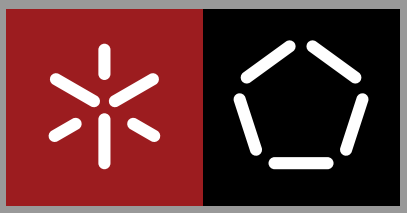
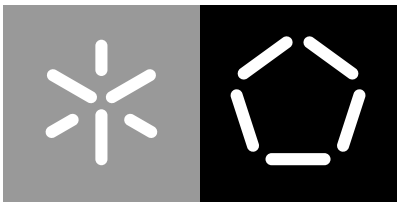


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Victor Cacciari Miraldo

Proof by Rewriting in Agda

June 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)

June 2015

ABSTRACT

It is very common to hear that *pen-and-paper* mathematics run smoother than computer-aided reasoning. The biggest pointed problem being the overhead of information one has to provide a computer to be able to perform simple reasoning, with the upside being the absolute certainty that one is correct. One such tool for computer reasoning is Agda, a dependently typed, pure, functional language. We explore this language in its two fronts: a proof-assistant and a programming language. On the proof-assistant side, we have developed a library for Relational Algebraic reasoning, which handles a big portion of the relational calculus, including a prototype of generic catamorphisms. However, the overhead of code is still present when reasoning with our library. We tackle this by automating the context inference of substitutions, based on the reflection mechanism of Agda. Our *tactics* allow one to perform rewrites just like on *pen-and-paper* mathematics. Several tactics, some of which are experimental generalizations of simpler variants, are presented with a thorough discussion on how do they work. The full source code corresponding to this dissertation is available on the internet.

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*, [31], 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.

As we said above, proof assistants depend on the logic they run on top of. Some of them, however, support some form of meta programming. That is, we can write programs that generate programs. This is achieved by having the programming language itself as a first class datatype. Examples include LISP and Prolog. One of the goals of this project is to explore how can we exploit such feature to automate the repetitive task of rewriting terms for other, proven to be equal, terms.

The tool of choice for this project is the Agda language, developed at Chalmers [23]. Agda uses a intensional variant of Martin-Löf's theory of types and provides a nice interactive construction feature. After the Curry-Howard isomorphism [14], 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 [6], 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[23] is one such language, other big contributions being *Epigram* [19] and *Coq* [5].

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 λ-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[32] 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 $\bot\text{-}elim$, 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.

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

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 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 $(x : A) \rightarrow B\ x$, in general, x can appear on the right-hand side of the function type constructor, $\rightarrow$. 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 *Finite Types*

The theory of types is not only roses. The soundness requirement is that every function must terminate. In fact, Agda passes all definitions through its termination checker and if it does not recognize a definition as terminating, it will output an error.

An interesting technique to write terminating functions is to use finite types. One example where we need to use such finite types is first-order unification, as in [18]. As the unification algorithm progresses, the terms are growing bigger, since variables are being substituted by (possibly) more complex terms. Yet, the number of variables to be substituted decreases by one in each step. This is modeled by a function with type Fin $(\mathsf{suc}\ n) \to$ Fin n, where Fin n is a type with n elements.

Agda, and dependently typed languages in general, allows one to speak about type *families* in a very structured way. To define the family $(\mathsf{Fin}\ n)_{n \in \mathbb{N}}$ of types with *exactly* n elements we proceed as follows.

```
data Fin : ℕ → Set where
  fz : {n : ℕ} → Fin (suc n)
  fs : {n : ℕ} → Fin n → Fin (suc n)
```

If we wish to enumerate, for instance, the inhabitants of type Fin 3, we would find that they are: fz, fs fz and fs fs fz. Now, how many inhabitants does Fin 0 have? It is indeed an uninhabited type, which is congruent with the intuition of having *exactly* zero elements.

The careful reader might have noticed a slight syntax difference between Fin's definition and the other datatypes presented before. In general, Agda allows one to define datatypes with parameters $p_1, \cdots, p_n$ and indices $i_1, \cdots, i_k$. The parameters works just like a Haskell type parameter. Indices, on the other hand, is what allows us to define inductive families in Agda. As stated in the excerpt above, the *type* of Fin is $\mathbb{N} \to$ Set. In a data declaration, parameters are the variables before the

colon, whereas indices appear after the colon. Vectors of a fixed length are another famous inductive family example.

1.2.4 *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[6], 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 [24], from where the *twice preserves even numbers* case study was taken (page 216).

1.4 STRUCTURE OF THE DISSERTATION

This document starts with a basic mathematical background, in chapter 2, where we try to help the unfamiliar reader to understand how programming and proving are intrinsically connected. These ideas are very important to have a solid understanding of the Agda language, which we introduced, shallowly, in section 1.2.

We follow with the description of our Relational Algebra library, chapter 3, where we explore the full expresivity of dependent types to encode relations in such a way that they remain useful for automatic proving. The library is by no means complete and should be taken as an exploration of what is possible or not. The code behaves very well until we start to use generic catamorphisms.

Chapter 4 will focus on rewriting. The context in which we immerse ourselves is given by sections 4.1 and 4.2, the rest of the chapter is concerned with explaining the basic form of rewriting one can automate. Generalizations on this rewriting framework follows on chapter 5.

On chapter 6 we give pointers for future work and conclude the work.

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 ideas are 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) λ-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 λ-CALCULUS AND TYPES

What is known as the λ-calculus is a collection of various formal systems based on the notation invented by Alonzo Church in [9, 8]. 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 [3, 12]. The goal of this chapter is to provide a minimal understanding of the λ-calculus, which will allow a better understanding of Martin-Löf's type theory and how logic is encoded in such formalism.

2.1.1 *The λ-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 Λ 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 α-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 λ-calculus. A term with the form $(\lambda x.M)N$ is called a β-redex and can be reduced to $[N/x]M$. If a given term has no β-redexes we say it is in β-normal form.

Definition 2.3 (β-reduction)**.** Let M, M' and N be lambda-terms and x a variable. Let $\rightarrow_\beta$ be the following binary relation over Λ 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 β-reductions from M.

Definition 2.4 (β-equality)**.** Let M and N be lambda terms, M and N are said to be β-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 λ-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 [3].

2.1.3 *Simply typed λ-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 Γ be a context, M a lambda-term and σ 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 β-reductions on it.

The simply-typed λ-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 λ-calculus we refer the reader to [4]. We're not interested in that aspect of the λ-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 Γ a context such that $\Gamma \vdash M : \sigma$ is derivable. We can look at σ 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[25] (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
σ	$\dfrac{}{\Gamma \vdash M : \sigma}\ (Ax)$
$\dfrac{\begin{array}{c}[\tau] \\ \vdots \\ \sigma\end{array}}{\tau \supset \sigma}\ (I \supset)$	$\dfrac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash (\lambda x.M) : (\tau \to \sigma)}\ (I \to)$
$\dfrac{\tau \supset \sigma \quad \tau}{\sigma}\ (E \supset)$	$\dfrac{\Gamma \vdash M : (\tau \to \sigma) \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}\ (E \to)$

This seemingly shallow equivalence is a remarkable result in Computing Science, discovered by Curry and Howard in [10, 14]. 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 [16] 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 [22] or, for a more practical view, [26, 7]

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 [11].

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 π_1 and π_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 [20] $E \sim \Sigma\{a.E' \mid E \xrightarrow{a} E'\}$[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 Σ 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 α and β are themselves arities. The informal meaning of such arity is *"give me an expression of arity α and I give an expression of arity β"*, 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 [22].

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	$fst\ \langle a, b\rangle$	$succ\ zero$	$true$
$false$	3×3	$succ\ (3+3)$	$succ\ zero$
$(\lambda x.x)$	$(\lambda x.snd\ x)p$	$cons(4, app(nil, nil))$	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 [16, 17].

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 Π, 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 omitted) 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[6] 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 [21]. The second, which is more abstract, is due to Kahl, at [15].

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, [15], 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 [21]. 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 [21], 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 [6].

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

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 being 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 [21], 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 [21] 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) in the anti-symmetry proof:

```
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 (otherwise we cannot perform rewrites). 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 $\mathbb{P}$-*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 ($\mathbb{P}$) 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[29]. 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 is 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 [29].

```
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 being:

```
{-# 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 be completely formalized 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.

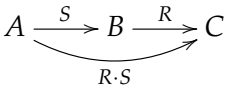
The design adopted for the lower level constructions has a somewhat heavy notation. The main reason for this choice (which differs significantly from other Relational Algebra implementations) is its ease of use when coupled with reflection techniques. If we provide all definitions as Agda functions, 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)
```

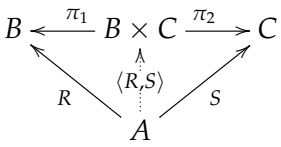
Agda also provides an abstract keyword. An abstract definition is treated just like a postulate, but with a proof. This is another way of forcing definitions to *not* expand or evaluate. It does not serve our purpose, however. If we declare a relational construct using an abstract (or a postulate for that matter) definition, Agda does not know how to introduce or eliminate it from terms and evaluation is blocked.

The decidability of relational composition is complicated. Given R : Rel B C and S : Rel A B decidable relations, for every $a, c \in A \times C$ we would like to be able to compute $(R \cdot S)\ c\ a \uplus ((R \cdot S)\ c\ a \to \bot)$. But $(R \cdot S)\ c\ a$ holds if and only if there exists a b that witness the composition. As we already know, there is only one way of proving an existential constructively: to construct the quantified object. However, we have to assume no structure for B whatsoever. For an enumerable set B, we could use some constraint solving technique, as in KodKod [28].

In some cases it is easy to compute such $b : B$. Take $R \cdot$ fun f for instance. Given a, we have no choice but $b = f\ a$ and this indeed solves the problem. In our library, we provided a *quickfix* for this if the user can provide another relation *when* : Rel A C such that *when* c a is inhabited if and only if the given relations composes (encoded by typeclass Composes). Although not the most general, it allows one to give alternative formulations of relational composition, so Agda can use this alternative formulations for computing decidability whereas it uses the normal composition for regular proofs. This task deserves more time, which is why it was left as future work. If we could provide a class that is sufficient to decide relational composition, we would be able to run almost every relation we use. This is a non-trivial problem, however.

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*

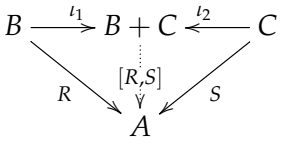
```
prod-uni-r1 : ∀{A B C} → {X : Rel C (A × B)}
  → (R : Rel C A) → (S : Rel C B)
  → X ⊆ ⟨ R , S ⟩
  → π₁ • X ⊆ R

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.

3.4.4 *Relators and Catamorphisms*

Our library already handles a nice portion of Relational Algebra but one of the most useful operations, induction! Relational catamorphisms allows one to define relations by induction over some specified structure.

Catamorphisms are defined as the *least* fixed points of certain recursion equations. Their dual are Anamorphisms, which are defined as *greatest* fixed points. There recursion equations are modeled by recursive functors (which must also be Relators, in the relational setting).

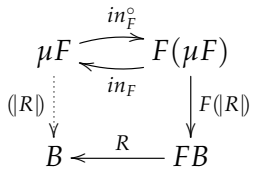
Definition 3.4 (Relator)**.** Let F be a monotonic endofunctor in **Rel**. We say that F is a *relator* iff, if $R \subseteq S$, then $F\ R \subseteq F\ S$.

It can be proved that relators also preserve converse, or, an even more interesting result: a functor is a relator if and only if it preserves converses, theorem 5.1 in [6]. Unfortunately, in Agda things are not so simple. The best option is to encode the relator laws as a type-class.

