Experimentation Project: Exploring Metaprogramming in Agda

Carlos Tomé Cortiñas
Utrecht University

September 24, 2017

1 Introduction

In this experimentation project we aim to investigate the possibilities offered by the recently introduced metaprogramming facilities in Agda (since version 2.5.2) for automatic program construction. The present project is built upon the previous work AutoInAgda, by Kokke and Swiestra [3].

The document is organized as follows. In section 2 we give a small overview of the original AutoInAgda library, which is used as starting point for this work. Consequently, in subsection 2.1, we explain the improvements made on top of it.

In section 3, we explain the original motivation for the project and in subsection 3.1 we proceed with a throughly explanation of the work we have implemented.

In section 4 we present some the benchmarking results of the library implemented against the original AutoInAgda library implementation.

Finally, section 5 concludes this report with a discussion about the pros and cons of our approach and hint some directions for a possible future line of work.

Accompanying this report, a working implementation of the work explained here can be found in the GitHub repository:

https://github.com/carlostome/AutoInAgda-ECOSC

2 AutoInAgda

AutoInAgda is a library for automatic construction of Agda programs. Given a function definition whose type signature is complete but a hole stands for the body, the library provides a function `auto` that can be used in the place of the hole.

If `auto` is able to find an Agda term of the required type, then the Agda file will typecheck as if the term was manually entered. Otherwise, a compile time type error will be raised and the programmer will be left with the responsibility of filling the goal.

In the rest of this document we will work with the following definitions in order to explain the different concepts we are concerned.

```
data Even : ℕ → Set where
  isEven0 : Even 0
  isEven+2 : ∀ {n} → Even n → Even (suc (suc n))
  even+ : ∀ {n m} → Even n → Even m → Even (n + m)
  even+ isEven0 e2 = e2
  even+ (isEven+2 e1) e2 = isEven+2 (even+ e1 e2)
```

Figure 1: Agda code setup

As a first example, in the following function definition, we use auto instead of manually entering a term of the correct type. The program successfully typechecks which means the library is able to find a suitable term of the type Even n → Even (n + 2).

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

Figure 2: Code example

It is convenient that we are able to parametrize auto both by the depth it will look for the solution in the search tree and the database of rules it will use to prove the goal.

Under the hood, the procedure that AutoInAgda uses to solve the goal is as follows. First, the type of the hole (the type of the Agda term the hole stands for) is reified into a datatype (included in the reflection API) that represents Agda terms (Because of the dependently typed nature of Agda, there is no difference between terms at the term level and terms in the type level). The Agda term representing the type is then converted into to an internal first-order representation. Then a proof search procedure 'a la Prolog is executed in order to find a term satisfying the type. Finally, in case of success the resulting term is reflected back into an Agda term and 'replaced' (this happens internally with the reflection mechanism) in the hole.

The process we just described is depicted in figure 3.

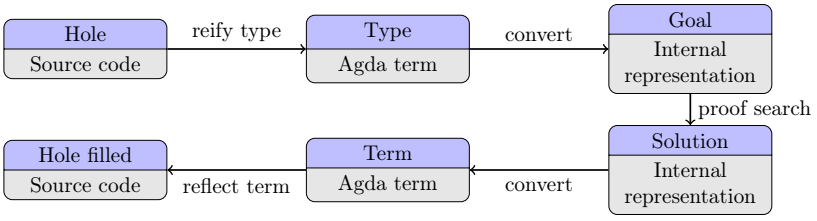

Figure 3: How AutoInAgda works

2.1 Improving vanilla AutoInAgda

The first part of this experimentation project consisted on getting acquainted with the library and update it to work with the latest version of the metaprogramming API.

Moreover, this project also addresses some issues regarding the usability of the library from an engineering point of view. In the following listing, we explain some of the main issues.

- Inspecting the generated term In its form, AutoInAgda automatically generated term is directly plugged into the hole. Transformed from the first-order term representation the library uses internally to the Agda terms the library does not provide a means of actively inspecting the term.
- Proof search debugging Proof search of terms that satisfy the type is done in a all-or-nothing fashion. Either the search does yield one or more terms of the required type or it does not. When this latter case happens it is not possible to recover the proof search space tree to understand where Agda gets stuck.
- Holes in proofs Besides searching for a term with the given hint database, sometimes the programmer would like to introduce further local hints to help with the proof search procedure. Such thing is just not possible in AutoInAgda.

