Auto in Agda

Programming proof search using reflection

Pepijn Kokke and Wouter Swierstra

Universiteit Utrecht
pepijn.kokke@gmail 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. 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 [Norell, 2007] or Coq [2004], 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 [Chlipala, 2013]. 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 [Braibant, 2012].

Agda does not have Coq's segregation of proof and programming language. Instead, programmers are encouraged to automate proofs by writing their own solvers [Norell, 2009b]. In combination with Agda's reflection mechanism [Agda development team, 2013, van der Walt and Swierstra, 2013], developers can write powerful automatic decision procedures [Allais, 2010]. Unfortunately, not all proofs are easily automated in this fashion. If this is the 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. In other words, we have defined a *program* for the automatic *construction* of *mathematical* proofs. More specifically, this paper makes several novel contributions:

– We show how to implement a Prolog interpreter in the style of Stutterheim et al. [2013] 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 enables it to produce a proof term that shows the resulting substitution is valid.
– 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.

Although Agda already has built-in proof search functionality [Lindblad and Benke, 2004], our approach has several key advantages over most existing approaches to proof automation:

– Our library is highly customizable. We may parametrize our tactics over the search depth, hint database, or search strategy. Each of these is itself a first-class Agda value, that may be inspected or transformed, depending on the user's needs.
– Although we limit ourself in the paper to a simple depth-first search, different proofs may require a different search strategy. Such changes are easily made in our library. To illustrate this point, we will develop a variation of our tactic which allows the user to limit the number of times certain rules may be applied (Section 5).
– Users need not learn a new programming language to modify existing tactics or develop tactics of their own. They can use a full-blown programming language to define their tactics, rather than restrict themselves to a domain-specific tactic language such as Ltac.
– Finally, users can use all the existing Agda technology for testing and debugging *programs* when debugging the generation of *proofs*. Debugging complex tactics in Coq requires a great deal of expertise – we hope that implementing tactics as a library will make this process easier.

We will compare our library with the various alternative forms of proof automation in greater depth in Section 6, after we have presented our development.

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.

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

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 [Pitman, 1980], MetaML [Taha and Sheard, 1997], and Template Haskell [Sheard and Peyton Jones, 2002]. This reflection mechanism makes it possible to convert a program fragment into its corresponding abstract syntax tree and vice versa. We will introduce Agda's reflection mechanism here with several short examples, based on the explanation in previous work [van der Walt and Swierstra, 2013]. A more complete overview can be found in the Agda release notes [Agda development team, 2013] and Van der Walt's thesis [2012].

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.

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 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 [])))
```

[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.

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 **unquote** construct then type-checks the given term, and turns it into the definition $\lambda\,x \to \lambda\,y \to x$.

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 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 [Peyton Jones, 2003] and Idris [Brady, 2013].

As shown by Van der Walt and Swierstra [2013], 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.

```
trivial : Even n → Even (n + 2)
trivial 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 trivial lemma, provided 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 = ε « quote isEven0 « quote isEven+2 « quote even+
```

To construct such a database, we use **quote** to obtain the names of any terms that we wish to include in it and pass them to the right-hand side of the _«_ function, which will insert them into a hint database to the left. Note that ε represents the empty hint database. We will describe the implementation of _«_ in more detail in Section 4. For now it should suffice to say that, in the case of even+, after the **quote** construct obtains an Agda Name, _«_ uses the built-in function type to look up the type associated with even+, and generates a derivation rule which states that given two proofs of Even n and Even m, applying the rule even+ will result in a proof of Even (n + m).

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 trivial lemma using the auto tactic and the hint database defined above:

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

Or, using the newly added Agda tactic syntax[3]:

```
trivial : Even n → Even (n + 2)
trivial = tactic (auto 5 hints)
```

The notation **tactic** f is simply syntactic sugar for **quoteGoal** g **in unquote** (f g), for some function f.

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
```

[3] Syntax for Agda tactics was added in Agda 2.4.2.

When no proof can be found, the auto function generates a dummy term with a type that explains the reason 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.

Overview The remainder of this paper describes how the auto function is implemented. Before delving into the details of its implementation, however, we will give a high-level overview of the steps involved:

1. The **tactic** keyword converts the goal type to an abstract syntax tree, i.e., a value of type Term. In what follows we will use AgTerm to denote such terms, to avoid confusion with the other term data type that we use.
2. Next, we check the goal term. If it has a functional type, we add the arguments of this function to our hint database, implicitly introducing additional lambdas to the proof term we intend to construct. At this point we check that the remaining type and all its original arguments are are first-order. If this check fails, we produce an error message, not unlike the searchSpaceExhausted term we saw above. We require terms to be first-order to ensure that the unification algorithm, used in later steps for proof search, is decidable. If the goal term is first-order, we convert it to our own term data type for proof search, PsTerm.
3. The key proof search algorithm, presented in the next section, then tries to apply the hints from the hint database to prove the goal. This process coinductively generates a (potentially infinite) search tree. A simple bounded depth-first search through this tree tries to find a series of hints that can be used to prove the goal.
4. If such a proof is found, this is converted back to an AgTerm; otherwise, we produce an erroneous term describing that the search space has been exhausted. Finally, the **unquote** keyword type checks the generated AgTerm and splices it back into our development.

