Under preparation for ICFP 2014

Auto in Agda

Programming proof search

Pepijn Kokke Wouter Swierstra

Universiteit Utrecht

pepijn.kokke@gmail.com

Abstract

We present the reader with an implementation of Prolog-style proof search in Agda. We then use this implementation, together with Agda's Reflection mechanism, to implement an auto tactic for first-order Agda terms. Last, we demonstrate one possible usage of this tactic, by implementing modular instance search for Agda-style type classes.

Wouter: Still need to finalize the abstract

1. Introduction

Writing proof terms in type theory is hard and often tedious. Interactive proof assistants based on type theory, such as Agda [14] 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 [15]. In combination with Agda's reflection mechanism [22], developers can write powerful automatic decision procedures [2]. Unfortunately, not all proofs are easily automated in this fashion. When 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. More specifically, this paper makes the following novel contributions:

- After illustrating the usage of our library with several motivating examples (Section 2), we show how to implement a Prolog interpreter in the style of Stutterheim et al. [20] in Agda (Section 3). Note that, in contrast to Agda, resolving a Prolog query

w.s.swierstra@uu.nl

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 term that can be derived from the given rules. We extend our interpreter to produce a proof term that witnesses the validity of the resulting substitution (Section 4).
- We integrate this proof search algorithm with Agda's *reflection* mechanism (Section 5). 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 6). This resolves one of the major restrictions of instance arguments: the lack of a recursive search

procedure for their construction.

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 [17], MetaML [21], and Template Haskell [18]. This reflection mechanisms make 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 [22]. A more complete overview can be found in the Agda release notes [1] and Van der Walt's thesis [23].

The central type in the reflection mechanism is a type `Term : Set` that 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

[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.

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 labelled 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
  Base : Even 0
  Step : ∀ {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+ Base       e2 = e2
even+ (Step e1) e2 = Step (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 [16] and Idris [3].

As shown by Van der Walt and Swierstra [22], 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 (Step Base)
```

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*, containing a collection of 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. The resulting proof is robust against reformulation and refactoring. 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 such a tactic similar to auto in Agda.

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

```
hints : HintDB
hints = hintdb
  (quote Base :: quote Step :: 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 for the moment. 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. Prolog in Agda

Let us set aside Agda's reflection mechanism for the moment. In this section, we will present a standalone Prolog interpreter. Subsequently, we will show how this can be combined with the reflection mechanism and suitably invoked in the definition of the

auto function. The code in this section is contained in its own Agda module, parameterized by two sets:

```
module Prolog
  (TermName : Set) (RuleName : Set) where
```

Terms and Rules

The heart of our proof search implementation is the structurally recursive unification algorithm described by McBride [12]. Here

the type of terms is indexed by the number of variables a given term may contain. Doing so enables the unification algorithm to formulated by structural induction on the number of free variables. This yields the following definition of terms:

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

In addition to variables, we will encode first-order constants as a TermName with a list of arguments.

For instance, if we choose to instantiate the 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 : PrologTerm 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 : PrologTerm 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 precisily 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 ourself to presenting the interface that we will use:

```
unify : (t₁ t₂ : PrologTerm m) → Maybe (∃ (Subst m))
```

Substitutions are indexed by two natural numbers n and m. A substitution of type Subst m n can be applied to a PrologTerm m to produce a value of type PrologTerm n. The unify function takes two terms t_1 and t_2 and tries to compute the most general unifier. As unification may fail, the result is wrapped in the Maybe monad. The number of variables in the terms resulting from the unifying substition is not known *a priori*, hence this number is existentially quantified over.

This unification function is defined using an accumulating parameter, representing an approximation of the final substitution. In

what follows, we will use the following, more general, function:

```
unifyAcc : (t₁ t₂ : PrologTerm m) →
           ∃ (Subst m) → Maybe (∃ (Subst m))
```

Next we define Prolog rules as records containing a name and terms for its premises and conclusion:

```
record Rule (n : ℕ) : Set where
  field
    name       : RuleName
    conclusion : PrologTerm n
    premises   : List (PrologTerm n)
```


Again the data type is quantified over the number of variables used by its constituents. Note that variables are shared between the premises and conclusion.