```
record IsRelator (F : Set → Set → Set) {{ p : IsWFunctor1 F }}
       : Set1 where
  field
    fmap-id : ∀{B A} → (F_r Id) ≡r Id

    fmap-• : ∀{I A B C}{R : Rel B C}{S : Rel A B}
      → F_r (R • S) ≡r F_r R • F_r S

    fmap-° : ∀{I A B}{R : Rel A B}
      → F_r (R °) ≡r (F_r R) °

    fmap-⊆ : ∀{I A B}{R S : Rel A B}
      → R ⊆ S → F_r R ⊆ F_r S
```

Yet, the purpose of this dissertation is not to explain Relators or Functors, but how to encode them in Agda. Going one step at a time, let us look at the traditional definition of a catamorphism.

Categorically speaking, let $F : \mathbf{C} \to \mathbf{C}$ be an endofunctor with initial algebra $(\mu F, in_F)$, then, for any object B and F-algebra $R : FB \to B$ the following diagram commutes.



Reading the universal depicted above, we have:

$$(\!|R|\!) \equiv_r R \cdot F\ (\!|R|\!) \cdot in_F^\circ$$

Although it looks (and it indeed is) pretty tricky to encode this directly into Agda, at least it gives us a type to follow. Since almost all useful functors are polymorphic, we will now denote F by F_A, and regard F as a functor with type $\mathbf{D} \times \mathbf{C} \to \mathbf{C}$.

We seek to define a relation $(\!|R|\!) : \mu F_A \to B$, given $R : F_A B \to B$, or, with Agda types, given a $B \to F_A B \to$ Set, we need to build a $B \to \mu F_A \to$ Set.

Shape and Positioning

The problem's core lies in how to define a fixed point in Agda, which turns out to be very complicated, if not impossible[1]. One solution is to look at a functor from a different angle. Playing around with the List functor we arrive at:

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

The last step of the above derivation, were we transform 1 into $X^{\perp}$, can not be proven in Agda. However, it is evident to see it is a valid transformation from the initiality of $\perp$.

It is important to note that the resulting term is an instance of a more general form:

$$T_{S,P} X = \Sigma\ (s : S)\ (X^{P\ s})$$

For $S = 1 + A$ and $P = [\text{const}\ \perp, \text{const}\ 1]$. It can be proved that all polynomial functors can be encoded in that fashion[2], where S represents a set of constructors of our *datatype* and P represents

1 With the help of Co-induction one can define fixed points, but this triggers some other problems when using such fixed points. Not to mention the instability of the Co-induction module.

the arity of each constructor. In the List example, $nil = \mathsf{i1}\ \mathsf{unit}$ has arity 0 and $cons = \mathsf{i2}\ (\mathsf{x}, \_)$ receives one recursive argument.

```
record IsWFunctor1 (F : Set → Set → Set) : Set1 where
  field
    S : Set → Set
    P : {A : Set} → S A → Set

    inF  : {A : Set} → F A (W (S A) P) → W (S A) P
    outF : {A : Set} → W (S A) P → F A (W (S A) P)

    F_r : {X A B : Set} → Rel A B → Rel (F X A) (F X B)

  μF : Set → Set
  μF A = W (S A) P

  inR : {A : Set} → Rel (F A (μF A)) (μF A)
  inR = fun inF

  outR : {A : Set} → Rel (μF A) (F A (μF A))
  outR = fun outF
```

In the excerpt above we have a polymorphic shape S, this parameter only represents the type our container is storing. For the sake of simplification we will forget about it. The set $W_S P$ is the set of all *well founded trees* constructed by regarding each $s : S$ as a constructor of arity $P\ s$. The first definition of W-types is from [16].

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

The important question being: are we talking about *the same* lists? And the answer is yes. Abbot et al. did a lot of work on container types and encoding inductive types as W-types. The most relevant result for us can be found in [2] and says that $W_S P$ coincide with the least fixed point of a functor identified by S and P:

$$W_S P \cong \mu X\ .\ T_{S,P} X$$

The elimination rule for W-types is

```
W-rec : ∀{c}{S : Set}{P : S → Set}
  → {C : W S P → Set c}
  → (c : (s : S)
    → (f : P s → W S P)
    → (h : (p : P s) → C (f p))
    → C (sup s f))
  → (x : W S P) → C x
W-rec {C = C} c (sup s f) = c s f (W-rec {C = C} c ∘ f)
```

which gives out the generic induction principle for W-types. The dependent type setting, however, allows one to tweak the conclusion C, depending on the $W_S P$ being *folded*. If we let $C = A \to \mathsf{Set}$, quantified on A, we arrive at something very close to:

```
W-rec-rel : {S : Set}{P : S → Set}{A : Set}
  → ((s : S) → (p : P s → W S P) → Rel (W S P) A → A → Set)
  → Rel (W S P) A
W-rec-rel h a w = W-rec (λ s p c → h s p (W-rec-rel h) a) w
```

Where its parameter function h receives the *shape* and *positioning* of a list l (that is, the $in^\circ\ l$), a relation built recursively, and something of type A. The universal law can provide some interesting intuition. It says that $(\!|R|\!)$ is the same as opening up the list, checking if the recursive parts *relate* with something, then relating the head and that something with R. That is exactly how we encode it:

```
W-cata-F : {A B : Set}{F : Set → Set → Set}{{ prf : IsWFunctor1 F }}
  (R : Rel (F A B) B) → Rel (μF A) B
W-cata-F {A} {B} {F} R = W-rec-rel (λ s p h n → gene h n (sup s p))
  where
    gene : Rel (μF A) B → Rel (μF A) B
    gene h n l = (R • F_r h) n (outF l)
```

Now we just need to wrap it over a record to prevent evaluation, but looking at the type of W-cata-F, it is already the type of a relational catamorphism. The final piece of the puzzle looks like:

```
record (|_|)_1 {A B : Set}{F : Set → Set → Set}{{ prf : IsWFunctor1 F }}
  (R : Rel (F A B) B)(b : B)(μFa : μF A) : Set
  where constructor cons-cata
    field un : W-cata-F R b μFa
```

Wrapping it up, in order to handle arbitrary[2] functors the idea is to: (a) encode it using W-types; (b) Use a custom elimination rule, derivable from the standard W-elimination and (c) translate the relational gene using the cata universal. A small problem with this technique is the need to explicitly

2 Not at all that arbitrary, we can just handle strictly positive inductive types, of which the polynomial functors are part of.

prove the cata universal for every functor we wish to use. One way of bypassing this would be to postulate it. Since we mainly need the universal for rewriting purposes, no actual computation needs to be performed.

Proving the universal law for a specific functor is straight-forward. Nevertheless, we adopted the postulate strategy to ease the development. Remember that in the relational case, the universal law is split in two separate laws:

```
postulate
  cata-uni-1 : {A B : Set}{F : Set → Set → Set}
    {{ pF : IsWFunctor1 F }}{{ pR : IsRelator F }}
    {R : Rel (F A B) B}{X : Rel (μF A) B}
  → X ⊆ R • F_r X • outR
  → X ⊆ (| R |)_1

