Relational Equality in the Intensional Theory of Types

Victor Cacciari Miraldo

University of Utrecht, the Netherlands
University of Minho, Portugal

victor.cacciari@gmail.com

September 29, 2015

Prelude

During my Master's dissertation, we investigated how to encode Relational Algebra in the Intensional Theory of Types and created a *rewriting* engine that uses Agda's reflection mechanism to help the development of proofs using equational reasoning.

This presentation will address the Relational Algebraic part of our project.

Prelude

Topics

- How to encode relations in a dependently typed language.
- Which difficulties arise from using the standard notion of relational equality.

Introduction

Agda is a dependently typed, pure, language built on top of the intensional variant of the Theory of Types.

Our exploratory project relies on **Agda** to provide support for different types of equational reasoning. Our main focus is **relational algebraic** reasoning.

We examine **Agda** in its two flavors:

- Constructing a Relational Algebra (RA) Library in **Agda**, using its proof-assistant side.
- Developing a smarter *rewrite* functionality for the language, delving into the programming part of **Agda**.

We used the version 2.4.3, with standard library 0.9 for development.

A First Example

Let us imagine one doubts whether lists are functors or not. A good start would be checking distribution over composition:

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

or, expanding $\mathsf{L}\ \cdot$,

$$\mathsf{Id} + (\mathsf{Id} \times (f \cdot g)) \equiv \mathsf{Id} + (\mathsf{Id} \times f) \cdot \mathsf{Id} + (\mathsf{Id} \times g)$$

This proof is easy to complete with the universal law for coproducts, given that $+$ and $\times$ are bi-functors. Let us, then, illustrate one branch of the proof, in Agda!

A First Example

Let us imagine one doubts whether lists are functors or not. A good start would be checking distribution over composition:

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

or, expanding L $\cdot$,

$$\mathsf{Id} + (\mathsf{Id} \times (f \cdot g)) \equiv \mathsf{Id} + (\mathsf{Id} \times f) \cdot \mathsf{Id} + (\mathsf{Id} \times g)$$

This proof is easy to complete with the universal law for coproducts, given that $+$ and $\times$ are bi-functors. Let us, then, illustrate one branch of the proof, in Agda!

A First Example

```
begin
    (Id + (Id * R) • Id + (Id * S)) • ι₂
≡r⟨ (tactic (by (quote +-bi-functor))) ⟩
    (Id • Id) + (Id * R • Id * S) • ι₂
≡r⟨ (tactic (by (quote ι₂-natural))) ⟩
    ι₂ • Id * R • Id * S
≡r⟨ (tactic (by (quote *-bi-functor))) ⟩
    ι₂ • (Id • Id) * (R • S)
≡r⟨ ≡r-cong (λ Z → ι₂ • Z * (R • S)) (≡r-sym (•-id-r Id)) ⟩
    ι₂ • Id * (R • S)
□
```

Relational Algebra in **Agda**

Introduction

The state of the art, at the time we began our research, consistes in two existing libraries:

- Mu et al., developed a RA Library focused on Program Refinement by Sub-relation reasoning.
- Kahl worked on a generic Category Theory *standard library* for **Agda**, from which Rel. Algebra arises as an instance of some structures.

Relational Algebra in **Agda**

Challenges

- Relational Equality was hardest challenged we faced. The problem was solved by borrowing concepts from the Homotopy Type Theory.
- Adapting the code for automatic processing was another threat. The amount of boilerplate code we had to cope is significant.
- The general unstable state of **Agda** makes the development of a stable product complicated. The language suffered major changes meanwhile, which led us to rewrite a substantial amount of code.

Relational Algebra in **Agda**

Challenges

- Relational Equality was hardest challenged we faced. The problem was solved by borrowing concepts from the Homotopy Type Theory.
- Adapting the code for automatic processing was another threat. The amount of boilerplate code we had to cope is significant.
- The general unstable state of **Agda** makes the development of a stable product complicated. The language suffered major changes meanwhile, which led us to rewrite a substantial amount of code.

Relational Algebra in **Agda**

Challenges

- Relational Equality was hardest challenged we faced. The problem was solved by borrowing concepts from the Homotopy Type Theory.
- Adapting the code for automatic processing was another threat. The amount of boilerplate code we had to cope is significant.
- The general unstable state of **Agda** makes the development of a stable product complicated. The language suffered major changes meanwhile, which led us to rewrite a substantial amount of code.

Relations

A binary relation R is defined as a subset of $A \times B$, knowing that A and B are sets. We usually write $A \xrightarrow{R} B$.

