A specification for Agda Core's unification algorithm for generic pattern matching

EWEN BROUDIN-CARADEC, TU Delft University, Netherlands and ENS Paris Saclay, France

Category: Graduate **Advisor:** JESPER COCKX

1 Introduction

In programming languages, pattern matching allows a program to analyze the different possible values a term can take described by so-called patterns. It is therefore present in a lot of functional programming languages, including Haskell, OCaml and Scala. In the case of proof assistants like Agda, totality is important so these patterns have to cover all possible cases. One way to do so is to propose a pattern for each constructor of the type of the eliminated term. However, in the presence of indexed datatypes, some patterns can often be ruled out by looking at the type of the term. For example, in Agda, given the datatypes $\mathbb{N}$ and `Vec` for natural numbers and lists indexed by their size (defined in Appendix A), one can define a head function that operates on any non-empty list as follows:

```
head : {A : Set} {n : ℕ} → Vec A (Suc n) → A
head (Cons _ a _) = a
```

Head of a vector in Agda

Here, there is no case for the empty list (`Nil`). It can safely be omitted. Indeed, the type that this pattern would have (`Vec A Zero`) is incompatible with the type that we already know the term has (`Vec A (Suc n)`). This incompatibility is found via unification. As the unification algorithm is at the heart of Agda, it will be present in Agda Core, a work-in-progress core language for Agda. However, the unification algorithm might rely on implicit assumptions – like uniqueness of identity proofs [2]. In a core language made to study mathematical theories like homotopy type theory, this is inconsistent. To prevent that, a specification of this algorithm is needed. The formalisation of this specification in Agda, as a part of Agda Core, is the topic of this work.

2 Agda Core

The main use of proof assistants, like Agda, is to increase the trust that we have in mathematical proofs by verifying them. It is especially useful for long and fastidious proofs where small mistakes can easily be overlooked. However, the code behind these proof assistants is also quite complex. For example, Agda 2 was first released in 2007 and has been continuously improved since then with more than 20 000 commits and almost 6 000 issues. This complexity is an obstacle to the trustworthiness as it can hide bugs which could lead Agda to accept invalid proofs. In fact, several ways to prove false have already been found (many were solved) and others could still be unnoticed.

To solve this issue, several proof assistants have a core language: a minimal fragment of their language used to rebuild everything from it. The aim of core language is to increase the trust in a proof assistant, therefore they are most of the time formalised like MetaRocq [8] for the Rocq Prover or Lean4Lean [1] for Lean. Agda Core is no exception and is mainly implemented in Agda [5]. Agda Core's implementation also uses a correct-by-construction approach. The data structures used in the code are made as restrictive as possible to ensure by their definition that they can only

Author's Contact Information: Ewen Broudin-Caradec, TU Delft University, Delft, Netherlands and ENS Paris Saclay, Gif-Sur-Yvette, France, ewen.broudin-caradec@lmf.cnrs.fr.

be used for their intended purpose. For example, in a typing judgement, $\Gamma \vdash a : A$, the construction of $\_ \vdash \_ : \_$ makes it impossible for a or A to contain variables not present in Γ. Hence, a and A are well scoped by construction. This is made possible by indexing terms, types, contexts, typing judgements, etc by a scope, a list of variables: it makes dependencies explicit. This syntax prevents many mistakes but makes it harder to manipulate variables which is needed for unification.

Most core language do not use a unification algorithm and do not accept generic pattern matching where some patterns are omitted because their types do not correspond to the type of the eliminated term. For example, in Rocq, there is an elaboration from the syntax used by the Rocq to the syntax of the core language. It transforms generic pattern matching into a pattern matching on all possible constructors of the type of the eliminated term (Appendix C). In order to recheck Agda files via Agda Core, an elaboration algorithm (programmed in Haskell[1]) also exists for Agda Core. However to keep this algorithm as simple as possible, Agda Core syntax is close to the internal syntax of Agda.[2] As such, it supports generic pattern matching in its full generality. Hence, Agda Core needs a unification algorithm.