The rest of this paper will explain these steps in greater detail.

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 one for term constructors. These data types will be referred to as RuleName and TermName, and will be instantiated with concrete 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 [2003]. 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

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

We will use the name PsTerm to stand for *proof search term* to differentiate them from the terms from Agda's *reflection* mechanism, AgTerm. In addition to variables, represented by the finite type Fin n, we will allow first-order constants encoded as a name with a list of arguments.

For instance, if we choose to instantiate TermName 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) :: 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 substitution—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.

Inference rules

The hints in the hint database will form *inference rules* that we may use to prove a goal term. We represent such rules as records containing a rule name, a list of terms for its premises, and a term for its conclusion:

```
record Rule (n : ℕ) : Set where
  field
    name       : RuleName
    premises   : List (PsTerm n)
    conclusion : PsTerm n
  arity : ℕ
  arity = length premises
```

Once again the data-type is quantified over the number of variables used in the rule. Note that the number of variables in the premises and the conclusion is the same.

Using our newly defined Rule type we can give a simple definition of addition. In Prolog, this would be written as follows.

```
add(0, X, X).
add(suc(X), Y, suc(Z)) :- add(X, Y, Z).
```

Unfortunately, the named equivalents in our Agda implementation given in Figure 1 are a bit more verbose. Note that we have, for the sake of this example, instantiated the RuleName and TermName to String and Arith respectively.

A *hint database* is nothing more than a list of rules. As the individual rules may have different numbers of variables, we existentially quantify these:

```
HintDB : Set
HintDB = List (∃ [n] Rule n)
```

Generalised injection and raising

Before we can implement some form of proof search, we need to define a pair of auxiliary functions. During proof resolution, we will work with terms and rules containing a different number of variables. We will use the following pair of functions, inject and raise, to weaken bound variables, that is, map values of type Fin n to some larger finite type.

```
inject : ∀ {m} n → Fin m → Fin (m + n)
inject n zero    = zero
inject n (suc i) = suc (inject n i)
raise : ∀ m {n} → Fin n → Fin (m + n)
raise zero    i = i
raise (suc m) i = suc (raise m i)
```

```
AddBase : Rule 1
AddBase = record {
  name       = "AddBase"
  conclusion = con Add ( con Zero []
                       :: var (# 0)
                       :: var (# 0)
                       :: [])
  premises   = []
  }
AddStep : Rule 3
AddStep = record {
  name       = "AddStep"
  conclusion = con Add ( con Suc (var (# 0) :: [])
                       :: var (# 1)
                       :: con Suc (var (# 2) :: [])
                       :: [])
  premises   = con Add ( var (# 0)
                       :: var (# 1)
                       :: var (# 2)
                       :: [])
               :: []
  }
```

Fig. 1. Agda representation of example rules

On the surface, the inject function appears to be the identity. When you make all the implicit arguments explicit, however, you will see that it sends the zero constructor in Fin m to the zero constructor of type Fin (m + n). Hence, the inject function maps Fin m into the *first* m elements of the type Fin (m + n). Dually, the raise function maps Fin n into the *last* n elements of the type Fin (m + n) by repeatedly applying the suc constructor.

We can use inject and raise to define similar functions that work on our Rule and PsTerm data types, by mapping them over all the variables that they contain.

Constructing the search tree

Our proof search procedure is consists of two steps. First, we coinductively construct a (potentially infinite) search space; next, we will perform a bounded depth-first traversal of this space to find a proof of our goal.

We will represent the search space as a (potentially) infinitely deep, but finitely branching rose tree.

```
data SearchTree (A : Set) : Set where
  leaf  : A → SearchTree A
  node  : List (∞ (SearchTree A)) → SearchTree A
```

We will instantiate the type parameter A with a type representing proof terms. These terms consist of applications of rules, with a sub-proof for every premise.

```
data Proof : Set where
  con : (name : RuleName) (args : List Proof) → Proof
```

Unfortunately, during the proof search we will have to work with *partially complete* proof terms.

