# AUTOMATIC GENERATION OF PROOF TERMS IN DEPENDENTLY TYPED PROGRAMMING LANGUAGES

Franck Slama

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews

2018

# Automatic Generation of Proof Terms in Dependently Typed Programming Languages

by

Franck Slama

University
of
St Andrews

This thesis is submitted to the

UNIVERSITY OF ST ANDREWS

in conformity with the requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

submitted on

02-03-2018

# Abstract

Dependent type theories are a kind of mathematical foundations investigated both for the formalisation of mathematics and for reasoning about programs. They are implemented as the kernel of many proof assistants and programming languages with proofs (Coq, Agda, Idris, Dedukti, Matita, etc). Dependent types allow to encode elegantly and constructively the universal and existential quantifications of higher-order logics and are therefore adapted for writing logical propositions and proofs. However, their usage is not limited to the area of pure logic. Indeed, some recent work [7, 10, 32, 33, 35, 36, 44] has shown that they can also be powerful for driving the construction of programs. Using more precise types not only helps to gain confidence about the program built, but it can also help its construction, giving rise to a new style of programming called *Type-Driven Development* [9].

However, one difficulty with reasoning and programming with dependent types is that proof obligations arise naturally once programs become even moderately sized. For example, implementing an adder for binary numbers indexed over their natural number equivalents naturally leads to proof obligations for equalities of expressions over natural numbers. The need for these equality proofs comes, in intensional type theories (like CIC and ML) from the fact that in a non-empty context, the propositional equality allows us to prove as equal (with the induction principles) terms that are not judgementally equal, which implies that the typechecker can't always obtain equality proofs by reduction.

As far as possible, we would like to solve such proof obligations automatically, and we absolutely need it if we want dependent types to be use more broadly, and perhaps one day to become the standard in functional programming. In this thesis, we show one way to automate these proofs by reflection in the dependently typed programming language Idris. However, the method that we follow is independent from the language being used, and this work could be reproduced in any dependently-typed language. We present an original *type-safe reflection* mechanism, where reflected terms are indexed by the original Idris expression that they represent, and show how it allows us to easily construct and manipulate proofs. We build a hierarchy of *correct-by-construction* tactics for proving equivalences in semi-groups, monoids, commutative monoids, groups, commutative groups, semi-rings and rings. We also show how each tactic reuses those from simpler structures, thus avoiding duplication of code and proofs. Finally, and as a conclusion, we discuss the trust we can have in such machine-checked proofs.

# Candidate’s Declaration

I, Franck Slama, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 60000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree.

I was admitted as a research student at the University of St Andrews in November 2012.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

Signature of Candidate: ................................................

Date: 02-03-2018

# Supervisor’s Declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of philosophy in computer science in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Signature of Supervisor: ................................................

Date: 02-03-2018

## Permission for Publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Franck Slama confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

No embargo on any electronic nor print copy.

Signature of Candidate: ................................................

Date: 02-03-2018

Signature of Supervisor: ................................................

Date: 02-03-2018

# Underpinning Research Data or Digital Outputs

I, Franck Slama, hereby certify that no requirements to deposit original research data or digital outputs apply to this thesis and that, where appropriate, secondary data used have been referenced in the full text of my thesis.

Signature of Candidate: ................................................

Date: 02-03-2018

# Acknowledgements

I'd like to sincerely thank the following people:

Edwin, for giving me the freedom and the encouragements to investigate what I wanted to, for your constant support throughout these years, and for always being so positive and motivating. It sincerely was a pleasure to work with you and I can't express how grateful I am.

Kevin, for accepting to be my second supervisor, and also for the work you are doing within the Functional Programming group at St Andrews to make it grow healthy.

The examiners Thorsten Altenkirch and Susmit Sarkar, for accepting the extra load of work to review this thesis, and for many improvements that you suggested. Also, thank you Susmit for letting me lecture parts of the module on computational complexity to the third year students. I've really enjoyed doing it.

Roy Dyckhoff for the feedback you gave me on an early version of my work and for the discussions we had at several seminars across Scotland.

My office mates Chris, Matus and Adam, and my *almost* office mates David C. (who visited us regularly from the end of the corridor) and Jan (who even had a chair at his name in our office) for all the interesting discussions, for your help, but also for the laughs and the needed distractions, especially the boardgames at Matus' place.

The technicians and the administrative team of the School of Computer Science for your support in many daily tasks, and for making our department such a nice and pleasant place to work.

The University of St Andrews and the School of Computer Science for funding this work.

My good old friend Mathias, for all the work we've done together while we were undergraduates in Toulouse, for all the projects we've been hacking on, and also for the good time we've had talking about so many things around some infused rums.

Ludovic, for reminding me to sometimes forget about my research, and for coming on some hiking trips around Scotland with me. Not to forget all the pubs and restaurants we visited in St Andrews!

Nikitas, alias Mouglon for playing some video games with me despite my poor skills at gaming, for all the interesting discussions we regularly have, and for the good ales we had together.

David S. for all the good time we had together sharing this house in St Andrews, and for helping me to improve my English when I first arrived in the UK five years ago.

Cyril, Charlotte, Lionel, Chloé, Adrien and Justine for all the good time we had when we were students in Toulouse. I can't believe it was many years ago.

Christine Maurel for your amazing lecture on lambda calculus that got me into it, Ralph Matthes, Sergei Soloviev and Celia Picard for my first internships and for introducing me to Coq, coinduction and category theory. And of course many thanks to Armelle Bonenfant for being the one who encouraged me to move to St Andrews for doing this PhD with Edwin; it was indeed a beautiful experience.

Alain Prouté for all the interesting discussions we had about the formalisation of mathematics, and for inviting me to the very first meeting on the Saunders system at your home by a sunny afternoon of July 2013.

Frédéric Blanqui and Gilles Dowek for the post-doc in your group that I am now about to start. I'm really looking forward to work with you in the Deducteam research group at École Normale Supérieure.

My Mother, my Father, my Brother, my Sister, and my Grandparents for your constant encouragements and support, and for your understanding when I am terrible at giving news.

Gisèle, Alain, Jade, Michel and Aude's grandparents, for opening up your home to me and always making me feel welcome from the moment I first walked through the door. Thank you for making me feel like a part of your family.

Aude, for everything you've done for me and for having changed my life. You know, it's only because of you that I've managed to finish this thesis, but I owe you so much more than that. There's no words to thank you properly for your unwavering support, or to express how thrilled I am to have you in my life. I love you.

Franck Slama
St Andrews
02-03-2018

*To Aude*

# Contents

# Chapter 1

# Introduction

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.*
>
> — C.A.R. Hoare

## 1.1 The need of formal certification

### 1.1.1 Critical software and formal methods

We use software everywhere and all the time. They are involved when we travel, when we buy or order something, when we call someone, when we write something on a computer or on one of these smart-phones or pads that we now take everywhere with us. They are also involved in all these devices that we forget to see, from fire-alarm systems to heating control systems that we find in many of our houses. They are also running in places and items that aren't familiar to many of us, like nuclear plants and in military devices. We can also find them in many medical devices, from diagnostic systems to computer-assisted surgery. Even though they are everywhere, we usually only realise that we need them so frequently when one of them suddenly stops working. When it's late, when the last train of the evening is approaching, and when the only machine that sells tickets seems to be stuck in an infinite loop. When we really need to call someone in great hurry, but the voice-over-IP software doesn't want to let the call go through. We're used to being annoyed by software that doesn't work as intended when we really need them, and we usually accommodate it. Sometimes, and when we can, turning them off and on again make them back to work. We're used to losing data, time or some money because of them. However, there are software on which we rely completely and that take vital decisions for ourselves, and the situation is completely different with these ones. These software are called *critical software*. Many of them are embedded, like the one implemented in airplanes and rockets. For this kind of software, it would definitely be a bad idea to try to restart them suddenly: no one wants to restart the flight system of an airplane in the middle of a flight, or to restart the control program of a nuclear plant. These software are absolutely vital, and we can't accept any bugs in them, as the consequences of a single failure can be disastrous. Therefore, these software need more care, more time and more energy to develop. But they need more than that : they also need new methods.

The usual approach for making sure that a software effectively fulfils its specification is to test it. There are several types of tests, but the general idea of a test is to run a component of a program on various inputs, and to compare the results produced by the program with the expected ones. This activity can be automated to some extent in order to perform a large number of tests that would be otherwise beyond reach. But even with a big number of tests, tests are still incomplete : it is impossible to test all the situations that can happen and all the input that could be processed, because usually the domains of functions are infinite (think of a function taking a number in input for instance). Thus, "testing can be used to show the presence of bugs, but never to show their absence" [Dijkstra-1970], and that's the reason why new methods for critical software have been developed in the last decades. We can divide these methods that bring more confidence about the correctness of programs into the following categories : model checking, abstract interpretation, static typing, and formal certification within logical frameworks.

- Model checking : This is a technique that applies for systems that have a finite number of states or that may be reduced to it by abstraction, and is based on the study of such automatons. This technique has appeared in the early 80s. The goal is to verify if some properties (like the program is free of deadlocks, a pointer is never null, etc) are verified by the model of a system. These properties are often written in temporal logic, a logic that enables to qualify propositions in terms of time, for expressing things like "the system always verifies $P$", or "$P$ will be true until the system reaches the state $S$". The general goal of model checking is to check if these formulae are valid on an automaton that models the system, and these verification can often be done automatically. When a proposition doesn't hold on the model, we are generally interested in producing a counter-example, i.e. an execution of the system that invalidates the proposition. Model-checking is nowadays one of the most used formal technique in the industrial sphere, as it can often be relatively well automated, compared to the other approaches.

- Abstract interpretation : This is a technique that aims to compute an approximation of the semantics of a program which can be used for answering some questions, like "may this pointer be null" or "may this value be bigger than X". There is necessarily a loss of information between the concrete semantic (that describes the real execution of the program) and the approximated one, which is the price to pay in order to have a computable semantic. Abstract interpretation was originally conceived by Patrick and Radhia Cousot in the late 70s, and is based on monotonic functions over ordered sets. It can be viewed as a partial execution of a program which gains information about its semantics without performing all the calculations. This technique is used when computing the exact semantics is impossible or too expensive. Abstract interpretation isn't only used in formal certification, but is also used in compilers to decide if certain transformations (mostly optimisations) can be done.

This thesis will however completely focus on the next two methods :

- Static typing : Some languages like Haskell and Ocaml have a strong and restrictive notion of type. In these statically typed languages, types are semantic entities that ensures that the programmer is not mindlessly mixing objects of different sorts, as it is known to cause many bugs. All the operations will be only applicable to the objects of the right type, without any implicit conversion of type. Therefore, in these languages, it is impossible to write a completely senseless operation.

The benefits of having a strong and static *type system* are now well known, as they eliminate at the root –by simply rejecting ill-typed programs– many bugs that are otherwise hard to debug. Types can also be a guide for the programmer, since the typechecker will automatically report all programming mistakes that are related to the sort of objects that are manipulated and the operations that can be applied to them. For instance, if `n` is a natural number, the expression `if n then 0 else 1` won't typecheck as `n` should be a boolean but is here a natural number. The expression `0 + True` will also be rejected by the typechecker as `True` is expected to be a natural number, but is a boolean. In most programming languages (like C) that do not have such a powerful notion of types, these examples would be unfortunately valid code, and the latter would typically be evaluated to 1 because the value `True` would have been implicitly converted to the integer 1.

Typing is a kind of static analysis, done at the first stage of the compilation process : the source of the program is analysed, without being executed, and this analysis decides if the program will be compiled or rejected. As these types become richer, the more bugs can be avoided. We will see in this thesis some very rich types, called *dependent types*, that enable to capture very precise informations about the sort of object manipulated, so precise that we will be able to use these types to carry *proofs*.

- Logic, formal specifications and proofs :

  Another possibility in order to increase the confidence in software is to formally prove its correctness, and this approach is often

> a complement to static typing. Such proofs are made within a logical theory (simply called a logic) that is suited for talking and reasoning about programs. Some logics, like Hoare logic [24] and its variants, are designed to reason about imperative programs, and constructive logics, like the Calculus of Constructions (CoC) [15] and Martin Löf type theory (ML) [29], are shaped to reason about functional programs. We will explain the meaning of the adjective "constructive" later. For both imperative and functional programs, the general idea is to express a formal specification within this logic, and to prove that the program fulfils this formal specification. Sometimes, the program is instead derived from the specification, by iterative refinements. This thesis will focus on this activity of proving.

For imperative programs, the specification is often expressed as pre and post conditions, which can be seen as a contract. If the caller of a function respects the conditions expressed in the pre-conditions, then we have the guarantee that the properties expressed by the post-conditions will be valid after the execution of this function. This idea comes from Hoare in 1969, and originally these verifications were done on paper. It's only a few decades ago that tools like the B-toolkit [1] appeared (in 1996), with the goals of helping the construction of the proof, and, more essentially, of verifying the complete proof.