3 Unification

The specification of unification is one of the challenges of Agda Core. It involves choosing the right data structures, and mistakes in the specification can lead to unwanted assumptions.

Let us first define what a unification problem is. Unification problems are stated using telescopic equalities. They are lists of named equalities. Each equality is written $e : t \stackrel{?}{=}_A u$ where e is the name of the equality and t and u are terms of type A. t and u are what should be unified. In the case of the empty telescopic equality, $()$ there is nothing to unify. A unification problem is, given a context and a telescopic equality, to determine if, by following a given set of unification rules, there exists a path to an updated context and an empty telescopic equality. The name "telescopic equality" comes from the fact that the type of an equality can depend on the previous ones. This allows us to write heterogeneous equalities (like in Appendix B).

With the datatypes **Vec** and $\mathbb{N}$ for vectors and natural numbers and a type A, each line of the example on the right is a unification problem. In (5) the problem is solved because the telescopic equality is empty.

$$\text{Let } \Gamma := a : A, n : \mathbb{N}, b : A$$

$$(\Gamma), (e : \text{Cons } 1\ a\ \text{Nil} \stackrel{?}{=}_{\mathbf{Vec}\ A\ 1} \text{Cons } n\ b\ \text{Nil}) \quad (1)$$

$$\rightsquigarrow (\Gamma), (e_0 : 1 \stackrel{?}{=}_{\mathbb{N}} n)(e_1 : a \stackrel{?}{=}_A b)(e_2 : \text{Nil} \stackrel{?}{=}_{\mathbf{Vec}\ A\ 0} \text{Nil}) \quad (2)$$

$$\rightsquigarrow (a : A, b : B), (e_1 : a \stackrel{?}{=}_A b)(e_2 : \text{Nil} \stackrel{?}{=}_{\mathbf{Vec}\ A\ 0} \text{Nil}) \quad (3)$$

$$\rightsquigarrow (a : A), (e_2 : \text{Nil} \stackrel{?}{=}_{\mathbf{Vec}\ A\ 0} \text{Nil}) \quad (4)$$

$$\rightsquigarrow (a : A), () \quad (5)$$

Basic unification example

To solve these problems, unification rules are used to transform a problem into an equivalent one until either the unification fails or the telescopic equality of the problem is reduced to the empty one. There are two kinds of unification rules: positive and negative. During unification, if a negative rule is encountered, it is possible to directly conclude that unification is impossible. In total, there are only 5 rules, 3 positive and 2 negative. The positive are, *solution* when a variable is replaced by a term, the rule used to go from (2) to (3) and from (3) to (4), *deletion* when two terms are the same (only valid for terms with uniqueness of identity proofs

[1]This process of rechecking the files is entirely run in Haskell but the rest of the Haskell files are extracted from Agda Core source files (in Agda) via agda2hs[3]. It is still a work in progress and only works on simple examples.

[2]However, the *surface* syntax of Agda used for pattern matching is quite different from Agda Core's syntax and Agda's *internal* syntax which both use case trees.

as otherwise it would create this assumption) and *injectivity* to advance when we face two identical constructors like in (1) or (4). The negative rules are, *conflict* which states that if two terms are created from different constructors, they cannot be unified, and *cycle*, to prevent the unification of a variable with a term using this variable.

While these rules are rather intuitive, formulating them in the wrong way can lead to properties like uniqueness of identity proofs or injectivity of type constructors (incompatible with univalence and excluded middle [2]). However, Agda is a proof assistant and aims to cover as many type theories as possible, including homotopy type theory, so staying agnostic on these properties is important in the context of Agda Core. The paper of Cockx and Devriese [2] presents a unification system with rules preventing this loss of generality. These rules have been implemented in the unification algorithm of Agda 2.5.1 and a formalisation work has already been done by Lieverse [6] using axiom K. In the Equations extension for Rocq, another unification system was used which is comparable but slightly more restrictive than these rules [9]. If unification is a solved problem for first order[7], it is undecidable for higher order[4][10].