As a poor man's technology for user-computer interaction, we choose to interact with the user through the *emacs* type error buffer included in the **agda-mode**.

The following program prints the automatically generated term as found by AutoInAgda. It is important to remark, that the use of the type error mechanism to interact with the user prevents the file from typechecking until print is changed for apply (this tactic replaces the original tactic function).

```
test : ∀ {n} → Even n → Even (4 + n)
test = print (auto 6 rules)
```

```
Success: The Term found by auto is:
λ z → even+ (isEven+2 (isEven+2 isEven0)) z
```

Figure 4: Example program and its output to the error buffer

Another improvement made to the original library is allow the user to print the proof search tree as has been explored during proof search. This can be displayed by using the tactic info. The display shows how the rules in the hint database were applied (by name) and whether they were successfully applied.

Furthermore, the printed tree shows exactly the trace of how the proof search tree was explored, thus depending on the strategy chosen it might be different.

Continuing with the previous example we can print the proof search tree that lead to that term.

Our approach emulates to some extent the same characteristic that it is available to Coq users and is extensively explained by Chlipala in [1].

The real power of the info tactic comes into play when AutoInAgda is not able to find a term to fill in the goal. From the following example, it is clear that such a term cannot be constructed but by visualizing the proof search tree, one can understand that at one point (when trying to fulfill Even 1) no more rules can be applied.

On the other hand, if the problem is that the given depth is not enough to construct the term we will see that at the given depth there are some applied

```
test : ∀ {n} → Even n → Even (4 + n)
test p = info (auto 6 rules)
```

```
Success Solution found. The trace generated is:
1.1 depth=6 isEven0 ×
1.2 depth=6 isEven+2 ×
1.3 depth=6 even+ ✓
1.3 .1 depth=5 isEven0 ×
1.3 .2 depth=5 isEven+2 ✓
1.3 .2.1 depth=4 isEven0 ×
1.3 .2.2 depth=4 isEven+2 ✓
1.3 .2.2.1 depth=3 isEven0 ✓
1.3 .2.2.1.1 depth=2 isEven0 ×
1.3 .2.2.1.2 depth=2 isEven+2 ×
1.3 .2.2.1.3 depth=2 even+ ×
1.3 .2.2.1.4 depth=2 var 0 ✓
1.3 .2.2.1.5 depth=2 var 1 ×
1.3 .2.2.2 depth=3 isEven+2 ×
1.3 .2.2.3 depth=3 even+ ×
1.3 .2.2.4 depth=3 var 0 ×
1.3 .2.2.5 depth=3 var 1 ×
1.3 .2.3 depth=4 even+ ×
1.3 .2.4 depth=4 var 0 ×
1.3 .2.5 depth=4 var 1 ×
1.3 .3 depth=5 even+ ×
1.3 .4 depth=5 var 0 ×
1.3 .5 depth=5 var 1 ×
1.4 depth=6 var 0 ×
1.5 depth=6 var 1 ×
```

Figure 5: Example program and its output to the error buffer

```
test : Even 5
test = info (auto 5 rules)
```

```
Error: Solution can't be found. The trace generated is:
1.1 depth=6 isEven0 ×
1.2 depth=6 isEven+2 ✓
1.2 .1 depth=5 isEven0 ×
1.2 .2 depth=5 isEven+2 ✓
1.2 .2.1 depth=4 isEven0 ×
1.2 .2.2 depth=4 isEven+2 ×
1.2 .2.3 depth=4 even+ ×
1.2 .3 depth=5 even+ ×
1.3 depth=6 even+ ×
```

Figure 6: Example program and its output to the error buffer

rules that succeeded (✓) but where not further explored. For example, in figure 7 the rule generated by the argument `Even n` can be applied although the depth limit did not allow for further exploration.

The original library required the type of the goal to be of the form `A → B → C` and could not make use of any local variables introduced in the left hand side of the equals symbol.

We have improved this, with the added benefit that now with little effort we can make the proof search aware of local variables and variables defined within a **with** clause. As a silly example of this, consider the program in figure 8. The program is able to typecheck but the user is left with filling the remaining hole.