For functional programs, the formal specifications and the proofs are made within a logical framerwork called *type theory*. Type theory is, in modern presentations, a formal system that contains computational rules (defining a rewriting system) and typing rules (defining a logic). The computational rules express how the computations are performed in order to reduce expressions to their results, and the typing rules describe how the different objects of the theory can be manipulated and combined (and which ones are valid objects). Because of a deep correspondence between types and logical propositions (described in 2.3), these typing rules also describe the kind of logical properties that can be expressed and proven about these programs.

Although there exists different kind of type theories (intensional, extensional, with or without cumulativity of universes, univalent or not, etc), and many different implementations of them, they all share some basic concepts [2]. This situation can be compared to set theory, where concrete systems like ZF, ZFC and Kripke-Platek set theory, all share the same intuitive notion of *set*, which comes directly from naive set theory. In type theory, *types* are used as a way to discriminate the "valid objects". This is used in every type system as a way to reject badly behaving terms (like $\Omega = (\lambda x.\ xx)\ (\lambda x.\ xx)$, which reduces to itself), and as a way to avoid paradoxes, like Russell's paradox, where self-references create inconsistencies. Type theory is even a branch of mathematics that has initially been developed for this purpose, when in 1908 Russel proposed his "ramified theory of types" in order to address the paradox that he discovered when he studied the book [20] on logic written by Frege.

Type theory has gained momentum after Howard and Curry discovered independently (in 1958 and 1969) that proofs correspond to computable functions (lambda-terms or combinators), and logical formulae to types. This discovery has reunited two pieces considered previously as separate : logic and computations. Because with this correspondence a logical proposition can be represented by a type, any type theory embeds a logic, and the rules of this logic are expressed by the typing rules of the theory considered. Different flavours of type theory have been developed and explored since then, with different inference rules and axioms, that has lead to various expressivity and different usages. We will say more about type theory and the Curry-Howard correspondence in section 2.2.

In this thesis, we will use the type theory implemented as the kernel of the Idris language, but all the ideas presented in this thesis can be applied to any dependently-typed programming language or proof assistant. The rest of this introduction will talk about such proof-oriented languages and proof assistants in 1.1.2, before presenting the Idris programming language in 1.1.3. Then, we will focus on dependent types and the problems that they bring in 1.2, and we will present on a concrete example in 1.3 the specific problem that we want to address in this thesis.

### 1.1.2 Proof assistants and programming languages

Proof assistants are software that enable the writing of code, logical statements and proofs. Each proof assistant is based on a formal system (or theory), which is often a type theory. On top of this trusted kernel, they can offer various features and automations that aim to help the construction of programs and proofs, but their most important aspect is the automatic verification of proofs. A proof done on paper can be wrong, because a wrong assumption can be made or a theorem badly applied. This is not something specific to proofs about programs, as some theorems in maths have had false proofs for quite a while before that the mathematical community realised that the proof was actually false. The paper never complains about a false proof, which makes these mistakes hard to spot, and they can ruin the efforts done to make a software safe. Thus, the great advantage of proof assistants is that they make sure that the proof built by the user is valid and effectively proves the statement claimed by the lemma. The way they verify the validity of proofs will be described later, in 2.5.

For proof assistants based on constructive logics, Coq [6] and Agda [35] are certainly the two most famous nowaday. They are based on fairly similar type theories (CIC [14] and an extension of ML respectively), but their spirit is slightly different. Coq is a proof assistant in the traditional sense : the proofs are often done in a proof mode by applying tactics, that have an effect on the current context and on the goal to prove. If we want to prove the following lemma of propositional logic : $\forall (P\ Q : Prop),\ P \land Q \rightarrow P \lor Q$, the proof goes like this :

1. We use `intros P Q`, the context becomes $\Gamma = \{P : Prop,\ Q : Prop\}$, and the goal $P \land Q \rightarrow P \lor Q$.

2. We use `intro H` to introduce the hypothesis $P \land Q$ in the context, which is now $\Gamma = \{P : Prop,\ Q : Prop,\ H : P \land Q\}$. The current goal becomes $P \lor Q$.

3. From the available proof $H$ of $P \land Q$ that we have in the context, we can extract a proof $H1$ of $P$ and a proof $H2$ of $Q$ by using

`destruct H as [H1 H2]`, and the context becomes $\Gamma = \{P : Prop, Q : Prop, H1 : P, H2 : Q\}$. The goal is unchanged.

4. Now that we have a proof of $P$ and a proof of $Q$ available in the context, we have two possibilities for proving $P \vee Q$. We can prove it by proving only $P$, and in this case we apply the tactic `left`, and the goal will become $P$, or we can prove it by proving only $Q$, and in this case we apply the tactic `right`, and the goal will become $Q$.

5. If we've chosen to prove $P$ at the previous step, then we can simply `exact H1`, which is a proof of $P$. If we've chosen to prove $Q$, then we can `exact H2`. In both cases, that finishes the proof.

Once the proof is complete (i.e. when there is no more subgoals to prove), we must tell the system that the proof is finished, and it will verify it. This is done by the command *Qed*. Therefore, a tactic could be wrongly implemented, it would not create any inconsistency in the system, as anyway, the complete proof –represented by a lambda-term that has been built by the application of these tactics– is going to be completely checked at the end.

```
Lemma and_imp_or : forall (P Q : Prop), P /\ Q -> P \/ Q.
Proof.
intros P Q.
intro H.
destruct H as [H1 H2].
left.
exact H1.
Qed.
```

Figure 1.1: A proof script in Coq

In Agda, the complete lambda-term representing this proof will be very similar, but the way to build it will be slightly different. Agda looks more like a traditional functional language and we write proofs in it as we write usual programs. The same proof would be written like this :

```
andImpliesOr : {P Q : Set} → (P ∧ Q) → (P ∨ Q)
andImpliesOr (∧-intro p q) = ∨-intro-left p
```

This proof is written as a function that does pattern matching on its only argument. Since the type $\wedge$ contains only one constructor called `∧-intro`, of type $P \rightarrow Q \rightarrow P \wedge Q$, the only possibility for being a proof of $(P \wedge Q)$ is that the input is `(∧-intro p q)` where `p` and `q` are respectively proofs of $P$ and $Q$. In order to build a proof of $P \vee Q$, we can use either the constructor `∨-intro-left` or the constructor `∨-intro-right` that are the two constructors of the type `∨`. The former one expects a proof of $P$, like `p`, and the latter a proof of $Q$, like `q`. That's why the term `∨-intro-left p` has the right type, i.e. is a valid proof of the lemma.

For inductive proofs, that are done with the `induction` tactic in Coq, they are directly written as a recursive function in Agda, where the recursive call on the smaller argument produces the induction hypothesis. That does not changes much the lambda-term that we obtain in both cases at the end, but the intellectual gymnastic of producing a proof is a bit different. Sometimes, the "script-style" of proofs in Coq is more efficient or seems more natural, and sometimes, writing directly the corresponding program in the Agda-style seems more straightforward. Note that this second style of proofs (writing them directly as lambda-terms) is also doable in Coq, but it is often more cumbersome than using the interactive proof mode.

Proof assistants like Coq and programming languages like Agda have in common the fact that they enable the writing of formal specifications and proofs that are verified by type-checking within the theory that they implement. The differences between these two categories are more cosmetic : in proof assistants, the emphasis is on the writing of proofs, while it is on the writing of programs in programming languages.

In this thesis, we will use the programming language Idris [8], but almost all the ideas that we will present can be adapted to any dependently typed functional programming language (dependent types will be described soon in the section 1.2). Idris is closer to the family of programming languages that enables to reason about code, like Agda, than it is to the family of proof assistans like Coq. However, all these tools share the essential property that proofs are verified by a small trusted kernel, which is absolutely needed for believing in machine-checked proofs. In Idris, proofs are usually directly written as a function (i.e. as a lambda-term), even though Idris also has a proof mode with a few tactics available. However, unlike Coq and Agda, Idris is intended to be a general purpose programming language, suited for developing real-world applications, and therefore includes support for things like system programming and network programming. We describe briefly the basics of Idris in the next subsection.

### 1.1.3 Programming and proving in Idris

The use of proof assistants is currently limited to researchers trained in type theory and logic, and systems like Coq and Agda haven't really penetrated the industrial sphere, and aren't particularly suited for "real" industrial applications. Idris [8] is a functional programming language that has been created in order to address these shortcomings, and has been designed from the start with the aim to bring formal verification to programmers. The intention behind it is to convince programmers that type-based verification can greatly enhance software's safety, even for real world applications, like the one interacting with the system and the user, communicating through the network, etc. We start by presenting the basics of this language with proofs.

#### Functions definitions in Idris

Idris' syntax is heavily influenced by Haskell. For instance, the definition of a function that composes two functions given in input is very similar to its Haskell equivalent :

```
compose : {A:Type} → {B:Type} → {C:Type}
           → (A → B) → (B → C) → (A → C)
compose f g x = g (f x)
```

First, the type of `compose` is claimed by the programmer, and then its definition is given. In the type, the element in curly braces {...} are representing implicit arguments, i.e. arguments that won't have to be passed to the function because they can be inferred automatically from the other arguments. Here, in order to call `compose`, we will only have to provide two functions `f` and `g` [1] such that the domain of `g` coincides with the co-domain of `f`. The types `A`, `B`, and `C` can be retrieved from the type signatures of these functions.

It is interesting to notice a first advantage of static typing : the type of the function *compose* brings the guarantee that it won't be possible to compose two functions that do not have a corresponding domain and co-domain, as the typechecker will make sure -by unification- that this constraint is preserved. Of course, this function *compose* is completely useless as we can always directly use the composition $g\ (f\ x)$. In order to write more interesting functions, we need to introduce some types like `Bool`, `Nat` and `List`.

### Inductive types and recursive functions

These datatypes `Bool`, `Nat` and `List` are defined as inductive types, which are sometimes also called (recursive) sum types, disjoint union or more occasionally variant types and tagged union in the literature. An inductive type contains some data constructors, that all have a type, and that represent the different way of constructing terms of this type. One of the easiest example, that has two constants constructors, is `Bool`. There are two possibilities for being a boolean : being either `True` or `False`. This leads to the following definition in Idris [2]

```
data Bool : Type where
   True : Bool
   False : Bool
```

Both constructors `True` and `False` are constants in the sense that they do not expect any parameter. Thus, both can generate only one inhabitant of this type, leading to a type with only two inhabitants, often

[1] and eventually an element `x` of type `A`, depending on if we want to get as a result the composed function itself, or its application to `x` (`compose` is a curried function, and the result `compose f g` is itself a function)

[2] The definition of `Bool` by the user is not needed, as this type is already provided by the language where it is defined exactly like this.

called **2** in type theory. We can see such a type with only constant constructors as an *enumeration*.

A bit more interesting than `Bool` is the case of the natural numbers `Nat`. A natural number is either the number zero, or the successor of another natural number. The first case gives a constant constructor, called `Z` in Idris (of arity 0, like `True` and `False`), and the second case a recursive constructor called `S`, as an abbreviation of "successor".

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

This definition is in fact an encoding of numbers in a unary numeral system, which generates the terms `Z, S Z, S (S Z), S (S (S Z)),` and so on, respectively representing the numbers 0, 1, 2, 3, etc.

There are some restrictions, and not all inductive definitions are acceptable, because the system could become otherwise inconsistent. In an inductive definition of a type $T$, each constructor must be strictly positive, which means that all occurrences of the type $T$ must be strictly positive. An occurrence is strictly positive if and only if it used in output position, but not in input position. This can be syntactically checked by not allowing $T$ to appear directly at the left of an arrow. For instance, the occurrence of $T$ in $(T \rightarrow nat)$ is negative. Thus, the following data constructor for $T$ would be rejected by the system : `isRejected:(T → nat) → T`. These positivity restrictions are needed in order to avoid non-terminating programs.

A type can also be parametrised by another type, and this is a case of polymorphism. For example, polymorphic lists are parametrised by the type of their elements, and their definition in Idris is in all aspects similar to what we find in Haskell or Ocaml :

```
data List : Type → Type where
   Nil : {T:Type} → List T
   (::) : {T:Type} → (h:T) → (t:List T) → List T
```

`Nil` represents the polymorphic empty list. The second constructor, written with the infix notation `(::)`, is called `Cons`. It takes an element of `T` (called `h`, for head), a `List` of `T` (called `t`, for tail), and it represents a lists made of h followed by t. In order to facilitate the writing of lists, Idris, like many other functional languages, uses the traditional concrete syntax for lists, where [a, b] represents the list a::b::Nil.