  cata-uni-2 : {A B : Set}{F : Set → Set → Set}
    {{ pF : IsWFunctor1 F }}{{ pR : IsRelator F }}
    {R : Rel (F A B) B}{X : Rel (μF A) B}
  → R • F_r X • outR ⊆ X
  → (| R |)_1 ⊆ X
```

As far as the author is aware, this is an original attempt at encoding generic catamorphisms in Agda. The List relator has already been handled in [21], yet we were looking forward to give a generic framework. Although theoretically sound, the implementation does not work as intended. Whenever we were working on proof objects involving catamorphisms Agda delivered a bunch of *stack overflows* and *internal error* messages, not to mention the very poor run time, which significantly slowed down development. Nevertheless, we are happy that it is possible to encode those generic functors. Fine tuning the implementation is left as future work.

3.4.5 *Properties*

Besides the main relational constructions, we developed a selection of property-specific modules. Given the amount of properties we have in relational algebra, we had to somehow categorize them. Modules are divided in the kind of property they provide. The naming scheme is straight forward: names start with a hyphen separated list of relevant operators, followed by the property name. If we are looking for $\cdot + \cdot$ and $(\cdot)^\circ$ distributivity, for instance, its name is going to be $+$-$^\circ$-distr. An additional identification: *-l*, *-r*, 1 or 2; might be found appended to the property name, indicating either the direction (left or right) or the part of the conclusion being extracted. One example is the split universal, section 3.4.2. A simple taxonomy of modules is provided below.

1. *Group* like properties, such as: neutral elements, absorption, inverse and associativity.
2. *Monotonicity* over _ ⊆ _.
3. *Idempotence* of operators.
4. *BiFunctoriality* for product and coproduct.
5. *Correflexive* specific properties.
6. *Galois* connections present in Relational Algebra, which provide distributivity over _ ∩ _ and _ ∪ _, cancellation and more monotonicity.

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 *Summary*

This chapter discussed, somewhat high level, the implementation and problems we found during the development of our Relational Algebra library. Even though some features might seem very complex, it is the price we have to pay in exchange for the expressiveness of dependent types.

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[29] 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.

Another complicated solution was the use of W-types to encode generic (polynomial) functors, with their respective catamorphism. Although some mechanical code had to be pushed to the user we finished with an interface very close to what one would write on paper, which closes the features we were looking forward to have.

4

TERMS AND REWRITING

Rewriting can be looked upon from a lot of different angles. Nevertheless, a very general approach is already largely used in logic: substitutivity of predicates. If we want to discuss the validity of an arbitrary statement, lets take $0 \leq 3^2 \leq 10$, for instance. We can substitute *sub-statements* by equal *sub-statements*. In our case, $3^2 = 9$, which renders the final proposition to be $0 \leq 9 \leq 10$. That is obviously true. In a more formal manner, let P be a predicate over a set S and $s_1, s_2 \in S$, if $P\ s_1$ holds and $s_1 \equiv s_2$, then, obviously $P\ s_2$ holds. This notion is called substitutivity.

Whenever we work with pen-and-paper mathematics, it is common to structure our thoughts in a series of *equivalent* statements. To our luck, Agda also has a very interesting customization feature called equational reasoning, which we will explore in this chapter. Yet, no matter when one performs a rewrite on paper, substitutivity rarely comes to mind. Mostly because it is purely mechanical, and one believes that such things are *evident*. We seek to make those steps evident to Agda, and recreate a validated *pen-and-paper* environment.

Agda introduced a reflection API in version 2.2.8. Although not a new feature in the world of functional programming (Lisp's *quoting* and *unquoting* and Template Haskell, for instance, are similar techniques) it is proving to be very useful for implementing interesting things. One application for reflection, which we chose to explore, is the possibility to write custom tactics for proving propositions. Somewhat close to how Coq proves propositions.

The representation of an Agda term has type Term, and can be obtained by *quoting* it. The keywords that bridge the world of programs and their abstract representations are quoteTerm and unquote, inverses of each other. The former transforms a normal term into it's Term representation whereas the later does exactly the opposite. Due to Agda's complexity, however, the type Term is difficult to handle. A much simpler option is to use a intermediate representation, which we called RTerm, for implementing the required operations.

After a simpler representation was figured out, we continued by implementing several RTerm operations. From these operations we could provide an interesting tactic, named by. Whenever the user

calls (tactic (by action)), we will get information about the goal and action at that point and try to derive the solution, as long as there exists a strategy that can handle the goal action pair.

This chapter starts by giving a brief introduction to Agda's Equational Reasoning framework, in section 4.1. Following with Agda's Term datatype and some of the reflection API, in section 4.2. On the sequence, section 4.3, we give a description of our RTerm representation and explain how we perform the conversion. A presentation of the operations required for automatic congruence and substitution guessing follows in section 4.4. We finish this chapter in section 4.5 where we explain the library entry-point function: the *by* function. Some guidance for when one wants to use our rewrite feature in different domains is also given.

The majority of the code is omitted from this chapter in order to improve readability. We stress that the project is publicly available at the author's GitHub repository.

4.1 EQUATIONAL REASONING

The possibility of defining mixfix operators makes Agda a very customizable language. This is, indeed, used very often to build domain specific languages on top of Agda itself, which gives a very convenient way of proving results.

By taking a look at a piece of squiggol, from a type-theoretic angle, one can identify a few components. As an example, let us take a proof of the shunting property, from [6], and look at its syntax.

$$
\begin{aligned}
& f \cdot R \subseteq S \\
\Leftarrow \quad & \{\ f \text{ is simple} \ \} \\
& f \cdot R \subseteq f \cdot f^{\circ} \cdot S \\
\Leftarrow \quad & \{\ \text{monotonicity}_2 \ \} \\
& R \subseteq f^{\circ} \cdot S \\
\Leftarrow \quad & \{\ f \text{ is entire} \ \} \qquad (1) \\
& f^{\circ} \cdot f \cdot R \subseteq f^{\circ} \cdot S \\
\Leftarrow \quad & \{\ \text{monotonicity}_1 \ \} \qquad (2) \\
& f \cdot R \subseteq S
\end{aligned}
$$

The first thing that pops up is the implication, $\Leftarrow$, which is not part of the relational calculus being developed in the subject of the demonstration. The justifications, however, must have type $e_{n+1} \Leftarrow e_n$. Well, implication in Agda is just the arrow type, as for functions, rendering the type of the justifications

to be $e_n \to e_{n+1}$. Since we have loads of justifications, it makes sense to associate them to some side. By assuming they are right-associative we arrive at a ternary mixfix operator $\_ \Leftarrow \{\_\} \_$. Transitivity and reflexivity of the underlying operation, in our case $\Leftarrow$, closes the box.

However, Agda already supports an even more generic approach to equational reasoning. All we need is a preorder relation $\_ \sim \_$ with a subjacent equivalence relation $\_ \approx \_$. Below we present the kinds of relational reasoning supported.

$$\begin{array}{lll} & R & \\ \equiv_r & \langle lemma_1 \rangle & \\ & S & \end{array} \qquad \begin{array}{lll} & R & \\ \subseteq & \langle lemma_2 \rangle & \\ & S & \end{array} \qquad \begin{array}{lll} & R \subseteq S & \\ \Leftarrow & \langle lemma_3 \rangle & \\ & R' \subseteq S' & \end{array}$$

Where the above lemmas must have type $R \equiv_r S$, $R \subseteq S$ and $R' \subseteq S' \to R \subseteq S$ respectively. The context of equational reasoning is a good starting point to build some intuition about rewriting.

The reader might have noticed a small problem in the above explanation of equational reasoning, in contrast with the shunting proof given as an example. Not every justification in the proof have the same *type*, as we claimed above. Looking at step 2, with type $\forall\ R\ S\ T\ .\ R \subseteq S \to C \cdot R \subseteq C \cdot S$, and step 1, with type $id \subseteq f^\circ \cdot f$. Our rewrite framework does exactly that job, of *adjusting* the lemmas that do not fit.

4.2 REFLECTION IN AGDA

As of version 2.4.8, Agda's reflection API provides a few keywords displayed in table 3. This feature can the thought of as the Template Haskell approach in Agda, or, meta programming in Agda.

The idea is to access the abstract representation of a term, during compile time, perform computations over it and return the resulting term before resuming compilation. We will not delve into much detail on Agda's abstract representation. The interested reader should go to Paul van der Walt's thesis[30], where, although somewhat outdated, Paul gives a in-depth explanation of reflection in Agda. However, the following excerpt gives a taste of how reflection looks like.

```
example : quoteTerm (λ x → suc x) ≡ con (quote suc) []
example = refl
```

Although very small, it states that the abstract representation of suc is a constructor, whose name is suc and has no arguments whatsoever. The Term datatype, exported from the Reflection module, has a large number of constructors and options. We are just interested in a small subset of such terms,

which is enough motive to build our own term datatype. That is the reason why we will not explain reflection in depth. Nevertheless, the next section provides a discussion on our interface to reflection.

Another interesting factor is to note how Agda normalizes a term before quoting it. Note how it performed a η-reduction automatically. To prevent this is the reason why we need all those record boilerplate in our Relational Algebra library.

quote	Returns a Name datatype, representing the quoted identifier.
quoteTerm	Takes a term, normalizes it and returns it's Term inhabitant.
quoteGoal g in f	Brings a quoted version of the goal in place into f's scope, namely, g.
quoteContext	Returns a list of quoted types. This is the ordered list of types of the local variables from where the function was called.
tactic f	Is syntax-sugar for quoteGoal g in (f quoteContext g).

Table 3: Agda 2.4.8 Reflection API

4.3 REPRESENTING TERMS

Before some confusion arises it is important to explain that we can not tackle rewriting in a standard fashion, like arbitrary term rewriting systems might do. Our problem is, in fact, slightly different. Given two structurally different terms and an action, we want to see: (A) if it is possible to apply such action in such a way that (B) it justifies the rewriting step. The first step is a simple matter of unification, which will be discussed later. However, the second part, justifying the rewrite, boils down to the generation of a subst that explains to Agda that what we are doing is valid. Our main goal here is to generate such subst.

The best way to explain how we implemented the *by* tactic is to work out a few examples by hand. This should provide the justification of some design decisions and give a tutorial on how to extend our framework.

We are going to focus on two scenarios. The first one is a simple associativity proof for list concatenation. Whereas the second is a relational proof of a trivial lemma.

```
++-assoc : ∀{a}{A : Set a}(xs ys zs : List A)
  → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc [] ys zs  = refl
++-assoc (x :: xs) ys zs = {! !}0
```

```
R⊆R•Id : {A B : Set}(R : Rel A B)
         → (R ⊆ R • Id) ⇐ Unit
R⊆R•Id R
  = begin
      R ⊆ R • Id
    ⇐⟨ {! !}1 ⟩
      R ⊆ R
    ⇐⟨ (λ _ → ⊆-refl) ⟩
      Unit
  □
```

One could fill in the holes manually, with the terms presented in table 4. Our task is to generate such terms automatically from the information supplied by Agda: the goal type and context list[1]; together with the action given by the user (identified by orange text in table 4).

```
Hole 0  =  cong (_::_ x) (++-assoc xs ys zs)
Hole 1  =  subst (λ x → R ⊆ x) (≡r-promote (•-id-r R))
```

Table 4: Solutions for holes 1 and 2

It is already easy to see how different relations will take different solutions. For hole one, a congruence suffices, since we're dealing with propositional equality and a data constructor (which is always a congruence). Hole two, on the other hand, needs a partially applied subst. And this will be the case whenever we're handling relational equality inside our *squiggol* environment. For the time being, however, let us forget about this detail. Indeed, let us explore the simpler example first.

Figure 1 gives the notation we are going to use to identify subterms in the given goal and action. We will denote the first goal by (hd_1, g_1, g_2) and a given action A by (hd_a, a_1, a_2). Inside the library, this is represented by a RBinApp, which forces all terms that we propose to handle to have a binary relation on their heads.

$$\text{Goal} \quad \underbrace{(x :: xs ++ ys) ++ zs}_{g_1} \overset{hd_g}{\equiv} \underbrace{x :: xs ++ (ys ++ zs)}_{g_2}$$

$$\text{Action} \quad \underbrace{(xs ++ ys) ++ zs}_{a_1} \overset{hd_a}{\equiv} \underbrace{xs ++ (ys ++ zs)}_{a_2}$$

Figure 1: Goal and Action for hole zero

An interesting observation is that if we take only the common subterms g_1 and g_2, we could obtain something like:

$$g_\square = g_1 \cap g_2 = (x :: \square ++ \square)$$

1 A context list is a list of types, in the same order as the De Bruijn indexes of the function parameters

Which is very close to the abstraction we are looking forward to derive. Yet, a congruence receives a unary function and our intersection is yielding a term with *two* holes. This behavior is actually correct, both g_1 and g_2 have a concatenation below _ :: _ x. This is easily fixed with a simple convention. We define a hole lifting operation where definitions whose arguments are all holes become a single hole. In the end, we arrive at:

$$g_{\square} = g_1 \cap g_2 = (x \ :: \ \square \ ++ \ \square) \overset{lift}{\rightarrow} (x \ :: \ \square)$$

This small intuition not only gives a simple decidable algorithm for deriving the abstraction from the goal type alone, but also gives some clues on how to represent terms. The term representation one is looking for has to support *holes*, or, behave as a zipper structure. So far we need a parametric simplification of Agda's Term. The representation we chose is:

```
data RTermName : Set where
  rcon : Name → RTermName
  rdef : Name → RTermName
  impl : RTermName


data RTerm {a}(A : Set a) : Set a where
  ovar : (x : A) → RTerm A
  ivar : (n : ℕ) → RTerm A
  rlit : (l : Literal) → RTerm A
  rlam : RTerm A → RTerm A
  rapp : (n : RTermName)(ts : List (RTerm A)) → RTerm A
```

Where RTermName distinguishes from definitions, constructors and function types. Such distinction is important once we need to convert these back to Agda's Term before unquoting. The RTerm functor is very straight-forward for a Haskell programmer. What might pass by unnoticed is how much more expressiveness one gets from a parametric type such as RTerm in Agda. The translation of an Agda term into a RTerm actually returns a RTerm ⊥, from which we can prove that there are no ovar occurrences (the actual object of type RTerm ⊥ *is* the proof).

It is also valid to ask why did we chose to add two constructors for representing variables. A specific, simple constructor, ivar, and a more general one, ovar. The reasons are manifold. Besides the usual excuse that a polymorphic type provides more expressiveness, we stress the importance of being able to differentiate *important* variables from the rest. Some algorithms also require more complicated types, rather than ℕ, for indexes.

Indeed, one important operation we can perform on RTems is the lifting of ivars to ovars, defined below.

