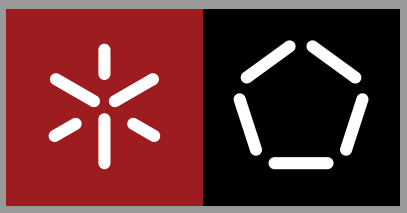
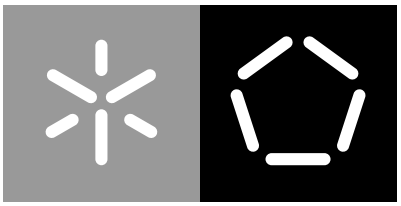


**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Victor Cacciari Miraldo

# Proof by Rewriting in Agda

January 2015



**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Victor Cacciari Miraldo

# Proof by Rewriting in Agda

Master dissertation
Master Degree in Computing Engineering

Dissertation supervised by
**José Nuno Oliveira (University of Minho)**
**Wouter Swierstra (University of Utrecht)**

January 2015

# DECLARATION

Following the rules of the *Dissertation* curricular unit of the Master in Computing Engineering (MEI) of the University of Minho, this document is submitted for evaluation as a Pre-Dissertation Report (RPD). Although already structured as a dissertation and following the template made available by the course staff, it is temporary in that it covers only the first part of the research work. This includes a review of the state of the art and identification of the main challenges to be addressed in the follow-up work. The last chapter includes the research plan which the candidate intends to follow towards making his contribution to the field which will be reported in the final submission of this dissertation.

# ACKNOWLEDGEMENTS

This dissertation and its underlying research are being conducted in the Netherlands. The author would like to thank the Erasmus+ programme for making this exchange possible.

# CONTENTS

# 1

# PRELUDE

## 1.1 INTRODUCTION

Although formal logic can be traced back to Aristotle, most of the groundbreaking work was done around the end of the 19th and early 20th centuries: the specification of propositional calculus; what we now know as predicate logic and other developments. The quest for bootstrapping mathematics, that is, formalizing mathematics in formal logic, was being pursued by many. A notorious attempt was made by Russel and Whitehead in *Principia Mathematica*, [26], where they believed that all mathematical truths could be derived from inference rules and axioms, therefore opening up the question of automated reasoning. Yet, in 1931, Kurt Gödel published his famous first and second incompleteness theorems. In a (very small) nutshell, they state that there are some truths that are not provable, regardless of the axiomatic system chosen. This question was further addressed by Alonzo Church and Alan Turing, in the late 1930s. This is when a definite notion of computability first arose (in fact they gave two, independent, definitions).

Armed with a formal notion of computation, mathematicians could finally start to explore this newly founded world, which we call Computing Science nowadays. Problems started to be categorized in different classes due to their complexity. In fact, some problems could now be proven to be *unsolvable*. Whenever we encounter such problem, we must work with approximations and subproblems that we know we can compute a solution for.

Given a formula in a logic system, the question of whether or not such a formula is true can vary from trivial to impossible. The simplest case is, of course, propositional logic, where validity is decidable but not at all that interesting for software verification in general. We need more expressive formal systems in order to encode software specifications, as they usually involve quantification or even modal aspects. A *holy grail* for formal verification would be the construction of a fully automatic theorem prover, which is very hard (if not impossible) to achieve. Instead, whenever the task requires an expressive system, we could only provide a guiding hand to our fellow mathematicians. This is what we call a *Proof Assistant*.

Proof Assistants are highly dependent on which logic they can understand. With the development of more expressive logics, comes the development of better proof assistants. Such tools are used to mitigate simple mistakes, automate trivial operations, and make sure the mathematician is not working on any incongruence. Besides the obvious verification of proofs, a proof assistant also opens up a lot of room for proof automation. By automating mechanical tasks in the development of critical software or models we can help the programmer to focus on what is really important rather than making he write boilerplate code that is mechanical in nature.

The tool of choice for this project is the Agda language, developed at Chalmers [21]. Agda uses a intensional variant of Martin-Löf's theory of types and provides a nice interactive construction feature. After the Curry-Howard isomorphism [12], interactive program construction and assisted theorem proving are essentially the same thing. The usual routine of an Agda programmer is to write some code, with some *holes* for the unfinished parts, and then ask the typechecker which types should such *holes* have. This is done interactively. Yet, trivial operations to write on paper usually require additional code for discharging in Agda. One such example is its failure to automatically recognize $i + 1$ and $1 + i$ as returning the same value. The usual strategy is to rewrite this subterm of our goal using a commutativity proof for $+$.

Small rewrites are quite simple to perform using the rewrite keyword. If we need to perform equational reasoning over complex formulas, though, we are going to need to specify the substitutions manually in order to apply a theorem to a subterm. The main objective of this project is to work around this limitation and provide a smarter rewriting mechanism for Agda. Our main case study is the equational proofs for relational algebra [5], which also involves the construction of a relational algebra library suited for rewriting.

## 1.2 THE AGDA LANGUAGE

Although functional languages have been receiving great attention and a fast evolution in recent years, one of the biggest jumps has been the introduction of dependent types. Agda[21] is one such language, other big contributions being *Epigram* [16] and *Coq* [4].

In languages such as Haskell or ML, where a Hindley-Milner based algorithm is used for type checking, values and types are clearly separated. Within dependent types, though, the story changes. A type can depend on any value the programmer wishes. Classical examples are the fixed size vectors `Vec A n`, where `A` is the type of the elements and `n` is the length of the vector. Readers familiar with C may argue that C also has fixed size arrays. The difference is that this `n` need not to be known at compile time: it can be an arbitrary term , as long as it has type `Nat`. That is, there is no difference between types and values, they are all sets, explained in the sequel.

This chapter introduces the basics of Agda and shows some examples in both the programming and the proof theoretic sides of the language. Later on, in section 2.2, we will see the theory in which Agda is built on top of, whereby a number of concepts introduced below will become clearer. Some background on the $\lambda$-calculus and the Curry-Howard isomorphism is presented in section 2.1.

### 1.2.1 *Peano Naturals*

In terms of programming, Agda and Haskell are very close. Agda's syntax was inspired by Haskell and, being also a functional language, a lot of the programming techniques we need to use in Haskell also apply for an Agda program. Knowledge of Haskell is not strictly necessary, but of a great value for a better understating of Agda.

The *Hello World* program in Agda is the encoding of Peano's Natural numbers, which follows:

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat
```

Data declarations in Agda are very similar to a GADTs[27] in Haskell. Remember that, in Agda, types and values are the same thing. The data declaration above states that Nat is *of type* Set. This Set is the type of types, to be addressed in more detail later. For now, think of it as Haskell's $*$ kind.

As expected, definitions are made by structural induction over the data type. Alas in Haskell, pattern matching is the mechanism of choice here.

```
_+_ : Nat → Nat → Nat
zero  + m = m
(succ n) + m = succ (n + m)

_*_ : Nat → Nat → Nat
zero  * _ = zero
(succ n) * m = m + (n * m)
```

There are a few points of interest in the above definitions. The underscore pattern (_) has the same meaning as in Haskell, it matches anything. The underscore in the symbol name, though, indicates where the parameters should be relative to the symbol name. Agda supports mixfix operators and has full UTF8 support. It is possible to apply a mixfix operator in normal infix form, for instance: a + b = _ + _ a b.

### 1.2.2 *Propositional Logic, a fragment*

Let us continue our exposition of Agda by encoding some propositional logic. First of all two sets are needed for representing the truth values:

```
data ⊤ : Set where
  tt : ⊤

data ⊥ : Set where

⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

The truth proposition has only one proof, and this proof is trivial. So we define $\top$ as a set with a single, constant, constructor. The absurd is modeled as a set with no constructors, therefore no elements. In the theory of types, this means that there is no proof for the proposition $\bot$, as expected. Note the definition of $\bot$-$elim$, whose type is $\bot \rightarrow A$, for all sets $A$[1]. This exactly captures the notion of proof by contradiction in logic. The definition is trickier, though. Agda's empty pattern, $()$, is used to discharge a contradiction. It tells that there is no possible pattern in such equation, and we are discharged of writing a right-hand side.

Let us now show how to encode a fragment of propositional logic in this framework. Taking conjunction as a first candidate, we begin by declaring the set that represents conjunctions:

```
data _/\_ (A B : Set) : Set where
  <_,_> : A → B → A /\ B
```

So, A /\ B contains elements with a proof of $A$ and a proof of $B$. Its elements can only be constructed with the <_,_> constructor. The reader familiar with Natural Deduction might recognize the following trivial properties. Given $D$ a set:

1. *Formation* rules for $D$ express the conditions under which $D$ is a set. In our example, whenever $A$ and $B$ are sets, then so is A /\ B.

2. *Introduction* rules for $D$ define the set $D$, that is, they specify how the canonical elements of $D$ are constructed. For the conjunction case, the elements of A /\ B have the form < a , b >, where $a \in A$ and $b \in B$.

3. *Elimination* rules show how to prove a proposition about an arbitrary element in $D$. They are very closely related to structural induction. The Agda equivalent is pattern matching, where we

1 Implicit parameters such as {A : Set} are enclosed in brackets. This can be read as universal quantification in the type level.

deconstruct, or eliminate, an element until we find the first constructor. Note that the proofs of /\-elim are simply pattern matching.

4. *Equality* rules give us the equalities which are associated with $D$. Without diving too much, in the simple case above, $< \mathsf{a}, \mathsf{b} >== < \mathsf{a}', \mathsf{b}' >$ whenever $\mathsf{a} == \mathsf{a}'$ and $\mathsf{b} == \mathsf{b}'$.

As detailed below in section 2.2, these properties come for free whenever we introduce a new set forming operation, that is, a data declaration in Agda. In fact, each set forming operation $D$ offers the four kinds or rules just given.