Many functions that operate on an inductive type are recursive : they solve a problem by calling themselves on a smaller instance of the problem, and eventually by doing some additional treatments. With List, one of the first function we want to write is usually a function that computes the length of a given list.

```
length : {T:Type} → List T → Nat
length Nil = Z
length (h::t) = S (length t)
```

When the input list is the empty list Nil), its length is zero (denoted by Z). When the input list is a Cons of a head h and a tail t, we can determine its length by computing recursively the length of the sublist t, and by adding one to this number, which is done by taking the successor of that number.

Another typical function with lists is append, which takes two lists l1 and l2 with elements of type T, and produces another list : the concatenation of the two input lists, often denoted l1++l2.

```
append : {T:Type} → List T → List T → List T
append Nil l2 = l2
append (h::t) l2 = h :: (append t l2)
```

When the first list is the empty list Nil, we can simply return the second list l2. When the first list is a Cons of a head h and a tail t, we can do it by appending t to l2 (which is done by the recursive call), and then by adding h at the head of this intermediate result.

### Predicates and proofs

We've mentioned earlier that Idris is one of these programming languages that enables to manipulate predicate and proofs, and which therefore makes it possible to do formal certification within the language. Thus, in order to gain confidence in the definitions that we have written above, we should now formally specify the behaviour of these functions [3], and prove the correctness of these functions according to these formal specifications. These formal specifications will use predicates and logical connectors. A predicate is a logical formulae that takes (at least) a parameter in input, and that can be proven for some of its possible inputs. Often, the predicates are themselves defined inductively, and we call them *inductive predicates*. Sometimes, and especially when the function being proven is very primitive, its lemma of correctness will use predicates that are extremely similar to the definition itself. This can become a problem, which is discussed more in depth in chapter 6. For example, the predicate "is the length of this list" will match very precisely the computational definition of the `length` function :

```
data HasLength : {T:Type} → List T → Nat → Type where
   NilHasLengthZero : HasLength [] Z
   ConsHasOneMoreElement : {T:Type} → (t:List T)
       → (n:Nat) → (pr:HasLength t n) → (h:T)
       → HasLength (h::t) (S n)
```

Even though the predicate `HasLength` is very similar to the function `length`, they shouldn't be confused : `length` is a function that can be immediately *computed* in order to return the length of the input list, but `hasLength` is a predicate that has to be *proven* on a given list. Its first constructor says that `[]` has length `Z`, and it precisely stands as a proof of it. As this definition is a logical one, the second constructor must be read by thinking of its arguments as universal quantifications. Its logical meaning is therefore :

```
∀ (T:Type) (t:List T) (n:Nat) (pr:HasLength t n) (h:T),
   hasLength (h::t) (S n)
```

which can be expressed in English by "for any type `T` and any list `t` of `T`, if `n` is the length of `t`, then for any element `h`, the list `(h::t)` has size `(S n)`".

[3] Some people would object that in fact, this activity of specification should have been done even before the writing of the functions.

The correctness lemma for `length` simply says that any list `l` has for length (according to the predicate `HasLength`) the result of the computation `length l`. This is expressed in Idris as :

```
length_correct : (l:List T) → HasLength l (length l)
```

The proof is not difficult and can be done by induction on the input list `l`. When `l` is `Nil`, we have to prove `HasLength [] (length [])`, which reduces to `HasLength [] Z` as `length []` reduces `Z`. But building a proof of `HasLength [] Z` is an easy task, because we have the first constructor of `HasLength`, called `NilHasLengthZero`, that says precisely what we need. Therefore, we can build the desired proof by simply returning this constant[4], which is the proof we need :

```
length_correct [] = NilHasLengthZero
```

The case where the input list `l` is a `Cons h t` is the inductive step. `length_correct`, although expressing a logical statement, is still an ordinary function that can be used as any other function. In particular, we can call it recursively on the smaller argument `t`, and if we do so, we will get the induction hypothesis, of type `HasLength t (length t)`, which states that `t` has provably the length `length t`, or said differently that the function `length` is behaving correctly on this smaller input `t`. Thus, according to the constructor `ConsHasOneMoreElement`, if we add any element in front of `t`, the result will provably have a length of one more, which is `S (length t)`. This can be applied to the abstract element `h` that we have, and this is precisely what we need, as the computation `length (Cons h t)` precisely returns the value `S (length t)` according to the second pattern of the definition of the `length` function. Thus, we can get the proof that we need by simply doing :

```
length_correct (Cons h t) =
    ConsHasOneMoreElement t (length t)
                          (length_correct t) h
```

[4] This constructor takes in fact a parameter, `T`, but as this parameter is only a type parameter, we can see this constructor as a constant (but a polymorphic one, that works for any type of elements).

We realise that this proof, in both cases, was really easy, and this is no surprise as the logical specification `HasLength` and the computational definition `length` are saying exactly the same thing. The only difference between the function `length` and the predicate `HasLength` is that the former one is a function that can be *computed* on a given list to produce a natural number, when the later can only be *proven* for a given list and a given natural number.

We could now do the same kind of work for `append`, by defining a formal specification for it with a predicate `isAppend`, that will take three lists in input, and that will be provable only when the third one is the concatenation of the first two lists. The definition of this predicate `isAppend` and the proof of correctness of `append` would not be more complicated than it was for `length`. However, it appears that for very primitive functions, like `length` and `append`, such proofs of correctness do not really bring any valuable guarantee because their specifications match too closely their definitions. For this kind of functions, highlighting logical links between these definitions would certainly give us a much more valuable guarantee than the actual proof of correctness that we just did, which connects two things (a logical definition and a computational one) that are too similar.

So far, we haven't expressed any link between these functions `length` and `append`, but there is a strong one, as appending two lists creates a list of a size that is the sum of the input's size. If we manage to make this link explicit and checkable by the machine, we will gain some additional confidence in these definitions. Of course, this property taken separately is not enough to completely validate the correctness of these functions. However, adding such links between different functions brings a good level of confidence, and is easily accessible to programmers. By continuing this activity with additional logical links when more functions will be defined, we will become more and more confident about this library of lists. This activity of proving lemmas that connects many definitions and objects is in fact a very common activity in mathematics where the behaviour of a new, freshly defined object is defined beside the objects and concepts that already exist, especially in the area of category theory. Computer scientists have already followed this path, and a good formalisation of lists, such as the one found in the Coq proof assistant, is expected to highlight many of these logical links.

An example of such a link is the fact that, for any lists `l1` and `l2`, the length of the append `(l1++l2)` must be `(length l1)` + `(length l2)`. We can state this property by writing a lemma, expressed in Idris as an ordinary function that has this type :

```
append_length : {T:Type} → (l1:List T) → (l2:List T)
   → length (l1 ++ l2) = length(l1) + length(l2)
```

We won't show the proof of this lemma as it would require some notions about equality that will be presented later in chapter 2, but this proof would again be done by induction on its first argument $l1$, and would consist of checking that in both cases (Nil and Cons) the computation of the length unfolds to two expressions that are equal.

### Types can express more

If the type of lists would encapsulate their length, we could state this logical property directly in the type of the function `append`, instead of having to define this auxiliary lemma `append_length`. Having this kind of property expressed directly in the type would be interesting because it would help the definition of the function : types can drive the development of software, as this is something we already know from Haskell and Ocaml, where having a strong type –that expresses more than its usual C or Java equivalent– leads to less errors and less debugging work, and can even drive the construction of the program. Here, that would mean that if the definition written for `append` doesn't verify this property about the sizes, then the definition would simply be rejected by the typechecker. Also, having the property about the size expressed in the type itself leads to less code and proof duplication, as the proof of `append_length` is looking at the definition of `append`, and is in fact following precisely the same recursive pattern.

In order to be able to express these links, dependently typed languages like Coq, Agda and Idris enable the manipulation of types that are indexed over some values. This enables to define the type `Vect`, that behaves like lists, but which has the novelty of being indexed over a natural number (in addition of being still polymorphic on a type T). We will describe dependent types more precisely in the next section, but we first give just a taste of what we can do with them. When an element v has type Vect n T (where n is a natural number and T a type), it means that v is a vector of size n, with elements of type T. The type of Vector indexed over their length can be written like this in Idris :

```
data Vect : Nat → Type → Type where
   Nil : {T:Type} → Vect Z T
   (::) : {T:Type} → {n:Nat} →
          (h : T) → (t : Vect n T) → Vect (S n) T
```

The first constructor, Nil, is still polymorphic over T, but it has now an encapsulated size, which is Z (denoting zero). The meaning is that Nil is a vector of size zero. The second constructor (::) can build an element of type Vect (S n) T, i.e. a vector of size (S n), when given in parameter a head h of type T, and a tail t of type (Vect n T), i.e. a vector of size n.

Thanks to this index, the type of functions –like append– that work with vectors can be a lot more informative. For example, we can now state directly in the type of append that when applied to a Vect of size n1 and a Vect of size n2, the result must have size n1+n2 :

```
append : {T:Type} → {n1:Nat} → {n2:Nat}
      → (Vect n1 T) → (Vect n2 T) → Vect (n1+n2) T
```

With this stronger type, there are several mistakes that can be avoided in the definition of append, since only an algorithm that builds a vector of size $n1 + n2$ will be accepted. For example, a function that always returns the first vector would be rejected by the system. We will now describe more in depth dependent types, their advantages and the problems that they bring in the next section.

# 1.2 Dependent types

## 1.2.1 What dependent types are

### Dependent types informally

The biggest novelty in functional programming compared to imperative or object-oriented programming is the fact that functions are first-class values[5]. In the same way, in dependently-typed programming languages, the novelty is that types –that were previously only a compile-time information– are now first-class values. They are something that can be computed, which means that they can be taken in parameter and returned by a function, just as any other value. It is for example possible to write this function :

```
NatOrList : (b:Bool) → Type
NatOrList False = Nat
NatOrList True = List Nat
```

This function returns a type, so for a given boolean `b`, the expression `(NatOrList b)` is a type, and this type depends on the value `b` : if `b` is `False` then this type is `Nat`, if `b` is `True` then this type is `List` of `Nat`.

With this, we can write a function that produces a value that can have two different possible types, depending on the actual value of the input.

```
ZeroOrNil : (b:Bool) → NatOrList b
ZeroOrNil False = Z
ZeroOrNil True = []
```

The *type* of the output of this function depends on the *value* of the input, and therefore this function has a *dependent type*.

With dependent types, not only a function can have the type of its output depend on the value of one of its input, but also pairs of values can become dependently typed, which means that the type of the second component `y` of a pair `(x,y)` can depend on the value of the first component. These *dependent pairs* are written `(x ** F(x))` in Idris. We

[5] Imperative and object-oriented languages are now starting to get some of the concepts coming from functional programming.

can for example define the type of pairs where the first component is a natural number `n`, and the second a vector of booleans of size `n` :

```
PairNatVect : Type
PairNatVect = (n:Nat ** Vect n Bool)
```

The value `(2 ** [True, True])` is an inhabitant of this type, but `(2 ** [True, True, True])` is not, because the second component has type `Vect 3 Bool` while it is expected to have type `Vect 2 Bool`.

As we've seen in the previous section with the example of the type `Vect` of "lists" indexed by their size, an inductive type can be indexed over a value. This is possible because inductive types are not different from other types in regard to dependent types, and it is therefore possible to inductively define `IndexedType` of type `TypeIndex → Type`. If `i` has type `TypeIndex`, then `(IndexedType i)` is a type. Therefore, `IndexedType` describes a family of types, indexed over `TypeIndex`. In practice, in order to define inductively such a family of types, we still use constructors, and each constructor targets a part of the family of types, by specifying what the index is for this constructor. To make things clearer, if `TypeIndex` is a type that contains the constants `I1` and `I2`, an indexed type can be defined inductively like this :

```
data IndexedType : TypeIndex → Type where
    Constr1 : ... → indexedType I1
    Constr2 : ... → indexedType I2
    ...
```

This definition means that `Constr1` targets the part that is indexed over `I1`, i.e. it builds values that have type `(indexedType I1)`, while `Constr2` builds values that have type `(indexedType I2)`.

### Dependent types more formally : $\Pi$ types and $\Sigma$ types

We've just seen with the function `ZeroOrNil` that functions can now have the type of their output depend on the values of their inputs. The usual function space (written with the arrow symbol →) with a fixed codomain found in non dependently-typed languages has been replaced by a dependent function space. This has been made possible by several changes, and the most important of them is that there is no longer a separation between terms and types. In non dependently-typed systems, the relation `x:T` relates a term `x` and a type `T`, and these two things live at different levels. However, in dependently-typed systems, there is no longer such a strict separation, and everything is a term. That means that types are no longer just a semantic information about a term, but they are ordinary values that can be computed. Being given a type $A$, we've seen that we can write a family of types $B : A \to Type$ which assigns to each term $a : A$ a type $B(a) : Type$. The function `NatOrList` that we've written above was such a family of type. We've also seen that we can use such a family of type in order to build a dependent function, i.e. a function where there is no fixed codomain. Formally, the type of such a function is a *pi-type*, denoted $\Pi_{(x:a)}B(x)$. In Idris, the type of such a function is simply written `(x:a) → (B x)`, and the fact that the right-hand side of the arrow uses the value `x` makes it explicit that this is a *dependent function*. The function `ZeroOrNil` that we've defined above was such a dependent function.