Let us take the relation $\mathbb{N} \xrightarrow{Succ} \mathbb{N}$ (which is also a function!) as an example.

$$\begin{aligned} Succ &= \{(n, n+1) \in \mathbb{N}^2\} \\ &= \{(n, k) \in \mathbb{N}^2 \mid k = n+1\} \\ &= \{(n, k) \in \mathbb{N}^2 \mid isSucc(n, k)\} \end{aligned}$$

All we need to know to work with $Succ$ is which elements of $\mathbb{N}^2$ also belong in $Succ$, but those are the ones satisfying $isSucc$.

This provides valuable insight: Encoding a relation is encoding its defining predicate!

Relations

A binary relation R is defined as a subset of $A \times B$, knowing that A and B are sets. We usually write $A \xrightarrow{R} B$.

Let us take the relation $\mathbb{N} \xrightarrow{Succ} \mathbb{N}$ (which is also a function!) as an example.

$$\begin{aligned} Succ &= \{(n, n+1) \in \mathbb{N}^2\} \\ &= \{(n,k) \in \mathbb{N}^2 \mid k = n+1\} \\ &= \{(n,k) \in \mathbb{N}^2 \mid isSucc(n,k)\} \end{aligned}$$

All we need to know to work with $Succ$ is which elements of $\mathbb{N}^2$ also belong in $Succ$, but those are the ones satisfying $isSucc$.

This provides valuable insight: Encoding a relation is encoding its defining predicate!

Relations

A binary relation R is defined as a subset of $A \times B$, knowing that A and B are sets. We usually write $A \xrightarrow{R} B$.

Let us take the relation $\mathbb{N} \xrightarrow{Succ} \mathbb{N}$ (which is also a function!) as an example.

$$\begin{aligned} Succ &= \{(n, n+1) \in \mathbb{N}^2\} \\ &= \{(n, k) \in \mathbb{N}^2 \mid k = n+1\} \\ &= \{(n, k) \in \mathbb{N}^2 \mid isSucc(n, k)\} \end{aligned}$$

All we need to know to work with $Succ$ is which elements of $\mathbb{N}^2$ also belong in $Succ$, but those are the ones satisfying $isSucc$.

This provides valuable insight: Encoding a relation is encoding its defining predicate!

Encoding Relations

A predicate over a set A, in dependent types, is usually encoded as a function $A \to Set$.

For our relations, we have $A \times B \to Set \equiv B \to A \to Set$.
Therefore, we will adopt the following encoding:

```
Rel : Set → Set → Set1
Rel A B = B → A → Set
```

Let us define our $Succ : Rel\ \mathbb{N}\ \mathbb{N}$ relation in Agda:

```
Succ (suc zero) zero = Unit
Succ (suc x) (suc y) = Succ x y
Succ _ _ = ⊥
```

Hence, if $(2, 3) \in Succ$, then the *type* $Succ\ 3\ 2$ is *inhabited*:

```
isSucc-3-2 : Succ 3 2
isSucc-3-2 = unit
```

Encoding Relations

A predicate over a set A, in dependent types, is usually encoded as a function $A \to Set$.

For our relations, we have $A \times B \to Set \equiv B \to A \to Set$.
Therefore, we will adopt the following encoding:

```
Rel : Set → Set → Set1
Rel A B = B → A → Set
```

Let us define our $Succ : Rel\ \mathbb{N}\ \mathbb{N}$ relation in Agda:

```
Succ (suc zero) zero = Unit
Succ (suc x) (suc y) = Succ x y
Succ _ _ = ⊥
```

Hence, if $(2, 3) \in Succ$, then the *type* $Succ\ 3\ 2$ is *inhabited*:

```
isSucc-3-2 : Succ 3 2
isSucc-3-2 = unit
```

Encoding Relations

A predicate over a set A, in dependent types, is usually encoded as a function $A \to Set$.

For our relations, we have $A \times B \to Set \equiv B \to A \to Set$.
Therefore, we will adopt the following encoding:

```
Rel : Set → Set → Set1
Rel A B = B → A → Set
```

Let us define our $Succ : Rel\ \mathbb{N}\ \mathbb{N}$ relation in Agda:

```
Succ (suc zero) zero = Unit
Succ (suc x) (suc y) = Succ x y
Succ _ _ = ⊥
```

Hence, if $(2, 3) \in Succ$, then the *type* $Succ\ 3\ 2$ is *inhabited*:

```
isSucc-3-2 : Succ 3 2
isSucc-3-2 = unit
```

Encoding Relations

A predicate over a set A, in dependent types, is usually encoded as a function $A \to Set$.