3 The project

The starting objective of this project was to exploit the new metaprogramming API that Agda offers to try to improve AutoInAgda both in terms of perfor-

```
test : ∀ {n} → Even n → Even (n + 16)
test = apply (auto 2 rules)
```

```
Error: Solution can't be found. The trace generated is:
1.1 depth=2 isEven0 ×
1.2 depth=2 isEven+2 ×
1.3 depth=2 even+ ✓
1.3 .1 depth=1 isEven0 ×
1.3 .2 depth=1 isEven+2 ×
1.3 .3 depth=1 even+ ×
1.3 .4 depth=1 var 0 ✓
1.3 .5 depth=1 var 1 ×
1.4 depth=2 var 0 ×
1.5 depth=2 var 1 ×
```

Figure 7: Example program and its output to the error buffer

```
_∋_ : (A : Set) → A → A
_ ∋ x = x
test : ∀ {n} → Even n → Even (4 + n)
test p with Even 0 ∋ {!!}
... | _ = apply (auto 5 (ϵ « isEven+2))
```

Figure 8: Example program with locally bound variables

mance and expressiveness.

In order to do so, the first step is to understand where advantage can be taken. From the figure in 9 it can be understood that there exists a great deal of overhead due to the translation of the Agda term representing the type of a hole into the internal first-order representation that the library uses.

Moreover, the unification function that the library uses, based on the work by McBride [4], can be replaced by the unify : Term → Term → TC () function that Agda provides in a primitive manner.

Because unify reflects the internal function used by Agda's typechecker (written in Haskell) whose code has been compiled, its usage will drastically improve the performance of unification.

In a graphical manner, figure 9 depicts the conceptual changes to the library and how they affect the overall procedure of automatic program construction.

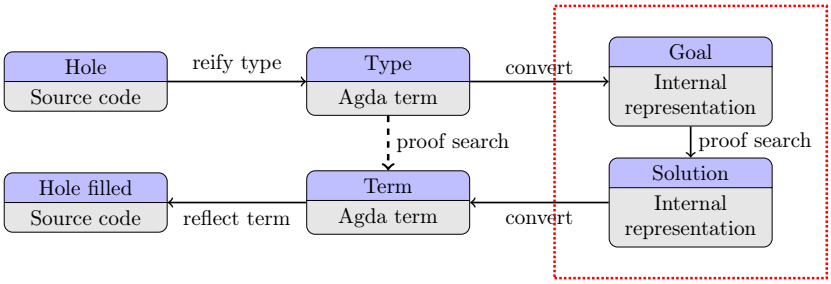

Figure 9: Improving AutoInAgda

3.1 Implementation

In this section, we explain the main changes the original implementation of AutoInAgda suffered in order to support the new API. Because the monadic nature of the reflection API, these changes make use of rather unsatisfying Agda flags such as TERMINATING or NO_POSITIVITY_CHECK. It is possible that alternative implementations of the ideas presented here can overcome such pitfalls but doing so would have been by a large extend out of the scope of this project.

The three main components of the library that are affected by the changes are:

- Term and rule representations
- Proof search tree
- Unification function over terms

In following subsections we proceed to explain one by one the changes made in these three components.

3.2 Term and rule representations

AutoInAgda original library used a type of first-order terms that are the basic building brick of the library. This datatype, Term, is a type family indexed by the number of free variables a term may contain.

```
data Term (n : ℕ) : Set where
  var : (x : Fin n) → Term n
  con : (s : Name) (ts : List (Term n)) → Term n
  lit : (l : Literal) → Term n
```

Figure 10: Term datatype in AutoInAgda

The Term datatype uses the constructor var to stand for unification variables. The constructor con then is used for function application, constructor application and skolem variables (i.e. variables that can not be instantiated during unification). The lit constructor is used for Agda's builtin literals such as the natural number datatype Nat.

The datatype we just described is essential to the proof search algorithm, which is reflected in how the proof search rules are encoded within the library. A rule is composed of a Term standing for its conclusion and a list of Terms that need to be satisfied so the rule can be applied. The following record type accounts for this fact.

The last key ingredient of the library is the unification function. As expected the unification function works directly on the Term datatype yielding a substitution for the unification variables in case the terms are unifiable. It is given the following type.