Such partial completed proofs are represented by the PartialProof type. In contrast to the Proof data type, the PartialProof type may contain variables, hence the type takes an additional number as its argument:

```
PartialProof : ℕ → Set
PartialProof m = ∃ [k] Vec (PsTerm m) k × (Vec Proof k → Proof)
```

A value of type PartialProof m records three separate pieces of information:

- a number k, representing the number of open subgoals;
- a vector of length k, recording the subgoals that are still open;
- a function that, given a vector of k proofs for each of the subgoals, will produce a complete proof of the original goal.

Next, we define the following function to help construct partial proof terms:

```
apply : (r : Rule n) → Vec Proof (arity r + k) → Vec Proof (suc k)
apply r xs = new :: rest
  where
    new  = con (name r) (toList (take (arity r) xs))
    rest = drop (arity r) xs
```

Given a Rule and a list of proofs of subgoals, this apply function takes the required sub-proofs from the vector, and creates a new proof by applying the argument rule to these sub-proofs. The result then consists of this new proof, together with any unused sub-proofs. This is essentially the 'unflattening' of a rose tree.

We can now finally return to our proof search algorithm. The solveAcc function forms the heart of the search procedure. Given a hint database and the current partially complete proof, it produces a SearchTree containing completed proofs.

```
solveAcc : HintDB → PartialProof (δ + m) → SearchTree Proof
solveAcc rules (0     ,     [] , p) = leaf (p [])
solveAcc rules (suc k , g :: gs , p) = node (map step rules)
```

If there are no remaining subgoals, i.e., the list in the second component of the PartialProof is empty, the search is finished. We construct a proof p [], and wrap

this in the leaf constructor of the SearchTree. If we still have open subgoals, we have more work to do. In that case, we will try to apply every rule in our hint database to resolve this open goal—our rose tree has as many branches as there are hints in the hint database. The real work is done by the step function, locally defined in a where clause, that given the rule to apply, computes the remainder of the SearchTree.

Before giving the definition of the step function, we will try to provide some intuition. Given a rule, the step function will try to unify its conclusion with the current subgoal g. When this succeeds, the premises of the rule are added to the list of open subgoals. When this fails, we return a node with no children, indicating that applying this rule can never prove our current goal.

Carefully dealing with variables, however, introduces some complication, as the code for the step function illustrates:

```
step : ∃ [δ] (Rule δ) → ∞ (SearchTree Proof)
step (δ, r)
  with unify (inject δ g) (raise m (conclusion r))
... | nothing       = ♯ node []
... | just (n, mgu) = ♯ solveAcc prf
  where
  prf : PartialProof n
  prf = arity r + k, gs′, (p ∘ apply r)
    where
    gs′ : Vec (Goal n) (arity r + k)
    gs′ = map (sub mgu) (raise m (fromList (premises r)) ++ inject δ gs)
```

Note that we use the function sub to apply a substitution to a term. This function is defined by McBride [2003].

The rule given to the step function may have a number of free variables of its own. As a result, all goals have to be injected into a larger domain which includes all current variables *and* the new rule's variables. The rule's premises and conclusion are then also raised into this larger domain, to guarantee freshness of the rule variables.

The definition of the step function attempts to unify the current subgoal g and conclusion of the rule r. If this fails, we can return node [] immediately. If this succeeds, however, we build up a new partial proof, prf. This new partial proof, once again, consists of three parts:

- the number of open subgoals is incremented by arity r, i.e., the number of premises of the rule r.
- the vector of open subgoals gs is extended with the premises of r, after weakening the variables of appropriately.
- the function producing the final Proof object will, given the proofs of the premises of r, call apply r to create the desired con node in the final proof object.

The only remaining step, is to kick-off our proof search algorithm with a partial proof, consisting of a single goal.

```
solve : (goal : PsTerm m) → HintDB → SearchTree Proof
solve g rules = solveAcc (1, g :: [], head)
```

Searching for proofs

After all this construction, we are left with a simple tree structure, which we can traverse in search of solutions. For instance, we can define a bounded depth-first traversal.

```
dfs : (depth : ℕ) → SearchTree A → List A
dfs zero     _          = []
dfs (suc k) (leaf x)    = return x
dfs (suc k) (node xs)   = concatMap (λ x → dfs k (♭ x)) xs
```

It is fairly straightforward to define other traversal strategies, such as a breadth-first search. Similarly, we could define a function which traverses the search tree aided by some heuristic. We will explore further variations on search strategies in Section 5.

4 Adding reflection

To complete the definition of our auto function, we still need to convert between Agda's built-in AgTerm data type and the data type required by our unification and resolution algorithms, PsTerm. Similarly, we will need to transform the Proof produced by our solve function to an AgTerm that can be unquoted. These are essential pieces of plumbing, necessary to provide the desired proof automation. While not conceptually difficult, this does expose some of the limitations and design choices of the auto function. If you are unfamiliar with the precise workings of the Agda reflection mechanism, you may want to skim this section.