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

```
add(0, x, x).
add(x, y, z) :- add(suc(x), y, suc(z)).
```

Unfortunately, the named equivalents in our Agda implementation 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.

```
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)
                       :: [])
               :: []
  }
```

Lastly, before we can implement some form of proof search, we define a pair of auxiliary functions. During proof resolution, we will need to 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)
```

We have tried to visualize the behaviour of inject and raise, embedding Fin 3 into Fin (3 + 1) in Figure 1. 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 these inject and raise to define similar functions that work on our Rule and Term data types, by mapping them over all the variables that they contain.

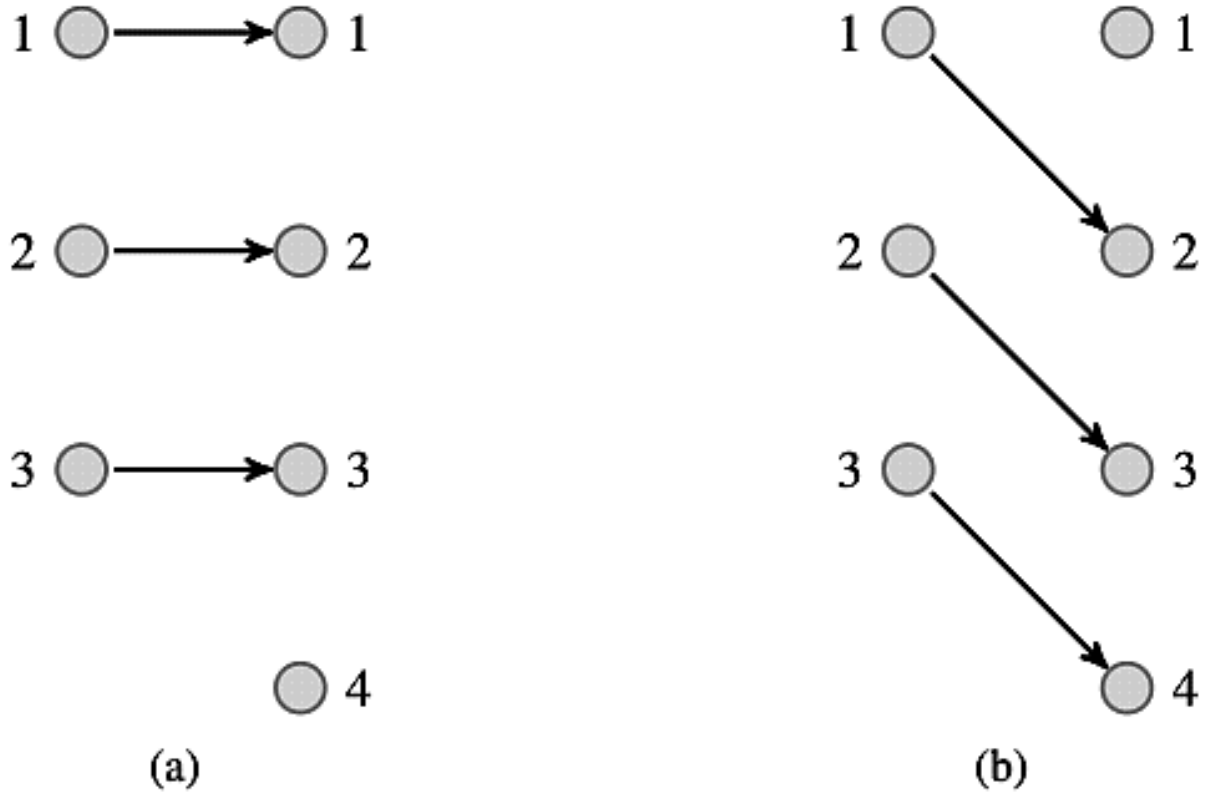

Figure 1. The graph of the inject function (a) and the raise function (b) embedding Fin 3 in Fin $(3 + 1)$

Proof search

Our implementation of proof search is split into two steps. In the first step we set up an higher-order representation of the search space, where we branch over some collection of undetermined rules at every step. In the second step we flatten this abstract representation to a first-order search tree.

The distinction between these two phases keeps the nitty gritty details involved with unification and weakening used in the first phase separate from the actual proof search. By doing so, we can implement various search strategies, such as breadth-first search, depth-first search or an heuristic-driven algorithm, by simply traversing the final search tree in a different order.

Setting up the search space

We start by defining the following type synonym to distinguish goals from regular Prolog terms:

```
Goal : ℕ → Set
Goal n = Term n
```

Next we define the data type that we will use to model the abstract search space.