An analogous technique can be employed to encode disjunction. Note the similarity with Haskell's Either datatype (they are the same, in fact). The elimination rule for disjunction is a proof by cases.

```
data _\/_ (A B : Set) : Set where
  inl : A → A \/ B
  inr : B → A \/ B

\/-elim : {A B C : Set} → (A → C) → (B → C) → A \/ B → C
\/-elim p1 p2 (inl x) = p1 x
\/-elim p1 p2 (inr x) = p2 x
```

As a trivial example, we can prove that conjunction distributes over disjunction.

```
\/-/\-dist : {A B C : Set} → A /\ (B \/ C) → (A /\ B) \/ (A /\ C)
\/-/\-dist < x , y > = \/-elim (λ b → inl < x , b >) (λ c → inr < x , c >) y
```

Using this fragment of natural deduction, we can start proving things. Consider a simple proposition saying that an element of a nonempty list $l_1 ++ l_2$, where $++$ is concatenation, either is in $l_1$ or in $l_2$. For this all that is required is to: encode lists in Agda, provide a definition for $++$, encode a *view* of the elements in a list and finally prove our proposition.

Defining the type of lists is a boring task, and the definition is just like its Haskell counterpart. One can just import the definition from the standard library and go to the interesting part right away.

```
open import Data.List using (List; _::_; [])

_++_ : ∀{a}{A : Set a} → List A → List A → List A
[] ++ l = l
(x :: xs) ++ l = x :: (xs ++ l)

data In {A : Set} : A → List A → Set where
  InHead : {xs : List A}{x : A} → In x (x :: xs)
  InTail : {x : A}{xs : List A}{y : A} → In y xs → In y (x :: xs)
```

Looking closely to this code, we one finds a set forming operation In, that receives one implicit parameter $A$ and is indexed by an element of $A$ and a $List\ A$. Indeed, In $x\ l$ is the proposition that states that $x$ is an element of $l$. How is such a proof constructed? There are two ways of doing so. Either $y$ is in $l$'s head, as captured by the InHead constructor, or it is in $l$'s tail, and a proof of such statement is required, in the InTail constructor. The reader may wonder about the statement In $y$ [], which should be impossible to construct. One can see, just from the types, that we cannot construct such a statement. The result of both In constructors involve non-empty lists and they are the only constructors we are allowed to use. So, the *non-empty list* requirement in the proposition we are trying to prove comes for free.

We proceed to showing how to state the proposition mentioned above and carry out its proof step by step, as this is an interesting example. The proposition is stated as follows:

```
inDistr : {A : Set}(l₁ l₂ : List A)(x : A)
  → In x (l₁ ++ l₂) → In x l₁ ∨ In x l₂
```

The variables enclosed in normal parentheses are called *telescopes*, and they introduce the dependent function type $(x : A) \to B\ x$, in general, $x$ can appear on the right-hand side of the function type constructor, $\to$. In this proof, we need an implicit set $A$, two lists of $A$, an element of $A$ and a proof that such element is in the concatenation of the two lists. The proof follows by *induction* on the first list.

```
inDistr [] l₂ x prf = inr prf
```

If the first list is empty, $[] ++ l_2$ reduces to $l_2$. So, prf has type In $x\ l_2$, which is just what we need to create a disjunction. If not empty, though, then it has a head and a tail:

```
inDistr (x :: l) l₂ .x InHead = inl InHead
```

Now we have to also pattern-match on $prf$. If it says that our element is in the head of our list, the result is also very simple. The repetition of the symbol $x$ in the left hand side is allowed as long as all duplicate occurrences are preceded by a dot. These are called dotted patterns and tell Agda that this is *the only possible* value for that pattern. We can see this by reasoning with InHead type, In $x\ (x :: l)$. So, if $l_1 = (x :: l)$ and we have $prf = In\ x\ (x :: l)$, that is, a proof that $x$ is in $l_1$'s head, we can only be looking for $x$.

```
inDistr (x :: l₁) l₂ y (InTail i)
  = ∨-elim (λ a → inl (InTail a)) inr (inDistr l₁ l₂ y i)
```

In case our element is not in $l_1$'s head, it might be either in the rest of $l_1$ or in $l_2$. Note that we pass a *structurally smaller* proof to the recursive call. This is a condition required by Agda's termination checker.

### 1.2.3 *Closing Remarks*

The small introduction to Agda given above tells a tiny bit of what the language is capable of, both in its programming side and its proof assistant side. All the concepts were introduced informally here, with the intention of not overwhelming a unfamiliar reader. A lot of resources are available in the Internet at the Agda Wiki page [1].

## 1.3 THE PROBLEM

As seen in section 1.2, Agda is a very expressive language and it allows us to build smaller proofs than in the great majority of proof assistants available. The mixfix feature gives the language a very customizable feel, one application being the equational reasoning framework. In the following illustration of propositional equality we prove the associativity of the concatenation operation.

```
open import Relation.Binary.PropositionalEquality
open ≡-Reasoning

++-assocH : ∀{a}{A : Set a}(xs ys zs : List A) →
      (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assocH [] ys zs = refl
++-assocH (x :: xs) ys zs =
  begin
    ((x :: xs) ++ ys) ++ zs
  ≡⟨ refl ⟩
    x :: (xs ++ ys) ++ zs
  ≡⟨ refl ⟩
    x :: ((xs ++ ys) ++ zs)
  ≡⟨ cong (_::_ x) (++-assocH xs ys zs) ⟩
    x :: (xs ++ (ys ++ zs))
  ≡⟨ refl ⟩
    (x :: xs) ++ (ys ++ zs)
  □
```

The notation is clear and understandable, indeed looking very much what a *squiggolist*[2] would write on paper. One of the main downsides to it, which is also inherent to Agda in general, is the need to specify every single detail of the proof, even the trivial ones. Note the trivial, yet explicit, $(\_::\_x)$ congruence being stated.

Aiming somewhat higher-level, we could actually generalize the congruences to substitutions, as long as the underlying equality exhibits a substitutive behavior. We can borrow a excerpt from a relational proof that the relation twice, defined by $(a, 2 \times a) \in \mathit{twice}$ for all $a \in \mathbb{N}$, preserves even numbers. The actual code is much larger and will be omitted, as this is for illustration purposes only:

```
twiceIsEven : (twiceR • evenR ⊆ evenR • twiceR) ⇐ Unit
twiceIsEven
  = begin

    twiceR • evenR ⊆ evenR • twiceR

  ⇐⟨ substitute (λ x → twiceR • evenR ⊆ x • twiceR) evenLemma ⟩

    twiceR • evenR ⊆ ρ twiceR • twiceR

  ⇐⟨ substitute (λ x → twiceR • evenR ⊆ x) (ρ-intro twiceR) ⟩

    twiceR • evenR ⊆ twiceR

  ⇐⟨ substitute (λ x → twiceR • evenR ⊆ x) (≡r-sym (•-id-r twiceR)) ⟩

    twiceR • evenR ⊆ twiceR • Id

  ⇐⟨ •-monotony ⟩

    (twiceR ⊆ twiceR × evenR ⊆ Id)

  ⇐⟨ (λ _ → ⊆-refl , ϕ⊆Id) ⟩

    Unit
  □
```

Besides the obvious Agda boilerplate, it is simple to see how the substitutive[3] behavior of Relational Equality can become a burden to write in every single step, nevertheless such factor is also what allows us to rewrite arbitrary terms in a formula. The main idea is to provide an automatic mechanism

2 *Squiggol* is a slang name for the Bird–Meertens formalism[5], due to the squiggly symbols it uses.

3 As we shall see later, this is not the case, and this example was heavily modified due to the encoding of relational equality not being a substitutive relation in Agda.

to infer the first argument for the substitute function, making equational reasoning much cleaner in Agda.

Alongside the development of such functionality, based on Agda's reflection capabilities (that is, to access and modify a program AST in compile time), we also want to develop a library for Relational Algebra, focused on its equational reasoning aspect. Both tasks have to walk with hands tied, since their design is mutually dependent as illustrated later.

The work documented by this thesis is, therefore, split into two main tasks:

1. Provide an Agda library for Relational Algebra that is suitable for
2. Equational Reasoning with automatic substitution inference.

It is worth mentioning that although Relations are our main case study, we want our substitution inference to work independently of the proof context. For instance, a very direct application would be to facilitate the calculation of extended static checking, as in [22].

## 1.4 STRUCTURE OF THE DISSERTATION

This document is a preliminary dissertation and only describes half of our research plan. In Chapter 2 the basic mathematical notions for a better understanting of Agda will be introduced. Later on, in chapter 3, the implementation of a Relational Algebra library will be discussed. The problems we faced, which are relative both to the Agda language and the model that was chosen for relations, will also be presented together with the solutions that were found so far. Chapter 4 contains a description of what is planned for the next six months of research.

# 2

# BACKGROUND

This chapter is concerned with background concepts. The idea is to keep this dissertation as mathematically self contained as possible, although for a full understanding of this kind of topics, further reading is necessary. Some very important basic ideasare introduced below that provide the underlying foundations of proof assistants such as Agda.

As already mentioned in section 1.2, Agda is both a programming language and a proof assistant. The programming side of Agda is a pure functional language and is built on top of the (dependently typed) $\lambda$-calculus. Such a formalism will be briefly presented in section 2.1. The proof assistant view, on the other hand, lies on top of the Curry-Howard isomorphism, which will be presented in section 2.1.4. Yet, Agda seems to be more general than the Curry-Howard isomorphism since it can handle first order logic out of the box. This generalization is explained in 2.2, where a surface description of Martin Löf's theory of types is given.

The reader familiar with these topics can safely skip this chapter.

## 2.1 NOTES ON $\lambda$-CALCULUS AND TYPES

What is known as the $\lambda$-calculus is a collection of various formal systems based on the notation invented by Alonzo Church in [8, 7]. Church solved the famous *Entscheidungsproblem* (the German for *decision problem*) proposed by David Hilbert, in 1928. The challenge consisted in providing an algorithm capable of determining whether or not a given mathematical fact was valid in a given language. Church proved that there is no solution for such problem, that is, it is an undecidable problem.