The first thing we will need are concrete definitions for the TermName and RuleName data types, which were parameters to the development presented in the previous section. It would be desirable to identify both types with Agda's Name type, but unfortunately Agda does not assign a name to the function space type operator, _→_; nor does Agda assign names to locally bound variables. To address this, we define two new data types TermName and RuleName.
First, we define the TermName data type.

```
data TermName : Set where
  name : Name → TermName
  pvar : ℕ → TermName
  impl : TermName
```

The TermName data type has three constructors. The name constructor embeds Agda's built-in Name in the TermName type. The pvar constructor describes locally bound variables, represented by their De Bruijn index. Note that the

pvar constructor has nothing to do with PsTerm's var constructor: it is not used to construct a Prolog variable, but rather to be able to refer to a local variable as a Prolog constant. Finally, impl explicitly represents the Agda function space.

We define the RuleName type in a similar fashion.

```
data RuleName : Set where
  name : Name → RuleName
  rvar : ℕ → RuleName
```

The rvar constructor is used to refer to Agda variables as rules. Its argument i corresponds to the variable's De Bruijn index – the value of i can be used directly as an argument to the var constructor of Agda's Term data type.

As we have seen in Section 2, the auto function may fail to find the desired proof. Furthermore, the conversion from AgTerm to PsTerm may also fail for various reasons. To handle such errors, we will work in the Error monad defined below:

```
Error : (A : Set a) → Set a
Error A = Message ⊎ A
```

Upon failure, the auto function will produce an error message. The corresponding Message type simply enumerates the possible sources of failure:

```
data Message : Set where
  searchSpaceExhausted : Message
  unsupportedSyntax    : Message
```

The meaning of each of these error messages will be explained as we encounter them in our implementation below.

Finally, we will need one more auxiliary function to manipulate bound variables. The match function takes two bound variables of types Fin m and Fin n and computes the corresponding variables in Fin (m ⊔ n) – where m ⊔ n denotes the maximum of m and n:

```
match : Fin m → Fin n → Fin (m ⊔ n) × Fin (m ⊔ n)
```

The implementation is reasonably straightforward. We compare the numbers n and m, and use the inject function to weaken the appropriate bound variable. It is straightforward to use this match function to define similar operations on two terms or a term and a list of terms.

Constructing terms

We now turn our attention to the conversion of an AgTerm to a PsTerm. There are two problems that we must address.

First of all, the AgTerm type represents all (possibly higher-order) terms, whereas the PsTerm type is necessarily first-order. We mitigate this problem by

allowing the conversion to 'fail', by producing a term of the type Exception, as we saw in the introduction.

Secondly, the AgTerm data type uses natural numbers to represent variables. The PsTerm data type, on the other hand, represents variables using a finite type Fin n, for some n. To convert between these representations, the function keeps track of the current depth, i.e. the number of Π-types it has encountered, and uses this information to ensure a correct conversion. We sketch the definition of the main function below:

```
convert : (binders : ℕ) → AgTerm → Error (∃ PsTerm)
convert b (var i [])      = inj₂ (convertVar b i)
convert b (con n args)  = convertName n ∘ convert b ⟨$⟩ args
convert b (def n args)  = convertName n ∘ convert b ⟨$⟩ args
convert b (pi (arg (arg-info visible _) (el _ t₁)) (el _ t₂))
  with convert b t₁ | convert (suc b) t₂
... | inj₁ msg      | _           = inj₁ msg
... | _             | inj₁ msg  = inj₁ msg
... | inj₂ (n₁, p₁) | inj₂ (n₂, p₂)
  with match p₁ p₂
... | (p₁′, p₂′) = inj₂ (n₁ ⊔ n₂, con impl (p₁′ :: p₂′ :: []))
convert b (pi (arg _ _) (el _ t₂)) = convert (suc b) t₂
convert b _               = inj₁ unsupportedSyntax
```

We define special functions, convertVar and name2term, to convert variables and constructors or defined terms respectively. The arguments to constructors or defined terms are processed using the convertChildren function defined below. The conversion of a pi node binding an explicit argument proceeds by converting the domain and then codomain. If both conversions succeed, the resulting terms are matched and a PsTerm is constructed using impl. Implicit arguments and instance arguments are ignored by this conversion function. Sorts, levels, or any other Agda feature mapped to the constructor unknown of type Term triggers a failure with the message unsupportedSyntax.