4 Contribution

I have implemented two distinct data structures for telescopic equality in Agda. The one presented in Section 3, which is easier to read, and another, composed of two lists of terms and a telescope, the type of these lists. I proved an equivalence between the two representation and began to use both in order to find the most suitable one in this case. The problem was to construct and eliminate a structure where, in each equality, the type can depend on the previous equalities but the terms cannot. For example, given a type A and a dependant type $B\ x$, in the context $\Gamma' := a : A, a' : A, b : B\ a, b' : B\ a'$, $(e_0 : a \overset{?}{=}_A a')(e_1 : b \overset{?}{=}_{B\ e_0} b')$ is a valid telescopic equality but $(e_0 : a \overset{?}{=}_A a')(e_1' : a \overset{?}{=}_A e_0)$ is not. With the correct-by-construction approach where each type or term is indexed by the variables that it can use, the first structure where each equality was an object became very hard to manipulate because, in the same equality, terms and types were constructed in a very different way and did not use the same variables. In the previous example, in e_1, b and b' are "a, a', b, b' terms" but $B\ e_0$ is a "a, a', b, b', e_0 type". In the second structure, the separation in distinct lists made variable's handling much simpler.

Using this second definition, I wrote down a specification of the unification rules of Cockx and Devriese [2]. The *injectivity* rule was the most technical as it starts with a constructor and two sets of arguments (implemented as lists of independents terms), and generates a telescopic equality, with potential dependencies in the types. For example, it happens in the unification of a dependant pair, $\Gamma', (e : (a, b) \overset{?}{=}_{\Sigma_{x:A}, B\ x} (a', b'))$ becomes $\Gamma', (e_a : a \overset{?}{=}_A a')(e_b : b \overset{?}{=}_{B\ e_a} b')$ where the type in e_b depends on e_a. Many substitutions were also needed because of the explicit dependencies. Due to the technicality of the *injectivity* rule, a first version was implemented with axiom K before a second version, less powerful and without this assumption, was added.

I also wrote an algorithm to check if variables can be swapped in a context. This algorithm is needed because unification can change the order of the variables. For example, it happens in the case of $(a : A)(n : \mathbb{N})(v : \mathbf{Vec}\ A\ n)(b : A)(m : \mathbb{N})(w : \mathbf{Vec}\ A\ (\mathrm{Suc}\ m)), (e_0 : n \overset{?}{=}_{\mathbb{N}} \mathrm{Suc}\ m)$. After n is unified with Suc m, v depends on m so the initial order of the variables cannot be kept ((6) to (7) in Appendix B).

5 Future work

The next step is, of course, to use this specification in the implementation of a unification algorithm for Agda Core. Once this algorithm is programmed, further work will be needed on the existing type checking algorithm to call the unification algorithm when it is needed.

References

[1] Mario Carneiro. 2025. Lean4Lean: Verifying a Typechecker for Lean, in Lean. arXiv:2403.14064 [cs.PL] https://arxiv.org/abs/2403.14064

[2] Jesper Cockx and Dominique Devriese. 2018. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *Journal of Functional Programming* 28 (2018). https://api.semanticscholar.org/CorpusID:44145310

[3] Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. 2022. Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium* (Ljubljana, Slovenia) *(Haskell 2022)*. Association for Computing Machinery, New York, NY, USA, 108–122. doi:10.1145/3546189.3549920

[4] Warren D. Goldfarb. 1981. The undecidability of the second-order unification problem. *Theoretical Computer Science* 13, 2 (1981), 225–230. doi:10.1016/0304-3975(81)90040-2

[5] Bohdan Liesnikov and Jesper Cockx. 2025. Building a Correct-by-Construction Type Checker for a Dependently Typed Core Language. In *Programming Languages and Systems*, Oleg Kiselyov (Ed.). Springer Nature Singapore, Singapore, 63–83.

