Relational Equality in the Intensional Theory of Types

Victor Cacciari Miraldo

University of Minho
University of Utrecht

1 Introduction

Relational Algebra [BdM97] has already proven to be a very expressive formalism for calculating with programs. In particular, relational shrinking can be used to derive a program as an optimization of its specification [MO12]. Nevertheless, there is still lack of computer support for relational calculus. Our approach is basically *piggybacking* on Agda[Nor09], an emerging language with a dependent type system, instead of building such a support system from scratch. Agda has a series of features that make it a very interesting target for such system. One such feature is being able to define mix-fix operators, from where its *Equational Reasoning* framework arises. This framework can be modified to closely resemble what a *squiggolist* would write on paper.

The task of encoding Relational Algebra in Martin-Löf's theory of types [ML84], however, is not as straightforward as one might think. There exists two separate efforts in such a direction. One is due to Mu et al., where a library targeted at program refinement is presented [MKJ09]; the other, more general approach, is due to Kahl [Kah14] and provides a complete categorical library for Agda, where the category of relations arises as a specific instantiation. Our goal is somewhat different, making both approaches unsuitable for us.

This *student-track* paper will focus on the difficulties we encountered when encoding relational equality in Agda in way suitable for (automatic) rewriting. We start with a (very) small introduction to Agda, section 2, aimed at readers without any Agda background whatsoever. We, then, explain how to perform syntactical rewrites in Agda, section 3. This should give a fair understanding of whats going to happen on sections 4, 5 and 6. Where we explain the encoding of relations, more specifically relational equality, in Agda and introduces some concepts from Homotopy Type Theory that allows one to fully formalize our model. We conclude on section 7 providing a summary of what was done and an example that illustrates the application of the tool we developed.

2 Agda Basics

In languages such as Haskell or ML, where a Hindley-Milner based algorithm is used for type-checking, values and types are clearly separated. Values are the objects being computed and types are simply tags to *categorize* them. In Agda,

however, this story changes. There is no distinction between types and values, which gives a whole new level of expressiveness to the programmer.

The Agda language is based on the intensional theory of types by Martin-Löf [ML84]. Great advantages arise from doing so, more specifically in the programs-as-proofs mindset. Within the Intensional Theory of Types, we gain the ability to formally state the meaning of quantifiers in a type. We refer the interested reader to [NPS90].

Datatype definitions in Agda resemble Haskell's GADTs syntax [VWPJ06]. Let us illustrate the language by defining fixed-size Vectors. For this, we need natural numbers and a notion of sum first.

```
data Nat : Set where
  Z : Nat
  S : Nat → Nat

_+_ : Nat → Nat → Nat
Z + n = n
(S m) + n = S (m + n)
```

The $Nat : Set$ statement is read as *Nat is of kind* $*$, for the Haskell enthusiast. In a nutshell, *Set*, or, Set_0, is the first universe, the type of small types. For consistency reasons, Agda has an infinite number of non-cumulative universes inside one another. That is, $\mathrm{Set}_i \nsubseteq \mathrm{Set}_i$ but $\mathrm{Set}_i \subseteq \mathrm{Set}_{i+1}$.

The underscores in the definition of + indicate where each parameter goes. This is how we define mix-fix operators. We can use underscores virtually anywhere in a declaration, as long as the number of underscores coincide with the number of parameters the function expects.

Leaving details aside, and jumping to Vectors, the following a possible declaration of fixed-size vectors in Agda.

```
data Vec (A : Set) : Nat → Set where
  Nil : Vec Z
  Cons : A → Vec A n → Vec A (S n)
```

Here we can already see something different from *Nat*. The type *Vec* takes one parameter, which must be a small type, we called it A and it is indexed by a natural number. Given an $A : Set$, *Vec A* has *kind* $Nat \to *$. This correctly models the idea of an inductive type family. For every natural number n, there is one type *Vec A n*.

Intuitively, if we concatenate a *Vec A n* to a *Vec A m*, we will obtain a *Vec A* $(n + m)$. This is exactly how vector concatenation goes in Agda.

```
_++_ : {A : Set} {n m : Nat} → Vec A n → Vec A m → Vec A (n + m)
Nil ++ v = v
(Cons a as) ++ v = Cons a (as ++ v)
```

The parameters enclosed in curly brackets are known as *implicit parameters*. These are values that the compiler can figure out during type-checking, so we need not to worry.

3 Rewriting in Agda

The steps of mathematical reasoning one usually writes on a paper have a fair amount of implicit rewrites. Yet, we cannot skip these steps in a proof assistant. We need to really convince Agda that two things are equal, by Agda's equality notion, before it can rewrite.