```
lift-ivar : RTerm ⊥ → RTerm ℕ
lift-ivar = lift-ivar' 0
  where
    lift-ivar' : ℕ → RTerm ⊥ → RTerm ℕ
    lift-ivar' _ (ovar ())
    lift-ivar' d (ivar n) with d ≤? n
    ...| yes _ = ovar n
    ...| no  _ = ivar n
    lift-ivar' _ (rlit l) = rlit l
    lift-ivar' d (rlam t) = rlam (lift-ivar' (suc d) t)
    lift-ivar' d (rapp n ts) = rapp n (map (lift-ivar' d) ts)
```

Unfortunately, we can not simply substitute all ivar constructor by ovar. We could be shadowing variables by performing this operation carelessly. Another reason for adding a polymorphic constructor to our RTerm type will become clear in the types of the term operations we are going to declare. We will not provide the full source-code for them to improve readability.

The central operation is the intersection of two RTerms. It is used to find out *where* the differences are, and, therefore, providing the context for applying congruences or substitutions later.

```
_∩_ : ∀{A} ⦃ eqA : Eq A ⦄
  → RTerm A → RTerm A → RTerm (Maybe A)
```

Where a just is used to keep equal ovars and a nothing represents a hole, or, a difference in the arguments. Note that some representations are exactly the same. After getting a intersection term, with type RTerm (Maybe ⊥), we actually cast it to RTerm Unit. They are isomorphic. One could argue that the intersection type could be made simpler, but it wouldn't be as generic as we wanted our operations to be.

Having defined an intersection, which works somewhat as a term *road map*. It makes sense to be able to fetch the differences in two terms, or, in a term and a *road map*:

```
_-_ : ∀{A} ⦃ eqA : Eq A ⦄
  → RTerm (Maybe A) → RTerm A → Maybe (List (RTerm A))
```

Which takes two terms, one with holes and a regular one, and if they *match*, return a list of subterms from it's second argument that overlap the holes in the first one. As an example:

$$(x :: \square) - (x :: (xs ++ ys) ++ zs) = just\ ((xs ++ ys) ++ zs :: [])$$

$$(x :: \square) - ((xs ++ ys) ++ zs) = nothing$$

$$(x :: \square ++ \square) - (x :: (xs ++ ys) ++ zs) = just\ (xs ++ ys :: zs :: [])$$

From the intersection and difference operators we ca already extract a lot of information to complete our goal. The general algorithm is given in figure 2

1. Given the goal (hd_g, g_1, g_2) and action type (hd_a, a_1, a_2), we start by calculating the abstraction

$$g\Box = \uparrow (g_1 \cap g_2)$$

   where $\uparrow$ _ represents hole-lifting.

2. We proceed by separating the parts of the goal which still need to be handled, they are, for $i \in \{1,2\}, u_i = g\Box - g_i$.

3. Instantiation of u_1 with a_1 and u_2 with a_2 follows, this should give two substitutions σ_1 and σ_2.

4. Afterwards, it's just a matter of converting all this data back to a term. $\sigma = \sigma_1 +\!+ \sigma_2$ will contain the list of arguments we should pass to our action function.

Figure 2: Converting a RWData to a UData

The last step, of converting everything back to Agda's Term type, should be modular. A simple idea is to use different strategies triggered by specific (hd_g, hd_a) pairs. This should give us enough freedom to use our library even on very particular scenarios. Remembering, though, that the specific constructors of a given case study should be deeply embedded into, at least, a record. Otherwise, normalization is going to render terms incompatible. We are going to discuss this topic later.

4.3.1 *Complexity Remarks*

Since we do not provide the full code here, it is also out of scope to give a detailed complexity analysis. For the interested reader, the code is annotated with such analysis. In this section we will present only a few results where our reasoning is always in a worst-case setting. Two important measures over RTerms have to be introduced, that is, the size of a term and the set of free variables of a term:

```
S : {A : Set} → RTerm A → ℕ
S (ovar _) = 1
S (ivar _) = 1
S (rlit _) = 1
S (rlam t) = 1 + S t
S (rapp n ts) = 1 + sum (map S ts)
  where open import Data.List using (sum)
```

```
Fv : {A : Set} → RTerm A → List A
Fv (ovar a) = a :: []
Fv (ivar _) = []
Fv (rlit _) = []
Fv (rlam t) = Fv t
Fv (rapp _ ts) = concatMap Fv ts
  where open import Data.List using (concatMap)
```

Given a $t \in \mathsf{RTerm} A$, for all A, S_t denotes $(S\ t)$ and $\#\ Fv_t$ denotes $length(Fv\ t)$.

Term intersection $t_1 \cap t_2$ was found to run in $\mathcal{O}(min(S_t, S_u))$. Same thing goes for lifting after intersection, $\_\cap \uparrow \_$. An ideal version of term subtraction $t - u$ will run in $\mathcal{O}(\#Fv_t)$, the same reasoning applies to $\_- \downarrow \_$. Instantiation, inst t, which is discussed in the next section, given that t has at most n free variables, will run in $\mathcal{O}(n^2 + (S_t + 1) \times n + S_t)$.

Note that we chose to leave the complexities dependent both on the size of the term and on the number of free variables. This is because more often than not we will have terms with 2 or 3 free variables, with a size of 10 to 15. It is therefore wrong to prune out either parameter from our analysis.

Our unification strategy, displayed in figure 2, lets call it basic, when called as basic $(g_1 \equiv_g g_2)$ $(t_1 \equiv_t t_2)$, in the worst case, runs in, approximately, $\mathcal{O}(2t(t+S+1)+2S)$ time, for $t = max(\#Fv_{t_1}, \#Fv_{t_2})$ and $S = max(S_{t_1}, S_{t_2})$.

This shows that the action structure plays a bigger role than the goal itself, for the run time of our tactic. Such result was somewhat expected, since instantiation has a quadratic run time on its first parameter's ovars.

4.4 INSTANTIATION

Until now, we know how to extract a congruence (if possible) from the goal type only. Nevertheless, we have more information to use. The action type namely. From it, we should be able to extract the parameters we need to apply automatically. Let us take a look at the second example for this illustration, that is, hole 1 in table 4.

Lemma •-id-r receives a relation R and returns a proof of $R \equiv_r R \bullet Id$. Abstracting over variable names and adopting their De Bruijn index instead we get a tree, similar to figure 3. Whereas, performing the steps described in the previous section for the goal in question will result in $g_\square = R \subseteq \square$, $u_1 = g_\square - g_1 = R$ and $u_2 = g_\square - g_2 = R \cdot Id$.

Comparing u_1 with a_1 and u_2 with a_2 should give the intuition that, if the variable indexed by 0 is instantiated to R, the types match. In fact, R is the solution. Long story short, we need to obtain two agreeing substitutions σ_1 and σ_2 such that $\sigma_i\ a_i = u_i$, for $i \in \{1,2\}$. Such problem is known as instantiation. We heavily based ourselves in [27], with a few modifications to handle both implicit parameters and our more expressive term language. These adjustments will be discussed below, together with a more formal approach to the problem.

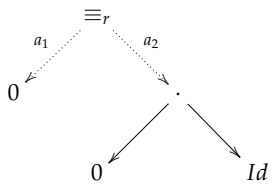


Figure 3: •-id-r's type

Before going any further with instantiation, we need to comply with Agda's termination checker. A neat way of doing so is to resort to RTerms indexed by a *finite* number of elements.

```
FinTerm : ℕ → Set
FinTerm = RTerm ∘ Fin
```

Given an action with n visible, that is, not implicit, variables, we therefore will need to instantiate all n of its parameters to be able to obtain the desired result. However, the order in which we find their instances might not be the usual natural number ordering. We therefore rely on a partial substitution datatype defined by a vector where each position i contains either a closed term, meaning that the i-th De Bruijn variable of our action should be instantiated to that closed term or a symbolical value representing that we did not find an instance for such variable.

```
X : ℕ → Set
X = Vec (Maybe (RTerm ⊥))
```

The empty substitution is easily defined by recursion on the number of elements we wish to instantiate.

```
empty-X : {n : ℕ} → X n
empty-X {zero}  = []
empty-X {suc n} = nothing :: empty-X
```

A few simple operations are also needed before going further:

```
lookup-X : {n : ℕ} → Fin n → X n → Maybe (RTerm ⊥)
lookup-X fz  (x :: _) = x
lookup-X (fs i) (_ :: s) = lookup-X i s

{-# TERMINATING #-}
apply-X : {n : ℕ} → X n → FinTerm n → Maybe (RTerm ⊥)
apply-X s (ovar x) = lookup-X x s
apply-X s (ivar n) = just (ivar n)
apply-X s (rlit l) = just (rlit l)
apply-X s (rlam t) = rlam <$> apply-X s t
apply-X s (rapp n ts) = rapp n <$> mapM (apply-X s) ts
```

Where _<$>_ is the same as in Haskell's Control.Applicative, and mapM is the lifting of map to the Maybe monad. The apply-X function is straight forward. An important note, however, is that we just substitute ovars. This distinction between *out* variables and *in* variables is more important than what initially meets the eye.

4.4. Instantiation

Relaxing the syntax to improve readability, we can say that the general form of an Agda type is composed of n implicit parameters, k explicit parameters and a, possibly dependent, conclusion.

$$aux \ : \ \{i_1\}\{i_2\}\cdots\{i_n\} \to (e_1)(e_2)\cdots(e_k) \to C\ i_1 \cdots i_n\ e_1 \cdots e_k$$

Therefore we have a total of $n+k$ parameters. Yet, whenever we need the conclusion from aux we just need to provide k arguments, since the first n implicit parameters are going to be figured out by Agda. It follows that our instantiation algorithm only needs to figure out k parameters. Rewriting the above type with De Bruijn indices we have:

$$aux \ : \ \{n+k-1\}\{n+k-2\}\cdots\{k\} \to (k-1)(k-2)\cdots(0) \to C\ (n+k-1)\cdots 0$$

We are not really concerned with all the implicit parameters anyway. If they can not be guessed by Agda, an error will happen at unquoting. We proceed to model the variables we need to instantiate as an ovar, whereas the other ones become a ivar. The instantiation algorithm is given below.

```
mutual
  instAcc : {n : ℕ} → FinTerm n → RTerm ⊥ → X n → Maybe (X n)
  instAcc (ovar x) t s = extend-X x t s
  instAcc (ivar _) _ s = just s
  instAcc (rlit l) (rlit k) s with l ≟-Lit k
  ...| yes _ = just s
  ...| no  _ = nothing
  instAcc (rlam t) (rlam u) s = instAcc t u s
  instAcc (rapp n ts) (rapp m us) s with n ≟-RTermName m
  ...| no  _ = nothing
  ...| yes _ = instAcc* ts us s
  instAcc _ _ _ = nothing

  instAcc* : {n : ℕ} → List (FinTerm n) → List (RTerm ⊥) → X n → Maybe (X n)
  instAcc* [] [] s = just s
  instAcc* [] (_ :: _) _ = nothing
  instAcc* (_ :: _) [] _ = nothing
  instAcc* (x :: xs) (y :: ys) s = instAcc x y s >>= instAcc* xs ys

inst : {n : ℕ} → FinTerm n → RTerm ⊥ → Maybe (X n)
inst t u = instAcc t u empty-X
```

The inst function receives one term with n ovars and a closed term. It returns a partial substitution for n variables and it works by calling an auxiliary version with a substitution as its state. Notice how an

ivar does not change the substitution and is simply accepted, whereas an ovar triggers an extension, which will be discussed later.

The return type being a Maybe (X n) has a somewhat confusing interpretation. If instantiation returns nothing, it means that the terms have an incompatible structure. On the other hand, if it returns a just σ there are two possible cases. Either σ is complete or not. In the former situation, everything is fine and we have just figured out our parameters. The later scenario means that we did not have enough information to instantiate every variable *yet*. Nevertheless, given two partial substitutions we might be able to build a complete one, as long as they agree on the variables defined on both and there is no variable undefined on both. Substitution concatenation is given by $\_{+}{+}_x\_$.