When $B : A \ \to \ Type$ is a constant function, the type $\Pi_{(x:A)}B(x)$ becomes completely equivalent to $A \ \to B$, which means that dependent functions and "ordinary" functions coincide when the type of the output doesn't change with the value of the input.

In the same way that the ordinary function space has been generalised with pi-types, ordinary pairs are generalised with sigma-types, that are written $\Sigma_{(x:A)}B(x)$, where $B$ is still a family of types. For example, the type of dependent pairs with a natural number `n` as first component and a vector of booleans of size `n` as second component –that we've defined above in Idris as `(n:Nat ** Vect n Bool)`– is formally written $\Sigma_{(n:Nat)}(\texttt{Vect}\ n\ \texttt{Bool})$ in type theory. This generalisation of pairs is completely equivalent to ordinary pairs when the type of the second component does not change with the value of the first : the type $\Sigma_{(x:A)}B(x)$ is the same as the product type $A * B$ when $B$ is a constant function.

### 1.2.2 Dependent types' expressivity

One of the great advantages of dependent types is that they enable the programmer to be a lot more precise about the sort of elements that are being manipulated. That avoids the situation where one has to use a type containing more inhabitants than needed (which means that the type therefore contains junks), as using too-large types is often a source of bugs. For example, if one defines an integer division function as `divide : Nat → Nat → Nat`, then what should be the result when it is called with zero as the second argument? The answer is that there is no good value to produce for this case, since we've left an unwanted value come in. The traditional solution is to use exceptions, or to change the type of the output type from `Nat` to the monad `Either Nat Error`[6] , but that still doesn't make explicit that the error appears only when the second argument is zero, and it still doesn't prevent it to happen, it just deals with it. Instead of making the output type more complicated –and therefore complicating the use of this function, as we would have to check if the result is a `Nat` or an `Error`–, dependent types allow us to simply reject the value zero by being more precise about the type of this second argument. This second argument can become a dependent pair $\Sigma_{b:Nat}(b{>}0)$, which carries a natural number $b$ and a proof that $b$ is strictly greater than zero. With this type, the problem of dividing by zero won't happen at all, since it is impossible to pass to the function a proof of $b{>}0$ when $b$ is zero. We see here that dependent types allow us to define very precise types by cutting down unwanted values with logical formulae.

It is now important to mention that the ability to write logical formulae also comes from the fact that types are first class objects. These logical formulae (like $b{>}0$ that we've just encountered) are made of predicates (> in this example), and these predicates themselves use types as first class objects : the predicate > is defined as a family of types, of type $Nat \to Nat \to Type$. For any $x$ and $y$ of type $Nat$, the expression $x{>}y$ is a type, and if this type is inhabited, then $x$ is effectively greater than $y$, and

[6] These two possibility are semantically equivalent. The difference is that exceptions are supported directly by the language, when one would have to do a few more things by hand with the type `Either Nat Error`.

all the inhabitants of this type stand as proofs of this fact.

The predicate "strictly greater than" is defined by using the other predicate "strictly lower than" : $(a{>}b) \ := \ b{<}a$. Then, the predicate "strictly lower than" is defined using the predicate "lower or equal" : $(a{<}b) \ := \ (S \ a) \leq b$. Finally, the predicate "lower or equal" is defined inductively, and as usual, each constructor targets a part of the family. In Idris, this predicate $\leq$ is called `LTE` and is defined as follow :

```
data LTE : Nat → Nat → Type where
  LTEZero : LTE 0 right
  LTESucc : LTE left right → LTE (S left) (S right)
```

While it is defined like this in Coq:

```
Inductive le (n:nat) : nat → Prop :=
  le_n : n ≤ n
  le_S : ∀m : nat,  n ≤ m → n ≤ Sm
```

It is interesting to realize that the same predicate can often be defined in different ways, as it is the case for this $\leq$ predicate : although these two definitions are logically equivalent, they are two different constructions that are not only syntactically different.

Let's mention that in Coq, these predicates have type $Nat \rightarrow Nat \rightarrow Prop$, because Coq separates the world of computations (that lie in `Type`) from the world of logical formulae (that lie in the impredicative sort `Prop`). That's not the case in Idris which only has a predicative `Type`, but in which $x{>}y$ must still be thought of as a logical proposition.

In summary, in dependently typed theories, we have the following properties :

- We can now return a type or pass a type as parameter

- Because of the first point, we can now write family of types, with the signature $A \rightarrow B \rightarrow ... \rightarrow Type$, where $A$, $B$, ... are types. And when such a family is inductively defined, we talk more often of an indexed type, rather than a family of types. The type `Vect`, with embedded size is the first example of indexed type that we've encountered. And for non-inductive definition, we've seen the type family `NatOrList`.

- We can now write dependent functions ($\Pi$-types), like the function `ZeroOrNil` and dependent pairs ($\Sigma$-types), like $\Sigma_{(n:Nat)}(\texttt{Vect}\ n\ \texttt{Bool})$, where the type of an argument/component depends on the value of a previous argument/component.
- We can write predicates as indexed types, as we've seen with the predicate "greater than", of type $Nat \rightarrow Nat \rightarrow Type$.
- Thanks to dependent pairs and predicates, we can specify more precisely a type, for example in order to avoid unwanted values in input of a function. More generally, dependent types enable to reason about programs, as any dependently-typed theory contains a logical counterpart, as we will show in the next subsection.

### 1.2.3 Strong specification as dependent types

We've just seen that in this kind of dependently-typed language, we can write predicates (like ">"), and expressions (like $b$>0), that help to create pre-conditions that will be verified at compile-time by the machine. The activity of specifying formally the behaviour of a function doesn't stop here. In fact, we have all the necessary bricks for writing any logical formulae that can be expressed in higher-order logic, which gives us the ability to write post-conditions as well. If one wants to completely specify the behaviour of the previous integer division function, one would usually write the logical formulae $\forall a : Nat,\ \forall b : Nat,\ \forall p : (b > 0),\ (\texttt{divide}\ a\ (b,p))\ *\ b\ \leq a\ \wedge\ ((\texttt{divide}\ a\ (b,p)) + 1)\ *\ b > a$. This formulae says that the result is such that if we multiply it by the denominator `b` then we get back at most the numerator `a`; and also that if this result would be larger by one, then multiplying it by the denominator $b$ would give strictly more than the numerator `a`. We've already mentioned that in a constructive theory, logical formulae correspond to types. That means that this logical formulae will become a type, and more precisely the type of a dependent function, as each universal quantification will become a pi-abstraction. It is a dependent type (and not an ordinary function with the usual function space $\rightarrow$) because the third argument (the proof `p`) uses the value of the second argument (the denominator `b`), and more significantly because the type of the output (which is `divide` $a\ (b,p))\ *\ b\ \leq a\ \wedge\ ((\texttt{divide}\ a\ (b,p)) + 1)\ *\ b > a$) uses the values of all the inputs. Therefore, this logical formulae becomes the following pi-type: `divide_correct` : $\Pi_{(a:Nat)}\,\Pi_{(b:Nat)}\,\Pi_{(p:GT\ b\ 0)}\,(divide\ a\ (b,p)) * b \leq a \wedge ((divide\ a\ (b,p)) + 1)\ *\ b > a$. A proof of it will be any inhabitant of this type, i.e. any dependent function that has this type.

We've just encountered the logical side of the dependent function space : pi-types $\Pi_{x:A}(P\ x)$ can be read as universal quantifications $\forall x : A,\ P\ x$ when they are used to state a logical proposition. There was no existential quantification in this formulae, but we can note that $(\exists x : T,\ P\ x)$ would become a sigma-type $\Sigma_{(x:T)}P\ x$ that carries both the value $x$ and the proof that $x$ is conform, i.e. an inhabitant of $(P\ x)$.

The logical formula that we have expressed above can be expressed as a lemma after the definition of the function. This approach is what we call the "usual approach" for formal certification, where we first write a function, and later complete the type of the function by one or a few additional lemmas, that all together specify completely the behaviour of the function. This is what we've done for the `length` function for lists, with the lemma `length_correct` that we've presented in section 1.1.3 and this is what is usually done with the Coq proof assistant. When the behaviour of a function has to be specified by one or more additional lemmas like `length_correct` and `divide_correct`, we say that the type of the function is a weak type. It is weak in the sense that it is not sufficient for specifying completely the expected behaviour of the function.

However, now that we have dependent types, it becomes possible to write strong types that will specify completely the behaviour of a function. The type becomes of course more complicated, and the same goes for the definition of the function itself, but it has the advantage to ensure from the start that the function being defined is exactly what we need. With the usual approach it was possible to only realise that the function is incorrect when we're trying to prove its correctness in vain. However, with stronger types, this risk is reduced. In the case of `divide`, a possible strong type would be : `divide` : $\Pi_{(a:Nat)}\ \Pi_{(b:Nat)}\ \Pi_{(b0)}\ \Sigma_{(c:Nat)}((c * b \leq$

$a) \wedge ((c+1) * b > a))$. The output is a dependent pair, whose first component `c` is the result of the division, and the second is the proof that this result has all the properties that we need in order to consider this function as correct.

We have to be careful with dependent types and strong specification as dependent types, as they can make both types and definitions a lot more complicated and hard to maintain. Putting all the properties directly into the type is not necessarily always the best thing to do. Instead, trying to find some important properties, that are easy enough to express, and powerful enough for ensuring a good level of safety is often more interesting. Dependent types bring the possibility to add logical properties into our types, but we are not limited to have nothing or everything into them, and the difficult but rewarding goal is to find a good compromise. Often, the entire correctness isn't what really matters. Ensuring some properties or invariants can already give some very strong guarantees and dependent types are very useful for this task.

Let's recall that everything is not only about post-conditions. There's also the fact that some functions are not defined if their inputs does not verify some conditions (called pre-conditions), as we have seen with the function `divide` that is not defined when the denominator is zero. We can also mention the traditional `zip` function for lists, because the two lists in input must have the same length. In a more traditional language that does not have dependent types (like Haskell or Ocaml), we usually return an error, raise an exception, or do the treatment partially and as much as possible, stopping when the end of the shortest list is reached. But when working in a dependently-typed theory, we can write the most precise type `zip : Vect n A → Vect n B → Vect n (A*B)`. Here, the typechecker will (statically) make sure that the two vectors are always forced to have the same size, and it will do that by inspecting their type, as their size is a part of their type.

### 1.2.4 Common problems with dependent types

Thanks to the introduction of dependent types, we can now define more precise types, and we have the ability to write logical formulae. However, these new possibilities also bring new problems. The first of them is that dependent types induce logical links between values and their types, and therefore obtaining an information about the shape of a term might give an information about some other terms. If the system doesn't support these logical links, we will run into problems when defining functions. For example, when a function manipulates a natural number `n` and a vector `v` of type `Vect n T`, in the case where `n` is zero, we know that `v` has type `Vect 0 T`. However, in many systems, including the Coq proof assistant –outside of the proof mode–, `v` will still be seen as an element of type `Vect n T`. Said differently, Coq doesn't natively propagate the informations that we get from the shape of a value to the other types involved in the function. In Coq, one possiblity is to switch to the proof-mode and to do the case analysis with the tactic "destruct", or to still do it in the language-mode, but with the "match ... as ... return ..." construct which makes the definition slightly more complicated. Idris doesn't have this complication, and if the shape of a term gives us informations about some other values, the system will know and use these informations. For example, the following (incomplete) code will typecheck :

```
f : (x:Nat) → (v: Vect x Bool) → Vect 0 Bool
f Z v = v
```

And that typechecks because Idris knows that $v$ has effectively type *Vect* 0 *Bool* in the case where $x$ is zero.

This good support for dependent types that Idris has does not only propagate the information that we get by inspecting some values into the types of other values, but it also uses these informations to enforce the shape of these other values. For example, if a vector of type `Vect x Bool` happens to be a `Nil` then `x` is forced to be `Z`, and the system knows it.

Every time that a term `t` of a dependent type (often a predicate) contains some information about some components of `t`, the system will collect this information and they will be facts that can then be used during the type-checking. For example, one way to define an `Even` predicate is :

```
data Even : Nat → Type where
    IsEven : {x:Nat} → Even (x + x)
```

We can use an element of `Even n` in order to know that `n` has the shape `x+x` for some `x`. This is extremely powerful because, as we notice here, the information about the shape isn't forced to be a constructor : it can be a "non-primitive shape", such as `x+x`. Let's use this information in the definition of a function that divides by two any even natural number :

```
divideByTwo : {n:Nat} → (Even n) → Nat
divideByTwo {n=k+k} IsEven = k
```

We see that this definition of `Even` is particularly useful when the point of view being taken is that an even number can be divided by two[7]. Such a definition is possible because we have this powerful dependent pattern-matching which reveals informations not only about the shape of the expression which is being matched (here, the value of type `Even n`), but also about other expressions (`n` in this case). We will see in chapter 5 that this powerful dependent matching also enables matching on some intermediate computations (still on the left hand side of definitions), and that it can deduce even more informations about the other expressions, with what will be called *views*.