In Agda, writing $x \equiv y$ means that x and y *evaluate* to the *same* value. This can be seen from the definition of propositional equality, where we only allow one to construct an equality type using reflexivity:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Having a proof $p : x \equiv y$ convinces Agda that x and y will *evaluate* to the same value. Whenever this is the case, we can rewrite x for y in a predicate. The canonical way to do so is using the *subst* function:

```
subst : {A : Set} (P : A → Set) {x y : A} → x ≡ y → P x → P y
subst P refl p = p
```

Here, the predicate P can be seen as a context where the rewrite will happen. From a programming point of view, Agda's equality notion makes perfect sense! Yet, whenever we are working with more abstract concepts, we might need a finer notion of equality. However, this new equality must agree with Agda's equality if we wish to perform syntactical rewrites. As we will see in the next section, this is not always the case.

It is worth mentioning a subtle detail on the definition of *subst*. Note that, on the left hand side, the pattern p has type $P\ x$, according to the type signature. Still, Agda accepts this same p to finish up the proof of $P\ y$. What happens here is that upon pattern matching on *refl*, Agda knows that x and y evaluate to the same value. Therefore it basically substitutes, in the current goal, every y for x. As we can see here, pattern-matching in Agda actually allows it to infer additional information during type-checking.

4 Relations and Equality in Agda

In order to have a Relational Reasoning framework, we first need to have relations. We follow the same powerset encoding of [MKJ09], and encode a subset of a given set by:

```
ℙ : Set → Set1
ℙ A = A → Set
```

In Agda, Set is the type of types, which allows us to encode a subset of a set A as a function $f : \mathbb{P}\ A$. Such subset is defined by $\{a \in A \mid f\ a \text{ is inhabited}\}$. A simple calculation will let one infer $B \overset{R}{\leftarrow} A = B \to A \to Set$ from $\mathbb{P}\ (A \times B)$. The subrelation notion is intuitively defined by:

$$\begin{aligned}&\_ \subseteq \_ : \{A\ B : Set\} \to (B \overset{R}{\leftarrow} A) \to (B \overset{S}{\leftarrow} A) \to Set\\ &R \subseteq S = \forall\ a\ b \to R\ b\ a \to S\ b\ a\end{aligned}$$

This makes sense because we are saying that $B \overset{R}{\leftarrow} A$ is a subrelation of $B \overset{S}{\leftarrow} A$ whenever the set $R\ b\ a$ being inhabited implies that the set $S\ b\ a$ is also inhabited, for all $b\ a$. For this matter, a set S being inhabited means that there exist some $s : S$. This follows from the inclusion of (mathematical) sets.

The relation $\subseteq$ is an order: it is reflexive, transitive and anti-symmetric. Reflexivity and transitivity are straight-forward to prove, but there is a catch in anti-symmetry. Remember that relational equality is defined by mutual inclusion:

$$\begin{aligned}&\equiv_r : \{A\ B : Set\} \to (B \overset{R}{\leftarrow} A) \to (B \overset{S}{\leftarrow} A) \to Set\\ &R \equiv_r S = R \subseteq S \times S \subseteq R\end{aligned}$$

On paper, anti-symmetry follows by construction. But for rewriting purposes in Agda, we need to find a way to prove $R \equiv S$ from $R \equiv_r S$. Unfortunately, this is not possible without additional machinery. We would like to have:

$$\frac{\begin{array}{l}\forall\ a\ b \to R\ b\ a \to S\ b\ a\\ \forall\ a\ b \to S\ b\ a \to R\ b\ a\end{array}}{R \equiv S}\ \equiv_r \text{ promote}$$

We could postulate function extensionality[1] and, if the functions are isomorphisms, this would finish the proof. But this is somewhat cheating. By pinpointing the problem one can give a better shot at solving it.

Well, if $R \equiv S$, then $R\ b\ a \equiv S\ b\ a$ at least propositionally. On the other hand, if $R \equiv_r S$ then we might have propositionally different relations. Consider the following relations (where $\mathbb{1}$ is the unit type and $+$ is the coproduct).

$$\begin{aligned}&Top : Rel\ \mathbb{N}\ \mathbb{N} && Top' : Rel\ \mathbb{N}\ \mathbb{N}\\ &Top\ \_\ \_ = \mathbb{1} && Top'\ \_\ \_ = \mathbb{1} + \mathbb{1}\end{aligned}$$

Although they are equivalent, it is clear that $Top\ b\ a = \mathbb{1} \not\equiv \mathbb{1} + \mathbb{1} = Top'\ b\ a$. To prove propositional equality from relational equality we depend on the user not making stupid decisions. We are thus facing a subtle encoding problem.