As we mentioned before, there are some caveats by using such representations. First, Agda's type system is a higher-order language which cannot be fully encoded by using a first-order representation. AutoInAgda's choice is to

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

Figure 11: Rule datatype in AutoInAgda

```
unify : ∀ {m} → (t₁ t₂ : Term m) → Maybe (∃ (Subst m))
```

Figure 12: Unification function type in AutoInAgda

throw a type error when a type could not be converted because for example used higher-order arguments.

Secondly, there is an implicit overhead in the conversion of the Agda term representing the type into the internal representation.

Last but not least, the unification function used by AutoInAgda is not as performant as the one provided internally by the reflection API.

For all this reasons, we decided that we can take good advantage of the new reflection API by changing the internal representation of terms to Agda's Term datatype and using the function unify provided by the API.

The Term datatype exposed by the reflection API is the following (module Reflection).

```
data Term where
  var      : (x : Nat) (args : List (Arg Term)) → Term
  con      : (c : Name) (args : List (Arg Term)) → Term
  def      : (f : Name) (args : List (Arg Term)) → Term
  lam      : (v : Visibility) (t : Abs Term) → Term
  pat-lam  : (cs : List Clause) (args : List (Arg Term)) → Term
  pi       : (a : Arg Type) (b : Abs Type) → Term
  agda-sort : (s : Sort) → Term
  lit      : (l : Literal) → Term
  meta     : (x : Meta) → List (Arg Term) → Term
  unknown  : Term
```

Figure 13: Term datatype exposed in the module Reflection

In comparison with the Term type as used by AutoInAgda (figure 3.2) this datatype is much richer as can capture a wider variety of constructions (literally it can represent any Agda type).

This representation, uses the var constructor to represent locally bound variables by they DeBruijn index. (this the argument (x : Nat)). If our intention is to use this type and be able to have unification variables on it (i.e. variables that can be unified with constructors or other variables) we need to rely in the

```
record Rule : Set where
  constructor rule
  field
    rname      : RuleName
    conclusion : Term
    premises   : List (Arg Term)
```

Figure 14: Rule type in this project

meta constructor that stands for Agda's metavariables (or holes).

Once, we settled on a representation for terms we need as well to define the rules that the proof search procedure will use in its operation. In order to do so, we define the following record.

The Rule type has been adapted to make use of the Agda Term datatype as defined in figure 3.2. As locally bound variables have to be distiguished from functions or type constructors, we keep the rule RuleName field to do so.

In order to generate a new rule, we use the reflection API to retrieve the type of the function or constructor as a Term value. Consequently, the term is procecessed such that every ocurrence of the pi constructor is striped off and the left handside (the argument) added as a premise. Finally, the remaining term is used as conclusion of the rule.

We haven't mentioned yet why the premises hold the type Arg Term instead of simply Term. In a moment this will become clear why we do so.

The purpose of a rule is to be used in the process of proof search. Because of this, the main interest we have is to understand how unification between the conclusion of the rule and the goal we need to solve is performed.

Each time the conclusion of the rule has to be unified with some other term we need to be able to distinguish in the term which subparts of it are open to be unified. For example, if we have a goal such as Even (suc (suc zero)) and the rule isEven+2 : $\forall$ {n} $\rightarrow$ Even n $\rightarrow$ Even (suc (suc n)) it is clear that n must stand for a unification variable so we can actually unify n $\rightarrow$ zero.

To achieve this, we decided that for a rule to be useful its local variables have to be replaced with Agda metavariables which will act as unification variables. Eeach time a rule has to be applied we have to create fresh metavariables that will substitute the the local variables within the rule.

However, it is not desired that for every premise of a rule a metavariable is created because even if the proof search succeeds and the goal is solved, Agda will complain that there are still metavariables to be solved. In order to prevent this from happening, our decision is to use the visibility of the premise of a rule (given by the Arg type defined in the reflection API) as an indicator of whether we need to instantiate the premise with a fresh metavariable should be created for this or not.

This process of intantiating a rule is done by the following functions:

The first function, inst-term, given a list of possible metavariables of the adecuate type (in order to create a fresh metavariable we need to know its type in advance) will replace the local variables, i.e. those created with the var constructor, for the metavariable if any that occupies the DeBruijn index in the

```
inst-term : List (Maybe Term) → Term → Term
inst-term = ...
inst-rule : Rule → TC (List (Maybe Term) × Rule)
inst-rule = ...
```

Figure 15: Functions to instantiate a Rule

list the variable holds a reference to.