```
_++x_ : {n : ℕ} → X n → X n → Maybe (X n)
[] ++x [] = just []
(x :: xs) ++x (y :: ys) with x | y
...| nothing | just y' = _::_ y <$> (xs ++x ys)
...| just x' | nothing = _::_ x <$> (xs ++x ys)
...| nothing | nothing = nothing
...| just x' | just y' with x' ≟-RTerm y'
...| no _ = nothing
...| yes _ = _::_ x <$> (xs ++x ys)
```

The extension of a substitution σ for $n = t$ is a simple traverse to the desired position, n, and, if such variable was not yet figured we simply return t. If it was already defined to a t', after all, then they must agree, that is, $t = t'$. This requirement preserves the injectivity of the substitution being computed.

```
extend-X : {n : ℕ} → Fin n → RTerm ⊥ → X n → Maybe (X n)
extend-X fz t (nothing :: s) = just (just t :: s)
extend-X fz t (just t' :: s) with t ≟-RTerm t'
...| yes _ = just (just t :: s)
...| no _ = nothing
extend-X (fs i) t (mt :: s) = _::_ mt <$> extend-X i t s
```

Representing terms abstractly, operations defined on such representation and the instantiation algorithm are all the pieces we need to figure out abstractions and parameters to actions. In the next section we are going to discuss how do they fit together and how the users can extend the library to suit their needs.

4.5 THE *by* TACTIC

Computing a term that allows a *rewrite*, which is our main task here, is divided in three parts. First we extract data from the goal and action type. We make sure that these terms have the expected form and satisfy the requirements imposed on them. The dependent type setting allows us to encode such requirements at the type-level. This data is called RWData.

```
record RWData : Set where
  constructor rw-data
  field
    goal   : RBinApp ⊥
    actℕ   : ℕ
    act    : RBinApp (Fin actℕ)
    ctx    : List (RTerm ⊥)
```

Note how we can be sure that both the goal is a closed binary application and the action has the correct number of variables to be instantiated, since $act\mathbb{N}$ represents the action arity. The context is unused so far, but since it is given by the tactic keyword we decided to keep it there. To compute such structure one calls:

```
make-RWData : Name → AgTerm → List (Arg AgType) → Err StratErr RWData
make-RWData act goal ctx
  with η (Ag2RTerm goal) | Ag2RTypeFin (type act) | map (Ag2RType ∘ unarg) ctx
...| g' | tyℕ , ty | ctx' with forceBinary g' | forceBinary (typeResult ty)
...| just g | just a = return (rw-data g tyℕ a ctx')
...| _ | _ = throwError (Custom "...")
```

where Ag2RTypeFin is defined by:

```
Ag2RTypeFin : AgType → ∃ FinTerm
Ag2RTypeFin = R2FinType ∘ lift-ivar ∘ η ∘ Ag2RType
```

Which means that we first translate a type normally, for which we get a RTerm ⊥, then we lift all ivars there to ovars. R2FinType finishes up by converting the variables n such that $n \leq actℕ$ to a finite representation and downgrades other variables back to ivars. This gives a term that can be passed directly to instantiation without further processing. Note the η-reduction being forced upon the terms before instantiation. This is due to relations being of type $A \to B \to$ Set, and, sometimes, Agda will add superfluous abstractions during normalization.

Having computed a RWData, we need to extract the congruence (resp. substitution) and the parameters, if any, to give to the action. That is, we are now going to compute some *unification data* out of a RWData.

```
record UData : Set where
  constructor u-data
  field
    □ : RTerm Unit
    σ : RSubst
    trs : List Trs
```

Which is composed of a single-hole abstraction, named □. A substitution for the action parameters, translated to a list of $\mathbb{N} \times$ RTerm $\bot$ for easier handling and a list of transformations to be applied. Let us postpone this last detail a bit longer. Section 4.3 gives a description of how to compute □ and what to instantiate with what afterwards. These could be summarized by the following unification strategy.

```
basic : RWData → Err StratErr UData
basic (rw-data (hd_x , g1 , g2) tn (hd_a , ty1 , ty2) _)
  = let g□ = g1 ∩↑ g2
    u1 = (g□ -↓ g1) »= (inst ty1)
    u2 = (g□ -↓ g2) »= (inst ty2)
    σ  = μ ((_++_x_ <$> u1) <*> u2)
  in maybe (λ s → i2 (u-data (⊥2UnitCast g□) s [])) (i1 NoUnification)
       (σ »= X2RSubst)
```

Some housekeeping functions are present to make sure we perform the correct conversions. Nevertheless, the statements inside the let block correspond to the description given in section 4.3. It is evident that wrapping everything in an Error monad makes programming much easier. Not only that, but it also allow us to compose strategies in a neat fashion:

```
runUStrats : RWData → Err StratErr UData
runUStrats = basic <|> basic-sym
```

Where basic-sym is the same as basic, but instantiates the first part of the goal with the second parameter of the action and vice-versa. If such instantiation yields a substitution, we add *symmetry* to the list of transformations to be applied. In fact, it is the only transformation we support so far. The alternative operator has the same Haskell semantics for the Error monad.

If we are able to compute a UData successfully, the last step is to convert all that information back into a term. This is where customization can take place.

```
record TStrat : Set where
  field
    when : RTermName → RTermName → Bool
    how  : Name → UData → Err StratErr (RTerm ⊥)

TStratDB : Set
TStratDB = List TStrat
```

The when field is a predicate over the goal relation name and the action relation name, in that order, which specifies when how should be applied. Whereas how receives the action name and unification data and should generate a closed term to close the goal in question.

As a simple example, the TStrat for handling propositional equality is shown below.

```
module PropEq where

  pattern pat-≡ = (rdef (quote _≡_))

  private
    open UData

    ≡-when : RTermName → RTermName → Bool
    ≡-when pat-≡ pat-≡ = true
    ≡-when _   _ = false

    fixTrs : Trs → RTerm ⊥ → RTerm ⊥
    fixTrs Symmetry term = rapp (rdef (quote sym)) (term :: [])

    ≡-how : Name → UData → Err StratErr (RTerm ⊥)
    ≡-how act (u-data g□ σ trs)
      = i2 (rapp (rdef (quote cong))
                 ( hole2Abs g□
                 :: foldr fixTrs (makeApp act σ) trs
                 :: [])
           )

  ≡-strat : TStrat
  ≡-strat = record
    { when = ≡-when
    ; how  = ≡-how
    }
```

4.5.1 *Interfacing*

One interesting feature of Agda is the possibility of providing *module parameters*. Importing the rewrite module requires a *term strategy* database:

```
module RW (db : TStratDB) where
```

Not only this makes the code very modular, but also allows one to use different strategies for the same relations, if the need arise.

Finally we have been through all the ingredients we need to build our automatic rewrite tactic. It comes in two flavors. The first one returns a bunch of information useful for debugging whereas the later is intended to be used for production.

```
by' : Name → List (Arg AgType) → AgTerm → (RWData × UData × RTerm ⊥)
by' act ctx goal with runErr (make-RWData act goal ctx »= RWerr act)
...| i1 err  = RW-error err
...| i2 term = term

by : Name → List (Arg AgType) → AgTerm → AgTerm
by act ctx goal = R2AgTerm ∘ p2 ∘ p2 $ (by' act ctx goal)
```

Using the by tactic should be straight-forward. Just import the desired term strategy, pass it as a parameter to the RW module and it will be good to go. The overhead introduced by our tactic at compile time is very acceptable.

4.5.2 *Closing Remarks*

This chapter presented the construction of a meta-programming tool to help mathematicians with mechanical steps in the development process they usually face. Agda proved to be a tool with enough strength for that task. In fact, a very interesting detail is how we can use Agda itself, to generate Agda programs. This contrasts with most proof assistants that provide custom tactics. Coq, for instance, has three different languages for creating customized tactics.

We reiterate that, although this chapter does not display the full source code of our tactic, the tactic is available on GitHub for the community to test, fork and modify! The repository can be found at the following url.

`https://github.com/VictorCMiraldo/agda-rw`

5

VARIATIONS OF THE *BY* TACTIC

Chapter 4 discussed the construction of a tactic that handles *one* goal, given *one* action. What if we tried to generalize this? The best case scenario would be the handling a number of goals and actions at the same time, maintaining a linear time complexity. That is a very hard achievement! Such tool would be a completely automatic theorem prover. By compiling all theorems and lemmas of a given theory in Agda and ask for proofs, such ideal tool would return the proof objects regardless of the theorem to be proved.

It is plausible, however, to try to handle either *multiple* goals, with given actions or to find the correct action, given *multiple* actions, to solve *one* goal. These possible extensions to the library are the object of discussion of this chapter. Sections 5.1 and 5.2 explore those options, respectively.

5.1 MULTIPLE GOALS

Given a transitive rewrite relation, we can, at least, specify how one would compute the intermediate goals. The implementation, however, is somewhat complex. The biggest problem, which is the size of the search space, does not have an easy workaround. Nevertheless, some contexts have an acceptable run time.

Lets imagine we want to prove an exchange law for natural numbers and sum, something in the line of:

$$\mathsf{exch} : (x\ y : \mathbb{N}) \to x + y \equiv y + (x + 0)$$

Since that sum is commutative and associative, this should be pretty straight forward given such lemmas.

$$\mathsf{+\text{-}comm} : \forall\ m\ n \to m + n \equiv n + m$$

```
+-id : ∀ n → n + 0 ≡ n
```

One possible implementation for exch could be:

```
exch x y = trans (+-comm x y) (cong (λ w → y + w) (sym (+-id x)))
```

Which is a good place to start. We are looking forward to derive something close the above right-hand-side term. The two arguments for trans can already be derived using the by tactic. Unfortunately, we need a concrete goal to call by, otherwise we can not infer the context (abstraction) where the rewrite happens.

The term intersection we defined in the previous chapter is of no use here, since in general, rewrites do *not* happen in independent subterms. One option is to simply substitute, according to +-comm's type, *one* occurrence of $a + b$ for $b + a$ in $g_1 = x + y$, repeating the procedure for +-id. The possibilities are displayed in figure 4

Note that if we apply first +-id, the amount of possibilities increases drastically.

The implementation uses the List monad to model non-deterministic computations. The core idea is modeled by the following three functions. For the more curious reader, we reiterate that the complete code is available online.

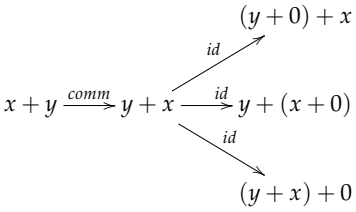


Figure 4: Substitution outcomes

As we already mentioned, the ability to non-deterministically perform a *single* substitution in a term is central. Given a term g and a pair of terms m, n, $g[m, n]$ will return a term with *one* occurrence of m substituted by n.