The convertChildren function converts a list of Term arguments to a list of Prolog terms, by stripping the arg constructor and recursively applying the convert function. We only give its type signature here, as the definition is straightforward:

```
convertChildren : ℕ → List (Arg Term) → Error (∃ (List ∘ PsTerm))
```

To convert between an AgTerm and PsTerm we simply call the convert function, initializing the number of binders encountered to 0.

```
agda2term : AgTerm → Error (∃ PsTerm)
agda2term t = convert 0 t
```

Constructing rules

Our next goal is to construct rules. More specifically, we need to convert a list of quoted Names to a hint database of Prolog rules. To return to our example in Section 2, the definition of even+ had the following type:

```
even+ : Even n → Even m → Even (n + m)
```

We would like to construct a value of type Rule that expresses how even+ can be used. In Prolog, we might formulate the lemma above as the rule:

```
even(add(M,N)) :- even(M), even(N).
```

In our Agda implementation, we can define such a rule manually:

```
Even+ : Rule 2
Even+ = record {
  name       = name even+
  conclusion = con (name (quote Even)) (
               con (name (quote _+_)) (var (# 0) :: var (# 1) :: [])
               :: []
               )
  premises   =   con (name (quote Even)) (var (# 0) :: [])
               :: con (name (quote Even)) (var (# 1) :: [])
               :: []
  }
```

In the coming subsection, we will show how to generate the above definition from the Name representing even+.

This generation of rules is done in two steps. First, we will convert a Name to its corresponding PsTerm:

```
name2term : Name → Error (∃ PsTerm)
name2term = agda2term ∘ unel ∘ type
```

The type construct maps a Name to the AgTerm representing its type; the unel function discards any information about sorts; the agda2term was defined previously.

In the next step, we process this PsTerm. The split function, defined below, splits a PsTerm at every top-most occurrence of the function symbol impl. Note that it would be possible to define this function directly on the AgTerm data type, but defining it on the PsTerm data type is much cleaner as we may assume that any unsupported syntax has already been removed.

```
split : PsTerm n → ∃ (λ k → Vec (PsTerm n) (suc k))
split (con impl (t₁ :: t₂ :: [])) = Product.map suc (_::_ t₁) (split t₂)
split t = (0, t :: [])
```

Using all these auxiliary functions, we now define the name2rule function below that constructs a Rule from an Agda Name.

```
name2rule : Name → Error (∃ Rule)
name2rule nm with name2term nm
... | inj₁ msg          = inj₁ msg
... | inj₂ (n , t)     with split t
... | (k , ts)         with initLast ts
... | (prems , concl , _) = inj₂ (n , rule (name nm) concl (toList prems))
```

We convert a name to its corresponding PsTerm, which is converted to a vector of terms using split. The last element of this vector is the conclusion of the rule; the prefix constitutes the premises. We use the initLast function from the Agda standard library, to decompose this vector accordingly.

Constructing goals

Next, we turn our attention to converting a goal AgTerm to a PsTerm. While we could use the agda2term function to do so, there are good reasons to explore other alternatives.

Consider the example given in Section 2. The goal AgTerm we wish to prove is Even n → Even (n + 2). Calling agda2term would convert this to a PsTerm, where the function space has been replaced by the constructor impl. Instead, however, we would like to *introduce* arguments, such as Even n, as assumptions to our hint database.

In addition, we cannot directly reuse the implementation of convert that was used in the construction of terms. The convert function maps every AgTerm variable is mapped to a Prolog variable *that may still be instantiated*. When considering the goal type, however, we want to generate *skolem constants* for our variables. To account for this difference we have two flavours of the convert function: convert and convert4Goal. Both differ only in their implementation of convertVar.

```
agda2goal×premises : AgTerm → Error (∃ PsTerm × HintDB)
agda2goal×premises t with convert4Goal 0 t
... | inj₁ msg          = inj₁ msg
... | inj₂ (n , p)      with split p
... | (k , ts)          with initLast ts
... | (prems , goal , _) = inj₂ ((n , goal) , toPremises k prems)
```

Fortunately, we can reuse many of the other functions we have defined above, and, using the split and initLast functions, we can get our hands on the list of premises prems and the desired return type goal. The only missing piece of the puzzle is a function, toPremises, which converts a list of PsTerms to a hint database containing rules for the arguments of our goal.

```
toPremises : ∀ {k} → ℕ → Vec (PsTerm n) k → HintDB
toPremises i [] = []
toPremises i (t :: ts) = (n , rule (rvar i) t []) :: toPremises (suc i) ts
```

The toPremises converts every PsTerm in its argument list to a rule, using the argument's De Bruijn index as its rule name.

Reification of proof terms

Now that we can compute Prolog terms, goals and rules from an Agda Term, we are ready to call the resolution mechanism described in Section 3. The only remaining problem is to convert the witness computed by our proof search back to an AgTerm, which can be unquoted to produce the desired proof. This is done by the reify function that traverses its argument Proof; the only interesting question is how it handles the variables and names it encounters.