```
data SearchSpace (m : ℕ) : Set where
  fail : SearchSpace m
  retn : Subst (m + δ) n → SearchSpace m
  step : (∃ Rule → ∞ (SearchSpace m))
       → SearchSpace m
```

Ignoring the indices for the moment, the SearchSpace type has three constructors: fail, retn and step. In the case of retn, we have found a substitution that resolves the goal we are trying to prove. In the step constructor, we have not yet resolved the goal, and instead have a choice of which Rule to apply. Note that we do not specify *which* rules may be used; only how the choice of *any* rule determines the remainder of the search. As a search need not terminate, the SearchSpace resulting from applying a rule are marked as coinductive. The fail constructor is used to mark branches of the search space that fail, i.e., where the selected rule is not unifiable with the current goal.

Note that we rename Agda's notation for coinduction to more closely resemble notation already familiar to Haskell programmers. Coinductive suspensions are created with the prefix operator $\sim$ rather than $\sharp$; such suspensions can be forced using a bang, !, rather than the usual $\flat$.

Now let us turn our attention to the indices. The variable m denotes the number of variables in the goal; δ denotes the number of fresh variables necessary to apply a rule; and n will denote the number of variables remaining after we have resolved the goal. This naming will be used consistently in subsequent definitions.

We can now define a function resolve that will be in charge of building up a value of type SearchSpace from an initial goal:

```
resolve : Goal m → SearchSpace m
resolve {m} g = resolveAcc (just (m, nil)) [g]
```

The resolve function is once again defined by calling an auxiliary function defined using an accumulating parameter. It starts with an empty substitution and a list of goals that only contains the initial goal g. The resolveAcc function will attempt to resolve a list of sub-goals, accumulating a substitution along the way:

```
resolveAcc : ∀ {m δ : ℕ}
  → Maybe (∃ (λ n → Subst (m + δ) n))
  → List (Goal (m + δ)) → SearchSpace m
resolveAcc (just (n, subst)) []                = retn subst
resolveAcc nothing _                           = fail
resolveAcc (just (n, subst)) (goal :: goals)   = step next
```

If we have no remaining goals, we can use the retn constructor to return the substitution we have accumulated so far. If at any point, however, the conclusion of the chosen rule was not unifiable with the next open subgoal—and thus the accumulating parameter has become nothing—the search will fail. The interesting case is the third one. If there are remaining goals to resolve, we recursively

construct a new SearchSpace. To do so, we use the step constructor and branch over the choice of rule. The next function defined below computes the remainder of the SearchSpace after trying to apply a given rule:

```
next : ∃ Rule → ∞ (SearchSpace m)
next (δ', rule) =
  ~ resolveAcc mgu (prems' ++ goals')
  where
    mgu : Maybe (∃ (λ n → Subst (m + (δ + δ')) n))
    mgu = unifyAcc goal' concl' subst'
      where
        goal' : PrologTerm (m + (δ + δ'))
        goal' = injectTerm δ' goal

        subst' : ∃ (Subst (m + (δ + δ')))
        subst' = n + δ', injectSubst δ' subst

        concl' : PrologTerm (m + (δ + δ'))
        concl' = raiseTerm (m + δ) (conclusion rule)

    goals' : List (PrologTerm (m + (δ + δ')))
    goals' = injectTermList δ' goals

    prems' : List (PrologTerm (m + (δ + δ')))
    prems' = raiseTermList (m + δ) (premises rule)
```

For the moment, try to ignore the various calls to raise and inject. Given the rule that must be applied, the next function computes most general unifier of the conclusion of rule and our current goal. The resulting substitution is passed to resolveAcc, which continues the construction of the SearchSpace. The premises of the rule are added to the list of open goals that must be resolved. The apparent complexity of the next function comes from the careful treatment of variables.

First of all, note that we pass the substitution accumulated so far to unifyAcc. This ensures that the constraints on any variables occurring in the two terms being unified are taken into account.

Next, there is the problem of avoiding variable capture. We can only unify two terms that have the same type. Therefore we must

ensure that the goal, the rule's conclusion and its premises have the same number of variables. At the same time, the substitution we are accumulating should be kept in synch with the variables used in the initial goal. Furthermore, the variables mentioned in the rule are implicitly universally quantified. We need to instantiate them with fresh variables to avoid introducing unintended constraints. This is where inject and raise come in.

Recall that injecting a variable into a larger set would keep its value the same, whereas raise maps the variable into a 'fresh' portion of the set that was previously unused. Therefore we will always take care to inject our goal terms and our accumulating substitution, whereas we raise the terms in the applied rule. This ensures that the substitution and goals are kept in synch, whereas any variables mentioned in the rule are fresh.

Note the number of free variables in the chosen rule, δ_2, is added to the amount of space that had to be made for previous rule applications, δ_1. As a result, we need to raise by more and more as the proof search proceeds.

Constructing search trees

The second step in our proof search implementation is to transform the SearchSpace we have just constructed into a first-order rose tree. We do this by branching once for every rule at every step constructor. The result of this transformation shall be expressed in terms of the following data type.