```
_[_] : {A : Set}{{eqA : Eq A}} → RTerm A → RTerm A × RTerm A → NonDet (RTerm A)
```

Note how the type of _[_] requires all terms involved to have the same parameter A, also remember that in the previous section we modeled actions with n explicit parameters as $FinTerm\,n$. Yet, our goal have type $RTerm\,\bot$. This type mismatch captures the instantiation problem, depicted in figure 4. In fact, we need to chose parameters from our goal to feed the given action, and only then substitute.

```
{-# TERMINATING #-}
apply : {n : ℕ} → RTerm ⊥ → RBinApp (Fin n) → NonDet (RTerm ⊥)
```

The function apply g t, for t : FinTerm n, is computed by induction in n. When $n = 0$ it means our action have no more free variables and we can substitute $g[t_1, t_2]$. When $n = k + 1$, however,

apply will non-deterministically instantiate the last parameter, transforming t to t' : FinTerm k, and call itself recursively. The finite type workaround does not fully work to prove termination, which have to still be explicitly mentioned here.

We finish off with a function that will select, from all the chains produced with _[_] and apply, those that start and finish with the correct terms.

```
divideGoal : RBinApp ⊥ → List (Σ ℕ (RBinApp ∘ Fin)) → Maybe (List (RTerm ⊥))
```

After dividing our goal into a list of intermediate terms, it is just a matter of applying the tactic to every single one of them and combining everything with a suitable transitivity function. It is important to note that there are two different explosions of our search space here. The number of actions we want to apply in a row will determine how *tall* our tree will grow, whereas a bigger number of parameters of the actions itself will result in a *wider* tree, that is, more bifurcations at our nodes.

Our approach here is very naive. The apply function will search every possible instantiation of a given action's parameters. We could rule out those that do not typecheck, for instance. This also motivates an addition to Agda's reflection API. A function with type AgTerm → AgTerm → Bool that returns true when its first argument has its second argument as a possible type. Such optimization would reduce the search space, unfortunately not in a significant manner. Using heuristics could also be an interesting optimization.

5.2 MULTIPLE ACTIONS: A BETTER SEARCHING ALGORITHM

The handling of multiple actions for the same goal would bring us closer to what *auto* or *tauto* do in Coq. We could have whole theories compiled in some sort of database and let our tool figure out which of those lemmas to apply. Since the goal is fixed, there should be one solution. A naive extension to our library could be made:

```
by+ : List Name → List (Arg AgType) → AgTerm → AgTerm
by+ [] _ _ = RW-error "No suitable action"
by+ (a :: as) ctx goal with runErr (make-RWData a goal ctx >>= RWerr a)
...| i1 _ = by+ as ctx goal
...| i2 t = R2AgTerm ∘ p2 ∘ p2 $ t
```

However, a List of names is a bad data structure to use here. We can see that by borrowing a few definitions from the complexity of our by tactic, section 4.3.1. Without loss of generality, let us assume that we are going to search a list of n tactics. Let us assume that a given action t maximizes the function $2 \times \#Fv_t(\#Fv_t + S_t + 1) + 2S_t$. Our by+ tactic will belong to $\mathcal{O}(n \times (2 \times \#Fv_t(\#Fv_t + S_t + 1) + 2S_t))$, for we might need to search every other $n-1$ actions before finding the correct one.

Is it possible, however, to rule out actions purely based on their structure? Indeed, we are going to explore an alternative data structure to speed up search and get instantiation for free, during lookup.

This section starts with a summarized explanation of Tries and then follows with our generalization of the same concept. The structure we arrive at is a type-indexed data structure, in the sense of [13]. Yet, our lookup and insertion methods have to be significantly different since we have to handle variables, that can be arbitrarily instantiated.

5.2.1 *Tries*

There a few variations on of a Trie, suited for different applications. We will, however, restrict ourselves to the original definition, which is also the simplest. This small digression has the purpose of introducing the underlying idea without too many technicalities.

Figure 5 shows a trie that stores the string set $S = \{"t","to","tot","tota","total","te","tea","ten"\}$. A few important notions to note are that, even though it has a tree structure, the keys are not associated with a specific node, but with its positioning in the trie. In the end of the day, we can look at Tries as finite deterministic acyclic automatons.

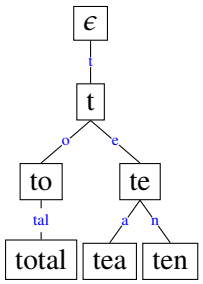


Figure 5: Trie example

The specification of a Trie is fairly simple. Taking already a small step towards a generalization and assuming that instead of storing lists of characters (strings) we are going to store arbitrary lists, the Trie functor is given by:

$$T\ a\ k = \mu X.(k+1) \times (a \to X+1)$$

So, each node contains a possible key and a partial map from a's to Tries. The retrieval of the key associated with a list l is performed by recursion on l. When we reach the empty list we return the $(k+1)$ part of the current node. Otherwise, we try to traverse the trie associated with the current node partial map applied to the head of the list.

Insertion is also very straight forward. Inserting a key k indexed by a list l, assuming l does not belong in the trie, yet, consists of a walk in the trie, by induction l, adding entries to the partial maps, if needed, until we reach the empty list, and then register k at that node.

The biggest limitation of a Trie, however, is that it can only be indexed by a linear data-structure, such as a list. R. Hinze, J. Jeuring and A. Löh provides a solution to that, in [13], by defining a special

kind of Tries that can be indexed by arbitrary data-types. The idea, in a nutshell, is to work with the algebraic representation of the indexes (data-types) and have nested maps at every node. This is a bit too general for our needs here. We are looking for a structure that can be indexed by a given term language, which follows the typical sum-of-products form. An additional complication arises when we start to allow variables in the index.

5.2.2 *Towards a Generalization*

Following the popular saying – A picture is worth a thousand words – let us begin with a simple representation. Just like a trie, our RTrie trie was designed to store names of actions to be performed, indexed by their respective type. Figure 7 shows the RTrie that stores the actions from figure 6. For clarity, we have written the De Bruijn indexes of the respective variables as a superscript on their names.

$$\begin{aligned} x^0 + 0 &\equiv x^0 \\ x^0 + y^1 &\equiv y^1 + x^0 \\ x^2 + (y^1 + z^0) &\equiv (x^2 + y^1) + z^0 \end{aligned}$$

Figure 6: Identity, Commutativity and Associativity for addition.

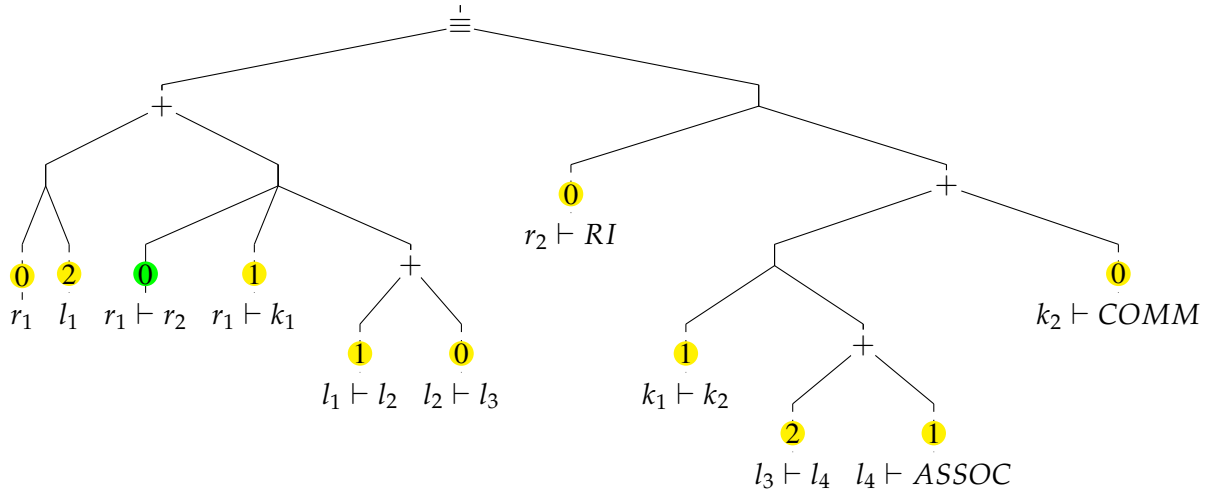


Figure 7: RTrie for terms of figure 6. Yellow and green circles represent DeBruijn indexes and literals, respectively.

Let us illustrate the lookup of, for instance, $(2 \times x) + 0 \equiv 2 \times x$. The search starts by searching in the root's partial map for $\equiv$. We're given two tries! Since $\equiv^{\star_1}$ is a binary constructor. We proceed by

looking for $(2 \times x) + 0$ in the left child of $\equiv$. Well, our topmost operator is now a $+^{\star_2}$, we repeat the same idea and now, look for $(2 \times x)$ in the left child of the left $+$. Here, we can choose to instantiate variable 0 as $(2 \times x)$ and collect label r_1 or instantiate variable 2, with the same term, and collect label l_1. Since we cannot know beforehand which variable to instantiate, we instantiate both! At this point, the state of our lookup is $(0 \mapsto 2 \times x, r_1) \vee (2 \mapsto 2 \times x, l_1)$. We proceed to look for the literal 0 in the right trie of $+^{\star_2}$, taking us to a leaf node with a rewrite rule stating $r_1 \vdash r_2$. This reads r_1 should be rewritten by r_2. We apply this to all states we have so far. Those that are not labeled r_1 are pruned. However, we could also instantiate variable 1 at that node, so we add a new state $(0 \mapsto 2 \times x, 1 \mapsto 0, k_1)$. At this step, our state becomes $(0 \mapsto 2 \times x, r_2) \vee (0 \mapsto 2 \times x, 1 \mapsto 0, k_1)$.

We go up one level and find that now, we should look for $2 \times x$ at the right child of $\equiv^{\star_1}$. We cannot traverse the right child labeled with a $+$, leaving us with to compare the instantiation gathered for variable 0 in the left-hand-side of $\equiv^{\star_1}$ to $2 \times x$. They are indeed the same, which allows us to apply the rule $r_2 \vdash RI$. Which concludes the search, rewriting label r_2 by RI, or, any other code for +-right-identity. By returning not only the final label, but also the environment gathered, we get the instantiation for variables for free. The result of such search should be $(0 \mapsto 2 \times x, RI) :: []$.

This small worked example already provides a few insights not only on how to code lookup, but also on how to define our RTrie. We have faced two kinds of nodes. Fork nodes, which are composed by a list of cells, and, Leaf nodes, which contain a list of (rewrite) rules. Indeed, we define RTrie by:

```
data Rule : Set where
  Gr : ℕ  → Rule
  Tr : ℕ → ℕ → Rule
  Fr : ℕ → Name → Rule

mutual
  Cell : Set
  Cell = IdxMap.to RTrie default
    × List (ℕ × List Rule)

  data RTrie : Set where
    Fork : List Cell → RTrie
    Leaf : List Rule → RTrie
```

Rewrite rules are divided in three different kinds: Gather, Transition and Final rules. The type to comes from `RW.Data.PMap X IsEqX`, and represents a partial map from X to some other type. The default modifier adds a default value to the map, making it possible to totalize it.

The idea is to separate the binding symbols from the non-binding ones. The second component of the Cell product correspond to these binding symbols. We used a list to be able to easily map over it. The IdxMap.to on the other hand, is used to store non-binding indexes, where the default value of such

map will give us *free* rules to apply when nothing else can be done. They act in a similar way as just accepting ivars, like we did with the instantiation algorithm, in section 4.3.

5.2.3 *Polymorphic vs. Monomorphic*

In a first attempt to encode our RTrie in Agda, we went generic, polymorphic. Type variable t is the index type and c is the conclusion type, which we are being indexed by different elements $t_1, \cdots t_n : t$. The assumption we make, however, states that t must have a IsTrie instance. Such record is defined by:

```
record IsTrie (t : Set) : Set1 where
  field
    Idx : Set
    idx≡ : Eq Idx

    toSym  : Idx → Maybe ℕ
    fromSym : ℕ  → Maybe Idx

    in_t   : Idx × (List t) → t
    out_t  : t → Idx × (List t)
```

Stating that for a given t, there exists an index type Idx, with a decidable underlying equality. We must be able to at least differentiate between different indexes, after all. We also state the we must be able to open and close such term. It is easy to prove that for each sum-of-products recursive type, we can construct a type Idx satisfying these properties. Just ignore the recursive arguments and make a label for each constructor.

This generic approach works reasonably well for simple cases. In our scenario, however, we need more control on the lookup and insertion functions. This topic will be explained further in the following sections. Nevertheless, we have the indexes for RTerm datatype defined by:

```
data RTerm_i {a}(A : Set a) : Set a where
  ovar_i : (x : A) → RTerm_i A
  ivar_i : (n : ℕ) → RTerm_i A
  rlit_i : (l : Literal) → RTerm_i A
  rlam_i : RTerm_i A
  rapp_i : (n : RTermName) → RTerm_i A
```

One interesting part, however, is the toSym function. These Symbols are the binding symbols. We are stating that some of our indexes are in fact binding symbols, therefore we need a way to both dis-

tinguish between binding and non-binding indexes, and convert to a standard representation. DeBruijn indexes were adopted as this representation.

```
out : ∀{a}{A : Set a} → RTerm A → RTerm_i A × List (RTerm A)
out (ovar x)  = ovar_i x , []
out (ivar n)  = ivar_i n , []
out (rlit l)  = rlit_i l , []
out (rlam t)  = rlam_i , t :: []
out (rapp n ts) = rapp_i n , ts

toSymbol : ∀{a}{A : Set a} → RTerm_i A → Maybe A
toSymbol (ovar_i a) = just a
toSymbol _  = nothing

idx-cast : ∀{a b}{A : Set a}{B : Set b}
  → (i : RTerm_i A) → (toSymbol i ≡ nothing)
  → RTerm_i B
idx-cast (ovar_i x) ()
idx-cast (ivar_i n) _ = ivar_i n
idx-cast (rlit_i l) _ = rlit_i l
idx-cast (rlam_i  ) _ = rlam_i
idx-cast (rapp_i n) _ = rapp_i n
```

Note that whenever an index is *not* a symbol, we can cast it to arbitrary types. This will be very important if we want to maintain the type of the interface, in the sense that we only return closed instantiations, or, maps from $\mathbb{N}$ to RTerm $\bot$.

5.2.4 *Lookup*

Looking up a term in a RTrie, as we stated in the example given in the beginning of this section, will involve a fair amount of components. Here, we will just give the outline of the algorithm. The details can be checked directly in the source code. The same will happen with insertion.

Most of the lookup functionality will happen inside a reader monad, that is, the partially applied function type: $(r \rightarrow)$, for some r. In our case, we will read list of configurations, representing the disjunctions used in the example. Henceforth, we have

```
L : Set → Set
L = Reader (List Lst)
```

where Lst represents a single state, or, an instantiation with a possible label representing the last path we took on the RTrie. In Agda:

```
Label : Set
Label = ℕ ⊎ Name

Lst : Set
Lst = ℕmap.to (RTerm ⊥) × Maybe Label
```

Retrieving a term from a RTrie relies mostly on the rewrite rules. The ruleList function will try the application of a list of rules to *all* states it receives. If we cannot apply a rule, we prune that state out of the search.

```
ruleList : List Rule → L (List Lst)
```

The main component is the following function, with its smaller helper function types presented below.

```
lkup-aux : RTerm ⊥ → RTrie → L (List Lst)
lkup-aux _ (Leaf r) = ruleList r
lkup-aux k (Fork (((d , rs) , bs) :: []))
  = let tid , tr = out k
  in lkup-inst k bs
  >>= λ r → maybe (lkup≡just tr) (lkup-aux k d) (IdxMap.lkup tid rs)
  >>= return ∘ (_++_ r)
  where
    lkup≡just : List (RTerm ⊥) → RTrie → L (List Lst)
    lkup≡just [] (Leaf r) = ruleList r
    lkup≡just _  (Leaf _) = return []
    lkup≡just tr (Fork ms) = lkup-list (zip tr ms)

lkup-list : List (RTerm ⊥ × Cell) → L (List Lst)

lkup-inst : RTerm ⊥ → List (ℕ × List Rule) → L (List Lst)
```

Spelling out the definition, we can see that looking something up in a Leaf is just a simple application of the rules in that leaf. The term itself does not matter, since even if we can extract an index out of it, we have nowhere to traverse next. Retrieving data from a Fork with one cell is a bit trickier, though. We can either instantiate some variables and proceed, or lookup the term index to find out which subtree to traverse. If we do not find any, we look something up in the default branch. This default branch is useful for simulating the behavior instantiation had with ivars, of accepting ivars without checking anything. We return the concatenation of the configurations produced by instantiation and recursion.

Instantiation deserves a more in-depth discussion. Given a state lst and, a cell with possible instantiations bs : List ($\mathbb{N}\times$List Rule), and a term t. We can choose to instantiate *one* $b \in bs$ at a given time. We will end up with a list of states $map\ (instantiate\ lst)\ bs$, representing the disjunction of possible instantiation choices we have at the moment. Now, lets fix a $b \in bs$ and explore the $instantiate$ function. If b is unbound in lst yet, we simply add an entry $b \mapsto t$ and apply the rules associated with such binding. If b is already instantiated to $lst(b)$, we need to check whether or not everything is right. If $lst(b) \equiv t$, we can apply the rules and continue. Otherwise, we discard such state since it is carrying an inconsistent instantiation.

5.2.5 *Insertion*

Inserting a new key in an already existing RTrie is not too different from insertion in a regular trie. Lets say we want to insert term t, with $out\ t \equiv (t_{id}, t_r)$, in a cell $((def, m_{idx}), m_{sym})$, where (def, m_{idx}) is the total map from indexes to RTries and m_{sym} is the list of possible bindings at this node. Now, everything depends on what t_{id} turns out to be. Case t_{id} is:

1. rlit$_i$ l; which implies $t_r \equiv []$, and it means that we are going to either add a Leaf node in our m_{idx} map, or append rules to an existing $m_{idx}(rlit_i\ l)$.
2. rlam$_i$; with $t_r \equiv [t']$. Just like the above case, we are going to traverse m_{idx}. Lets assume $m_{idx}(rlam_i) = r$ (if $rlam_i \notin dom(m_{idx})$, we simply add a $rlam_i \mapsto RTrieEmpty$), we proceed by inserting t' in r.
3. rapp$_i$ n; in this situation t_r might be an arbitrary list of terms. Assuming $m_{idx}(rapp_i\ n) = Fork\ l$, we proceed by inserting each t_r in their respective cell in l. Or, $map(uncurry\ insert)(zip\ t_r\ l)$.
4. ovar$_i$ x; we found a binding symbol! We need to modify m_{sym} here. We simply construct a new rule and either add or append it to $m_{sym}(n)$, depending on whether or not $m_{sym}(n)$ was already defined.
5. ivar$_i$ x; we will change the default value of our index map. This means that we are going to ignore ivar$_i$, just like we did with instantiation, in our first approach. The modification applied to d is to simply add a new rule to it.

The implementation revolves around a state monad, carrying a label and a counter to generate new rules. That is considered housekeeping and we will skip through it.

```
I : ∀{a} → Set a → Set a
I {a} = STₐ {Iz} {a} (Σ ℕ (λ _ → Maybe ℕ))
```

Steps 1 to 5 have a interesting similarity. They all traverse the trie and, some of them, must perform changes *after* the recursive step has been completed. This is very close to what a zipper does, and can be achieved with one.

```
CellCtx : Set _
CellCtx = RTrie → Cell
```

Let us define a function $\mathcal{M}$ that given a Cell and an index i, will traverse the cell following i's *direction*, just like described above. The return value, however, is a pair containing a RTrie to be traversed by the caller for the recursive case and a context, to reconstruct the cell with the given RTrie.

```
ℳ : {A : Set}{{enA : Enum A}}
  → Cell → RTerm_i A → I (CellCtx × RTrie)
ℳ {{enum aℕ _}} c tid with toSymbol tid | inspect toSymbol tid
...| nothing | [ prf ] = mIdx c (idx-cast tid prf)
...| just s  | _  = maybe (mSym c) enum-err (aℕ s)
  where postulate enum-err : I (CellCtx × RTrie)
```

We divide it in two other functions. Either we are handling a binding symbol or a simple index.