In Coq, because there isn't such a powerful dependent pattern matching built to its core type-theory, some functions are very difficult to define, and often, the last hope will be to define them in the proof-mode, as one would prove a theorem. However, the definition of a function done in proof-mode is very opaque, hard to understand and hard to maintain.

The presence of this powerful dependent pattern matching in Idris makes the definitions of many functions easier, especially the ones that manipulate terms of dependent types which contain links between several values, because the system will take in account the fact that knowing the shape of a term can affect the forms of other terms. This feature brings more structure to the programming activity, as the links between various data are becoming explicit.

However, there is still an important problem that will be encountered frequently when defining a function that returns a value of a dependent type. Often, the type of the function makes the claim that the output

[7] Another definition, more conventional but which does not help for defining the division by two, is to say that 0 is even, and that $\forall n,\ Even\ n\ \rightarrow\ Even\ (S\ (S\ n))$.

has some specific indices (i.e. the output has type $T\ i_1\ i_2\ ...\ i_n$), but the definition creates a term that has some different indices (i.e. a term of type $T\ i'_1\ i'_2\ ...\ i'_n$), and the typechecker will simply reject this definition, as it produces something that does not have the expected type. If the definition makes sense, of course, the corresponding indices must be equal. By "equal", we mean something very specific called propositional equality, that will be detailed in chapter 2, but intuitively this equality means "provably equal", and implies "replaceable". This is in fact the very common and intuitive equality used in mathematics, where the left and right hand side of an $=$ symbol can be written down differently, but must describe objects that are replaceable by each other. A propositional equality is therefore something that has to be proven, and in the case of such a function definition, the user of the system will have to prove that each index $i_k$ is equal to its corresponding $i'_k$ produced in the definition, in order to make the definition accepted by the system. Said differently, when the programmer is defining a function with a slightly different type than expected, that's his duty to prove the convertibility between the two types. That means that as soon as we start to introduce dependent types, we have proof obligations coming along the way. We will now present a concrete example of how these proof obligations arise in practice.

## 1.3 How proof obligations arise on a small example

Proving that one term is equal to another is common in formal verification, and proof obligations arise naturally in dependently typed programming when indexing types over values in order to capture some logical properties. To demonstrate this, we revisit[8] an example from previous work [11] which shows how proof obligations arise when a type is indexed by natural numbers. Our goal is to implement a verified library of binary numbers. To ensure functional correctness, we define the types `Bit` and `Binary` indexed over the value they represent (expressed as a natural number):

```
data Bit : Nat → Type where
     b0 : Bit Z
     b1 : Bit (S Z)

data Binary : (width : Nat) → (value : Nat) → Type where
     zero : Binary Z Z
     (#)  : Binary w v → Bit bit
            → Binary (S w) (bit + 2 * v)
```

The constructor `b0` represents the single bit `0`, so it builds a `Bit` indexed over the natural number zero. Similarly, the constructor `b1` represents the single bit `1`, therefore it builds a `Bit` indexed over the natural number one.

The type `Binary` allows the representation of a binary number of width and value zero with the constructor `zero`, and the representation of larger numbers with the constructor `#`, which extends a binary number `bin` by adding a digit to its right, and the value represented by this new binary number is two times the value (`v`) represented by `bin` plus the value (`bit`) represented by the added digit.

We will write a function to add two binary numbers. To do so, we begin with an auxiliary function, which adds three bits (the third is a

[8] see the file binary.idr on the Github project

carry bit), and produces the two bits of the result, where the first is the more significant bit:

```
addBit : Bit x → Bit y → Bit c → (bX ** (bY **
       (Bit bX, Bit bY, c + x + y = bY + 2 * bX)))
addBit b0 b0 b0 = (_ ** (_ ** (b0, b0, Refl)))
addBit b0 b0 b1 = (_ ** (_ ** (b0, b1, Refl)))
addBit b0 b1 b0 = (_ ** (_ ** (b0, b1, Refl)))
addBit b0 b1 b1 = (_ ** (_ ** (b1, b0, Refl)))
addBit b1 b0 b0 = (_ ** (_ ** (b0, b1, Refl)))
addBit b1 b0 b1 = (_ ** (_ ** (b1, b0, Refl)))
addBit b1 b1 b0 = (_ ** (_ ** (b1, b0, Refl)))
addBit b1 b1 b1 = (_ ** (_ ** (b1, b1, Refl)))
```

Figure 1.2: Addition of three bits

The syntax `(n ** t)` denotes a *dependent pair*, where the type of the second argument `t` can refer to the first argument `n`. So, we can read this type as: "there exists a number `bX`, and a number `bY`, such that we have two bits `Bit bX` and `Bit bY` and the sum of the input bits `c`, `x` and `y` equals `bY + 2 * bX`." For example, on the second line, which corresponds to the computation $1_2 + 0_2 + 0_2 = (01)_2$, the function produces this bits `b0` and `b1`, and a proof that $1 + 0 + 0 = 1 + (2 \times 0)$.

It should be pointed that the proof part of the output is systematically produced by using only `Refl`. Indeed, for each of the eight patterns, a simple computation reduces the two expressions to the same value, and `Refl` therefore stands as a proof of equality of these two identical entities. Also note the usage of the underscore symbol for inferable terms, which let the system fill in these holes.

There is no need to produce any lemma of correctness about `addBit` afterwards as the correct by construction style in which it is written already gives the needed property: the two bits produced effectively represent the addition of the three bits given in input.

We then define the function `adc` that adds two binary numbers and a carry bit. This works for two binary numbers with the same number of bits, and produces a result with one more bit. This result represents the value `c + x + y`, where `c` represents the value of the input carry bit, and `x` and `y` represent the values of the two binary numbers given in input. We would like to write:

```
adc : Binary w x → Binary w y → Bit c
      → Binary (S w) (c + x + y)
adc zero zero carry = zero # carry
adc (numx # bX) (numy # bY) carry
   = let (vCarry0 ** (vLsb ** (carry0, lsb, _)))
          = addBit bX bY carry in
          adc numx numy carry0 # lsb
```

The first pattern performs the addition between the values zero, zero and a carry. In this simple case, we simply have to produce the one-digit value `zero # carry`. The second pattern is when the two binary numbers are respectively `(numx # bX)` and `(numy # bY)` where `numx` and `numy` are smaller binary numbers and `bX` and `bY` are two bits. In this case, we start by adding the least significant bits `bX` and `bY` with the `carry`, and this is performed by calling the function `addBit`. It will essentially produce 2 bits, the first one representing the carry of the result (called `carry0`), and the second one the least-significant bit of the result (called `lsb`). Once we have this intermediate result, we can call resursively `adc` on the smaller terms `numx` and `numy` with `carry0` as the carry that we need to propagate, and finally add `lsb` to the right of this result.

Unfortunately, this definition is rejected by idris' type-checker because the types do not match for both patterns. The result of the first line `adc zero zero carry` is expected to have the type :

```
Binary 1 ((c + 0) + 0)
```

but we provide a term of type :

```
Binary 1 (c + 0).
```

For the second case, the expected index is:

```
((c + (bit2 + (v1 + (v1 + 0)))) + (bit + (v + (v + 0))))
```

while we're trying to provide a term indexed over:

```
   vLsb + (((vCarry0 + v1) + v) + (((vCarry0 + v1) + v)
+ 0)).
```

The definition of adc we have given would behave correctly, and it has *provably* the expected type, but it does not have it *immediately* or *judgementally*: after full reductions[9] the expected and provided types are still different.

For these situations where we want to keep the code simple and to work on the type mismatch problem separately, Idris provides *provisional definitions* with the syntax ?= instead of =. Using provisional definitions here will make the definition accepted, but it will also generate two proof obligations, one for each pattern. Each of these proof obligations requires us to prove an implication P → E where P is the *provided* type and E is the *expected* one, meaning that it is possible to transform (by a computation) the provided term into a term that has the right type[10]. One way to do so is to show that the expected and provided types are in fact *provably equal* and to transform the output along such an equality proof. Without automations, these proof of equalities have to be done by the programmer, and they consist of a series of rewriting steps, where each step uses a property about the operation + on natural numbers. The proof for the second pattern is shown underneath, with some detail elided:

```
adc_lemma_2 = proof {
  intros;
  rewrite sym (plusZeroRightNeutral x);
  [...]
  rewrite (plusAssociative c (plus bit0 (plus v v)) bit1);
  rewrite plusCommutative bit1 (plus v v);
  [...]
  rewrite (plusAssociative (plus (plus x v) v1) (plus x v) v1);
  trivial;
}
```

Such proofs consist of a potentially long sequence of rewriting steps,

[9] In dependently typed theories, the presence of types predicated on values requires also using reductions inside the types.

[10] It is therefore unfortunately possible to throw away the initial output of the function by doing this "conversion", leading to a very misleading code. Idris will be improved in this respect in a future version.

each using one of the properties: neutral element, commutativity, associativity. Without some automation, this sequence of rewritings must be done by the programmer. Not only is this time consuming, but a small change in the definition may lead to a different proof obligation, thus invalidating the proof. A minor change in the datatype, or the definition of `addBit` or `adc` will require us to do a new proof, and thus, without support from the machine, these small proofs can become the everyday routine in any dependently-typed language. It is worth mentioning that even without using dependent types, these proof obligations for equalities happen very frequently during the formal certification of most applications. The proofs are absolutely not reusable because they perform rewritings for very specific terms. Even though they are usually omitted "on paper" for everyday mathematics, they are nonetheless required by proof assistants, no matter how obvious they can be to a human reader.

Our handwritten proof `adc_lemma_2` uses only the existence of a neutral element, and the associativity and commutativity of $+$ on `Nat`. Thus, we're rewriting a term by using the properties of a commutative monoid. With the right choice of combinators [12] such proofs could be made much simpler, but we would like to let a generic prover for commutative monoids to find a proof automatically.

Such an automatic prover is possible to build, because algebraic structures like monoids, groups and rings (commutative or not) all have a very useful property: every expression in them can be normalised to a canonical representation in the absence of extra axioms, i.e. in the case of "pure" algebraic structures. Thus this kind of proofs of equality can be automatically produced by computing the normal forms for each side of the equality, and then comparing them using the syntactical equality, because when they are in normal form, being equal is just a matter of being syntactically the same entity. A few proof automations have already been developed for some structures (ring and commutative rings mostly), for some proof assistants. In the next chapter, we will precisely discuss the aspects of proof engineering and we will review some of the main tools that already exist in the area of proof automation, but we first close this chapter by stating the contributions of this thesis.

## 1.4 Contributions and outline of the thesis

Ring solvers are already implemented for various proof assistants, including Coq [22] and Agda[11]. In this thesis, I describe a certified implementation[12] of an automatic prover for equalities in a *hierarchy* of algebraic structures, including Monoid, Groups and Rings (all potentially commutative), for the Idris language.

The principal novelty is in the approach that we follow, using a new kind of *type-safe reflection*. Working by reflection for implementing tactics has been done several times, including for the implementation of a ring solver for the Coq proof assistant, but without providing any guarantees, contrary to our type-safe reflection. I will compare our approach with other implementations in Section 4.8.

The contributions of this thesis are the following :

1. In chapter 2, we present important concepts of *logic* and *type theory*, including the correspondence between proofs and programs, and we define the notion of equality in the setting of intentional type theories. Then we discuss about proof engineering and we show the current state of the art in the area of proof automation.

2. We present a new *type-safe reflection* mechanism, where the reflected terms are indexed over the concrete inputs, thus providing a direct way to pull out the proofs, and providing the guarantee that the reflection of a term is indeed a faithful representation of the term. The basic ideas of the technique are first presented in chapter 3 on a smaller problem with only natural numbers and addition, and are later adapted for a hierarchy of tactics proving equalities in algebraic structures, in chapter 4.

3. The normalisation procedures are implemented by following a *correct by construction* approach, instead of implementing a normalisation procedure, and proving afterwards that this function is

[11] `http://wiki.portal.chalmers.se/agda/%22?n=Libraries.UsingTheRingSolver`

[12] The implementation of our hierarchy of tactics can be found online at `https://github.com/FranckS/RingIdris`

correct. This approach is much more suitable for programming languages like Idris. Again, this approach will be presented in chapter 3 on the smaller problem, and in chapter 4 for the complete hierarchy of algebraic structures.