One of Church's main objectives was to build a formal system for the foundations of Mathematics, just like Martin-Löf's type theory, which was to be presented much later, around 1970. Church dropped his work when his basis was found to be inconsistent. Later on it was found that there were ways of making it consistent again, with the help of types.

The notion of type is paramount for this dissertation as a whole. This notion arises when we want to combine different terms in a given language. For instance, it makes no sense to try to compute $\int \mathbb{N}\ \mathrm{d}x$. Although syntactically correct, the subterms have different *types* and, therefore, are not compatible. A type can be seen as a categorization of terms.

For a thorough introduction of the lambda-calculus, the reader is directed to [2, 11]. The goal of this chapter is to provide a minimal understanting of the $\lambda$-calculus, which will allow a better understanding of Martin-Löf's type theory and how logic is encoded in such formalisms.

### 2.1.1 *The $\lambda$-calculus*

**Definition 2.1** (Lambda-terms)**.** Let $\mathcal{V} = \{v_1, v_2, \cdots\}$ be a infinite set of variables, $\mathcal{C} = \{c_1, c_2, \cdots\}$ a set of constants such that $\mathcal{V} \cap \mathcal{C} = \emptyset$. The set $\Lambda$ of lambda-terms is inductively defined by:

ATOMS

$$\mathcal{V} \cup \mathcal{C} \subset \Lambda$$

APPLICATION

$$\forall M, N \in \Lambda.\ (MN) \in \Lambda$$

ABSTRACTION

$$\forall M \in \Lambda, x \in \mathcal{V}.\ (\lambda x.M) \in \Lambda$$

Let us adopt some conventions that will be useful throughout this document. Terms will usually be denoted by uppercase letters $M, N, O, P, \cdots$ and variables by lower cases $x, y, z, \cdots$. Application is left associative, that is, the term $MNO$ represents $((MN)O)$ whereas abstractions are right associative, so, $\lambda xy.K$ represents $(\lambda x.(\lambda y.K))$.

Suppose we have a term $\lambda yz.x(yz)$. We say that variables $y$ and $z$ are bounded variables and $x$ is a free variable (as there is no visible abstraction binding it). From now on, we'll use Barendregt's convention for variables and assume that terms do not have any name clashing. In fact, whenever we have two terms $M$ and $N$ that only differ in the naming of their variables, for instance $\lambda x.x$ and $\lambda y.y$, we say that they are $\alpha$-convertible.

**Definition 2.2** (Substitution)**.** Let $M$ and $N$ be lambda-terms where $x$ has a free occurrence in $M$. The substitution of $x$ by $N$ in $M$, denoted by $[N/x]M$ is, informally, the result of replacing every free occurrence of $x$ in $M$ by $N$. It is defined by induction on $M$ by:

$$\begin{aligned} [N/x]x &= N \\ [N/x]y &= y,\ y \neq x \\ [N/x](M_1 M_2) &= [N/x]M_1\ [N/x]M_2 \\ [N/x](\lambda y.M) &= (\lambda y.[N/x]M) \end{aligned}$$

### 2.1.2 *Beta reduction and Confluence*

Equipped with both a notion of term and a formal definition of substitution, we can now model the notion of computation. The intuitive meaning is very simple. Imagine a normal function $f(x) = x+3$ and suppose we want to compute $f(2)$. All we have to do is to substitute $x$ for 2 in the body of $f(x)$, resulting in $2+3$.

This notion is followed to the letter in the $\lambda$-calculus. A term with the form $(\lambda x.M)N$ is called a $\beta$-redex and can be reduced to $[N/x]M$. If a given term has no $\beta$-redexes we say it is in $\beta$-normal form.

**Definition 2.3** ($\beta$-reduction)**.** Let $M, M'$ and $N$ be lambda-terms and $x$ a variable. Let $\rightarrow_\beta$ be the following binary relation over $\Lambda$ defined by induction on $M$.

$$\frac{}{(\lambda x.M)N \rightarrow_\beta [N/x]M} \qquad \frac{M \rightarrow_\beta M'}{(\lambda x.M) \rightarrow_\beta (\lambda x.M')}$$

$$\frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N} \qquad \frac{M \rightarrow_\beta M'}{NM \rightarrow_\beta NM'}$$

We use the notation by $M \twoheadrightarrow_\beta N$ when $N$ is obtained through zero or more $\beta$-reductions from $M$.

**Definition 2.4** ($\beta$-equality)**.** Let $M$ and $N$ be lambda terms, $M$ and $N$ are said to be $\beta$-equal, and denote this by $M =_\beta N$ if $M \twoheadrightarrow_\beta N$ or $N \twoheadrightarrow_\beta M$.

**Theorem 2.1** (Confluency)**.** *Let $M, N_1$ and $N_2$ be lambda terms. If $M \rightarrow_\beta N_1$ and $M \rightarrow_\beta N_2$ then there exists a term $Z$ such that $N_i \twoheadrightarrow_\beta Z$, for $i \in \{1,2\}$.*

**Theorem 2.2** (Church-Rosser)**.** *Let $M$ and $N$ be lambda-terms such that $M =_\beta N$. Then there exists a term $Z$ such that $M \twoheadrightarrow_\beta Z$ and $N \twoheadrightarrow_\beta Z$.*

Note that the aforementioned results are of enormous relevance not only for the $\lambda$-calculus, but for similar formalisms too. They allow us to prove that, for instance, the normal form of a lambda-term (if it exists[1]) is unique. In fact, lambda-calculus consistency is proved using these results [2].

### 2.1.3 *Simply typed $\lambda$-calculus*

**Definition 2.5** (Type)**.** Given $\mathcal{C}_\mathcal{T} = \{\sigma, \sigma', \cdots\}$ a set of atomic types, we define the set $\mathbb{T}$ of simple types by induction, as the least set built by the clauses:

1. $\mathcal{C}_\mathcal{T} \subset \mathbb{T}$
2. $\forall \sigma, \tau \in \mathbb{T}.\ (\sigma \to \tau) \in \mathbb{T}$.

In any programming context, one is always surrounded by variable declarations. Some languages (the strongly-typed ones) expect some information about the type of such variables. This is what we call a *context*. Formally, a context is a set $\Gamma \subseteq \mathcal{V} \times \mathbb{T}$, whose elements are denoted by $(x : \sigma)$.

This allows us to define the notions of derivation and derivability, almost closing the gap between programming and logic.

**Definition 2.6** (Derivation)**.** We define the set of all type derivations by induction in the target lambda-term:

1. $$\frac{}{\Gamma \vdash x : \sigma}\ (Ax)$$

2. $$\frac{\begin{array}{c}\vdots\\ \Gamma, x : \tau \vdash M : \sigma\end{array}}{\Gamma \vdash (\lambda x.M) : (\tau \to \sigma)}\ (I \to)$$

3. $$\frac{\begin{array}{c}\vdots\\ \Gamma \vdash M : (\tau \to \sigma)\end{array} \quad \begin{array}{c}\vdots\\ \Gamma \vdash N : \tau\end{array}}{\Gamma \vdash MN : \sigma}\ (E \to)$$

**Definition 2.7** (Derivability)**.** Let $\Gamma$ be a context, $M$ a lambda-term and $\sigma$ a type. We say that the sequent $\Gamma \vdash M : \sigma$ is derivable if there exists a derivation with such sequent as its conclusion.

1 There are terms that do not have a normal form. A classical example is $(\lambda x.xx)(\lambda x.xx)$. The reader is invited to compute a few $\beta$-reductions on it.

The simply-typed $\lambda$-calculus is a model of computation. It has the same expressive power as the Turing Machine for expressing computability notions. This is a very well studied subject and the references provided in this chapter are a compilation of everything that has been studied so far. For more typed variations of the $\lambda$-calculus we refer the reader to [3]. We're not interested in that aspect of the $\lambda$-calculus, though. We rather want to explore the connection with Mathematical logic.

### 2.1.4 *The Curry-Howard Isomorphism*

On one hand we have the models of computation, on the other hand we have the proof systems. A a first glance, they look like very different formalism's, but they turned out to be structurally the same. Let $M$ be a term and $\Gamma$ a context such that $\Gamma \vdash M : \sigma$ is derivable. We can look at $\sigma$ as a propositional formula[2] and to $M$ as a proof of such formula. There are other ways to show this connection, but we will illustrate it using the Natural Deduction[23] (propositional implication will be denoted by $\supset$). Let's put the rules presented in definition 2.6 side-by-side with the Axiom, $\supset$-elimination and $\supset$-introduction rules from Natural Deduction;

| Natural Deduction | Type Derivation |
|---|---|
| $\sigma$ | $\dfrac{}{\Gamma \vdash M : \sigma}\ (Ax)$ |
| $\dfrac{\begin{array}{c}[\tau]\\ \vdots \\ \sigma\end{array}}{\tau \supset \sigma}\ (I\supset)$ | $\dfrac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash (\lambda x.M) : (\tau \to \sigma)}\ (I\to)$ |
| $\dfrac{\tau \supset \sigma \quad \tau}{\sigma}\ (E\supset)$ | $\dfrac{\Gamma \vdash M : (\tau \to \sigma) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \sigma}\ (E\to)$ |

This seemingly shallow equivalence is a remarkable result in Computing Science, discovered by Curry and Howard in [9, 12]. This was the starting point for the first proof checkers, since checking a proof is the same as typing a lambda-term. If the term is typeable, then the proof is valid. This far we have only presented the simpler version of this connection. Another layer will be built on top of it and add all ingredients for working over first-order logic, in the next section. Behind the curtains, all Agda does is type-checking terms.

2 Remember that the implication, here denoted by $\subset$, forms a minimal complete connective set and is, therefore, enough to express the whole propositional logic.

The understanding of this connection is of major importance for writing proofs and programs in Agda (or any other proof-assistant based on the Curry-Howard isomorphism, for that mater).