```
data SearchTree (A : Set) : Set where
  fail : SearchTree A
  retn : A → SearchTree A
  fork : List (∞ (SearchTree A)) → SearchTree A
```

Note that this SearchTree is finitely branching, but potentially infinitely deep. At every fork we may branch over some finite set of rules, but there is no guarantee that we can construct the entire SearchTree in finite time.

In our case, we will instantiate the type variable A with a tuple containing a substitution together with a trace that keeps track of all the applied rules. In order to keep the code readable, let us introduce the following alias.[3]

```
Result m = ∃₂ (λ δ n → Subst (m + δ) n) × Rules
```

The existential quantifier $\exists_2$ hides both the number of fresh variables that we need to introduce, δ, and the number of variables in the terms produced by the final substitution, n.

The function that takes care of the transformation is almost trivial. For a given set of rules, we simply traverse the SearchSpace structure, where at every step we apply the continuation to every rule. Since we also wish to maintain a trace of the rules that have been applied, we shall define this transformation using an auxiliary function with an accumulating parameter:

```
mkTree : Rules → SearchSpace m
       → SearchTree (Result m)
mkTree rs₀ s = go s []
  where
  go : SearchSpace m → Rules → SearchTree (Result m)
  go fail     _   = fail
  go (retn s) acc = retn ((_, (_, s)), acc)
  go (step f) acc =
```

$$\mathsf{fork}\ (\mathsf{map}\ (\lambda\ \mathsf{r} \to \sim\ \mathsf{go}\ (!\ \mathsf{f}\ \mathsf{r})\ (\mathsf{acc} ::^{\mathsf{r}} \mathsf{r}))\ \mathsf{rs}_0)$$

Note that we accumulate the trace of rules applied in the order in which they are applied: new rules are added to the end of the list with the snoc operator $::^{\mathsf{r}}$.

[3] Rules is an alias for a list of existentially quantified rules List (∃ Rule).

In the implementation of mkTree, Agda's guardedness checker cannot tell that the call to map is size-preserving, and therefore safe. To show this definition is suitably guarded, we need to inline the definition of map and explicitly recurse over the list of rules rs_0.

After the transformation, we are left with a first-order tree structure, that we can traverse in search of solutions. For example, we can define a simple bounded depth-first traversal as follows:

```
dfs : (depth : ℕ) → SearchTree A → List A
dfs zero     _          = []
dfs (suc k) fail        = []
dfs (suc k) (retn x)    = return x
dfs (suc k) (fork 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 can also vary the rules used to construct the SearchTree. For example, you may want to define a function that constructs a 'linear' proof, where every rule is applied at most once. All these search strategies are simple variations of the solution presented here.

Putting all these pieces together, we can define a function searchToDepth, which implements proof search up to a given

depth d, i.e. it constructs the SearchSpace, flattens this to a SearchTree, and finally traverses the resulting tree in depth-first order up to depth d.

```
searchToDepth : ℕ → Rules → Goal m → List (Result m)
searchToDepth depth rules goal =
  dfs depth (mkTree rules (resolve goal))
```

Example

Using this implementation of proof search, together with the terms and rules defined above, we can compute, for instance, the sum $3 + 1$. First we define a query, corresponding to the Prolog query `add(3,1,x)`:

```
query : Term 1
query =
  con Add (inject 1 Three :: inject 1 One :: var (# 0) :: [])
```

Note that we must inject the terms Three and One, which are closed terms, in order to make it match the variable domain of our variable var (# 0).

Second, we use searchToDepth to search for a substitution. We use a function apply which applies a list of solutions to a goal term:

```
apply : List (Result m) → Goal m → List (Term 0)
```

Since we do not wish to go into the details of unification and substitution, we shall leave this function undefined. Instead we will present a complete usage of searchToDepth, resolving the previously defined query:

```
result : List (Term 0)
```

```
result = apply substs (var (# 0))
  where
    rules  = (1, AddBase) :: (3, AddStep) :: []
    substs = searchToDepth 5 rules query
```

Once we have this, we can show that the result of adding 1 and 3 is indeed 4.

```
test : result ≡ (Four :: [])
test = refl
```


4. Constructing proof trees

The Prolog interpreter described in the previous section returns a substitution. To use such an interpreter to produced proof terms, however, we need to do a bit more work.