For our relations, we have $A \times B \to Set \equiv B \to A \to Set$.
Therefore, we will adopt the following encoding:

```
Rel : Set → Set → Set1
Rel A B = B → A → Set
```

Let us define our $Succ : Rel\ \mathbb{N}\ \mathbb{N}$ relation in Agda:

```
Succ (suc zero) zero = Unit
Succ (suc x) (suc y) = Succ x y
Succ _ _ = ⊥
```

Hence, if $(2, 3) \in Succ$, then the *type* $Succ\ 3\ 2$ is *inhabited*:

```
isSucc-3-2 : Succ 3 2
isSucc-3-2 = unit
```

Relational Equality

A relation $A \xrightarrow{R} B$ is a sub-relation of S, written $R \subseteq S$, when:

$$\forall (a, b) \in A \times B \;.\; b \; R \; a \Rightarrow b \; S \; a$$

Equality is defined by mutual inclusion: $R \equiv_r S$ iff $R \subseteq S \wedge S \subseteq R$.

Or, in Agda:

```
_⊆_ : {A B : Set}(R S : Rel A B) → Set
R ⊆ S = ∀ a b → R b a → S b a
```

In words:

> *Given an $a : A$, $a\ b : B$ and an inhabitant of $R\ b\ a$, produce an inhabitant of $S\ b\ a$.*

Relational Equality

A relation $A \xrightarrow{R} B$ is a sub-relation of S, written $R \subseteq S$, when:

$$\forall (a, b) \in A \times B \;.\; b \; R \; a \Rightarrow b \; S \; a$$

Equality is defined by mutual inclusion: $R \equiv_r S$ iff $R \subseteq S \wedge S \subseteq R$.

Or, in Agda:

```
_⊆_ : {A B : Set}(R S : Rel A B) → Set
R ⊆ S = ∀ a b → R b a → S b a
```

In words:

> *Given an $a : A$, $a\ b : B$ and an inhabitant of $R\ b\ a$, produce an inhabitant of $S\ b\ a$.*

Relational Equality, in Agda

The Problem

Given that $\_ \subseteq \_$ is anti-symmetric (with respect to $\equiv_r$) by definition, proving equality should be simple.

However, propositional equality in Agda, $x \equiv y$ means that x and y *evaluate* to the *same* value.

Take $\mathbb{N} \xrightarrow{T,U} \mathbb{N}$ as the *Top* relation on Natural numbers and a dumb variant of it.

```
T _ _ = Unit
U _ _ = Unit ⊎ Unit
```

Note that, although we can prove, in Agda, that $T \equiv_r U$, they *do not* evaluate to the same value! They are not equal.

We need a better notion of equality!

Relational Equality, in Agda

The Problem

Given that $\_ \subseteq \_$ is anti-symmetric (with respect to $\equiv_r$) by definition, proving equality should be simple.

However, propositional equality in Agda, $x \equiv y$ means that x and y *evaluate* to the *same* value.

Take $\mathbb{N} \xrightarrow{T,U} \mathbb{N}$ as the *Top* relation on Natural numbers and a dumb variant of it.

```
T _ _ = Unit
U _ _ = Unit ⊎ Unit
```

Note that, although we can prove, in Agda, that $T \equiv_r U$, they *do not* evaluate to the same value! They are not equal.

We need a better notion of equality!

Relational Equality, in Agda

The Problem

Given that $\_ \subseteq \_$ is anti-symmetric (with respect to $\equiv_r$) by definition, proving equality should be simple.

However, propositional equality in Agda, $x \equiv y$ means that x and y *evaluate* to the *same* value.

Take $\mathbb{N} \xrightarrow{T,U} \mathbb{N}$ as the *Top* relation on Natural numbers and a dumb variant of it.

```
T _ _ = Unit
U _ _ = Unit ⊎ Unit
```

Note that, although we can prove, in Agda, that $T \equiv_r U$, they *do not* evaluate to the same value! They are not equal.

We need a better notion of equality!

Relational Equality, in Agda

Proof Irrelevance

Finding inspiration in the Homotopy Type Theory (HoTT), we see that this has been done before.