[1] Function extensionality is expressed by point-wise equality. It is known not to introduce any inconsistency and it is considered to be a safe postulate. In short, given $f, g : A \to B, f \equiv g$ only if $\forall\ x \to f\ x \equiv g\ x$.

5 Homotopy Type Theory

Luckily people from the Univalent Foundations have thought of this problem. In this section we will borrow a few concepts from Homotopy Type Theory [Uni13] (HoTT) and discuss how these concepts can contribute to a solution for our encoding. There is no *final* solution, though, since they will boil down to design decisions.

Recalling our problem, given $R \equiv_r S$, $R\ b\ a$ and $S\ b\ a$ might evaluate to different types. But we do not care to which values they evaluate to, as long as one is inhabited iff the other is so. This notion is called *proof-irrelevance* in HoTT jargon, and the sets which are proof irrelevant are called *mere propositions*.

$$
\begin{aligned}
&\mathit{isProp} : \mathit{Set} \to \mathit{Set} \\
&\mathit{isProp}\ P = (p1\ p1 : P) \to p1 \equiv p2
\end{aligned}
$$

Let us denote the set of all *proof irrelevant* types, or $(\Sigma\ \mathit{Set}\ \mathit{isProp})$, by $\mathbb{MP}$. Should we have defined our relations as $B \to A \to \mathbb{MP}$, our problem would be almost done. The drawback of such a decision is the evident loss of expressiveness, for instance, coproducts are already not proof-irrelevant. Not to mention that users would have to be familiar with such notions before encoding their relations. We chose to encode this as a *typeclass* and, for using a fully formal definition of anti-symmetry, both relations must belong into that typeclass.

There is a very useful result we exploit. Given P a mere proposition. If we find an inhabitant $p : P$, then $P \approx \mathbb{1}$. If we find a contradiction $P \to \bot$, then $P \approx \bot$, where $\approx$ means univalence. For the unfamiliar reader, univalence can be thought of as some sort of isomorphism. The concept is too deep to be introduced in detail here. We refer the reader to [Uni13].

6 Anti-Symmetry of Relational Inclusion

Being a mere proposition is of no interest if we cannot compute the set $R\ a\ b$, for a given R , a and b. Yet, we can also make this explicit in Agda, by means of another *typeclass*. We require our relations to be decidable.

$$
\begin{aligned}
&\mathit{isDec} : \{A\ B : \mathit{Set}\}\ (B \xleftarrow{R} A) \to \mathit{Set} \\
&\mathit{isDec}\ R = (a : A)\ (b : B) \to (R\ b\ a) + (R\ b\ a \to \bot)
\end{aligned}
$$

On these terms, together with function extensionality, we are able to provide a proof of anti-symmetry in Agda's terms. The type[2] is:

$$
\begin{aligned}
&\subseteq\text{-antisym} : \{A\ B : \mathit{Set}\}\ \{R\ S : \mathit{Rel}\ A\ B\} \\
&\quad \{\{\mathit{decr} : \mathit{IsDec}\ R\}\}\ \{\{\mathit{decs} : \mathit{IsDec}\ S\}\} \\
&\quad \{\{\mathit{prpr} : \mathit{IsProp}\ R\}\}\ \{\{\mathit{prps} : \mathit{IsProp}\ S\}\} \\
&\quad \to R \subseteq S \to S \subseteq R \to R \equiv S
\end{aligned}
$$

[2] The double braces $\{\{\_\}\}$ work almost like Haskell's type context syntax $(Fa) \Rightarrow$.

Yet, this model of anti-symmetry is too restrictive for the user. During development, base relations might change, which will trigger changes in all of their instances. For this reason, we provide a postulate of the $\equiv_r$-promote rule introduced in section 4. Note, however, that this postulate allows for a contradiction. It is easy to see why. Take the *Top* and *Top′* relations defined in section 4. It is easy to prove $Top \equiv_r Top'$, therefore, through $\equiv_r$-promote we have $Top \equiv Top'$, but $\mathbb{1} \not\equiv \mathbb{1} + \mathbb{1}$, and, through *cong* (*const* $\circ$ *const*), we have $\lambda \_ \_ \to \mathbb{1} \not\equiv \lambda \_ \_ \to \mathbb{1} + \mathbb{1}$. Therefore, $Top \not\equiv Top'$. The actual Agda code for this is trickier than one thinks, hence it is omitted here.

This is not inconsistent with our model, however. Since the definition of a relation is basically the encoding of its defining predicate in Agda, one should not need to use *Set*s which are not mere propositions. The $\equiv_r$-promote postulate is there to speed up development. As we proved in this section (more details on [Mir15]), if we use decidable mere-propositions in the domain of our relations, no inconsistency arises. The coproduct is not a mere-proposition.