Besides the resulting substitution, the Result type returned by the proof search process also contains a a trace of the applied rules. In the following section we will discuss how to use this information to reconstruct a proof term. That is, we will construct a closed term of the following type:

```
data ProofTerm : Set where
  con : RuleName → List ProofTerm → ProofTerm
```

It is easy to compute the arity of every rule: we simply take the length of the list of premises. After making this observation, we can define a function to construct such a ProofTerm as a simple fold:

```
toProofTerms : Rules → List ProofTerm
toProofTerms = foldr next []
  where
```

```
next : ∃ Rule → List ProofTerm → List ProofTerm
next (δ, r) pfs with arity r ⩽? length pfs
... | no  r>p = []   -- should not occur
... | yes r⩽p =
  con (name r) (take (arity r) pfs) :: drop (arity r) pfs
```

The next function combines a list of proof terms, produced by recursive calls, and the single rule r that has just been applied. If the list contains enough elements, we construct a new ProofTerm node by applying the rule to the first arity r elements of the list. This new ProofTerm is the head of the list, replacing the children terms that previously formed the prefix of the list. Essentially, this is the 'unflattening' of a rose tree using the the arities of the individual nodes. Upon completion, toProofTerms should return a list with a single element: the proof term that witnesses the validity of the our derivation. The function, toProofTerm, returns this witness if it exists:

```
toProofTerm : Rules → Maybe ProofTerm
toProofTerm rs with toProofTerms rs
... | []          = nothing
... | p :: []     = just p
... | p :: _ :: _ = nothing
```

Of course, the toProofTerms function may fail if there are not enough elements in the list to fully apply a rule. When run on the result of our proof search functions, such as searchToDepth, however, we know that the list has the right length, even if this is not enforced by its type. While we could use a clever choice of indexed data type to show that the toProofTerms can be defined in a *total* fashion, there is little benefit in doing so. The proof search functions such as searchToDepth are already *partial* by their very nature. Adding further structure to the accumulated list of rules to guarantee totality will not change this.

5. Adding reflection

To complete the definition of our auto function, we still need to convert between Agda's built-in Term data type and the data type required by our unification and resolution algorithms, PrologTerm. This is an essential piece of plumbing, necessary to provide the desired proof automation. While not difficult in principle, this does expose some of the limitations and design choices of the auto function.

The first thing we will need are concrete definitions for the TermName and RuleName data types, two were parameters to the development presented in the previous sections. It would be desirable to identify both types with Agda's Name type, but unfortu-

nately the Agda does not assign a name to the function space type operator, $\_\rightarrow\_$; 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 as follows:

```
data TermName : Set where
  pname : (n : Name) → TermName
  pvar  : (i : ℕ) → TermName
  pimpl : TermName
```

The TermName data type has three constructors. The pname constructor embeds Agda's built-in Name in the a TermName type. The pvar constructor describes locally bound variables, represent by their De Bruijn index. Note that the pvar constructor has nothing to do with PrologTerm'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, pimpl explicitly represents the Agda function space.

We define the RuleName type in a similar fashion:

```
data RuleName : Set where
  rname : (n : Name) → RuleName
  rvar  : (i : ℕ) → RuleName
```

The rvar constructor is used to refer to Agda variables as rules. Its argument i is corresponds to a 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 Agda Term to PrologTerm may also fail for various reasons. To handle such errors, we will work in the Error monad defined below:

```
Error : (A : Set) → Set a
Error A = Either 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
  indexOutOfBounds     : Message
  unsupportedSyntax    : Message
  panic!               : Message
```

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

Finally, we wil 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) variables—where m ⊔ n denotes the maximum of m and n:

$$\mathsf{match} \;:\; \mathsf{Fin}\ \mathsf{m} \to \mathsf{Fin}\ \mathsf{n} \to \mathsf{Fin}\ (\mathsf{m} \sqcup \mathsf{n}) \times \mathsf{Fin}\ (\mathsf{m} \sqcup \mathsf{n})$$

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, matchTerms, or a term and a lists of terms, matchTermAndList.