```
mIdx : Cell → RTerm_i ⊥
       → I (CellCtx × RTrie)
mIdx ((d , mh) , bs) (ivar_i _)
  = return $ (λ bt → (merge d bt , mh) , bs) , BTrieEmpty
  where
    merge : RTrie → RTrie → RTrie
    merge (Leaf as) (Leaf bs) = Leaf (as ++ bs)
    merge _ bt  = bt
mIdx ((d , mh) , bs) tid
  = let mh' , prf = IdxMap.alterPrf BTrieEmpty tid mh
  in return $ (λ f → (d , IdxMap.insert tid f mh) , bs)
    , (IdxMap.lkup' tid mh' prf)

mSym : Cell → ℕ
  → I (CellCtx × RTrie)
mSym (mh , bs) tsym with ℕmap.lkup tsym bs
...| nothing = makeRule
      >>= λ r → return (const (mh , (tsym , r :: []) :: bs)
        , BTrieEmpty)
...| just rs = handleRules rs
      >>= λ r → return (const (mh , ℕmap.insert tsym r bs)
        , BTrieEmpty)
```

If we are handling a binding symbol, then it is obvious we do not have any recursive argument to take care of. Note how mSym returns a constant context and the empty RTrie. When we handle indexes, however, note how we just change the default value of the index map in case we find an ivar_i.

Our life becomes much easier now. Imagine we need to insert an opened term (t_{id}, t_r) in a Cell c. Well, by calling $\mathcal{M}\ c\ t_{id}$ we are going to get a context ctx and a RTrie rt to insert our t_r into. Assuming this insertion results in an RTrie rt', we just need to reconstruct our current cell with $ctx\ rt'$.

Indeed, the main insertion function is defined by

```
{-# TERMINATING #-}
insCell : {A : Set}{{enA : Enum A}}
  → RTerm_i A × List (RTerm A) → Cell → I Cell
insCell (tid , tr) cell
  = M cell tid
  >>= λ { (c , bt) → insCellAux tid tr bt >>= return ∘ c }
  where
    tr≡[] : {A : Set}{{enA : Enum A}}
      → RTerm_i A → I RTrie
    tr≡[] tid with toSymbol tid
    ...| nothing = (Leaf ∘ singleton) <$> makeRule
    ...| _ = return $ Fork []

    insCellAux : {A : Set}{{enA : Enum A}}
      → RTerm_i A → List (RTerm A) → RTrie
      → I RTrie
    insCellAux tid _ (Leaf r) = return (Leaf r)
    insCellAux tid [] _ = tr≡[] tid
    insCellAux tid tr (Fork [])
      = Fork <$> insCell*ε tr
    insCellAux tid tr (Fork ms)
      = Fork <$> insCell* tr ms
```

Afterwards, it is just a matter of running the monad and providing a cleaner interface. After we finished inserting the term we traverse the tree again substituting the last used label by the name we want to store.

Insertion has poor performance, mostly due to the reconstruction needed at every step. A good idea is to trick Agda into normalizing the trie that represents an action database using unquote and quoteTerm. Quoting forces normalization, which will make Agda *compile* it to its normal form in the `agdai` file. This technique ensures that we only create the database once.

5.2.6 *The* auto *tactic*

Equipped with our RTries, we are ready to handle multiple actions in a manner far better than what we did with lists. We can also reuse a big portion of the library and, for instance, make the lookup actually return a Name × UData object. Making it possible to use the same user created term strategies, discussed in section 4.5, to reconstruct the Agda term filling the goal.

```
search-action : RTermName → RBinApp ⊥ → RTrie → List (Name × UData)
search-action hd (_ , g₁ , g₂) trie
  = let g□ = g₁ ∩↑ g₂
    u₁ = g□ -↓ g₁
    u₂ = g□ -↓ g₂
    ul = replicateM (u₁ :: u₂ :: [])
  in maybe (mkSearch g□) (ul-is-nothing :: []) ul
```

Note how we can not just look up the goal term. We need to find out where the changes are, just like we did with the *by* tactic, and construct a term using these differences and a RTermName to use as a new head.

The final tactic is provided inside a sub module of `RW.RW`, where we can abstract over the RTrie being used as a database and a function for modifying RTermNames.

```
module Auto
    (bt : RTrie)
    (newHd : RTermName → RTermName)
  where
```

The importance of newHd is not evident. When computing terms with the by tactic, we have both the goal and the action head (that is, the topmost operator of both terms), which allow us to choose which strategy to apply, if any. Using the auto tactic, however, we need to find an action to apply to our goal. Let $g_\square$ be the context inferred from our goal and u_1 and u_2 the subtraction of $g_\square$ from the goal. We will search for an action with type rapp (newHd g_{hd}) (u_1 :: u_2 :: []). The tactic is defined as follows.

```
auto : List (Arg AgType) → AgTerm → AgTerm
auto ctx goal with runErr (auto-internal ctx goal)
...| i1 err = RW-error err
...| i2 r  = r
```

```
auto-internal : List (Arg AgType) → AgTerm → Err StratErr AgTerm
auto-internal _ goal with forceBinary $ Ag2RTerm goal
...| nothing = i1 $ Custom "non-binary goal"
...| just (hd , g1 , g2)
  = let
      options = search-action (newHd hd) (hd , g1 , g2) bt
      strat   = uncurry $ our-strategy hd
      err     = Custom "No option was succesful"
    in try-all strat err options >>= return ∘ R2AgTerm
```

This abstraction over which relation to use given the topmost goal relation opens up a few interesting possibilities for future work. Let's assume that newHd had the following, more complex type:

$$\text{RTermName} \to \text{List}\ (\text{RTermName} \times (\text{Maybe RTerm}\ \bot))$$

Satisfying something property close to: $\forall(n,m) \in \text{newHd}\ g_{hd}$, if $m \equiv \text{just}\ t$ for some term t, then t has type $n \to g_{hd}$. This is roughly the introduction of a lattice of convertible relations, with g_{hd} being the minimal. If we imagine our goal head was $\_ \leq \_$, we could start by searching actions which had the same head, if that did not work, we could start looking for actions whose head is $\_ \equiv \_$. Although simple in its core, this idea has a few not so straight-forward details for the goal context $g_\square$ must also be taken into account. For this reason we left this loose end open, to be tied in the future.

5.2.7 *Summary*

Although a more formal complexity analysis is needed to provide a sound conclusion as to how much time do we save by encoding our term database as a Trie, it is straight forward to have an intuition to how it performs better, at the very least on spacial complexity, compared to a list-based implementation. A Haskell prototype was developed prior to the Agda code. Needless to mention, such prototype performed much better and could handle a very large number of terms, even for a real theory. The biggest problem, in Agda, is the insertion function. Trying to make a database with 30 terms, running Agda with RTS options: `-K64M -M2.5G -s`, results in the output presented in figure 8. That is, we need more than 2.5 gigabytes of heap space to finish the computation.

This behavior was somewhat expected, since the insertion algorithm needs to keep reconstructing the tree as it progresses. It is worth remembering that the RTerms corresponding to the lemmas are not as small as one might imagine. For instance, consider the distributivity of converse over coproducts, that is:

```
+-°-distr : {A B C D : Set}{R : Rel A B}{S : Rel C D}
  → (R ° + S °) ≡r (R + S) °
```

When translated to its corresponding meta-language representation, we have the following RTerm.

```
+-°-distr-size : S (Ag2RType (type (quote +-°-distr))) ≡ 22
+-°-distr-size = refl
```

One solution to this performance problem is to perform all processing in Haskell. We would need two additional keywords on Agda. One to add a given function to the internal term database and one to run the corresponding auto tactic with the persistent term database. Is it really a good idea to push reflection computations to Haskell instead of fixing the Agda side, though? This is a complicated question to answer and we will leave this paragraph as a thought digression only, since Agda is still a young tool.

```
Current maximum heap size is 2684354560 bytes (2560 MB);
use `+RTS -M<size>' to increase it.
  43,764,717,080 bytes allocated in the heap
  19,527,437,072 bytes copied during GC
   2,643,217,856 bytes maximum residency (40 sample(s))
      10,944,000 bytes maximum slop
            2637 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     74987 colls,     0 par   31.79s   31.83s     0.0004s    0.0033s
  Gen  1        40 colls,     0 par   393.98s   407.71s     10.1929s    27.7731s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time  191.16s  (192.32s elapsed)
  GC      time  425.78s  (439.54s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time  616.94s  (631.86s elapsed)

  %GC     time      69.0%  (69.6% elapsed)

  Alloc rate    228,946,026 bytes per MUT second

  Productivity  31.0% of total user, 30.3% of total elapsed
```

Figure 8: RTS dump for `Rel.Properties.Database`.

6

CONCLUSIONS AND FUTURE WORK

The overall task of adding rewriting functionality to Agda is quite an exploratory project for a couple reasons, the most relevant being the unstable state of Agda's standard library and, in particular, of the Reflection module. One workaround is to define our own Term datatype, and, whenever Agda changes its reflection module, we just need to fix the two conversion functions. One of the biggest difficulties we found is the steep learning curve of Agda's standard library.

Our goal was not only adding automatic rewriting to Agda, but to explore how closely to *pen-and-paper* we could model Relational Algebra in Agda. Disconsidering generic catamorphisms, the result is very satisfactory. Agda runs in an acceptable time and allows us to build some homework level proofs. When things got interesting though, that is, when we added (generic) catamorphisms to the recipe, Agda did not behave that well. We encountered several performance problems and *internal error* messages. The code still works and is able to (poorly) handle generic catamorphisms. From what we know, this is the first attempt at encoding relational generic catamorphisms in Agda. It is interesting to know that it is, indeed, possible to develop an encoding using W-types. Yet, work remains to be done in refining and fixing this approach. On this same note, the authors can not help but wish Agda provided more control or information over evaluation (type checking). Some sort of profiling, for instance, would really help developers.

On another interesting digression, we did fix the relational equality problem, in Agda, by using a few concepts from Homotopy Type Theory. Our fix still relies on one axiom, the Univalence Axiom. It is arguable, however, on how much of that was indeed necessary, since by assuming function extensionality one could already complete the proofs. Nevertheless, we do believe it is very good practice to keep postulates to an absolute minimum. They not only block evaluation but can lead to inconsistencies if not carefully analyzed. It is also worth noting that our library had special specifications in mind, when it was designed. Such as being suitable for automated processing. This constraint rendered a library with much more code overhead to give us finer control over evaluation.

We would like to reiterate that the Relational Algebra library we constructed is by no means final. A proper organization of the already provided properties, which do not encompass a full account of

Relational Algebra, is definitely needed. Bootstrapping the proofs with our rewriting engine would be also very interesting, even though not all proofs can be bootstrapped. A more detailed work on the decidability of relational composition could also be important for making catamorphisms usable in practice.

In parallel with the development of the aforementioned library, on a more practical front, we developed a few tactics for automated context inference on equational reasoning proofs. That is, we are able to infer the substitutions that justify the rewriting operation automatically. During a first iteration, the tactic was pretty straight forward and, once we figured out the necessary RTerm operations, its implementation was relatively simple. Generalizing it was a more complicated task. We did develop a data structure that seems to perform better than lists to handle multiple possible actions, yet, the construction of such object in Agda posed a performance problem, which we did not have time to overcome and was left as future work. Here again, a few debugging tools from Agda itself could have been of great help.

Long story short, however, our contributions were twofold. On one hand, a library pushing the limits of the Agda language, on another, a helpful automatic rewrite functionality. Although Agda is still a young tool, it has incredible potential. The unified language for programming and meta-programming, indeed, makes it possible to automate almost every mechanical task in a very natural way. From the tools available to us, it is evident that we are at the beginning of a new era for software verification and formal methods.

BIBLIOGRAPHY

[1] Agda wiki: Tutorials, Sep 2014.

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

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

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

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

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

[7] 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.

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

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

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

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

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

[13] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Science of Computer Programming*, pages 148–174, 2004.

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

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

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

[17] 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.

[18] Conor McBride. First-order unification by structural recursion. *Journal of functional programming*, 13(06):1061–1075, 2003.

[19] 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.

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

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

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

[23] 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.

[24] 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.

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

[26] Wouter Swierstra and Nicolas Oury. The Power of Pi. *ICFP08*, 2008.

[27] Wouter Swierstra and Thomas van Noort. A library for polymorphic dynamic typing. *Journal of Functional Programming*, 23:229–248, 5 2013.

[28] Emina Torlak. *A constraint solver for software engineering : finding models and cores of large relational specifications*. PhD thesis, MIT, 2009.

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

[30] Paul van der Walt. Reflection in Agda. Master's thesis, Utrecht University, the Netherlands, 2012.

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

[32] 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.

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