4. We develop a hierarchy of tactics where each tactic *reuses the rewriting machinery* of the structure from which it naturally inherits. For example, simplifying neutral elements is only implemented at the monoid level, and each level inheriting from it will reuse it. Re-usability is difficult to obtain when we want to reuse the prover of a less expressive structure. For example, reusing the monoid prover for building the group prover is not trivial, since we lose the possibility to express negations ($-x$) and subtractions ($x - y$). Some specific encodings will be presented in Section 4.6 for solving this problem.

These contributions 1-4 have led to the paper "Automatically Proving Equivalence by Type-Safe Reflection" [42], co-authored with Edwin Brady, which has been presented to the international Conference on Intelligent Computer Mathematics (CICM) 2017.

5. In chapter 5, we present a style of *programming with dependent types* that enables to gain structural information about terms. This style of programming enables to gain some confidence about programs, but often requires some proof automations to be practicable, as the ones developed in the previous chapter of this thesis.

6. As an opening in chapter 6, we discuss *the limits of formal proofs*, the imperfect guarantees that they represent, and we present ideas to increase the trust that we can have in formal proofs. These ideas have been published in a paper titled "Automatic Predicate Testing in Formal Certification" [41] authored by myself, which has been accepted to the international conference on Tests And Proofs (TAP) 2016.

# Chapter 2

# Logic, Type Theory and Equality

*It is with logic that we prove, and with intuition that we find.*

— H. Poincaré

In this chapter, we introduce some important notions of type-theory, especially the correspondence between proofs and programs that we have already mentioned, also called the Curry-Howard correspondence, which leads to constructive logics. Then, we present some well-known problems of type-theory and we show how the propositional equality is formally defined in the setting of intensional and constructive type theory.

## 2.1 Lambda calculus and simple types

Type theory is a formalism in which we develop and reason about functional programs, so it is natural that they are built on top of a micro-kernel that captures the essence of functional programming. This micro-kernel is the lambda calculus. In its simplest form, the untyped lambda calculus, we only have computational rules (together with conversion rules). The most important of them is the $\beta$-reduction, which captures the idea of function application. It is defined in terms of substitution :

$$(\lambda x.t)s \rightarrow_\beta t[x := s]$$

It relies on the notion of capture-avoiding substitution : the notation $t[x := s]$ denotes the term $t$ where all free occurrences of the variable $x$ have been replaced by the term $s$. Often, this notion of substitution is part of the metatheory : it is defined outside of the theory, unlike the abstraction and the function application. Otherwise, when the substitutions are part of the calculi, as it is for instance the case for logic $\mathcal{G}$ which uses the $\lambda$-tree syntax approach to representing syntactic structures, we talk about explicit substitutions, which enable new possibilities [4] such as reasoning about the substitutions themselves, using sharing in structures, etc.

In the untyped lambda calculus, the $\beta$-reduction as a rewriting rule is neither strongly nor weakly normalising, and therefore the untyped lambda calculus is not a terminating system[1]. Also, even if it is possible to simulate common data structure with Church encodings [5], there is no proper notion of type inside of the language, and any lambda term can always be applied to any other term. It is for example possible to pass a list to a function that is intended to operate on natural numbers. In order to avoid these issues, the lambda calculus can be equipped with simple types, and this has also the effect of turning the system strongly normalizing [21] : well-typed terms always reduce to a value (their normal form) after a finite series of reductions.

The simply typed lambda calculus adds to the computational rules some typing rules. The most two important are the well known typing rules of an abstraction and of an application :

$$\frac{\Gamma,\ x : T_1 \vdash e : T_2}{\Gamma \vdash (\lambda x : T_1.\ e) : (T_1 \rightarrow T_2)}\ abst,\ \rightarrow_{intro}$$

$$\frac{\Gamma \vdash f : T_1 \rightarrow T_2\ \Gamma \vdash x : T_1}{\Gamma \vdash f\ x : T_2}\ app,\ \rightarrow_{elim}$$

The first rule says that if the expression $e$ has type $T_2$ in a context where $x$ is declared and has type $T_1$, then the lambda expression that takes an $x$ of type $T_1$ and returns $e$ has type $T_1 \rightarrow T_2$. This typing rule of abstractions is often calls $\rightarrow_{intro}$ as it introduces the arrow type.

[1] As an example, consider the looping term $\Omega = (\lambda x.xx)(\lambda x.xx)$ which reduces to itself.

The second rule says that given a function $f$ of type $T_1 \to T_2$ and an $x$ of type $T_2$, the application of $f$ on $x$ has type $T_2$. This typing rule of the function application is often calls $\to_{elim}$ as it eliminates the arrow type.

We will see in the following sections that these typing rules also have a logical interpretation, which makes type theory not only adapted to computing, but also to reasoning.

## 2.2 Propositions as types : the Curry-Howard correspondence

Type theory is not based on predicate logic, in the sense that it does not require any pre-existing logical concept, unlike set-theory which uses an (unspecified) first-order logic that lies outside of the theory. Instead, all the logical constructions are interpreted within the type theory through the Curry-Howard correspondence, where a logical proposition is interpreted as a type whose inhabitants represent the proof of the proposition. This correspondence between propositions and types necessary leads to a correspondence between proofs and programs (i.e. lambda terms) : if $A$ and $B$ denotes two logical propositions, a proof of $A \to B$ is a function that takes as parameter a proof of $A$ (i.e. an element of the type $A$) and which produces a proof of $B$. For instance, a proof of $(A \to B) \to (B \to C) \to A \to C$ is a function that takes 3 arguments : a proof of $(A \to B)$ –let's call it $f$–, a proof of $(B \to C)$ –lets call it $g$–, and a proof of $A$ –let's call it $a$–, and which produces a proof of $C$. Such a proof of $C$ can be built by composing the two functions : $g\ (f\ a)$ has type $C$, i.e. it is a proof of $C$. This proof is nothing else but the `compose` function[2] that we've seen in 1.1.3. It becomes now clear that the typing rule of an abstraction (presented above) also expresses the fact that the implication $T_1 \to T_2$ can be proven in a context $\Gamma$ if one can prove $T_2$ in the context $\Gamma$ augmented with the hypothesis $T_1$. Correspondingly, the logical interpretation of the elimination rule is that one can prove the

[2] We clearly see that a lemma of propositional logic corresponds immediately to a type, where all the implications symbols are replaced by arrows of the function space, and where the propositional variables are interpreted as type variables.

proposition $T_2$ given a proof of the implication $T1 \rightarrow T_2$ and a proof of $T_1$. In summary, typing rules have a logical counterpart that can be revealed by simply forgetting the terms and keeping only their types.

With this correspondence between propositions and types in mind, it is easy to understand how the usual logical connectors, like $\wedge$ and $\vee$ are defined in constructive type-theory :

```
data And : Type → Type → Type where
  And_intro : {A:Type} → {B:Type} → A → B → And A B

data Or : Type → Type → Type where
  Or_intro_left : {A:Type} → {B:Type} → A → Or A B
  Or_intro_right : {A:Type} → {B:Type} → B → Or A B
```

Figure 2.1: Definition of logical connectors in Idris

The connector $\wedge$ is defined as an inductive type (or sum type), parametrised by the two propositions $A$ and $B$ of the conjunction. It has only one constructor –usually called `And_intro`– which expects two arguments : an element of $A$ and an element of $B$, standing respectively as proof of $A$ and proof of $B$. With this definition, the only way to build an inhabitant of the type $A \wedge B$ (i.e. to build a proof of this conjunction), is to build separately a proof of $A$ and a proof of $B$, and to use the constructor `And_intro` on these two proofs. Following the same idea, the connector $\vee$ is defined with two constructors : `Or_intro_left` that only expects a proof of $A$, and `Or_intro_right`, that only expects a proof of $B$. As the intuition demands, in order to prove $A \vee B$ there are two possibilities : either proving $A$ (and using `Or_intro_left`), or proving $B$ (and using `Or_intro_right`).

The correspondence between propositions and types that we've described leads to an internal logic that is said to be *constructive*.

## 2.3 Constructive logic and type theory

In order to describe constructive logic, we first need to talk about classical logics, because constructive logic is an alternative to it. In *classical logic*, each logical formulae has a truth value, which is either True or False. This is called the "principle of excluded middle". That means that in these systems, we have $\forall\ P,\ P \vee \neg P$. And from this principle, some important consequences can be derived, like the "principle of double negation", which says that a formulae and its double negation are equivalent : $\forall P,\ P \leftrightarrow \neg\neg P$. Often, the principle of excluded middle is not added as an axiom, but can be proven[3] from the definition of the implication, which is $P \rightarrow Q \equiv \neg P \vee Q$ in classical logic. That means that $P \rightarrow Q$, is completely equivalent to $\neg P \vee Q$ : one does not say anything more than the other. This is precisely this interpretation of the implication which is rejected[4] in constructive logics, because this definition does not make any link between the hypothesis $P$ and the conclusion $Q$ : this is a definition only based on a truth table that does not algorithmically construct a proof of $Q$ from a proof of $P$.

In *constructive logics*, the validity of a formulae is defined in terms of *demonstrability* instead of truth values in a truth table. The implication $P \rightarrow Q$ isn't thought anymore as its truth table (as it was the case in classical logic), but is defined as a function that transforms a proof of $P$ into a proof of $Q$. This is in this sense that this logic is "constructive" : it algorithmically constructs the demonstrability of the formulae. In the same spirit, a proof of $A \vee B$ needs to construct a term of type $A \vee B$, where $\vee$ is just a shorthand for the sum type $Or$ with two constructors that we've presented above. Thus, if we inspect the content of a proof of $A \vee B$, it will contain either a proof of $A$ or a proof of $B$. Finally, a proof of $\exists x : T,\ P(x)$ must contain an algorithm able to produce a witness $a : T$ and a proof of $P(a)$, i.e. an evidence of the fact that this element $a$ has effectively the desired property. Thus, in dependently-typed theories, the formulae $\exists x : T,\ P(x)$ is represented with the Sigma types $\Sigma_{(x:T)}P(x)$, and a proof of it is a *dependent pair*, introduced in 1.2. From these definitions comes the computational content of constructive logic.

[3] The proof is done by replacing $Q$ by $P$ in the definition of $P \rightarrow Q$, which gives $P \rightarrow P \equiv \neg P \vee P$. But since $P \rightarrow P$ is a tautology, we get that $\neg P \vee P$ is a tautology too.

[4] In constructive logic, it can be showed that $(\neg P \vee Q) \rightarrow (P \rightarrow Q)$ by analysing the two possibilities for the premise $(\neg P \vee Q)$, but we cannot extract a proof of $\neg P \vee Q$ from a proof of $P \rightarrow Q$ because a proof of a disjunction must tell which part is true by giving a proof of it.

Constructive logic is often inappropriately identified with intuitionistic logic, which is the first constructive system developed (by Brouwer) in 1907 in his "program of intuitionism". That's why many people still talk about intuitionistic logic to talk about constructive logic. This "program of intuitionism" was more a philosophical treaty and still quite far from a complete formalisation, and it's only in the 30s that Heyting started to formalise most of it [23].

In the seventies, Per Martin Löf built the first constructive type theory, referred to as "ML". Today, constructive logics are essential in the context of formal verification by theorem proving, because these logics are naturally well suited for talking about programs and proofs. These constructive theories are implemented as the kernel of most proof assistants and programming languages with proofs, like Coq, Agda and Idris that we have already mentioned.

## 2.4 Basic notions of type theory

The notion of *context* is central to everything in type theory. A context is a container for zero, one or more typing assumptions[5], like $[a : A,\ b : B]$. They are often denoted by the letter $\Gamma$, and they can be thought of as lists of assumptions.

In type theory, one of the most important notion is the one of *judgement*. A judgement is a statement in the meta-language. There are different kind of judgements, including :

- Context validity : $\vdash \Gamma\ Context$ which means that $\Gamma$ is a well formed context of typing assumptions. The condition for being a valid context usually is that a variable should not be assigned multiple types, and that types should be introduced before being used (for example, $[A : Type,\ a : A]$ is a valid context, but not $[a : A,\ A : Type]$). Often, when we will mention a context, we will make implicit the assumption that it is a valid one in order to make things more readable.

[5] These typing assumptions can be seen as "logical assumptions" for the reasons we've explained in 2.2.

- Type validity : $\Gamma \vdash A\ Type$, which means that A is a well-formed type in the context $\Gamma$. In dependently typed languages, everything is a term (including types), but not every term is a type. For example, a lambda-abstraction isn't a type. We will only be able to talk about type inhabitants for valid types.

- Typing judgement : $\Gamma \vdash a : A$, which means that $a$ is a well-formed term of type $A$ in the context $\Gamma$.