Constructing terms

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

First of all, the Agda Term type represents all (possibly higher-order) terms, whereas the PrologTerm type is necessarily first-order. We mitigate this problem, by allowing the conversion to fail, throwing an 'exception' with the message unsupportedSyntax.

Secondly, the Agda Term data type uses natural numbers to represent variables. The PrologTerm data type, on the other hand, represents variables using a finite type Fin n, for some n. To convert between these representations, we could compute the number of free variables in a Term, and use this information to map between the two different representations of bound variables. To keep matters simple, however, we allow the conversion to fail with an indexOutOfBounds message, even though this should never occur. While we could do more work to prove totality of the variable conversion, we are already defining a function that could fail. Totality of the variable conversion will still not make our conversion

total.

The conversion function, fromTerm, traverses the argument term, keeping track of the number of Π-types it has encountered. We sketch its definition below:

```
fromTerm : ℕ → Term → Error (∃ PrologTerm)
fromTerm d (var i [])      = fromVar d i
fromTerm d (con c args)  = fromDef c ⟨$⟩ fromArgs d args
fromTerm d (def f args)   = fromDef f ⟨$⟩ fromArgs d args
fromTerm d (pi (arg visible _ (el _ t₁)) (el _ t₂))
  with fromTerm d t₁ | fromTerm (suc d) t₂
... | left msg       | _          = left msg
... | _              | left msg   = left msg
... | right (n₁, p₁) | right (n₂, p₂)
  with matchTerms p₁ p₂
... | (p₁', p₂') = let term = con pimpl (p₁' :: p₂' :: [])
                   in right (n₁ ⊔ n₂, term)
fromTerm d (pi (arg _ _ _) (el _ t₂))
   = fromTerm (suc d) t₂
fromTerm _ _ = left unsupportedSyntax
```

We define special functions, fromVar and fromDef, to convert variables and constructors or defined terms respectively. The arguments to constructors or defined terms are processed using the fromArgs function defined below. The conversion of a pi node binding an explicit argument proceeds by converting the domain and codomain. If both conversions succeed, the resulting terms are matched and a PrologTerm is constructed using pimpl. 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 fromArgs function converts a list of Term arguments to a list of Prolog terms, by stripping the arg constructor and recursively

applying the fromTerm function. We only give its type signature here, as the definition is straightforward:

```
fromArgs : ℕ → List (Arg Term)
         → Error (∃ (List ∘ PrologTerm))
```

Next, the fromDef function constructs a first-order constant from an Agda Name and list of terms:

```
fromDef : Name → ∃ (λ n → List (PrologTerm n))
        → ∃ PrologTerm
fromDef f (n, ts) = n, con (pname f) ts
```

Lastly, the fromVar function converts a natural number, corresponding to a variable name in the Agda Term type, to the corresponding PrologTerm by taking the difference between the number of binders traversed and the De Bruijn index:

```
fromVar : ℕ → ℕ → Error (∃ PrologTerm)
fromVar n i with compare n i
fromVar n ⌊suc (n + k)⌋ | less    ⌊_⌋ k
  = left indexOutOfBounds
fromVar n ⌊n⌋           | equal   ⌊_⌋
```



```
  = right (suc 0, var (# 0))
fromVar ⌊suc (i + k)⌋ i | greater ⌊_⌋ k
  = right (suc k, var (# k))
```

To convert between an Agda Term and PrologTerm we simply call the fromTerm function, initializing the number of binders encountered to 0:

```
toPrologTerm : Term → Error (∃ PrologTerm)
toPrologTerm = fromTerm 0
```

Constructing rules

Our next goal is to construct rules. More specifically, we need to convert a list of quoted Names to a hint databases 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(Plus(m,n)) :- Even(m), Even(n).
```

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

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

```
fromName : Name → Error (∃ PrologTerm)
fromName = toPrologTerm ∘ unel ∘ type
```

The type construct converts a Name to the Agda Term representing its type; the unel function discards any information about sorts; the toPrologTerm was defined previously.

In the next step, we process this PrologTerm. The splitTerm function splits a PrologTerm at every top-most occurrence of the function symbol pimpl. Note that it would be possible to define this function directly on Agda's Term data type, but defining it on the PrologTerm data type is much cleaner as all unsupported syntax has been removed.

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

Using all these auxiliary functions, it is straightforward to construct the desired rule.

