : PsTerm 1
AddOne = con Add (var (# 0) :: con One [] :: [])
```

Note that this representation of terms is untyped. There is no check that enforces addition is provided precisely two arguments. Although we could add further type information to this effect, this introduces additional overhead without adding safety to the proof automation presented in this paper. For the sake of simplicity, we have therefore chosen to work with this untyped definition.

We shall refrain from further discussion of the unification algorithm itself. Instead, we restrict ourselves to presenting the interface that we will use:

```
unify : (t₁ t₂ : PsTerm m) → Maybe (∃ [n] Subst m n)
```

The unify function takes two terms t_1 and t_2 and tries to compute a sustitution—the most general unifier. Substitutions are indexed by two natural numbers m and n. A substitution of type Subst m n can be applied to a PsTerm m to produce a value of type PsTerm n. As unification may fail, the result is wrapped in the Maybe type. In addition, since the number of variables in the terms resulting from the unifying substitution is not known *a priori*, this number is existentially quantified over. For the remainder of the paper, we will write ∃ [x] B to mean a type B with occurrences of an existentially quantified variable x, or ∃ (λ x → B) in full.

Occasionally we will use a more general function unifyAcc, which takes a substitution as an additional parameter. It applies this substitution to t_1 and t_2 before attempting to unify.

```
unifyAcc : (t₁ t₂ : PsTerm m)
         → ∃ [n] Subst m n → Maybe (∃ [n] Subst m n)
```

[3] We will use the name PsTerm to stand for *proof search term* to differentiate them from Agda's *reflection terms*, or AgTerm.

Inference rules

Constructing the search tree

Searching for proofs

4. Adding reflection

5. Type classes

As a final application of our proof search algorithm, we show how it can be used to implement a *type classes* in the style of Haskell. Souzeau and Oury [22] have already shown how to use Coq's proof search mechanism to construct dictionaries. Using Agda's *instance arguments* [10] and the proof search presented in this paper, we mimic their results.

We begin by declaring our 'type class' as a record containing the desired function:

```
record Show (A : Set) : Set where
  field
    show : A → String
```

We can write instances for the Show 'class' by constructing explicit dictionary objects:

```
ShowBool : Show Bool
ShowBool = record {show = ...}
Showℕ : Show ℕ
Showℕ = record {show = ...}
```

Using instance arguments, we can now call our show function without having to pass the required dictionary explicitly:

```
open Show {{...}}
example : String
example = show 3
```

The instance argument mechanism infers that the show function is being called on a natural number, hence a dictionary of type Show ℕ is required. As there is only a single value of type Show ℕ, the required dictionary is inserted automatically. If we have multiple instance definitions for the same type or omit the required instance altogether, the Agda type checker would have given an error.

It is more interesting to consider parametrized instances, such as the Either instance given below.

```
ShowEither : Show A → Show B → Show (Either A B)
ShowEither ShowA ShowB = record {show = showE}
  where
    showE : Either A B → String
    showE (left x)  = "left " ++ show x
    showE (right y) = "right " ++ show y
```

Unfortunately, instance arguments do not do any recursive search for suitable instances. Trying to call show on a value of type Either ℕ Bool, for example, will not succeed: the Agda type checker will complain that it cannot find a suitable instance argument. At the moment, the only way to resolve this is to construct the required instances manually:

```
ShowEitherBoolℕ : Show (Either Bool ℕ)
ShowEitherBoolℕ = ShowEither ShowBool Showℕ
```

Writing out such dictionaries is rather tedious.

We can, however, use the auto function to construct the desired instance argument automatically. We start by putting the desired instances in a hint database:

```
ShowHints : HintDB
ShowHints = hintdb (quote ShowEither
                        :: quote ShowBool
                        :: quote Showℕ :: [])
```

The desired dictionary can now be assembled for us by calling the auto function:

```
example : String
example = show (left 4) ++ show (right true)
  where
    instance = quoteGoal g
               in unquote (auto 5 ShowHints g)
```

Note that the type of the locally bound instance record is inferred in this example. Using this type, the auto function assembles the desired dictionary. When show is called on different types, however, we may still need to provide the type signatures of the instances we desire. While deceptively simple, this example illustrates how *useful* it can be to have even a little automation.

6. Discussion

The auto function presented here is far from perfect. This section not only discusses its limitations, but compares it to existing proof automation techniques in interactive proof assistants.

Performance First of all, the performance of the auto function is terrible. Any proofs that require a depth greater than ten are intractable in practice. This is an immediate consequence of Agda's poor compile-time evaluation. The current implementation is call-by-name and does no optimization whatsoever. While a mature evaluator is beyond the scope of this project, we believe that it is essential for Agda proofs to scale beyond toy examples. Simple optimizations, such as the erasure of the natural number indexes used in unification [4], would certainly help speed up the proof search.

Restrictions The auto function can only handle first-order terms. Even though higher-order unification is not decidable in general, we believe that it should be possible to adapt our algorithm to work on second-order goals. Furthermore, there are plenty of Agda features that are not supported or ignored by our quotation functions, such as universe polymorphism, instance arguments, and primitive functions.

Even for definitions that seem completely first-order, our auto function can fail unexpectedly. Consider the following definition of the product type, taken from Agda's standard library:

```
_×_ : (A B : Set) → Set
A × B = Σ A (λ _ → B)
```

Here a (non-dependent) pair is defined as a special case of the type Σ, representing dependent pairs. We can define the obvious Show instance for such pairs:

```
Show× : Show A → Show B → Show (A × B)
```

Somewhat surprisingly, trying to use this rule to create an instance of the Show 'class' fails. The **quoteGoal** construct always returns the goal in normal form, which exposes the higher-order nature of A × B. Converting the goal Show (A × (λ _ → B)) to a PrologTerm will raises the 'exception' unsupportedSyntax; the goal type contains a lambda which we cannot handle.

Furthermore, there are some limitations on the hints that may be stored in the hint database. At the moment, we construct every hint by quoting an Agda Name. Not all useful hints, however, have a such a Name, such as any variables locally bound in the context by pattern matching or function arguments. For example, the following call to the auto function fails to produce the desired proof:

```
simple : Even n → Even (n + 2)
simple e = quoteGoal g in unquote (auto 5 hints g)
```

The variable e, necessary to complete the proof is not part of the hint database. We hope that this could be easily fixed by providing a variation of the **quoteGoal** construct that returns both the term representing to the current goal and a list of the terms bound in the local context.

Refinement The auto function returns a complete proof term or fails entirely. This is not always desirable. We may want to return an incomplete proof, that still has open holes that the user must complete. This difficult with the current implementation of Agda's reflection mechanism: it cannot generate an incomplete Term.

In the future, it may be interesting to explore how to integrate proof automation, as described in this paper, better with Agda's IDE. If the call to auto were to generate the concrete syntax for a (possibly incomplete) proof term, this could be replaced with the current goal quite easily. An additional advantage of this approach would be that reloading the file does no longer needs to recompute the proof terms.

Metatheory The auto function is necessarily untyped because the interface of Agda's reflection mechanism is untyped. Defining a well-typed representation of dependent types in a dependently typed language remains an open problem, despite various efforts in this direction [6, 8, 11, 16]. If we had such a representation, however, we might be able to use the type information to prove that when the auto function succeeds, the resulting term has the correct type. As it stands, proving soundness of the auto function is non-trivial: we would need to define the typing rules of Agda's Term data type and prove that the Term we produce witnesses the validity of our goal Term. It may be slightly easier to ignore Agda's reflection mechanism and instead verify the metatheory of the Prolog interpreter: if a proof exists at some given depth, searchToDepth should find it; any Result returned by searchToDepth should correspond to a valid derivation.

Related work

There are several other interactive proof assistants, dependently typed programming languages, and alternative forms of proof automation in Agda. In the remainder of this section, we will briefly compare the approach taken in this paper to these existing systems.

Coq Coq has rich support for proof automation. The Ltac language and the many primitive, customizable tactics are extremely powerful [7]. Despite Coq's success, it is still worthwhile to explore better methods for proof automation. Recent work on Mtac [27] shows how to add a typed language for proof automation on top of Ltac. Furthermore, Ltac itself is not designed to be a general purpose programming language. It can be difficult to abstract over certain patterns and debugging proof automation is not easy. The programmable proof automation, written using reflection, presented here may not be as mature as Coq's Ltac language, but addresses these issues.

Idris The dependently typed programming language Idris also has a collection of tactics, inspired by some of the more simple Coq tactics, such as rewrite, intros, or exact. Each of these tactics is built-in and implemented as part of the Idris system. There is a small Haskell library for tactic writers to use that exposes common commands, such as unification, evaluation, or type checking. Furthermore, there are library functions to help handle the construction of proof terms, generation of fresh names, and splitting subgoals. This approach is reminiscent of the HOL family of theorem provers [12] or Coq's plug-in mechanism. An important drawback is that tactic writers need to write their tactics in a different language to the rest of their Idris code; furthermore, any changes to tactics requires a recompilation of the entire Idris system.

Agsy Agda already has a built-in 'auto' tactic that outperforms the auto function we have defined here [13]. It is nicely integrated with the IDE and does not require the users to provide an explicit hint database. It is, however, implemented in Haskell and shipped as part of the Agda system. As a result, users have very few opportunities for customization: there is limited control over which hints may (or may not) be used; there is no way to assign priorities to certain hints; and there is a single fixed search strategy. In contrast to the proof search presented here, where we have much more fine grained control over all these issues.

Closure

The proof automation presented in this paper is not as mature as some of these alternative systems. Yet we strongly believe that this style of proof automation is worth pursuing further.

The advantages of using reflection to program proof tactics should be clear: we do not need to learn a new programming language to write new tactics; we can use existing language technology to debug and test our tactics; and we can use all of Agda's expressive power in the design and implementation of our tactics. If a particular problem domain requires a different search strategy, this can be implemented by writing a new traversal over a SearchTree. Hint databases are first-class values. There is never any built-in magic; there are no compiler primitives beyond Agda's reflection mechanism.

The central philosophy of Martin-Löf type theory is that the construction of programs and proofs is the same activity. Any external language for proof automation renounces this philosophy. This paper demonstrates that proof automation is not inherently at odds with the philosophy of type theory. Paraphrasing Martin-Löf [14], it no longer seems possible to distinguish the discipline of *programming* from the *construction* of mathematics.

Acknowledgements We would like to thank the Software Technology Reading Club at the Universiteit Utrecht for their helpful feedback.

References

[1] 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].

[2] Guillaume Allais. Proof automatization using reflection (implementations in Agda). MSc Intern report, University of Nottingham, 2010.

[3] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. doi: 10.1017/S095679681300018X.

[4] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, pages 115–129. Springer, 2004.

[5] Thomas Braibant. Emancipate yourself from Ltac. Available online `http://gallium.inria.fr/blog/your-first-coq-plugin/`, 2012.

[6] James Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham, 2009.

[7] Adam Chlipala. *Certified programming with dependent types*. MIT Press, 2013.

[8] Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*. Spring Verlag, 2006.

[9] The Coq development team. The Coq proof assistant reference manual. Logical Project, 2004.

[10] Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 143–155. ACM, 2011. doi: 10.1145/2034773.2034796.

[11] Dominique Devriese and Frank Piessens. Typed syntactic metaprogramming. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP 2013)*. ACM, September 2013. doi: 10.1145/2500365.2500575.

[12] M.J.C Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[13] Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in Agda. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs*, pages 154–169. Springer-Verlag, 2004.

[14] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 167–184. Prentice-Hall, Inc., 1985.

[15] Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13:1061–1075, 11 2003. ISSN 1469-7653. doi: 10.1017/S0956796803004957. URL `http://journals.cambridge.org/article_S0956796803004957`.

[16] Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. ACM. doi: 10.1145/1863495.1863497.

[17] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

[18] Ulf Norell. Playing with Agda. Invited talk at TPHOLS, 2009.

[19] Simon Peyton Jones, editor. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[20] Kent M. Pitman. Special forms in Lisp. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 179–187. ACM, 1980.

[21] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, 2002. doi: 10.1145/581690.581691.

[22] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008.

[23] Jurriën Stutterheim, Wouter Swierstra, and Doaitse Swierstra. Forty hours of declarative programming: Teaching Prolog at the Junior College Utrecht. In *Proceedings First International Workshop on Trends in Functional Programming in Education, University of St. Andrews, Scotland, UK, 11th June 2012*, volume 106 of *Electronic Proceedings in Theoretical Computer Science*, pages 50–62, 2013.

[24] Walid Taha and Tim Sheard. 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. doi: 10.1145/258993.259019.

[25] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 157–173. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41581-4. doi: 10.1007/978-3-642-41582-1_10.

[26] Paul van der Walt. 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`.

[27] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 87–100, 2013. doi: 10.1145/2500365.2500579.