The list of metavariables is of type Maybe Term because as we explained before, not all premises will be instantiated with fresh metavariables.

The function, inst-rule performs the process of instantiating every premise from left to right. Each time a premise is instantiated, depending of its attribute visibility a new metavariable of its type will be created or not.

As a small example, in the rule created from the function even+ : **forall** {n m : Nat} → Even n → Even m → Even (n + m) the variables n and m of type Nat will be replaced by fresh metavariables of type Nat while no metavariable will be created for either the premises Even n or Even m.

3.3 Proof search tree

The original approach to performing the proof search in AutoInAgda was to decouple the generation of the proof search tree from the actual traversing of it. In order to do so, the proof search tree was represented as a Rose tree of finitely branching capacity but possibly infinitely depth.

However, in our approach we have to change the represention of the tree in order to account for the monadic nature of the functions involved for instantiation of rules and unification. Moreover, we need the backtracking power of the TC monad during the generation of the proof search tree.

As in vanilla AutoInAgda, our design keeps a division between the generation of the proof search tree and the function that performs the traversal. This leaves open the possiblility of implementing different proof search strategies as before.

In the following figure we present both the new represention for search trees and the function solve that given a goal Term and a hint database generates the tree.

```
{-# NO_POSITIVITY_CHECK #-}
data SearchTree (A B : Set) : Set where
  succ-leaf : B → A → SearchTree A B
  fail-leaf : B → SearchTree A B
  node : B → List (TC (SearchTree A B)) → SearchTree A B
```

Figure 16: Search tree

Figure 3.3 exposes our representation of proof search trees. The main characteristic is that the node constructor holds a list of actions in the TC monad that will be used to generate subsequent trees.

This has the effect of having to declare that the typechecker must not perform the possitivity check on the SearchTree type because there is no information of whether the TC type is positive or not in its argument. For our purposes, we assume TC this is the case and therefore use the flag to bypass the typechecker.

The representation we have chosen is neccesary because each branching in the tree depends on an action performed in the TC monad. As the following function, solve, shows.

```
{-# TERMINATING #-}
solve : Term → HintDB → SearchTree Proof DebugInfo
solve g db = solveAcc (1 , g :: [] , Vec.head) (nothing , nothing) db
  where
    solveAcc : Proof′ → DebugInfo → HintDB → SearchTree Proof DebugInfo
    solveAcc (0 , [] , p) di _ = succ-leaf di (p [])
    solveAcc (suc k , g :: gs , p) di db = node di (map step (getHints db))
      where
        step : Hint → TC (SearchTree Proof DebugInfo)
        step h = catchTC (do g′ ← normalise g
                             -| ir              ← inst-rule (getRule h)
                             -| unify′ g′ (conclusion (proj₂ ir))
                             ~| return (solveAcc (prf (proj ir)) (just (rname (getRule h)) , nothing) db))
                         (return (fail-leaf (just (rname (getRule h)) , just g)))
          where
            prf : Rule → Proof′
            prf r = (length (premises r) + k) , prm′ , (p ∘ con′ r)
              where
                prm′ = Vec.map unArg (Vec.fromList (premises r))
                  Vec.++ gs
```

Figure 17: Solve function

For each goal we have, every rule in the hint database has to be instantiated in order to check whether the rule can be applied, its premises become new goals to solve, or the conclusion doesn't unify with the goal. In the latter case we must discard any new metavariable created for this purpose or Agda will complain about unsolved metavariables when trying to typecheck the final solution.

As a result, the instantiation and unification are done inside a branch of the TC monad combinator catchTC : TC A → TC A → TC A (provided by the reflection API) so in case of failure we can backtrack without having free metavariables lying around.

3.4 Unification function

At the beginning of this project our goal was to use the unification function provided by Agda natively in order to increase the performace of the proof searching. However, it turns out that unify is too liberal in what allows to unify and by using it in raw we are able to find solutions that are not solutions because when reflected back won't pass the typechecker.