7 Summary and Conclusions

On this very short paper we presented the problem of encoding relational equality in Agda, and gave a few pointers on how to tackle it. This is just a short summary of [Mir15], where we present our library in full. We provide standard relational constructs, including a prototype of catamorphisms, generic on their functor. Our encoding uses *W-types* and is built on top of [AAG04]. We had to build all of these constructs from scratch, since we had the goal of also providing automatic inference of rewriting context for them, which we accomplished. The full Agda code of our relational algebra library, together with the author's master dissertation is available on GitHub.

https://github.com/VictorCMiraldo/msc-agda-tactics

It is arguable that we did not really solve the problem, since our final solution still relies on the univalence axiom ($a \approx b \to a \equiv b$ [Uni13]). We reiterate that there is no final solution to this problem, since different options will lead to libraries with different designs, fit for different purposes.

Here we give a small example of the final product of this project, and how we use the lifting of relational equality to perform generic (automatic) rewrites in Agda. Note that Agda's mix-fix feature allows one to define operators such as the *squiggol* environment. The example below is one branch of the proof that Lists are functors. The `by` function was also developed by us, and it uses the Reflection mechanism of Agda (similar to Template Haskell) to infer the substitution to be performed.

The rewrites we perform are just *subst*s, as explained in section 3. The automation happens on compile time. The *tactic* keyword gives us the meta-representation of the relevant terms, our engine then generates a meta-representation of the *subst* that justifies the rewrite step, which is then plugged back in before

the compiler resumes type-checking. It is very relevant to state that, although the lemmas justifying the rewrite steps have a $\equiv_r$ as their type, this is then converted to Agda's $\equiv$ in order for us to use *subst*. Relational equality is *not* substitutive, in general.

Let us imagine we are in doubt whether lists are a functor or not. A good start is if they distribute over composition, that is,

$$\mathrm{L}\ (f \cdot g) \equiv \mathrm{L}\ f \cdot \mathrm{L}\ g$$

The proof itself is simple to complete with the universal law for coproducts. We illustrate the i_2 branch in Agda, using our library.

$$
\begin{aligned}
&ex\ :\ (Id + (Id\ \times\ R)\ \circ\ Id + (Id\ \times\ S))\ \circ\ i_2\ \equiv_r\ i_2\ \circ\ Id\ \times\ (R\ \circ\ S)\\
&ex = \mathit{begin}\\
&\qquad (Id + (Id\ \times\ R)\ \circ\ Id + (Id\ \times\ S))\ \circ\ i_2\\
&\equiv_r \qquad \{\ \text{tactic (by (quote +-bi-functor))}\ \}\\
&\qquad (Id\ \circ\ Id) + (Id\ \times\ R\ \circ\ Id\ \times\ S)\ \circ\ i_2\\
&\equiv_r \qquad \{\ \text{tactic (by (quote } i_2\text{-natural))}\ \}\\
&\qquad i_2\ \circ\ Id\ \times\ R\ \circ\ Id\ \times\ S\\
&\equiv_r \qquad \{\ \text{tactic (by (quote } \times\text{-bi-functor))}\ \}\\
&\qquad i_2\ \circ\ (Id\ \circ\ Id)\ \times\ (R\ \circ\ S)\\
&\equiv_r \qquad \{\ \text{tactic (by (quote } \circ\text{-id-r))}\ \}\\
&\qquad i_2\ \circ\ Id\ \times\ (R\ \circ\ S)\\
&\qquad \square
\end{aligned}
$$

References

[AAG04] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using w-types. In *In Automata, Languages and Programming, 31st International Colloqium (ICALP), pages 59 71*, pages 59–71, 2004.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall international series in computer science. Prentice Hall, 1997.

[Kah14] W. Kahl. Rath-agda, relational algebraic theories in agda, Dez 2014.

[Mir15] Victor Cacciari Miraldo. Proofs by rewriting in Agda. Master's thesis, Utrecht University and University of Minho, 2015. Submitted.

[MKJ09] S-C. Mu, H-S. Ko, and P. Jansson. Algebra of programming in agda. *Journal of Functional Programming*, 2009.

[ML84] P. Martin-Löf. Intuitionistic type theory, 1984.

[MO12] Shin-Cheng Mu and Jos Nuno Oliveira. Programming from galois connections. *The Journal of Logic and Algebraic Programming*, 81(6):680 – 704, 2012. 12th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2011).

[Nor09] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[VWPJ06] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. *SIGPLAN Not.*, 41(9):251–262, September 2006.