Another important notion is the one of *typing rule*, and we have already seen two typing rules with $\rightarrow_{intro}$ and $\rightarrow_{elim}$. A typing rule is an inference rule that has one or more judgements as hypothesis, and a judgement of conclusion. It says that the judgement of the conclusion can be derived from the hypothesis. Because of the correspondence between computations and proofs that we have presented, the reason why typing rules are presented like inference rules in logic becomes clear. The typing rule :

$$\frac{\Gamma_1 \vdash e_1 : T_1 \quad ... \quad \Gamma_n \vdash e_n : T_n}{\Gamma \vdash e : T}$$

has the following logical counterpart :

$$\frac{\Gamma_1 \vdash T_1 \quad ... \quad \Gamma_n \vdash T_n}{\Gamma \vdash T}$$

which can be retrieved from the typing rule by simply erasing the terms, and keeping only the contexts and the types.

## 2.5 Type theory and verification of proofs

In this thesis, the type theory that is being used is the one implemented as the kernel of Idris. It is a variant of the type theory ML. As explained in the previous chapter, this type theory allows types depending on terms [5]. The typing rules of Idris' type theory can be found in [8].

When a type theory is defined, it essentially contains typing rules and computational rules. If we focus on the typing rules, there are three problems of interest[6] :

- Type-checking : given a term $t$ and a type $T$, does $t$ have effectively type $T$ (or a type that can be converted to $T$)? Formally, the system is looking for a derivation of $\vdash t : T$ where both the term $t$ and the type $T$ have been given by the user of the system.
- Type-inference (typability) : given a term $t$, is this term typable, i.e. does there exist a type $T$ such that $\vdash t : T$ can be derived? Here, the type of $t$ (if it exists) is produced, not given in input.
- Type inhabitation (proof search) : given a type $T$, is this type inhabited, i.e. does there exist a term $t$ such that $\vdash t : T$ can be derived? Here, the problem is to find (if it exists) an inhabitant of a type, i.e. a proof of a proposition according to the Curry-Howard correspondence.

Internally, proofs are always represented as lambda-terms, regardless of the way used to write them (as a proof script, or directly as a function), because of the correspondence between proofs and functions described in 2.2. Therefore, in this setting, the verification of proofs is nothing more than type-checking with very rich types, especially with sigma-types and pi-types. Logical formulae correspond to types, and proofs of these formulae to functions. So in order to mechanically verify that a claimed proof $p$ (i.e. a function) is a valid proof of the formulae $F$ it is enough to type-check $p$ against $F$, or to infer the type of $p$ and to compare the obtained type with $F$. If they are the same (after full reductions and modulo alpha-conversion), then $p$ is effectively a valid proof of what has been claimed, i.e. a proof of $F$.

The decidability of type-inference and type-checking depends on the exact system considered. In the most general case, the problem of type-inference is undecidable in dependently typed calculus [19], but

[6] There are also interesting problems about the computational rules, like normalisation and strong normalisation, and interesting problems about both computational rules and typing rules, like preservation of type through reduction.

most implementations of such systems are designed to have at least a decidable type-checking for a big fragment of the language, which means that with some type annotations given by the user when needed, the type-checking becomes decidable (potentially if we put aside some pathological cases). The decidability of type-checking is often[7] an important property for these type theories that are implemented as the internal kernel of proof assistants and programming languages with proofs, as the same type-checking algorithm is also used to *verify* proofs under the Curry-Howard correspondence. When this verification of proofs can be done completely automatically by a decidable type-checking algorithm, the user of the system needs to write proofs, but at least their verification is automatic. However, and unfortunately, such proofs can not be produced automatically by the machine, because the problem of type-inhabitation[8] is undecidable in second-order systems (like system F) and above.

The situation that interests us is the type-checking one, where the user of the system writes a term and a type, and where the system must check if the definition is valid. As we have seen in the introduction with the example of binary numbers indexed over their natural number equivalents, in the presence of dependent types, it is often the case that the type-checking algorithm of the system will reject the definition, because the type claimed by the user and the one produced by the system are different and are not mechanically convertible by reductions and conversions. We also saw that in this case, only a proof –which is at the moment done by the user– can be used to explicitly transform the produced term into another –very similar– term with a different type, therefore making the definition acceptable.

[7] There are however also some interesting theories where type-checking is not decidable and requires some work from the user.

[8] The only interesting (but primitive) system in which type-inhabitation is decidable is the simply typed lambda calculus.

## 2.6 Terms transformation along equality proofs

Given an element $a : T$ it is possible to build an element of the type $T'$ if $T$ and $T'$ are replaceable. This notion of replaceability is implemented in intensional type theory by the propositional equality, which will be described in depth in section 2.7, and which means "being provably equal". One of the essential aspect of the equality, which is implemented as a specific type, is its elimination principle : a proof $p$ of $T = T'$ can be eliminated with an element $a : T$ to build an element of type $T'$, denoted as *rewrite p in a* :

$$\frac{a : T,\ p : T = T'}{(rewrite\ p\ in\ a) : T'} =_{elim}$$

Therefore, an element $a : T$ can be *transformed*[9] into an inhabitant of $T'$ if one can prove the equality between the types $T$ and $T'$. This is something essential, because when one tries to define a function, it is common that the typechecker reports an error because the result of the computation does not have the expected type –the one expressed in the function's type–. When this happens, it is enough to prove the equality between the two types, which is something necessarily doable if the definition makes sense, and to use this equality to transform the produced term into a term of the right type.

In the presence of dependent types, this situation happens very often, and a concrete example has been described in the introduction. The reason is that in presence of a type family $T : I \to Type$, a function with a dependent type can be defined to provide a term, which, according to the typing-rules, is of type $(T\ x)$ while it is expected to be an element of $(T\ x')$. In this case, it will be enough to build a proof of equality between these two indices $x = x'$ in order to obtain a proof of $T\ x = T\ x'$ that can be used to transform the term appropriately. Because of the very nature of the propositional equality, building such proofs of equality can be far

[9] Let's emphasize that the conclusion of the rule is not $a : T'$ in the system that we consider, which would make type-checking undecidable in the general case.

from trivial, and the next subsection will explain why and will describe the two notions of equality in intensional type theory.

## 2.7 Equalities in intentional type theory

### 2.7.1 Definitional and propositional equalities

In Mathematics, equality is a proposition, e.g. we can disprove an equality or assume an equality as a hypothesis. Since in type theory, propositions are seen as types [25], the proposition that two elements $x$ and $y$ are equal corresponds to a type. Thus, if $x$ and $y$ are of type $a$, then the type $Id_a(x,y)$ represents the proposition "$x$ is equal to $y$". If this type is inhabited, then $x$ is said to be provably equal to $y$. Thus, $Id$ is a type family (parametrised by the type $a$) indexed over two elements of $a$, giving $Id\ (a : Type)\ :\ a \rightarrow a \rightarrow Type$. For convenience, we write $(Id_a\ x\ y)$ as $(x\ =_a\ y)$, or even simply as $(x\ =\ y)$. This equality is the equality which can be manipulated in the language.

There is another, more primitive, notion of equality in Intensional Type Theory, called judgemental equality, or definitional equality. This second equality means "equal by definition". The judgemental equality cannot be negated or assumed; we cannot talk about this primitive equality inside the theory. Whether or not two expressions are equal by definition is a matter of evaluating the definitions. For example, if $f\ :\ \mathbb{N}\ \rightarrow\ \mathbb{N}$ is defined by $f\ x\ \equiv\ x\ +\ 2$, then $f$ 5 is definitionally equal to 7. Definitional equality entails unfolding of functions and reductions, until no more reduction can be performed. We denote the definitional equality by $\equiv$.

The judgemental equality has to be included in the propositional equal, because what is equal by definition must be provably equal. This is accomplished by giving a constructor for the type $Id(a,a)$ and nothing when "$a$ is not $b$". In these theories, `Id` is therefore implemented with the following type with one constructor :

```
data Id : a → a → Type where
     Refl : (x : a) → Id x x
```

The only way for $(Id_a \; x \; y)$ to be inhabited is therefore that x and y are equal by definition. In this case, the constructor Refl helps to create a proof of this equality : (Refl x) is precisely the proof which says that $x =_a x$. Here, we are using the notation of Idris where unbound variables like a in the definition of Id are *implicitly* quantified, as a concise programming notation.

### 2.7.2 Equality proofs in non-empty contexts

When reading these definitions, one could wrongly think that because the propositional equality captures exactly the judgemental equality, it will be impossible to prove the equality between syntactically different terms, like $a + b$ and $b + a$ for any $a$ and $b$. In fact, the propositional and the judgemental equality only coincide in an empty context. When proving $\forall a \; b, \; a + b = b + a$, the variables $a$ and $b$ are first abstracted, and the equality goal $a + b = b + a$ makes sense in a context that contains these abstracted variables $a$ and $b$. It is then possible to finish the proof because in these type theories a principle of induction is associated with each inductive type. An inductive principle says that if a proposition holds for the base cases (i.e. the constant constructors), and if it can be showed that when it holds for some terms then it will also hold for the bigger terms obtained by using the recursive constructors, then this proposition will hold for any term of this type. More formally, if a type $T$ is an inductive type with a constant constructor and a recursive constructor, i.e. $T = 1 + T$, defined in idris as :

```
data T : Type where
  T0 : T
  T1 : T → T
```

then we have :
$T\_ind : \forall P : T \to Type, \; (P \; T0) \to (\forall t : T, \; P \; t \to P \; (T1 \; t)) \to (\forall t : T, \; P \; t).$

**An aside about the axioms of the underlying theory :** The induction principles are not necessary pure axioms of the theory. For each type, the associated principle of induction can be proven by the use of a recursive definition[10]. Therefore, we can either take the inductive principles as the primitive brick, and consider that the theory automatically adds such an axiom every time that a new inductive type is defined, or we can consider that the theory allows the computation of fixpoints through well-founded recursive definitions and that this is the primitive construction. The first way to see things is the standard logical point of view where we see the internal type theory as a *logic* with clear axioms that are easy to state, and the second formalisation is a *computational* point of view, where the internal mechanisms of recursive definitions enabled by the language become part of the trusted kernel of the theory. The important thing is that either way, we now have the possibility to prove the equality of terms containing universally quantified variables. For the rest of the text, we will consider that we have these induction principles, and we will talk about them as *axioms* because they are either pure axioms, or they directly follow from the ability to write well-founded recursive definitions.

These principles of induction can be used for proving any proposition, so they also work for the propositional equality. For example, we can prove that $n + 0 = n$ for all $n$ by induction on the `Nat` $n$, even if $n + 0 \not\equiv n$ (in a context containing $n$) with the usual definition of $+$, recursive on its first argument. So, the axiom of induction enables us to prove the equality between terms that are not *definitionally* equal in non-empty contexts. Proving equalities is therefore in these theories something which isn't automatically decidable by the type-checker in the general case : when some variables $x$, $y$, etc are abstracted, evaluating completely the left and right hand sides $L(x,y,...)$ and $R(x,y,...)$ is not enough because the reductions will be stuck until actual values are passed for the variables $x$, $y$, etc.

Because of the dependent pattern-matching that Idris has, given an equality proof $p$ of $L = R$ (with $L$ and $R$ potentially containing free variables), it is possible to prove that necessarily $p$ is $Refl$ by pattern-matching on $p$. Said differently, the only acceptable shape of an equality proof is $Refl$, which means that axiom K is provable in Idris. That means that even for a universally quantified formulae like $\forall x\ y,\ L(x,y,...) =$

[10] see the file others/axiomInduction_byRecursivity.idr

$R(x,y,...)$, when given a proof $p$ of it, $p$ always reduces to $Refl$ when fully applied, which means it can be used as a definitional equality. Therefore, we can manipulate as *definitionally* equal things which have been proven to be *propositionally* equal.

The problem is that when trying to prove a lemma of the shape $\forall x\ y,\ L(x,y,...) = R(x,y,...)$, after abstracting the universally quantified variables, we are left with the goal $L(x,y,...) = R(x,y,...)$ that might not be provable by using directly $Refl$ because until we get actual values for the abstract variables $x$, $y$, etc, we have two things that might not be judgmentally equal. As an example, when trying to prove $\forall x\ y,\ x + y \ = \ y + x$, after the abstraction of $x$ and $y$, we are left with the goal $x + y \ = \ y + x$ that cannot be proven by $Refl$. But since we have these variables $x$ and $y$ in the context, we can use the induction principles (here either on $x$ or on $y$) to make some progress in the proof and ultimately to finish it. However, using the induction principle requires to make proofs (for the base case and the induction step), which, as explained in 2.4 is not fully automatically doable and is therefore the programmer's role.

However, it is often possible to indicate a normalisation procedure that transforms each term into the canonical representative of its equivalence class. This is often possible when the considered datatype and its operations happen to have some classical properties, like the associativity or the commutativity of an operator, or the existence of a neutral element. For example, the type $Nat$ and its operation of addition $+$, defined recursively on its first argument, verifies these three properties:

- $associativity : \forall (x\ y\ z : Nat),\ (x + y) + z = x + (y + z)$,
- $commutativity : \forall (x\ y : Nat),\ x + y = y + x$ and
- $neutral\_element : \forall x : Nat,\ x + 0 = 0 + x = x$.