## 2.2 MARTIN-LÖF'S TYPE THEORY

Type theory was originally developed with the goal of offering a clarification, or basis, for constructive Mathematics. However, unlike most other formalizations of mathematics, it is not based on first order logic. Therefore, we need to introduce the symbols and rules we'll use before presenting the theory itself. The heart of this interpretation of proofs as programs is the Curry-Howard isomorphism, already explained in section 2.1.

Martin-Löf's theory of types [14] is an extension of regular type theory. This extended interpretation includes universal and existential quantification. A proposition is interpreted as a set whose elements are proofs of such proposition. Therefore, any true proposition is a non-empty set and any false proposition is a empty set, meaning that there is no proof for such proposition. Apart from *sets as propositions*, we can look at sets from a *specification* angle, and this is the most interesting view for programming. A given element $a$ of a set $A$ can be viewed as: a proof for proposition $A$; a program satisfying the specification $A$; or even a solution to problem $A$.

This chapter we'll explain the basics of the theory of types (in its *intensional* variation) trying to establish connections with the Agda language. It begins by providing some basic notions and the interpretation of propositional logic into set theory. We'll follow with the notion of arity, which differs from the canonical meaning, finishing with a small discussion on the dependent product and sums operators, which closes the gap to first order logic. The interested reader should continue with [20] or, for a more practical view, [24, 6]

### 2.2.1 *Constructive Mathematics*

The line between Computer Science and Constructive Mathematics is very thin. The primitive object is the notion of a function from a set $A$ to a set $B$. Such function can be viewed as a program that, when applied to an element $a \in A$ will construct an element $b \in B$. This means that every function we use in constructive mathematics is computable.

Using the constructive mindset to prove things is also very closely related to building a computer program. That is, to prove a proposition $\forall x_1, x_2 \in A \,.\, \exists y \in B \,.\, Q(x_1, x_2, y)$ for a given predicate $Q$ is to give a function that when applied to two elements $a_1, a_2$ of $A$ will give an element $b$ in $B$ such that $Q(a_1, a_2, b)$ holds.

### 2.2.2 *Propositions as Sets*

In classical mathematics, a proposition is thought of as being either true or false, and it doesn't matter if we can prove or disprove it. On a different angle, a proposition is constructively true if we have a *method* for proving it. A classical example is the law of excluded middle, $A \vee \neg A$, which is trivially true since $A$ can only be true or false. Constructively, though, a method for proving a disjunction must prove that one of the disjuncts holds. Since we cannot prove an arbitrary proposition $A$, we have no proof for $A \vee \neg A$.

Therefore, we have that the constructive explanation of propositions is built in terms of proofs, and not an independent mathematical object. The interpretation we are going to present here is due to Heyting at [10].

ABSURD, $\bot$, is identified with the empty set, $\emptyset$. That is, a set with no elements or a proposition with no proof.

IMPLICATION, $A \supset B$ is viewed as the set of functions from $A$ to $B$, denoted $B^A$. That is, a proof of $A \supset B$ is a function that, given a proof of $A$, returns a proof of $B$.

CONJUNCTION, $A \wedge B$ is identified with the cartesian product $A \times B$. That is, a proof of $A \wedge B$ is a pair whose first component is a proof of $A$ and second component is a proof of $B$. Let us denote the first and second projections of a given pair by $\pi_1$ and $\pi_2$. The elements of $A \times B$ are of the form $(a, b)$, where $a \in A$ and $b \in B$.

DISJUNCTION, $A \vee B$ is identified with the disjoint union $A + B$. A proof of $A \vee B$ is either a proof of $A$ or a proof of $B$. The elements of $A + B$ are of the form $i_1\ a$ and $i_2\ b$ with $a \in A$ and $b \in B$.

NEGATION, $\neg A$, can be identified relying on its definition on the minimal logic, $A \supset \bot$

So far, we defined propositional logic using sets (types) that are available in almost every programming language. Quantifications, though, require operations defined over a family of sets, possibly *depending* on a given *value*. The intuitionist explanation of the existential quantifier is as follows:

EXISTS, $\exists a \in A\ .\ P(a)$ consists of a pair whose first component is one element $i \in A$ and whose second component is a proof of $P(i)$. More generally, we can identify it with the disjoint union of a

family of sets, denoted by $\Sigma(x \in A, B(x))$, or just $\Sigma(A, B)$. The elements of $\Sigma(A, B)$ are of the form $(a, b)$ where $a \in A$ and $b \in P(a)$.

FOR ALL, $\forall a \in A \,.\, P(a)$ is a function that gives a proof of $P(a)$ for each $a \in A$ given as input. The correspondent set is the cartesian product of a family of sets $\Pi(x \in A, B(x))$. The elements of such set are the aforementioned functions. The same notation simplification takes place here, and we denote it by $\Pi(A, B)$. The elements of such set are of the form $\lambda x \,.\, b(x)$ where $b(x) \in B(x)$ for $x \in A$.

### 2.2.3 *Expressions*

Thus far now we have identified the sets needed to express first order formulas, but we did not mention what an expression is. In fact, in the theory of types, an expression is a very abstract notion. We are going to define the set $\mathcal{E}$ of all expressions by induction shortly.

It is worth remembering that Martin-Löf's theory of types was intended to be a foundation for mathematics. It makes sense, therefore, to base our definitions in standard mathematical expressions. For instance, consider the syntax of an expression in the CCS calculus [17] $E \sim \Sigma\{a.E' \mid E \xrightarrow{a} E'\}$[3]; we have a large range of syntactical elements that we don't usually think about when writing mathematics on paper. For instance, the variables $a$ and $E'$ works just as placeholders, which are *abstractions* created at the predicate level in the common set-by-comprehension syntax. We then have the *application* of a summation $\Sigma$ to the resulting set, where this symbol represents a *built-in* operation, just like the congruence $\_ \sim \_$. These are exactly the elements that will allow us to build expressions in the theory of types.

**Definition 2.8** (Expressions)**.**

APPLICATION; Let $k, e_1, \cdots, e_n \in \mathcal{E}$ be expressions of suitable arity. The application of $k$ to $e_1, \cdots, e_n$ denoted by $(k\ e_1\ \cdots\ e_n)$, is also an expression.

ABSTRACTION; Let $e \in \mathcal{E}$ be an expression with free occurrences of a variable $x$. We denote by $(x)e$ the expression where every free occurrence of $x$ is interpreted as a hole, or context. So, $(x)e \in \mathcal{E}$.

3 Expansion Lemma for CCS; States that every process (roughly a LTS) is equivalent to the sum of its derivatives. The notation $E \xrightarrow{a} E'$ states a transition from $E$ to $E'$ through label $a$.

COMBINATIONS; Expressions can also be formed by combination. This is a less common construct. Let $e_1, \cdots, e_n \in \mathcal{E}$, we may form the expression:

$$e_1, e_2, \cdots, e_n$$

which is the combination of the $e_i$, $i \in \{1, \cdots, n\}$. This can be though of tupling things together. Its main use is for handling complex objects as part of the theory. Consider the scenario where one has a set $A$, an associative closed operation $\oplus$ and a distinguished element $a \in A$, neutral for $\oplus$. We could talk about the monoid as an expression: $A, \oplus, a$.

SELECTION; Of course, if we can tuple things together, it makes sense to be able to tear things apart too. Let $e \in \mathcal{E}$ be a combination with $n$ elements. Then, for all $i \in \{1, \cdots, n\}$, we have the selection $e.i \in \mathcal{E}$ of the $i$-th component of $e$.

BUILT-IN'S; The built-in expressions are what makes the theory shine. Yet, they change according to the flavor of the theory of types one is handling. They typically include *zero* and *succ* for Peano's encoding of the Naturals, together with a recursion principle *natrec*; Or *nil*, *cons* and *listrec* for lists; Products and Coproduct constructors are also built-ins: $\langle \_, \_ \rangle$ and *inl*, *inr* respectively.

**Definition 2.9** (Definitional Equality)**.** Given two expressions $d, e \in \mathcal{E}$, Should they be syntactical synonyms, we say that they are *definitionally* or *intensionally* equal. This is denoted by $d \equiv e$.

From the expression definition rules, we can see a few equalities arising:

$$\begin{aligned} e &\equiv ((x)e)\ x \\ e_i &\equiv (e_1, \cdots, e_n).i \end{aligned}$$

As probably noticed, we mentioned *expressions of suitable arity*, but did not explain what arity means. The notion of arity is somewhat different from what one would expect, it can be seen as a *meta-type* of the expression, and indicates which expressions can be combined together.

Expressions are divided in a couple classes. It is either *combined*, from which we might select components from it, or it is *single*. In addition, an expression can be either *saturated* or *unsaturated*. A single saturated expression have arity **0**, therefore, neither selection nor application can be performed. Unsaturated expressions on the other hand, have arities of the form $\alpha \twoheadrightarrow \beta$, where $\alpha$ and $\beta$ are themselves arities. The informal meaning of such arity is *"give me an expression of arity $\alpha$ and I give an expression of arity $\beta$"*, just like normal Haskell types.

For example, the built in *succ* has arity $\mathbf{0} \twoheadrightarrow \mathbf{0}$; the list constructor *cons* has arity $(\mathbf{0} \otimes \mathbf{0}) \twoheadrightarrow \mathbf{0}$, since it takes two expressions of arity $\mathbf{0}$ and returns an expression. We will not go into much more detail on arities. It is easy to see how they are inductively defined. We refer the reader to chapter 3 of [20].