The following snipped of code which doesn't fail although it warns us about unsolved metavariables ilustrates this behaviour.

```
example′ : TC ⊤
example′ =
  do m ← newMeta (def (quote ℕ) [])
  -| n ← newMeta (def (quote ℕ) [])
  -| unify m (lit (nat 0))
  ~| unify m (def (quote _+_)
     (arg (arg-info visible relevant) n ::
     arg (arg-info visible relevant) (lit (nat 2)) :: []))
  ~| return tt
example : ⊤
example = run (λ _ → example′)
```

Figure 18: Unification function example

The expected behaviour would be that the second call to unify fails because as m was unified to zero it cannot be unified to n + 2. However, the second unify does not fail and leaves the door open to generate non sense terms during proof search because unification won't fail even when is supposed to do so.

As a consequence of this, we have not to use the raw unify from the reflection API but to use it to build a more custom function that behaves more as we expect of it. To this matter, we have included an example unify function in the module Unification that is able to already rule out the example in figure 3.4.

However, it is not so clear how the unification function has to be customized in order to be able to find a solution for the programs we want but not generate terms that finally when reflected back into Agda will prevent the file from typechecking.

4 Benchmarking

In this section, we present a benchmark of the resulting implementation. For this matter, we have used the Haskell benchmarking library criterion . In order to do so we have selected two simple examples that exhibit the improvement achieved in performance terms.

```
evenBenchCase1 : Even _
evenBenchCase2 : forall {n} → Even n → Even (n + _)
```

The templates for the benchmark cases are parametrized with the size of the case that we are benchmarking again, because in order to test the performance of the libraries we would like to do so for different values.

[0]http://www.serpentine.com/criterion/

The benchmarking results of both implementations can be seen in figure 19 for the original AutoInAgda and in figure 20 for the performance of the implementation presented here.

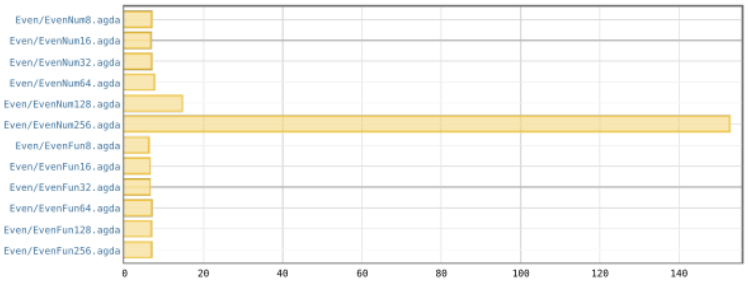

Figure 19: Benchmark using original AutoInAgda

Figure 20: Benchmark using our implementation

As we can see from the results above, our implementation has made anbig improvement in terms of performance in some cases in the range of $x15$ (case EvenNum256. For small cases the difference between implementations is despicable, although is worthy to note that there is always a lower limit of what we can achieve due to the requirement of Agda to check that every file we depend on has been already typechecked.

5 Conclusions and future work

This work has explored the possibilities that the new Agda's metaprogramming API has to offer in order to automatise program construction. We have been able as shown in section 4 to have a big performance boost with regard to the original implementation of the AutoInAgda library in some cases reaching a 10 . However, we have also noted that expressive wise our library deeply depends on how the internal unify (the one offered by the reflection API is just a reflection of the one used internally by the typechecker) function should be changed to allow automatic construction of more programs while retaining the ones it is already able to solve.

real number here

We believe that intimate knowledge of the inner workings of Agda's inference and typechecking algorithms is required to improve this situation. Nevertheless, this situation is not as bad because we have designed our framework to be unify

agnostic. This is, we have parametrized the proof search by the unification function so it can be easily interchanged and tested.

Moreover, we have introduced an implementation of some basic debugging facilities that work both for the original AutoInAgda library and our extended version, that should make the process of automatically filling Agda holes more pleasant for the Agda's programmer or type theorist practitioner.

As a future line of work, we should be able to integrate our framework into a sort of elaboration monad for metaprogramming as it was done by Christiansen and Brady [2] in Idris. An attempt to implement such elaboration monad can be found in the accompanying code of this work under the folder *ElaborationInAgda*.

References

[1] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant (MIT Press)*. The MIT Press, 2013.

[2] David Christiansen and Edwin Brady. Elaborator reflection: Extending idris in idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 284–297, New York, NY, USA, 2016. ACM.

[3] Pepijn Kokke and Wouter Swierstra. *Auto in Agda*, pages 276–301. Springer International Publishing, Cham, 2015.

[4] Conor McBride. First-order unification by structural recursion. *J. Funct. Program.*, 13:1061–1075, 2003.