When this is the case, we can prove that they are *univalent* to either Unit or $\bot$.

$$\begin{aligned} &\text{lemma-332} &&: \{P : \text{Set}\} \to \text{isProp } P \to (p_0 : P) \to P \approx \text{Unit} \\ &\neg\text{lemma-332} &&: \{P : \text{Set}\} \to \text{isProp } P \to (P \to \bot) \to P \approx \bot \end{aligned}$$

Thus, by requiring the user to provide only *proof-irrelevant* relations, we can always *transform* them into Unit or $\bot$.

Relational Equality, in Agda

Proof Irrelevance

Finding inspiration in the Homotopy Type Theory (HoTT), we see that this has been done before.

When this is the case, we can prove that they are *univalent* to either Unit or $\bot$.

$$\begin{aligned} &\text{lemma-332} &&: \{P : \text{Set}\} \to \text{isProp}\ P \to (p_0 : P) \to P \approx \text{Unit} \\ &\neg\text{lemma-332} &&: \{P : \text{Set}\} \to \text{isProp}\ P \to (P \to \bot) \to P \approx \bot \end{aligned}$$

Thus, by requiring the user to provide only *proof-irrelevant* relations, we can always *transform* them into Unit or $\bot$.

Relational Equality, in Agda

Proving anti-symmetry of $\_ \subseteq \_$

At this point, we do not care anymore about the structure of the underlying set of a relation, as it will resemble Unit or $\bot$.

The last ingredient we need, is to stick to *decidable* relations:

$$\begin{aligned} &\mathsf{isDec} : \{A\ B : \mathsf{Set}\} \to \mathsf{Rel}\ A\ B \to \mathsf{Set} \\ &\mathsf{isDec}\ R = \forall\ a\ b \to (R\ b\ a) \uplus (R\ b\ a \to \bot) \end{aligned}$$

The rest of the proof is a straight-forward combination of these ingredients.

Hence, given two relations R and S that are mere propositions and decidable, we can finally prove anti-symmetry:

$$\begin{aligned} &\subseteq\text{-antisym} : \{A\ B : \mathsf{Set}\}\{R\ S : \mathsf{Rel}\ A\ B\} \\ &\quad \to R \subseteq S \to S \subseteq R \to R \equiv S \end{aligned}$$

Relational Equality, in Agda

Proving anti-symmetry of _ ⊆ _

At this point, we do not care anymore about the structure of the underlying set of a relation, as it will resemble Unit or ⊥.

The last ingredient we need, is to stick to *decidable* relations:

```
isDec : {A B : Set} → Rel A B → Set
isDec R = ∀ a b → (R b a) ⊎ (R b a → ⊥)
```

The rest of the proof is a straight-forward combination of these ingredients.

Hence, given two relations R and S that are mere propositions and decidable, we can finally prove anti-symmetry:

```
⊆-antisym : {A B : Set}{R S : Rel A B}
  → R ⊆ S → S ⊆ R → R ≡ S
```

Relational Equality, in Agda

Proving anti-symmetry of _ ⊆ _

At this point, we do not care anymore about the structure of the underlying set of a relation, as it will resemble Unit or ⊥.

The last ingredient we need, is to stick to *decidable* relations:

```
isDec : {A B : Set} → Rel A B → Set
isDec R = ∀ a b → (R b a) ⊎ (R b a → ⊥)
```

The rest of the proof is a straight-forward combination of these ingredients.

Hence, given two relations R and S that are mere propositions and decidable, we can finally prove anti-symmetry:

```
⊆-antisym : {A B : Set}{R S : Rel A B}
  → R ⊆ S → S ⊆ R → R ≡ S
```

Conclusions

- We can see that although complicated in a few points, Agda is an excellent tool for working with (so far, basic) Relational Algebra. The mixfix feature is a great plus!
- A few debugging tools from Agda could be very helpful. For instance, we failed to pinpoint (in the code) where the performance explodes with our prototype of catamorphisms.
- Despite not being mentioned here, the Reflection mechanism allows one to create interesting meta-programming libraries. More can be found on my thesis.

Relational Equality in the Intensional Theory of Types

Victor Cacciari Miraldo

University of Utrecht, the Netherlands
University of Minho, Portugal

victor.cacciari@gmail.com

September 29, 2015

Catamorphisms

The last relational construct we encoded was the catamorphism notion. Our objective was to provide functor generic catas. The end result is still experimental, and shows very poor performance in Agda 2.4.3, besides requiring some *TERMINATING* pragmas.

We will not delve too deeply into the technicalities of our encoding. Instead, we chose to make explicit the theoretical connection between catamorphisms and W-types, which is the foundation of our work.