[6] K.Z. Lieverse. 2024. *A Generic Translation from Case Trees to Eliminators.* Master's thesis. TU Delft. https://resolver.tudelft.nl/uuid:e91ef1ea-942e-4f11-a1c4-bb82f444aaed

[7] CONOR MCBRIDE. 2003. First-order unification by structural recursion. *Journal of Functional Programming* 13, 6 (2003), 1061–1075. doi:10.1017/S0956796803004957

[8] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 8 (Dec. 2019), 28 pages. doi:10.1145/3371076

[9] Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 1–29. doi:10.1145/3341690

[10] Simon Spies and Yannick Forster. 2020. Undecidability of higher-order unification formalised in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 143–157. doi:10.1145/3372885.3373832

A Definition in Agda

```
data ℕ : Set where
  Zero : ℕ
  Suc : ℕ → ℕ
```

```
data Vec (A : Set ) : ℕ → Set where
  Nil : Vec A Zero
  Cons : (n : ℕ) → A → Vec A n → Vec A (Suc n)
```

B A more complete example

Let $\Gamma_0 := (a : A)(n : \mathbb{N})(v : \mathbf{Vec}\ A\ n)(b : A)(m : \mathbb{N})(w : \mathbf{Vec}\ A\ (\mathrm{Suc}\ m))$ in
  Let $\Gamma_1 := (a : A)(m : \mathbb{N})(v : \mathbf{Vec}\ A\ (\mathrm{Suc}\ m))(b : A)(w : \mathbf{Vec}\ A\ (\mathrm{Suc}\ m))$ in
  Let $\Gamma_2 := (a : A)(m : \mathbb{N})(v : \mathbf{Vec}\ A\ (\mathrm{Suc}\ m))(w : \mathbf{Vec}\ A\ (\mathrm{Suc}\ m))$ in
  Let $\Gamma' := (a : A)(m : \mathbb{N})(v : \mathbf{Vec}\ A\ (\mathrm{Suc}\ m))$ in

$$(\Gamma_0), (e_0 : n \stackrel{?}{=}_{\mathbb{N}} \text{Suc } m)(e : \text{Cons } n\ a\ v \stackrel{?}{=}_{\textbf{Vec}\ A\ e_0} \text{Cons (Suc } m)\ b\ w) \tag{6}$$

$$\rightsquigarrow (\Gamma_1), (e : \text{Cons (Suc } m)\ a\ v \stackrel{?}{=}_{\textbf{Vec}\ A\ (\text{Suc } m)} \text{Cons (Suc } m)\ b\ w) \tag{7}$$

$$\rightsquigarrow (\Gamma_1), (e_1 : \text{Suc } m \stackrel{?}{=}_{\mathbb{N}} \text{Suc } m)(e_2 : a \stackrel{?}{=}_{A} b)(e_3 : v \stackrel{?}{=}_{\textbf{Vec}\ A\ (\text{Suc } m)} w) \tag{8}$$

$$\rightsquigarrow (\Gamma_1), (e_2 : a \stackrel{?}{=}_{A} b)(e_3 : v \stackrel{?}{=}_{\textbf{Vec}\ A\ (\text{Suc } m)} w) \tag{9}$$

$$\rightsquigarrow (\Gamma_2), (e_3 : v \stackrel{?}{=}_{\textbf{Vec}\ A\ (\text{Suc } m)} w) \tag{10}$$

$$\rightsquigarrow (\Gamma'), () \tag{11}$$

Unification example

C The Rocq's elaboration

```
Inductive ℕ : Type :=
  | Zero : ℕ
  | Suc : ℕ → ℕ.
```

```
Inductive Vec (A : Type) : ℕ → Type :=
  | Nil : Vec A Zero
  | Cons : ∀ (n : ℕ), A → Vec A n → Vec A (Suc n).
```

```
Definition head {A : Type} {n : ℕ} :
  Vec A (Suc n) → A :=
  fun v ⇒ match v with
  | Cons _ _ a _ ⇒ a
  end.

Print head.
```

Output of the print:

```
head =
fun (A : Type) (n : ℕ) (v : Vec A (Suc n)) ⇒
match
  v in (Vec _ n0) return match n0 with
                         | Zero ⇒ IDProp
                         | Suc _ ⇒ A
                         end
with
| Nil _ ⇒ idProp
| Cons _ _ a _ ⇒ a
end
     : ∀ {A : Type} {n : ℕ}, Vec A (Suc n) → A
```