Evaluation of expressions in the theory of types is performed in a lazy fashion, the semantics being based in the notion of *canonical expression*. These canonical expressions are the values of programs and, for each set, they have different formation conditions. The common property is that they must be closed and saturated. It is closely related to the weak-head normal form concept in lambda calculus, as illustrated in table 1.

| Canonical | Noncanonical | Evaluated | Fully Evaluated |
|---|---|---|---|
| 12 | $fst\ \langle a, b\rangle$ | $succ\ zero$ | $true$ |
| $false$ | $3 \times 3$ | $succ\ (3+3)$ | $succ\ zero$ |
| $(\lambda x.x)$ | $(\lambda x.snd\ x)p$ | $cons(4, app(nil, nil))$ | 4 |

Table 1: Expression Evaluation State

### 2.2.4 *Judgement Forms*

Standing on top of the basic constructors of the theory of types, we can start to discuss what kind of judgement forms we can express and derive. In fact, Agda boils down to a tool that does not allow us to make incorrect derivations. Type theory provides us with derivational rules to discuss the validity of judgements of the form given in table 2

| | Sets | Propositions |
|---|---|---|
| *(i)* | $A$ is a set. | $A$ is a proposition. |
| *(ii)* | $A$ and $B$ are equal sets. | $A$ and $B$ are equivalent propositions. |
| *(iii)* | $a$ is an element of a set $A$. | $a$ is a proof of $A$. |
| *(iv)* | $a$ and $b$ are equal elements of a set $A$. | $a$ and $b$ are the same proof. |

Table 2: judgement Forms

When reading a set as a proposition, we might simplify $(iii)$ to *A is true*, disregarding the proof. What matters is the existence of the proof.

But then, let's take $(i)$ as an example. What does it means to be a set? To know that $A$ is a set is to know how to form the canonical elements of $A$ and under what conditions two canonical elements are equal. Therefore, to construct a set, we need to give a syntactical description of its canonical elements

and provide means to decide whether or not two canonical elements are equal. Let us take another look at the Peano Naturals:

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat
```

In fact defines a Set, whose canonical elements are either zero or succ applied to something. Equality in Nat is indeed decidable, therefore we have a set in the theory of types sense.

The third form of judgement might also be slightly tricky. Given that $A$ is a set, to know that $a$ is an element of $A$ amounts to knowing that, when evaluated, $a$ yields a canonical element in $A$ as value. Making the parallel to Agda again, this is what allows one to pattern match on terms.

To know that two sets are equal is to know that they have the same canonical elements, and equal canonical elements in $A$ are also equal canonical elements in $B$. Two elements are equal in a set when their evaluation yields equal canonical elements.

For a proper presentation of the theory of types, we should generalize these judgement forms to cover hypotheses. As this is not required for a stable understanding of Agda. we refer the reader to Martin-Löf's thesis [14, 15].

### 2.2.5 *General Rules*

A thorough explanation of the rules for the theory of types is outside the scope of this section. This is just a simple illustration that might help unfamiliar readers to grasp Agda with some comfort, so that they get a taste of what is happening behind the curtains. For this, it is enough to present the general notion of rule, their syntax and give a simple example.

In the theory of types there are some general rules regarding equality and substitution, which can be justified by the defined semantics. Then, for each set forming operation, there are rules for reasoning about the set and its elements. These foremost classes of rules are divided into four kinds:

1. *Formation* rules for $A$ describe the conditions under which $A$ is a set and when another set $B$ is equal to $A$.
2. *Introduction* rules for $A$ are used to construct the canonical elements. They describe how they are formed and when two canonical elements of $A$ are equal.
3. *Elimination* rules act as a *structural induction principle* for elements of $A$. They allow us to prove propositions about arbitrary elements.

4. *Equality* rules gives us the equalities which are associated with $A$.

The syntax will follow a natural deduction style. For example:

$$\frac{A \text{ set} \quad B(x) \text{ set } [x \in A]}{\Pi(A, B) \text{ set}}$$

The rule represents the formation rule for $\Pi$, and can be applied whenever $A$ and $B(x)$ are sets, assuming that $x \in A$. judgements may take the form $p = a \in A$, meaning that if we compute the program $p$, we get the canonical element $a \in A$ as a result.

So far so good. But how do we use all these rules and interpretations? Let us to take the remaining fog out of how Martin-Löf's theory *is* Agda with the next example. For this, we need the set of booleans and the set of intentional equality.

*Boolean Values*

The set $\{true, false\}$ of boolean values is nothing more than an enumeration set. There are generic rules to handle enumeration sets. For simplicity's sake we are going to use an instantiated version of such rules.

$$\frac{}{\mathbb{B} \text{ set}}\ \mathbb{B}_{form} \qquad \frac{}{true \in \mathbb{B}}\ \mathbb{B}_{intro_1} \qquad \frac{}{false \in \mathbb{B}}\ \mathbb{B}_{intro_2}$$

$$\frac{C(b) \text{ set } [b \in \mathbb{B}] \quad c \in C(true) \quad d \in C(false)}{(true\ ?\ c\ :\ d) = c \in C(true)}\ \mathbb{B}_{eq_1}$$

$$\frac{C(b) \text{ set } [b \in \mathbb{B}] \quad c \in C(true) \quad d \in C(false)}{(false\ ?\ c\ :\ d) = d \in C(false)}\ \mathbb{B}_{eq_2}$$

$$\frac{b \in \mathbb{B} \quad C(v) \text{ set } [v \in \mathbb{B}] \quad c \in C(true) \quad d \in C(false)}{(b\ ?\ c\ :\ d) \in C(b)}\ \mathbb{B}_{elim}$$

*Intensional Equality*

The set $Id(A, p, a)$, for $Id$ a primitive constant (with arity $\mathbf{0} \otimes \mathbf{0} \otimes \mathbf{0} \twoheadrightarrow \mathbf{0}$), represents the judgement $p = a \in A$. Put into words, it means that the program $p$ evaluates to $a$, which is a canonical element of $A$. The elimination and equality rules for $Id$ will not be presented here.

$$\frac{A \text{ set} \quad a \in A \quad b \in A}{Id(A,a,b) \text{ set}}\ Id_{form}$$

$$\frac{a \in A}{id(a) \in Id(A,a,a)}\ Id_{intro_1} \qquad \frac{a = b \in A}{id(a) \in Id(A,a,b)}\ Id_{intro_2}$$

From the introduction rules, we should be able to tell which are the canonical elements of $Id$. As an exercise, the reader should compare the set $Id$ with Agda's *Relation.Binary.PropositionalEquality*[4].

### 2.2.6 *Epilogue*

At this point we have briefly discussed the Curry-Howard isomorphism in both the simply-typed version and the dependently typed flavor. We have presented (the very surface of) the theory of types and which kind of logical judgements we can handle with it. But how does all this connect to Agda and verification in general?

Software Verification with a functional language is somewhat different from doing the same with a imperative language. If one is using C, for example, one would write the program and annotate it with logical expressions. They are twofold. We can use pre-conditions, post-conditions and invariants to deductively verify a program (these are in fact a variation of Hoare's logic). Or, we can use Software Model Checking, proving that for some bound of traces $k$, our formulas are satisfied.

When we arrive at the functional realm, our code is correct *by construction*. The type-system provides both the *annotation language* and you can construct a program that either type-checks, therefore respects its specification, or does not pass through the compiler. Remember that Agda's type system is the intensional variant of Martin-Löf's theory of types, which was presented above in this chapter.

Let us now prove that, for any set $A$ and $a \in A$, given a $b \in \mathbb{B}$ we have that $(b\ ?\ a\ :\ a) = a$. Translating it to a more formal version, we want to prove that:

$$Id(A, (b\ ?\ a\ :\ a), a)\ [b \in \mathbb{B}, a \in A] \text{ is inhabited.}$$

Well, it suffices to show the existence of some element in the aforementioned set.

Let us denote the set $Id(A, (x\ ?\ a\ :\ a), a)$ by $\phi(x)$, for any given set $A$ and element $a \in A$. The (partially ommited) derivation follows.

4 Available from the Agda Standard Library: https://github.com/agda/agda-stdlib

$$\dfrac{b \in \mathbb{B} \quad \phi(v)\ \text{set}\ [v \in \mathbb{B}] \quad \vdots \quad \dfrac{A\ \text{set} \quad a \in A}{(true\ ?\ a\ :\ a) = a \in A}\ \mathbb{B}_{eq_1} \quad \dfrac{A\ \text{set} \quad a \in A}{(false\ ?\ a\ :\ a) = a \in A}\ \mathbb{B}_{eq_2}}{(b\ ?\ id(a)\ :\ id(a)) \in \phi(b)}\ \mathbb{B}_{elim}$$

Now, take a look at the same proof, encoded in Agda without anything from the standard library. Note how some rules (like $\mathbb{B}_{elim}$) are built into the language, as pattern-matching, for instance.