```
toRule : Name → Error (∃ Rule)
toRule name with fromName name
... | left msg          = left msg
... | right (n, t)      with splitTerm t
```



```
... | (k, ts)           with initLast ts
... | (prems, concl, _) =
```

```
    right (n, rule (rname name) concl (toList prems))
```

We convert a name to its corresponding PrologTerm, which is split into a vector of terms using splitTerm. The last element of this vector is the conclusion of the rule; the initial prefix constitutes the premises.

Constructing goals

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

Consider the example given in Section 2. The goal Term we wish to prove is Even n → Even (n + 2). Calling toPrologTerm would convert this to a PrologTerm, where the function space has been replaced by the pimpl. What we would like to do, however, is to introduce as any available assumptions, such as Even n, and try to resolve the remaining goal Even (n + 2).

Fortunately, we can reuse many of the auxiliary functions we have defined already to achieve this. We convert a Term to the corresponding PrologTerm. Using the splitTerm and initLast function, we can get our hands on the list of arguments args and the desired return type goal.

```
toGoalRules : Term → Error (∃ PrologTerm × Rules)
toGoalRules t with fromTerm' 0 t
... | left msg          = left msg
... | right (n, p)      with splitTerm p
... | (k, ts)           with initLast ts
... | (args, goal, _)   = let rs = toRules 0 args
                          in right ((n, goal), rs)
```

The only missing piece of the puzzle is a function, toRules, that converts a list of PrologTerms to a Rules list.

```
toRules : ℕ → Vec (PrologTerm n) k → Rules
toRules i []       = []
toRules i (t :: ts) = (n, rule (rvar i) t [])
                      :: toRules (suc i) ts
```

The toRules converts every PrologTerm in its argument list to a rule, generating a fresh variable for each parameter.

There is one last technical point. In the previous version of fromTerm, an Agda Term variable was mapped to a Prolog variable. When considering the goal type, however, we want to generate skolem constants for our variables, rather than Prolog variables which may still be unified. To account for this difference, we use the fromTerm' function, a slight variation of the fromTerm function described previously.

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 Agda Term. The fromProof function does exactly that:

```
fromProof : ProofTerm → Term
fromProof (con (rvar i) ps) = var i []
fromProof (con (rname n) ps) with definition n
... | function _    = def n ∘ toArg ∘ fromProof ⟨$⟩ ps
... | constructor′  = con n ∘ toArg ∘ fromProof ⟨$⟩ ps
... | _             = unknown
  where
    toArg = arg visible relevant
```

Any bound variables, corresponding to usage of the local premises, can be mapped to the var constructor the Agda Term data type. As we know by construction that these correspond to rules without premises, these variables do not need any further arguments.

If the rule being applied is constructed using an rname, we do disambiguate whether the rule name refers to a function or a constructor. The definition function, defined in Agda's reflection library, returns information about how the piece of abstract syntax to which its argument Name corresponds. For the moment, we restrict this definition 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 an additional lambda around all the premises that were introduced by the toGoalRules function. To do so, we define the intros function that repeatedly wraps its argument term in a lambda:

```
intros : ℕ → Term → Term
intros zero t = t
intros (suc k) t = lam visible (intros k t)
```

Hint databases

We allow users to provide hints, rules that may be used during resolution, in the form of a *hint database*. Such a hint database is simply a list of Prolog rules:

```
HintDB : Set
HintDB = List (∃ Rule)
```

We can 'assemble' hint databases from a list of names using the function hintdb:

```
hintdb : List Name → HintDB
```

```
hintdb = concatMap (fromError ∘ toRule)
  where
    fromError : Error A → List A
    fromError = fromEither (const []) [_]
```

Note that if the generation of a rule fails for whatever reason, no error is raised, and the rule is simply ignored. This behaviour is easily adapted.

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. The only function that would need to be adapted to handle such requirements is the mkTree function in Section 3.

Error messages

Lastly, we need to decide how to report error messages. Since we are going to return an Agda Term, we need to transform the Message type we saw previously into an Agda Term. When unquoted, this term will cause an 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 Agda Term 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 (indexOutOfBounds)
```



```
    = quoteTerm (throw indexOutOfBounds)
quoteError (unsupportedSyntax)
    = quoteTerm (throw unsupportedSyntax)
quoteError (panic!)
    = quoteTerm (throw panic!)
```

Putting it all together

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

```
auto : (depth : ℕ) → HintDB → Term → Term
auto depth hints goalType
  with toGoal goalType
... | left msg  = quoteError msg
... | right ((n, goal), args)
  with searchToDepth depth (args ++ hints) goal
... | []          = quoteError searchSpaceExhausted
... | (_, trace) :: _
  with toProofTerm trace
... | nothing  = quoteError panic!
... | just p   = intros (fromProof p)
```