The Proof may contain two kinds of variables: locally bound variables, rvar i, or variables storing an Agda Name, name n. Each of these variables is treated differently in the reify function.

```
reify : Proof → AgTerm
reify (con (rvar i) ps) = var i []
reify (con (name n) ps) with definition n
... | function x    = def n (toArg ∘ reify ⟨$⟩ ps)
... | constructor′  = con n (toArg ∘ reify ⟨$⟩ ps)
... | _             = unknown
  where
    toArg : AgTerm → Arg AgTerm
    toArg = arg (arg-info visible relevant)
```

Any references to locally bound variables are mapped to the var constructor of the AgTerm data type. These variables correspond to usage of arguments to the function being defined. As we know by construction that these arguments are mapped to rules without premises, the corresponding Agda variables do not need any further arguments.

If, on the other hand, the rule being applied is constructed using a name, we do disambiguate whether the rule name refers to a function or a constructor. The definition function, defined in Agda's reflection library, tells you how a name was defined (i.e. as a function name, constructor, etc). For the sake of brevity, we restrict the definition here to only handle defined functions and data constructors. It is easy enough to extend with further branches for postulates, primitives, and so forth.

We will also need to wrap additional lambdas around the resulting term, due to the premises that were introduced by the agda2goal×premises function. To do so, we define the intros function that repeatedly wraps its argument term in a lambda.

```
intros : AgTerm → AgTerm
intros = introsAcc (length args)
  where
    introsAcc : ℕ → AgTerm → AgTerm
    introsAcc zero t = t
    introsAcc (suc k) t = lam visible (introsAcc k t)
```

Hint databases

Users to provide hints, i.e., rules that may be used during resolution, in the form of a *hint database*. These hint databases consist of an (existentially quantified) a list of rules. We can add new hints to an existing database using the insertion operator, «, defined as follows:

```
_«_ : HintDB → Name → HintDB
db « n with name2rule n
db « n | inj₁ msg = db
db « n | inj₂ r   = db ++ [r]
```

If the generation of a rule fails for whatever reason, no error is raised, and the rule is simply ignored. Our actual implementation requires an implicit proof argument that all the names in the argument list can be quoted successfully. If you define such proofs to compute the trivial unit record as evidence, Agda will fill them in automatically in every call to the _«_ function on constant arguments. This simple form of proof automation is pervasive in Agda programs [Oury and Swierstra, 2008, Swierstra, 2010].

This is the simplest possible form of hint database. In principle, there is no reason not to define alternative versions that assign priorities to certain rules or limit the number of times a rule may be applied. We will investigate some possibilities for extensible proof search in section 5.

It is worth repeating that hint databases are first-class objects. We can combine hints databases, filter certain rules from a hint database, or manipulate them in any way we wish.

Error messages

Lastly, we need to decide how to report error messages. Since we are going to return an AgTerm, we need to transform the Message type we saw previously into an AgTerm. When unquoted, this term will cause a type error, reporting the reason for failure. To accomplish this, we introduce a dependent type, indexed by a Message:

```
data Exception : Message → Set where
  throw : (msg : Message) → Exception msg
```

The message passed as an argument to the throw constructor, will be recorded in the Exception's type, as we intended.

Next, we define a function to produce an AgTerm from a Message. We could construct such terms by hand, but it is easier to just use Agda's **quoteTerm** construct:

```
quoteError : Message → Term
quoteError searchSpaceExhausted =
  quoteTerm (throw searchSpaceExhausted)
quoteError unsupportedSyntax    =
  quoteTerm (throw unsupportedSyntax)
```

Putting it all together

Finally, we can present the definition of the auto function used in the examples in Section 2:

```
auto : ℕ → HintDB → AgTerm → AgTerm
auto depth rules goalType
  with agda2goal×premises goalType
... | inj₁ msg = quoteError msg
... | inj₂ ((n, g), args)
  with dfs depth (solve g (args ++ rules))
... | [] = quoteError searchSpaceExhausted
... | (p :: _) = intros (reify p)
```

The auto function takes an AgTerm representing the goal type, splits it into PsTerms representing the goal g and a list of arguments, args. These arguments are added to the initial hint database. Calling the solve function with this hint database and the goal g, constructs a proof tree, that we traverse up to the given depth in search of a solution. If this proof search succeeds, the Proof is converted to an AgTerm, a witness that the original goal is inhabited. There are two places where this function may fail: the conversion to a PsTerm may fail because of unsupported syntax; or the proof search may not find a result.

5 Extensible proof search

As we promised in the previous section, we will now explore several variations and extensions to the auto tactic described above.