```
data Bool : Set where
  tt : Bool
  ff : Bool

if_then_else_ : {A : Set} → Bool → A → A → A
if tt then x else _ = x
if ff then _ else x = x

data Id (A : Set)(x : A) : A → Set where
  id : (a : A) → Id A x a

proof : {A : Set}{a : A}{b : Bool} → Id A (if b then a else a) a
proof {a = a} {b = tt} = id a
proof {a = a} {b = ff} = id a
```

The Agda snippet above is fairly more readable than the actual derivation of our example lemma. Understanding how one writes programs and very general proofs in the same language can be tough. We hope to have exposed how Agda uses Martin-Löf's theory of types in a very clever way, providing a very expressive logic for theorem proving. The programming part of Agda is directly connected to last chapter's Curry-Howard Isomorphism. This concludes a big portion of the background needed for understanding the rest of this dissertation.

# 3

# RELATIONAL ALGEBRA IN AGDA

This work takes Relational Algebra[5] as main case study for a couple of reasons. One of the most important is its expressive power and unquestionable advantage as a framework for reasoning about software, in an equational fashion. Relational Algebra is in fact a discipline that allows us to speak of software engineering through unambiguous equations, giving a definite meaning to the *engineering* part of software engineering.

We are not the first to use Agda and Relational Algebra together, though. There are two approaches that deserve to be mentioned and compared to what we are doing here. The first, which in fact should be regarded as a basis for our work, is due to Mu et al, at [18]. The second, which is more abstract, is due to Kahl, at [13].

This chapter will present and compare both approaches mentioned above, then introduce a basic encoding for relations. This will be followed by the discussion of several further options that would change the handling of equality, which turned out to be a very delicate matter.

## 3.1 STATE OF THE ART

The current developments within Relational Algebra in Agda vary a lot, mainly due to different goals. One has a plethora of options when developing a library. It is therefore important to weight options against goals. As far as we know, at the time of writing, there is no library that fit our specific goals, which are: (a) An expressive encoding of relational algebra; (b) easy to handle at the meta-level, for automatic rewriting.

The library built by Kahl, [13], is not specific to Relational Algebra. Instead, Kahl chose to build a library for Category Theory in general, from which one can use the specific category of relations, **Rel**. The theories in RATH-Agda are intended for high-level programming, not so much focused on theorem proving. Thus the equality problem we ran into is not handled in Kahl's library. RATH-Agda has both an equality up to isomorphism and intensional equality, which are not interchangeable.

Later on, the library defines the relational operations on top of the categorical basis, also provided by the library. Relational Equality is defined by mutual inclusion and, since no rewriting is intended, substitutivity of relational equality is not provided. Another problem we would run into had we used RATH-Agda is the quoted representation of terms. In Kahl's library every construction is defined as a top-level function, therefore it is expanded upon quoting.

Another interesting library is the one developed by Jansson, Mu and Ko at [18]. Their goal, however, is again very different from both RATH-Agda and this project. Mu, et al., are focusing on deriving programs given specifications, which is a big merit of functional programming. Indeed, they explore the equational reasoning of programs in order to keep *refining* programs. They also avoid the non-extensional equality problem by defining another equality type and proving its substitutiveness whenever necessary. Their main focus is on subrelation-reasoning, though.

## 3.2 ENCODING RELATIONS

Unlike most programming languages, an encoding of Relational Algebra in a dependently typed language allows one to truly see the advantage of dependent types in action. Most of the definitions happen at the *type level* (remember that there is no difference between types and values in Agda, this is just a mnemonic to help understanding the *functions*). The encoding presented below is based on [18], the most significant differences being due to extensional equality.

Long story short, a binary relation $R$ of type $A \to B$ can be thought of in terms of several mathematical objects. The usual definition is to say that $R \subseteq A \times B$, where $\cdot \times \cdot$ is the cartesian product of sets. In fact, $R$ contains pairs or related elements whose first component is of type $A$ and second component is of type $B$. Note this is more general than the concept of a function, where we can have $b_1\ R\ a$ and $b_2\ R\ a$, which means that $(a, b_1) \in R$ and $(a, b_2) \in R$, as a perfectly valid relation, but not a function, since $a$ would be mapped to two different values, $b_1$ and $b_2$.

Another way of speaking about relations, though, is to consider functions of type $A \to \mathcal{P}B$. If our previous $R$ is to encoded as a function $f$, we will then have $f\ a = \{b_1, b_2\}$. For the more mathematically inclined reader, the arrows $A \to \mathcal{P}B$ in the category **Sets**, of sets and functions, correspond to arrows $A \to B$ in the category **Rel**, of sets and relations. For this matter, we actually call **Rel** the Kleisli Category for the monad $\mathcal{P}$. In fact, we can make the encoding of relations as functions more explicit by defining the so-called powerset transpose of a relation $R$, which is the function that for each $a$ returns the exact (possibly empty) subset of $B$ that is related to $a$ trough $R$. This encoding is central in the Algebra of Programming book [5].

$$(\Lambda\ R)\ a = \{b \in B \mid b\ R\ a\}$$

### 3.2. Encoding Relations

For our Agda encoding of Relations, we shall use a slight modification of the "$\mathcal{P}$ approach". We begin by encoding set theoretic notions, the most important of all being undoubtedly the membership notion $\cdot \in \cdot$. One way of encoding a subset of a set $A$ is using a function $f$ of type $A \to Bool$, the subset beeing obtained by $\{\ a \in A \mid f\ a\ \}$. Yet, in Agda, this would force that we deal only with decidable domains, which is not a problem if everything is finite, but for infinite domains this would not work.

Another option, which is the one used in [18], is to use a function $f$ of type $A \to \mathsf{Set}$ to encode a subset of $A$. Remember that Set is the type of types in Agda. Although not very intuitive, this is much more expressive than the last option, whereby the induced subset is defined by $\{\ a \in A \mid f\ a$ is inhabited $\}$, which turns out to be:

```
ℙ : Set → Set1
ℙ A = A → Set
```

Extending from sets to binary relations is a very simple task. Besides the canonical steps, we'll also swap the arguments for a relation, following what is done in [18] and keeping the syntax closer to what one would write on paper, since we usually write a relational statement from left to right, that is, $y\ R\ x$ means $(x,y) \in R$.

$$\begin{aligned} \mathcal{P}(A \times B) &= \mathcal{P}(B \times A) \\ &= B \times A \to \mathsf{Set} \\ &= B \to A \to \mathsf{Set} \end{aligned}$$

Below we present the base encoding of relations in Agda following this plan, together with a few constructs to help in their definition and to illustrate usage.

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

_∪_ : {A B : Set} → Rel A B → Rel A B → Rel A B
(R ∪ S) b a = R b a ⊎ S b a

_∩_ : {A B : Set} → Rel A B → Rel A B → Rel A B
(R ∩ S) b a = R b a × S b a

fun : {A B : Set} → (A → B) → Rel A B
(fun f) b a = f a ≡ b
```

The operations are defined just as we would expect them to be. Note that _⊎_ represents a disjoint union of sets (equivalent to Either in Haskell) and _×_ represents the usual cartesian product. The function lifting might look like the less intuitive construction, but it's a very simple one. *fun f* is

a relation, therefore it takes a $b$ and a $a$ and should return a type that is inhabited if and only if $(a,b) \in fun\ f$, or, to put it another way, if $b \equiv f\ a$, almost like its textbook definition.

## 3.3 RELATIONAL EQUALITY

A notion of convertibility is paramount to any form of rewriting. In out scenario, whenever two relations are equal, we can substitute them back and forth in a given equation. There are a couple different, but equivalent notions of equality. The simplest one is borrowed from set theory.

**Definition 3.1** (Relation Inclusion)**.** Let $R$ and $S$ be suitably typed binary relations, we say that $R \subseteq S$ if and only if for all $a, b$, if $b\ R\ a$ then $b\ S\ a$.

**Definition 3.2** (Relational Equality)**.** For $R$ and $S$ as above, we say that $R \equiv_r S$ if and only if $R \subseteq S$ and $S \subseteq R$.

**Lemma 3.1.** Relation Inclusion is a partial order, that is, $\subseteq$ is reflexive, transitive and anti-symmetric. And $\equiv_r$ is an equivalence relation (reflexive, transitive and symmetric).

So far so good. At first sight one would believe that there are no problems with such definitions (and in fact there will be no problem as long as we keep away from Agda, but then it would not be fun, right?). Definitions were wrapped as datatypes in Agda to prevent expansion when using reflection, this implies that we have some boilerplate code that must be coped with.

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

The reader is invited to take a deeper look into what this definition means. Let $R$ : Rel $A$ $B$, for $A$ and $B$ Sets, $a : A$ and $b : B$. The statement $R\ b\ a$ yields a Set, and an element $r : R\ b\ a$ represents a proof that $b$ is related to $a$ trough $R$. We have no interest in the contents of such proof though, as its existence is what matters. In more formal terms, we would like that Rel returned a proof irrelevant set. In Coq this would be `Prop`, but Agda has no set of propositions.

Such a problem comes to the surface once we try to use Agda's propositional equality instead of $\equiv_r$ (and hence the name distinction):

```
postulate
  rel-ext : {A B : Set}{R S : Rel A B}
    → (∀ b a → R b a ≡ S b a)
    → R ≡ S

naive₁ : {A B : Set}{R S : Rel A B}
  → R ⊆ S → S ⊆ R → R ≡ S
naive₁ (⊆in rs) (⊆in sr)
  = rel-ext (λ b a → ? )
```

We have to use functional extensionality anyway (here, disguised as rel-ext). But the type of our goal is $R\ b\ a \equiv S\ b\ a$ and the proof gets stuck since set equality is undecidable. What we want is that $R$ and $S$ be inhabited at the same time.

So, it is clear that we also need some encoding of $\equiv_r$:

```
record _≡r_ {A B : Set}(R S : Rel A B) : Set where
  constructor _,_
  field
    p1 : R ⊆ S
    p2 : S ⊆ R
```

But this definition is not substitutive in Agda. Therefore we need to lift it to a substitutive definition. In fact, it is safe to lift it to the standard propositional equality. The advantages of doing so are that we win all of the functionality to work with $\equiv$ for free.

The solution we are currently adopting relies on using yet another disguised version of function extensionality (here as powerset transpose extensionality) and on postulating the proof-irrelevant notion of set equality (as in Set Theory):

```
≡r-promote : {A B : Set}{R S : Rel A B}
      → R ≡r S → R ≡ S
≡r-promote {R = R} {S = S} (⊆in rs , ⊆in sr)
  = Λ-ext (λ a → ℙ-ext (flip R a) (flip S a) (rs a) (sr a))
  where
    Λ : {A B : Set}(R : Rel A B) → A → ℙ B
    Λ R = λ a b → R b a

    postulate
      Λ-ext : {A B : Set}{R S : Rel A B}
        → (∀ a → (Λ R) a ≡ (Λ S) a)
        → R ≡ S

      ℙ-ext : {A : Set}(s1 s2 : ℙ A)
        → (∀ x → s1 x → s2 x)
        → (∀ x → s2 x → s1 x)
        → s1 ≡ s2
```

The powerset transpose postulate is pretty much function-extensionality with some syntactic sugar, and it is known to not introduce any contradiction. But ℙ-*ext* on the other hand, has to be justified. The reason for such a postulate is just because somewhere in our library, we need a notion of proof-irrelevance able to state relational equality. As discussed in the next section, there are a couple places were we can insert this notion, but postulating it at the subsets (ℙ) level is by far the easier to handle and introduces the least amount of boilerplate code. (Remember that if $a$ is a subset of $A$, that is $a : \mathbb{P}A$, stating $a\ x$ is stating $x \in a$, for any $x : A$.).