Catamorphisms

The last relational construct we encoded was the catamorphism notion. Our objective was to provide functor generic catas. The end result is still experimental, and shows very poor performance in Agda 2.4.3, besides requiring some *TERMINATING* pragmas.

We will not delve too deeply into the technicalities of our encoding. Instead, we chose to make explicit the theoretical connection between catamorphisms and W-types, which is the foundation of our work.

Catas, as fixed points

In a pen-and-paper setting, one defines a cata as a function (resp. relation) with its domain being the least fixed point of a given recursive functor.

Let us take lists as an example:

$$\begin{aligned} F_A\ X &= 1 + A \times X \\ L_A &= \mu\, X\ .\ F_A\ X \end{aligned}$$

This definition clearly doesn't work in Agda, as it is trivially non-terminating. In the dependent type setting, one uses *W-types* to encode recursive datatypes generically.

Catas, as fixed points

In a pen-and-paper setting, one defines a cata as a function (resp. relation) with its domain being the least fixed point of a given recursive functor.

Let us take lists as an example:

$$\begin{aligned} F_A\ X &= 1 + A \times X \\ L_A &= \mu\ X\ .\ F_A\ X \end{aligned}$$

This definition clearly doesn't work in Agda, as it is trivially non-terminating. In the dependent type setting, one uses *W-types* to encode recursive datatypes generically.

W-types

The trick lies in the following reasoning.

$$\begin{aligned} F_A X &\cong 1 + A \times X \\ &\cong 1 \times 1 + A \times X \\ &\cong 1 \times X^{\perp} + A \times X^{1} \end{aligned}$$

Which gives us a shape functor $S = 1 + A$ and a positioning functor $P = [\text{const } \perp, \text{const } 1]$ such that $W\ S\ P$ is isomorphic to $\mu X.F_A\ X$, where W is defined as:

```
data W (S : Set)(P : S → Set) : Set where
  sup : (s : S) → (P s → W S P) → W S P
```

W-types

The trick lies in the following reasoning.

$$\begin{aligned} F_A X &\cong 1 + A \times X \\ &\cong 1 \times 1 + A \times X \\ &\cong 1 \times X^{\bot} + A \times X^{1} \end{aligned}$$

Which gives us a shape functor $S = 1 + A$ and a positioning functor $P = [\text{const } \bot, \text{const } 1]$ such that $W\ S\ P$ is isomorphic to $\mu X.F_A\ X$, where W is defined as:

```
data W (S : Set)(P : S → Set) : Set where
  sup : (s : S) → (P s → W S P) → W S P
```

Catas as W-recursion

W-types come equipped with a recursion principle:

$$
\begin{aligned}
&\mathsf{W\text{-}rec} : \forall\{c\}\{S : \mathsf{Set}\}\{P : S \to \mathsf{Set}\} \\
&\quad \to \{C : \mathsf{W}\ S\ P \to \mathsf{Set}\ c\} \\
&\quad \to (c : (s : S) \to (f : P\ s \to \mathsf{W}\ S\ P) \\
&\qquad \to (h : (p : P\ s) \to C\ (f\ p)) \\
&\qquad \to C\ (\mathsf{sup}\ s\ f) \\
&\quad) \to (x : \mathsf{W}\ S\ P) \to C\ x
\end{aligned}
$$

Which, upon choosing $C = A \to \mathsf{Set}$ gives us:

$$
\begin{aligned}
&\mathsf{W\text{-}rec\text{-}rel} : \{S : \mathsf{Set}\}\{P : S \to \mathsf{Set}\}\{A : \mathsf{Set}\} \\
&\quad \to ((s : S) \to (p : P\ s \to \mathsf{W}\ S\ P) \to \mathsf{Rel}\ (\mathsf{W}\ S\ P)\ A \to A \to \mathsf{Set}) \\
&\quad \to \mathsf{Rel}\ (\mathsf{W}\ S\ P)\ A \\
&\mathsf{W\text{-}rec\text{-}rel}\ h\ a\ w = \mathsf{W\text{-}rec}\ (\lambda\ s\ p\ c \to h\ s\ p\ (\mathsf{W\text{-}rec\text{-}rel}\ h)\ a)\ w
\end{aligned}
$$

And this later W-rec-rel is what is exported as a catamorphism in our library. (plus a few conversions from a relational gene).

Relational Equality in the Intensional Theory of Types

Victor Cacciari Miraldo

University of Utrecht, the Netherlands
University of Minho, Portugal

victor.cacciari@gmail.com

September 29, 2015