Custom search strategies

The simplest change we can make is to abstract over the search strategy used by the auto function. In the interest of readability we will create a simple alias for the types of search strategies. A Strategy represents a function which searches a SearchTree up to depth, and returns a list of the leaves (or Proofs) found in the SearchTree in an order which is dependent on the search strategy.

```
Strategy = (depth : ℕ) → SearchTree A → List A
```

The changed type of the auto function now becomes.

```
auto : Strategy → ℕ → HintDB → AgTerm → AgTerm
```

This will allow us to choose whether to pass in dfs, breadth-first search or even a custom user-provided search strategy.

Custom hint databases

In addition, we have developed a variant of the auto tactic described in the paper that allows users to define their own type of hint database, provided they can implement the following interface:

```
HintDB  : Set
Hint    : ℕ → Set
getHints : HintDB → Hints
getRule  : Hint k → Rule k
getTr    : Hint k → (HintDB → HintDB)
```

Besides the obvious types for hints and rules, we allow hint databases to evolve during the proof search. The user-defined getTr function describes a transformation that may modify the hint database after a certain hint has been applied.

Using this interface, we can implement many variations on proof search. For instance, we could implement a 'linear' proof search function that removes a rule from the hint database after it has been applied. Alternatively, we may want to assign priorities to our hints. To illustrate one possible application of this interface, we will describe a hint database implementation that limits the usage of certain rules. Before we do so, however, we need to introduce a motivating example.

Example: limited usage of hints

We start by defining the following sublist relation, taken from the Agda tutorial [Norell, 2009a]:

```
data _⊆_ : List A → List A → Set where
  stop : [] ⊆ []
  drop : xs ⊆ ys →      xs ⊆ y :: ys
  keep : xs ⊆ ys → x :: xs ⊆ x :: ys
```

It is easy to show that the sublist relation is both reflexive and transitive—and using these simple proofs, we can build up a small hint database to search for proofs on the sublist relation.

```
hintdb : HintDB
hintdb = ε « quote drop « quote keep « quote ⊆-refl « quote ⊆-trans
```

Our auto tactic quickly finds a proof for the following lemma:

```
lemma₁ : ws ⊆ 1 :: xs → xs ⊆ ys → ys ⊆ zs → ws ⊆ 1 :: 2 :: zs
lemma₁ = tactic (auto dfs 10 hintdb)
```

The following lemma, however, is false.

```
lemma₂ : ws ⊆ 1 :: xs → xs ⊆ ys → ys ⊆ zs → ws ⊆ 2 :: zs
lemma₂ = tactic (auto dfs 10 hintdb)
```

Indeed, this example does not type check and our tactic reports that the search space is exhausted. As noted by Chlipala [2013] when examining tactics in Coq, auto will nonetheless spend a considerable amount of time trying to construct a proof. As the trans rule is always applicable, the proof search will construct a search tree up to the full search depth—resulting in an exponental running time.

We will use a variation of the auto tactic to address this problem. Upon constructing the new hint database, users may assign limits to the number of times certain hints may be used. By limiting the usage of transitivity, our tactic will fail more quickly.

To begin with, we choose the representation of our hints: a pair of a rule and a 'counter' that records how often the rule may still be applied:

```
record Hint (k : ℕ) : Set where
  field
    rule    : Rule k
    counter : Counter
```

These counter values will either be a natural number n, representing that the rule can still be used at most n times; or ⊤, when the usage of the rule is unrestricted.

```
Counter : Set
Counter = ℕ ⊎ ⊤
```

Next, we define a decrementing function, decrCounter, that returns nothing when a rule can no longer be applied:

```
decrCounter : Counter → Maybe Counter
decrCounter (inj₁ 0)  = nothing
decrCounter (inj₁ 1)  = nothing
decrCounter (inj₁ x)  = just (inj₁ (pred x))
decrCounter (inj₂ tt) = just (inj₂ tt)
```

Given a hint h, the transition function will now simply find the position of h in the hint database and decrement the hint's counter, removing it from the database if necessary.

We can redefine the default insertion function (_«_) to allow unrestricted usage of a rule. However, we will define a new insertion function which will allow the user to limit the usage of a rule during proof search:

```
_« [_] _ : HintDB → ℕ → Name → HintDB
db « [0] _ = db
db « [x] n with (name2rule n)
db « [x] n | inj₁ msg = db
db « [x] n | inj₂ (k, r) = db ++ [k, record {rule = r; counter = inj₁ x}]
```

We now revisit our original hint database and limit the number of times transitivity may be applied:

```
hintdb : HintDB
hintdb = ε «     quote drop
         «     quote keep
         «     quote refl
         « [2] quote trans
```

If we were to search for a proof of lemma_2 now, our proof search fails sooner. *A fortiori*, if we use this restricted database when searching for a proof of lemma_1, the auto function succeeds sooner, as we have greatly reduced the search space. Of course, there are plenty of lemmas that require more than two applications of transitivity. The key insight, however, is that users now have control over these issues – something which is not even possible in current implementations of auto in Coq.