There are a few other options of achieving a substitutive behavior for $\equiv_r$, as we shall present in the next sections.

### 3.3.1 *Hardcoded Proof Irrelevance*

This idea of a proof irrelevant set was mentioned before, as was the fact that we postulated such notion, which is not the ideal thing to do in Agda. We want to keep our postulates to an absolute minimum.

In order to formally talk about proof irrelevance we need some concepts from HoTT, the Homotopy Type Theory[25]. HoTT is an immense and complex newly founded field of Mathematics, and we are not going to explain it in detail. The general idea is to use abstract Homotopy Theory to interpret types. In normal type theory, a statement $a : A$ is interpreted as $a$ is an element of the set $A$. In HoTT, however, we say that $a$ is a point in space $A$. Objects are seen as points in a space and the type $a = b$ is seen as a path from point $a$ to point $b$. It iss obvious that for every point $a$ there is a path from $a$

to $a$, therefore $a = a$, giving us reflexivity. If we have a path from $a$ to $b$, we also have one from $b$ to $a$, resulting in symmetry. *Chaining* of paths together corresponds to transitivity. So we have an equivalence relation $=$.

Even more important, is the fact that if we have a proof of $P\ a$ and $a = b$, for a predicate $P$, we can transport this proof along the path $a = b$ and arrive at a proof of $Pb$. This corresponds to the substitutive behavior of standard equality. Note though, that in classical mathematics, once an equality $x = y$ has been proved, we can just switch $x$ and $y$ wherever we want. In HoTT, however, there might be more than one path from $x$ to $y$, and they might yield different results. So, it is important to state which path is being *walked through* when we use substitutivity.

The first important notion is that of homotopy. Traditionally, we take that two functions are the same if they agree on all inputs. A homotopy between two functions is very close to that classical notion:

```
_~_ : {A : Set}{B : A → Set}(f g : (x : A) → B x) → Set
f ~ g = ∀ x → f x ≡ g x
```

It is worth mentioning that $\_\tilde{}\_$ is an equivalence relation, although we omit the proof. We follow by defining an equivalence, which can be seen an encoding of the notion of isomorphism:

```
isequiv : {A B : Set}(f : A → B) → Set
isequiv f = ∃ (λ g → ((f ∘ g) ~ id) × ((g ∘ f) ~ id))
```

That is, $f$ is an equivalence if there exists a $g$ such that $g$ is a left and right-inverse of $f$.

Now we can state when two types are equivalent. For the reader familiar with algebra, it is not very far from the usual isomorphism-based equivalence notion. As expected, univalence is also an equivalence relation.

```
_≈_ : (A B : Set) → Set
A ≈ B = ∃ (λ f → isequiv f)
```

Following the common practice when encoding HoTT in Agda, we have to postulate the Univalence axiom, which in short says that univalence and equivalence coincide:

```
postulate
  ≈-to-≡ : {A B : Set} → (A ≈ B) → A ≡ B
```

Now, our job becomes much easier, and it suffices to show that if two relations are mutually included, then they are univalent.

*Mere Propositions*

As we mentioned earlier, proof irrelevance is a desired property in most systems. In HoTT, one distinguishes between mere propositions and other types, where mere propositions are defined by:

```
isProp : ∀{a} → Set a → Set a
isProp P = (p1 p2 : P) → p1 ≡ p2
```

This allows us to state some very useful properties, which leads us to handle propositions as both true or false, depending on whether or not they're inhabited. These corresponds to lemma 3.3.2 in [25].

```
lemma-332 : {P : Set} → isProp P → (p0 : P) → P ≈ Unit

¬lemma-332 : {P : Set} → isProp P → (P → ⊥) → P ≈ ⊥
```

Both are provable in Agda, but the proofs are omitted here.

*Adding Relations to the mix*

To exploit such fine treatment of equality in our favor, we need to add relation and a few other details to the mix. The relation definition given before is kept, while pushing to the users the responsibility of proving that their relations are both mere propositions and decidable.

This can be easily done with Agda's instance mechanism:

```
record IsProp {A B : Set}(R : Rel A B) : Set where
  constructor mp
  field isprop : ∀ b a → isProp (R b a)

record IsDec {A B : Set}(R : Rel A B) : Set where
  constructor dec
  field undec : Decidable R

open IsDec {{...}}
open IsProp {{...}}
```

This will treat both records as typeclasses in the Haskell sense. Now, for talking about subrelations they must be an instance of IsProp, and whenever we wish to use anti-symmetry they must also be an instance of IsDec. It turns out that anti-symmetry is now provable as long as we assume relational extensionality.

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

⊆-antisym : {A B : Set}{R S : Rel A B}
  {{ decr : IsDec R }} {{ decs : IsDec S }}
  {{ pir : IsProp R }} {{ pis : IsProp S }}
  → R ⊆ S → S ⊆ R → R ≡ S
```

Although we could arrive at the result we desired with a minimal number of postulates (univalence axiom, only), the user would be heavily punished when wanting to define a relation, not to say that decidability would give problems once relational composition (discussed below) enters the stage. For this reasons we chose not to adopt this solution *as is*, even though it is more formal than what we previously presented.

*Custom Universes*

We could remove the IsProp record from our code should we give relations a bit more structure, and, prove that every object in this new (more structured) world is a mere proposition. One good option would be to encode a universe $U$ of mere propositions and have relations defined as $B \to A \to U$. This extra structure allows us to prove proof irrelevance for all $u : U$ (which holds by construction, once $u$ is a mere proposition), but only lets one define relations over $U$, which is less expressive than *Set*. The additional boilerplate code is also big, once we have to define a language and all operations that we'll need to perform with it. Our universe is defined as:

```
mutual
  data U : Set where
    TT  : U
    FF  : U
    _∧_ : U → U → U
    ¬_  : U → U
    _⇒_ : U → U → U
    _∨_ : U → U → U
    all : {A : U} → (♯ A → U) → U
    exs : {A : U} → (♯ A → U) → U
    _==_ : {A : U}(x y : ♯ A) → U
```

With its interpretation back to Agda Sets beeing:

```
{-# TERMINATING #-}
♯_ : U → Set
♯ TT = Unit
♯ FF = ⊥
♯ (p ∧ q) = ♯ p × ♯ q
♯ (¬ p) = ♯ p → ⊥
♯ (p ⇒ q) = ♯ p → ♯ q
♯ (p ∨ q) = || ♯ p ⊎ ♯ q ||
♯ all p = ∀ a → ♯ p a
♯ exs {A} p = || ∃ (♯_ ∘ p) ||
♯ (x == y) = x ≡ y
```

Notation $||\ \_\ ||$ denotes propositional truncation. That is, for every type $A$, there is a type $||\ A\ ||$ with two constructors: (i) for any $x : A$ we have $|\ x\ |:||\ A\ ||$; (ii) for any $x, y :||\ A\ ||$, we have $x \equiv y$. This is also called *smashing* sometimes. Not every type constructor preserves mere propositions. A simple example is the coproduct $1 + 1$ itself. Even though 1 is a mere proposition, $1 + 1$ is not, since the elements of such type contain also information about which injection was used. Thus the need to smash this information out if we want to keep with mere propositions.

It turns out that we are just defining the logic we will need to define relations, but this structure is very helpful! Now we can prove that given $u : U$, $\sharp u$ is a mere proposition.

```
uProp : (P : U) → isProp (♯ P)
```

So far so good! We just removed one instance from the user and proving decidability becomes straightforward (but in a few, rare, complicated cases)! However, the changes were not only for the better. A new, minor, problem is the verbosity introduced by $U$, since everything is harder to write and read. But there is a more serious situation happening here. If we look at the existential quantifier defined in $U$, its witnesses must also come from $U$. This can be very restrictive once we start using relational composition (which is defined in terms of existentials).

*The Equality Model*

Given the options discussed above, with all their positive and negative aspects, it seems a little too restrictive to adopt only one option. We indeed mixed both the $\equiv_r$ promote with $\subseteq$ *-antisym*. The idea is that users not only can chose how formal they want their model to be, but this can significantly increase development speed. In the first stages of development, where the base relations might change a lot (and, if instances were written, they would consequently change), one can keep developing with the $\equiv_r$ promotion. Once this model is stable, it can completely formalizeb by adding the desired instances and using subrelation anti-symmetry.

## 3.4 CONSTRUCTIONS

After establishing a model for relations and relational equality, we follow by presenting some of the important constructions here. Note that contrary to *pen and paper* Mathematics, we provide an encoding of the constructions and then we prove that our encoding satisfies the universal property for the given construction, instead of adopting such property as the definition. This not only proves the encoding to be correct, but it is the only constructive approach we can use.

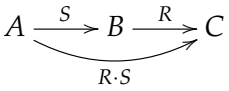
The design adopted for the lower level constructions has a somewhat heavy notation. The main reason for this choice (which differs significantly from other Relational Algebra implementations) is its ease of use when coupled with reflection techniques. If we provide all definitions as Agda functions, when we access a term AST, Agda will normalize and expand such definitions. By encapsulating it in records, we can stop this normalization process and use a (much) smaller AST representation.

### 3.4.1 *Composition*

Given two relations $R : B \to C$ and $S : A \to B$, we can construct a relation $R \cdot S : A \to C$, read as $R$ after $S$, and defined by:

$$R \cdot S = \{(a,c) \in A \times C \mid \exists b \in B \,.\, a\ S\ b \wedge b\ R\ c\}$$

Or, using diagrams:




As a first definition in Agda, one would expect something like:

```
_after_ : {A B C : Set} → Rel B C → Rel A B → Rel A C
R after S = λ c a → ∃ (λ b → R c b × S b a)
```

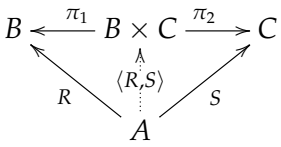
And here we can start to see the dependent types shining. The existential quantification is just some syntax sugar for a dependent product. Therefore, for constructing a composition we need to provide a witness of type $B$ and a proof that $cRb \wedge bSa$, given $c$ and $a$.

Yet, this suffers from the problem mentioned earlier. Agda will normalize every occurrence of _after_ to its right-hand side, which is a non-linear lambda abstraction, and will make subterm matching very complex to handle. An option is to use the exact definition of an existential quantifier, but expand it:

```
record _•_ {A B C : Set}(R : Rel B C)(S : Rel A B)(c : C)(a : A) : Set
  where
    constructor _,_
    field
      witness  : B
      composes : (R c witness) × (S witness a)
```

### 3.4.2 *Products*

Given two relations $R : A \to B$ and $S : A \to C$, we can construct a relation $\langle R, S\rangle : A \to B \times C$ such that $(b, c)\ \langle R, S\rangle\ a$ if and only if $b\ R\ a \wedge c\ S\ a$. Without getting in too much detail of what it means to be a product, we usually write it in the form of a commutative diagram:




That is,

$$\begin{aligned} R &= \pi_1 \cdot \langle R, S\rangle \\ S &= \pi_2 \cdot \langle R, S\rangle \end{aligned}$$

Products are unique up to isomorphism, which explains the notation without introducing any new names. The proof is fairly simple and can be conducted by *gluing* two product diagrams. The diagrammatic notation states the existence of a relation $\langle R, S\rangle$ and the dotted arrow states its uniqueness. $\pi_1\ (b, c) = b$ and $\pi_2\ (b, c) = c$ are the canonical projections.

**Definition 3.3** (Split Universal)**.** Given $R$ and $S$ as above, let $X : A \to B \times C$, then:

$$X \subseteq \langle R, S\rangle \Leftrightarrow \pi_1 \cdot X \subseteq R \wedge \pi_2 \cdot X \subseteq S$$

Encoding this in Agda is fairly simple, once we already have products (in their categorical sense) available. We just wrap everything inside a record:

```
record ⟨_,_⟩ {A B C : Set}(R : Rel A B)(S : Rel A C)(bc : B × C)(a : A) : Set
  where constructor cons-⟨,⟩
    field un : (R (proj₁ bc) a) × (S (proj₂ bc) a)
```

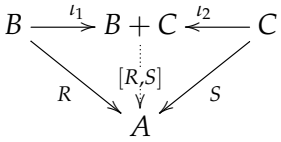
Its universal property can be derived from the following *lemmas*

```
prod-uni-r1 : ∀{A B C} → {X : Rel C (A × B)}
  → (R : Rel C A) → (S : Rel C B)
  → X ⊆ ⟨ R , S ⟩
  → π₁ • X ⊆ R

prod-uni-r2 : ∀{A B C} → {X : Rel C (A × B)}
  → (R : Rel C A) → (S : Rel C B)
  → X ⊆ ⟨ R , S ⟩
  → π₂ • X ⊆ S

prod-uni-l : ∀{A B C} → {X : Rel C (A × B)}
  → (R : Rel C A) → (S : Rel C B)
  → (π₁ • X) ⊆ R → (π₂ • X) ⊆ S
  → X ⊆ ⟨ R , S ⟩
```

In fact, the product of relations respects both decidability and propositionality (that is, given two mere propositions, their product is still a mere proposition). Therefore, such instances are already defined.

### 3.4.3 *Coproduct*

If we flip every arrow in the diagram for products, we arrive at a dual notion, usually called coproduct or sum. Given two relations $R : B \to A$ and $S : C \to A$, we can perform a *case analysis* in an element of type $B + C$ and relate it to an $A$. Indeed, the *either* of $R$ and $S$, denoted $[R, S]$, has type $B + C \to A$ and is depicted in the following diagram:




In Agda, we have:

```
record either {A B C : Set}(R : Rel A C)(S : Rel B C)(c : C)(ab : A ⊎ B) : Set
  where constructor cons-either
    field un : case (R c) (S c) ab
```

```
coprod-uni-r1 : ∀{A B C}{X : Rel (A ⊎ B) C}
  → (R : Rel A C) → (S : Rel B C)
  → (X ≡r either R S)
  → R ≡r X • ι₁

coprod-uni-r2 : ∀{A B C}{X : Rel (A ⊎ B) C}
  → (R : Rel A C) → (S : Rel B C)
  → (X ≡r either R S)
  → S ≡r X • ι₂

coprod-uni-l : ∀{A B C}{X : Rel (A ⊎ B) C}
  → (R : Rel A C) → (S : Rel B C)
  → (R ≡r X • ι₁) → (S ≡r X • ι₂)
  → X ≡r either R S
```

The coproduct has instances already defined for decidability and mere propositionality.

## 3.5 THE LIBRARY

The constructions and workarounds explained above do not constitute a full description of the library. There are still modules for equational reasoning and a few more constructs implemented. All the code is available in the following GitHub repository:

https://github.com/VictorCMiraldo/msc-agda-tactics/tree/master/Agda/Rel

. It is compliant with Agda's version 2.4.2.2; standard library version 0.9.

### 3.5.1 *Next Versions*

Our Relational Algebra in Agda library is no diferent from other library. It always has some feature to be implemented and requries constant maintenance work, to comply with the latest Agda standard library. Yet, there are two very important constructs that have not been implemented so far. We would like to have both a relational fold and shrinking[19] in the library. This would give us the expressivity we're looking for.

### 3.5.2 *Summary*

The problem imposed by Agda's (intensional) equality is that it implies convertibility. As seen in this chapter, we don't really care about convertibility of relations. They need to be inhabited at the same time, though. Borrowing some notions from Homotopy Type Theory[25] in order to hardcode a proof irrelevance notion was very helpful, besides the complex layer of boilerplate code for the end-user. The later option, postulating some axioms that allows one to *trick* Agda's equality, revealed itself to be cleaner and to provide both a better interface and a completely formal approach, when needed.

# 4

# SUMMARY AND FUTURE WORK

The overall task of adding rewriting functionality to Agda is quite an exploratory project for a couple reasons, the most important being the unstable state of Agda's standard library and, in particular, of the Reflection module.

The work started by getting familiar both with the Agda language and the required theoretical background. Our case study is the development of a library for Relational Algebra, that will be used as our main target for the rewriting functionality. During this development I stumbled across a few problems regarding the notion of equality, which were solved using a combination of techniques, as explained in section 3.2. This significantly slowed down the project, mostly because one had to be sure to get this right, otherwise the library wouldn't be suitable for automatic rewriting, as is the case of current state-of-the-art Relational Algebra libraries, in Agda.

The library is far from complete, as any library requires constant management and adaptation, but is stable enough for the work to proceed to the next task, which is exploring the reflection capabilities of Agda, and exploring how to use them to provide a general rewriting functionality. This is the form of the current work.

Once a reliable rewriting mechanism is ensured, we would like to run some bigger verification tasks, in order to *destructively* test our tool. Parallel to this, there is also the work of expanding the library with more Relational Algebra constructions. As mentioned before, this is constant work, at least until Agda's stdlib stabilizes. We're currently working with Agda version 2.4.2.2 with standard library version 0.9. The code we developed so far and the sources for this document are available at the git repository `https://github.com/VictorCMiraldo/msc-agda-tactics`.

The use of this library to support the relational derivation of functional programs in the algebra of programming style is the ultimate goal of this project, once the technical difficulties mentioned above are addressed and solved.

# BIBLIOGRAPHY


[1] Agda wiki: Tutorials, Sep 2014.

[2] H. Barendregt. *The lambda calculus: its syntax and semantics*. North Holland, 2nd. revised edition, 1984.

[3] H. Barendregt. *Handbook of Logic in Computer Science: Lambda Calculi with Types*, volume 2. Oxford University Press, 1993.

[4] Yves Bertot and Pierre Caterán. *Interactive Theorem Proving and Program Development*. Springer Berlin Heidelberg, 2006.

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

[6] Ana Bove and Peter Dybjer. Dependent types at work. In *Language Engineering andLecture Notes in Computer Science. International Summer School on Language Engineering and Rigorous Software Development. Piriapolis, URUGUAY. FEB 25-MAR 01, 2008*, pages 57–99, 2009.

[7] A. Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1, 1936.

[8] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2), 1936.

[9] H. B. Curry. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[10] A. Heyting. *Intuitionism: an introduction*. Studies in logic and the foundations of mathematics. North-Holland Pub. Co., 1971.

[11] J. Hindley and P. Seldin. *Introduction to combinators and [lambda]-calculus*. London Mathematical Society student texts. Cambridge University Press, 1986.

[12] W. A. Howard. The formulae-as-types notion of construction, 1969. manuscript.

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

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

[15] P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 167–184, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[16] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer Berlin Heidelberg, 2005.

[17] Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag Berlin Heidelberg, 1980.

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

[19] Shin-Cheng Mu and José 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).

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

[21] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[22] JoséN. Oliveira. Extended static checking by calculation using the pointfree transform. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *Lecture Notes in Computer Science*, pages 195–251. Springer Berlin Heidelberg, 2009.

[23] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Acta Universitatis Stockholmiensis, Stockholm, Göteborg, Uppsala: Almqvist, Wicksell., 1965.

[24] Wouter Sweirstra and Nicolas Oury. The Power of Pi. *ICFP08*, 2008.

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

[26] A.N. Whitehead and B. Russell. *Principia Mathematica*. Principia Mathematica. University Press, 1912.

[27] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 224–235, New York, NY, USA, 2003. ACM.