The auto function converts the Term to a PrologTerm, the return type of the goal, and a list of arguments that may be used to construct this term. It then proceeds by calling the searchToDepth function with the argument hint database. If this proof search succeeds, the Result is converted to an Agda Term, a witness that the original goal is inhabited. There are three places that this function

may fail: the conversion to a PrologTerm may fail, for instance because of unsupported syntax; the proof search may not find any result; or the final conversion to an Agda Term may fail unexpectedly. This last case should never be triggered, provided the toProofTerm function is only called on the result of our proof search.

6. 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 [19] 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 records:

```
ShowBool : Show Bool
ShowBool = record {show = showBool}

Showℕ : Show ℕ
Showℕ = record {show = showℕ}
```

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 $\mathbb{N}$ is required. As there is only a single value of type Show $\mathbb{N}$,

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 parameterised 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 (Inl x) = "Inl " ++ show x
    showE (Inr y) = "Inr " ++ show y
```

Unfortunately, instance arguments do not do any recursive search for suitable instances. Trying to call show on a value of type Either $\mathbb{N}$ 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 :: [])
```

Now we can call our proof search to assemble the instances for us:

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

The type of the locally bound instance record is inferred; the proof search manages to assemble the desired dictionary.

7. 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 or sharing 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 help speed up the proof search substantially.

Language The auto function can only handle first-order terms. Even if higher-order unification is undecidable in general, we believe we should be able to adapt our algorithm to work on second-order functions. Furthermore, there are plenty of Agda features that are not supported by our quotation or Agda's reflection mechanism, such as universe polymorphism, instance arguments, and primitive

functions. Even in the presence of simple dependent types, our resolution function can produce surprising results. Consider the following example, defining a show function on dependent pairs:

```
data _×_ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → A × B

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

Here we define a type for *dependent* pairs, but only use the degenerate, simply typed case. Although our proof search can construct the required dictionary, using the show function results in various unresolved metavariables. We suspect that this is because Agda cannot figure out how to instantiate the second argument of the dependent pair. We suspect this is a limitation of the reflection mechanism. **Wouter:** Pepijn: is dit opgelost in de HEAD?

Refinement and Recursion 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.

Another consequence of this restriction is that we cannot use induction hypotheses as hints. **Wouter:** Why is this exactly? Do we have a good story here?

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, 13]. 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, to do prove 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 represent a valid derivation.

Related work

There are several existing alternatives

Coq and Ltac

Mtac

Idris

Closure

Having said all of this, we have good reasons to believe the approach to proof automation described in this paper is interesting and worth exploring further. Unlike Coq, we do not need a custom language of proof tactics. We can debug and test our proof search mechanism just as easily as we debug any other Agda function. It is straightforward to record a log of all the rules that have been attempted, for example, which is invaluable information when trying to debug proof automation. It is easy to write variations of the proof

search resolution mechanism. We have first-class hint databases that can be assembled modularly, inspected by other functions, or even modified during proof search. This is super useful: consider the problem of having `trans` in a hint database.

Using the techniques described in this paper, it is possible to write many other pieces of proof automation. Automated rewriting, for example. Or a high-level, first-class tactic language: try this piece of automation, and if that fails try something else.

This is the way forward for proof automation.

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. ISSN 1469-7653. doi: 10.1017/S095679681300018X. URL `http://journals.cambridge.org/article_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 meta-programming. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP 2013)*. ACM, September 2013. doi: 10.1145/2500365.2500575. URL `https://lirias.kuleuven.be/handle/123456789/404549`.

[12] Conor McBride. First-order unification by structural recursion. *Jour-*

nal of Functional Programming, 13:1061–1075, 11 2003. ISSN 1469-7653. doi: 10.1017/S0956796803004957. URL `http://journals.cambridge.org/article_S0956796803004957`.

[13] 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. ISBN 978-1-4503-0251-7. doi: 10.1145/1863495.1863497. URL `http://doi.acm.org/10.1145/1863495.1863497`.

[14] 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.

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

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

2014/2/24

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

[18] 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.

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

[20] 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.

[21] 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. URL `http://doi.acm.org/10.1145/258993.259019`.

[22] 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. URL `http://dx.doi.org/10.1007/978-3-642-41582-1_10`.

[23] 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`.