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.

Restricted language fragment 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 pair type:

```
_×_ : (A B : Set) → Set
A × B = Σ A (λ _ → B)
pair : {A B : Set} → A → B → A × B
pair x y = x, y
```

Here a (non-dependent) pair is defined as a special case of the dependent pair type Σ. Now consider the following trivial lemma:

```
andIntro : (A : Set) → (B : Set) → A × B
```

Somewhat surprisingly, trying to prove this lemma using our auto function, providing the pair function as a hint, fails. The **quoteGoal** construct always returns the goal in normal form, which exposes the higher-order nature of $A \times B$. Converting the goal $(A \times (\lambda\ \_ \to B))$ to a PsTerm will raise the 'exception' unsupportedSyntax; the goal type contains a lambda which causes the proof search to fail before it has even started.

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. The difficulty lies with the current implementation of Agda's reflection mechanism, as it cannot generate an incomplete Term.

In the future, it may be interesting to explore how to integrate proof automation using the reflection mechanism better with Agda's IDE. For instance, we could create an IDE feature which replaces a call to auto with the proof terms that it generates. As a result, reloading the file would no longer need 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 [Chapman, 2009, Danielsson, 2006, Devriese and Piessens, 2013, McBride, 2010]. If we had such a representation, however, we could use the type information to prove that when the auto function succeeds, the resulting term has the correct type. As it stands, a bug in our auto function could potentially produce an ill-typed proof term, that only causes a type error when that term is unquoted.

Variables The astute reader will have noticed that the tactic we have implemented is closer to Coq's eauto tactic than the auto tactic. The difference between the two tactics lies in the treatment of unification variables: eauto may introduce new variables during unification; auto will never do so. It would be fairly straightforward to restrict our tactic to only apply hints when all variables known. A suitable instantiation algorithm, which we could use instead of the more general unification algorithm in this paper, has already been developed in previous work [Swierstra and van Noort, 2013].

Technical limitations The auto tactic relies on the unification algorithm and proof search mechanism we have implemented ourselves. These are all run *at compile time*, using the reflection mechanism to try and find a suitable proof term. It is very difficult to say anything meaningful about the performance of the auto tactic, as Agda currently has no mechanism for debugging or profiling programs run at compile time. We hope that further advancement of the Agda compiler and associated toolchain can help provide meaningful measurements of the performance of auto. Similarly, a better (static) debugger would be invaluable when trying to understand why a call to auto failed to produce the desired proof.

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 [Chlipala, 2013]. Despite Coq's success, it is still worthwhile to explore better methods for proof automation. Recent work on Mtac [Ziliani et al., 2013] 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.

More recently, Malecha et al. [2014] have designed a higher-order reflective programming language (MirrorCore) and an associated tactic language (Rtac). MirrorCore defines a unification algorithm – similar to the one we have implemented in this paper. Alternative implementations of several familiar Coq tactics, such as eauto and setoid_rewrite, have been developed using Rtac. The authors have identified several similar advantages of 'programming' tactics, rather than using built-in primitives, that we mention in this paper, such as manipulating and assembling first-class hint databases.

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 sub-goals. This approach is reminiscent of the HOL family of theorem provers [Gordon and Melham, 1993] 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 [Lindblad and Benke, 2004]. 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.

Conclusion

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 [1985], 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, and all our anonymous reviewers for their helpful feedback – we hope we have done their feedback justice.

Bibliography

Agda development team. Agda release notes documenting the reflection mechanism. 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].

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

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.

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

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

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

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.

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

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

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

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.

Gregory Malecha, Adam Chlipala, and Thomas Braibant. Compositional computational reflection. In *Proceedings of the 5th International Conference on Interactive Theorem Proving (ITP'14)*, 2014.

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.

Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13:1061–1075, 11 2003. doi: 10.1017/S0956796803004957.

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.

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.

Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009a.

Ulf Norell. Playing with Agda. Invited talk at TPHOLS, 2009b.

Nicolas Oury and Wouter Swierstra. The Power of Pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 39–50, 2008. doi: 10.1145/1411204.1411213.

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

Kent M. Pitman. Special forms in Lisp. In *Proceedings of the 1980 ACM conference on LISP and Functional Programming*, pages 179–187. ACM, 1980.

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.

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.

Wouter Swierstra. More dependent types for distributed arrays. *Higher-order and symbolic computation*, 23(4):489–506, 2010.

Wouter Swierstra and Thomas van Noort. A library for polymorphic dynamic typing. *Journal of Functional Programming*, 23:229–248, 5 2013. doi: 10.1017/S0956796813000063.

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.

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. doi: 10.1007/978-3-642-41582-1_10.

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`.

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.