For such datatypes, it will be possible to automatically decide the equality between two terms. In chapter 3 we will implement such a decision procedure for `Nat` and the property of associativity, and in chapter 4 we will see how this can be generalized to any datatype for various algebraic structures.

Prior to that and in the next subsection, we will talk about proof engineering and we will review the current state of the art in the area of proof automation.

## 2.8 Proof engineering and proof automation

### 2.8.1 Proof engineering

The last decades have seen the emergence of good practices for the development of software, including essential aspects like the ones of code-reusability and automation of the production of code and test cases that can be automatically generated. Unfortunately, we're still lagging behind in the area of formal certification, where the development of proofs is still a tedious activity reserved to experts who produce, often by hands, proofs that are most of the time not reusable.

The development of a formally certified application is usually done by following what we call the "standard approach of formal certification", which proceeds like this for all the functions that are part of the application :

1. The function is declared by stating its type, seen as a weak specification : the function maps a domain D to a codomain C. This is also what we do in a programming language like Ocaml or Haskell.

2. The implementation of the function is given, and it must satisfy its type.

3. Logical properties that will be needed for expressing the correctness lemma of the function are defined. For example, for building a certified sorting function, defining a predicate *isSorted* is an essential key (see the discussion in chapter 6 about the potential danger that such definitions can represent).

4. A correctness theorem which captures all the properties we want this function to verify is stated.

5. A proof that the given implementation of the function fulfils this stronger specification is developed, often using intermediate lemmas.

If the implementation changes (even just a bit, for a small optimisation, or for improving the clarity of the code for example), the proof of correctness will most of the time become invalid. This is a big issue because :

- Making proof is a tough activity, and we don't want to throw away the production of this tedious work
- We are therefore encouraged to never change the implementation

As we know, software naturally evolve with time, with the need of new features, and that often implies adding pieces of informations in types (like adding constructor to inductive types), and re-adapting the implementation of the existing functions with these new dataypes, which in turn implies the fact that the proof of correctness needs to be adapted or in the worst case completely redone. Most of the time, even a small change will completely break the proofs that have been built. Formal certification is, on this dimension, incompatible with the good practices of software engineering that have been discovered. This apparent incompatibility is one of the reason which currently prevents formal methods to be used broadly, and especially for real-world applications, as it does not scale well with large developments.

This is why, when this is doable, we would like to let the machine automatically generate the proof of correctness. Managing changes in data-types, algorithms and in proofs of correctness would become much more easy to handle if all the mechanical lemmas could be automatically proven by the machine, as less effort would have to be developed in order to certify (partially or totally) an application.

Therefore, the general tendency in formal certification is to equip programming languages and proof assistants with *proof automations* that discharge the programmer from proving all the mechanical lemmas that can be automatically proven. In this thesis, we will describe the implementation of a hierarchy of *tactics* that prove equalities in algebraic structures, like monoids, groups and rings.

### 2.8.2 State of the art in proof automation

Various proof automations have already been implemented for proof assistants and programming languages like Coq and Agda. Coq is equipped with a unified ring and semi-ring prover [22] which proves equalities in such structures (potentially commutative), with variables universally quantified. When given a goal of the form $forall\ (x\ y\ z ... : T),\ f(x,y,z,...) = g(x,y,z,...)$ where $T$ is a ring and $f$ and $g$ are compositions of operations of the ring (like $+$ and $*$), the prover will automatically generate a proof of it if such a proof exists. Otherwise, if it fails, then that means that either the given equality is false, or that proving it requires some properties that are external to the ring. This prover has been implemented by reflection : the left and the right hand side of the potential equality are being reflected in the language, with the use of specific data-types.

As well as the ring prover, Coq also provides the Omega solver [16], which solves a goal in Presburger arithmetic (i.e. a universally quantified formula made of equations and inequations), and a field [18] decision procedure for real numbers, which plugs to Coq's ring prover after simplification of the multiplicative inverses.

In Coq, most tactics are implemented in Ocaml (using the abstract syntax of Coq terms). A few others are implemented using Ltac [17], a proof dedicated and untyped meta-language for the writing of automations, which allows to do pattern-matching on goals and on proof contexts. More recently, the Mtac extension [51] provides a typed language for implementing proof automation which supports dependently-typed tactic programming.

Agda's reflection mechanism[11] gives access to a representation of the current goal (that is, the required type) at a particular point in a program. This allows various proof automations to be done in Agda [27, 28].

[11] `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Reflection`

Proofs by reflection has been intensively studied [13, 30], but without anything similar to the type-safe reflection that we will present in the next two chapters, and which are the essential part of thesis.

# Chapter 3

# Automating Proofs by Reflection

> *The fact that an opinion has been widely held is no evidence whatever that it is not utterly absurd; indeed in view of the silliness of the majority of mankind, a widely spread belief is more likely to be foolish than sensible.*
>
> — B. Russell, *Marriage and Morals*

We will first present the basic ideas of our technique for solving equalities on a simplified problem, in which we aim to deal with universally quantified natural numbers and their addition. What we want is to automatically generate proofs of this kind of goals :

**Example 1.** $\forall(x\ y\ z : Nat),\ (x+y)+(x+z) = x+((y+x)+z)$

For this smaller problem, we have decided to only work with the associativity of $+$, expressed formally by the property $plusAssociative :$ $\forall x1\ x2\ x3,\ (x1+x2)+x3 = x1+(x2+x3)$ and with the fact that Z is a right neutral element for the $+$ operation, expressed formally by the property $plusZeroRightNeutral : \forall x, x+Z = x$. Note that Z is also a left neutral element, but this property is not needed because we have this behaviour by reduction, as $+$ is defined recursively on its first argument. Of course, some datatypes can have more or less properties than these two, and this work will be extended in chapter 4 where we will present a very general hierarchy of provers for multiple algebraic structures.

Thus, in summary, in this section, we want to write a decision procedure, able to tell if two expressions composed of universally quantified natural numbers and additions of these numbers are equal, and to produce a proof of this equality if appropriate, where “equal" has the meaning syntactically equal or equal thanks to the properties of associativity and of right neutral element.

## 3.1 Working by reflection

When trying to prove this kind of equalities, the variables are abstracted, and they become part of the context. In the example 1 given above, after abstraction of the variables, the goal becomes simply $(x+y)+(x+z) = x+((y+x)+z)$, which is something of the general form $x = y$. The general idea –that will also apply for the more general problem detailed in the next chapter– will be to normalize both sides of the "potential equality" $x = y$, and afterwards to compare them with a syntactic equality test. The goal of the normalization is to compute a canonical representation for an expression, such that any other number provably equal to this expression (by using the two available properties) will have the same canonical representation, and reciprocally. For example, the normalisation might transform $x+((y+x)+z)$ into $((x+y)+x)+z$ if we decide that the normal form will be completely left associative. If so, then the normalisation of the right hand side will also give $((x+y)+x)+z$. It will then be possible to decide the equality by simply comparing the normalised left and right hand sides with a strict syntactic equality test. This is in fact what we do all the time when we have to decide if two things, written differently, are equal or not. For example, when given two mathematical polynomials, in order to decide the (pointwise) equality of these two functions, a technique that always works is to decide once and for all a canonical representation of polynomials, and to put both polynomials in this form. If the normalised forms are the same, then the two original polynomials are equal, otherwise they do not represent the same computation.

In fact, such a normalisation function can't be written directly because in the LHS and RHS of $x = y$, we potentially have variables which have been universally quantified. And the normalization function needs to do different treatments for a “variable natural number" (i.e. a number which has been universally quantified) and for the constant $Z$. This is not possible yet, because once the variables are abstracted, they are just ordinary values of type $Nat$, and there is no way to distinguish them. Indeed, this information only exists at the level of the abstract syntax trees representing the two terms, and we do not have a direct access to these ASTs. For this reason, we will work by reflection. In this example, it means that we will define a datatype that will be used as an encoding of natural numbers, or more precisely, as an encoding of natural numbers composed of “variable numbers", $Z$, and additions of these things. This datatype will let us inspect the internal structure of a number by pattern matching. Previously, we were only able to pattern match a natural number against the constructors `Z` and `S`, which wasn't what we needed. With the first version of the datatype `Expr` presented in Figure 3.1, we will be allowed to pattern match an encoding of number against the constructors `Plus`, `Var` and `Zero`.

```
data Expr : (Γ : Vect n Nat) → Type where
    Plus : {n:Nat} → {Γ:Vect n Nat} →
           Expr Γ → Expr Γ → Expr Γ
    Var  : {n:Nat} → {Γ:Vect n Nat} →
           (i : Fin n) → Expr Γ
    Zero : {n:Nat} → (Γ:Vect n Nat) →
           Expr Γ Z
```

Figure 3.1: First version of reflected natural numbers

Variables are represented using a De Brujin-like index: `Var FZ` denotes the first variable abstracted, `Var (FS FZ)` the second one, and so on.

The type `Expr` is indexed over a vector of natural numbers $\Gamma$, which is the context of all universally quantified variables. In the example 1, we will encode $(x + y) + (x + z)$ and $x + ((y + x) + z)$ in a context where three elements are present. The first element of this context denotes the variable $x$, the second denotes $y$, and the third denotes z. Thus, the left hand side will be encoded by :

```
e1 : (x, y, z : Nat) → Expr [x, y, z]
e1 x y z = Plus (Plus (Var FZ) (Var (FS FZ)))
                (Plus (Var FZ) (Var (FS (FS FZ))))
```

## 3.2 Type-safe reflection

If we continue with this first definition of `Expr`, the normalisation function will take an `Expr` and produce another `Expr`, and we will need to prove that after normalisation, the natural number represented by the resulting expression is the same than the one represented by the original expression. That can be expressed by the following correctness lemma :
`∀ e:Expr Γ, reify Γ (reduce e) = reify Γ e`
where `reify` is a function computing the interpretation of an `Expr` in a context $\Gamma$, that is to say, the natural number that this `Expr` is encoding.

Another possibility, which we follow, is to add an index to the type `Expr` that precisely captures the concrete number that the `Expr` is encoding. This representation creates a link between a syntactical expression and its semantics, and is the first brick to our type-safe reflection mechanism. With it, it won't be necessary to define the `reify` function, as the type of an encoded expression already gives the concrete value it represents : the concrete element reflected by a term of type `Expr Γ x` is precisely the index `x`. Therefore, we get the property that the reflection of a term `x` is guaranteed to be a faithful representation of `x`.

```
using (x : Nat, y : Nat, Γ : Vect n Nat)
  data Expr : (Vect n Nat) → Nat → Type where
      Plus : Expr Γ x → Expr Γ y →
             Expr Γ (x + y)
      Var  : (i : Fin n) → Expr Γ (index i Γ)
      Zero : Expr Γ Z
```

Figure 3.2: Second version of reflected number with embedded denotation

For an expression $e_x$ : `Expr Γ x`, we will say that $e_x$ denotes (or encodes) the number `x` in the context $\Gamma$. When an expression is a variable `Var i`, the denoted number is simply the corresponding variable in the context, i.e. `(index i Γ)`. Also, the `Zero` expression denotes the natural number `Z`. Finally, and fairly trivially, if $e_x$ is an expression encoding the number $x$, and $e_y$ is an expression encoding the number $y$, then the expression `Plus` $e_x$ $e_y$ denotes the concrete natural number $(x+y)$.

The second datatype `Expr` that we use makes a link between the syntax of expression and their semantics. We will take advantage of it when writing the normalisation function, thanks to Idris' dependent pattern-matching. However, this representation means that it is impossible to write a purely syntactic operation not married to any semantics. An alternative approach here would have been to define the type of expression as only indexed over the number of variables in which they make sense (i.e. `Expr: Nat → Type`) and to have separately a `reify` function of type `Expr n → Vect n Nat → Nat` that evaluates an expression in a context having the right size. We could then retrieve a type isomorphic to `Expr` by defining `Expr' Γ x` as the dependent pair `e:Expr n ** reify e Γ = x`.

## 3.3 A correct by construction approach

We want to write the reduction function on a *correct by construction* way, which means that no additional proof should be required after the definition of the function. Thus, `reduce` will produce the proof that the new `Expr` freshly produced has the same interpretation as the original `Expr`, and this will be made easier by the fact that the datatype `Expr` is now indexed over the real –concrete– natural number : a term of type `Expr Γ x` is the encoding of the number `x`. Thus, we can write the type of `reduce` like this :

```
reduce :  Expr Γ x → (x' ** (Expr Γ x', x = x'))
```

The function `reduce` produces a dependent pair : the new concrete number `x'`, and a pair made of an `Expr Γ x'` which is the new encoded term indexed over the new concrete number we have just produced, and a proof that old and new –concrete– natural numbers are equal.

This function doesn't simply produce an `Expr Γ x`, because the concrete number on which the resulting expression will be indexed is not necessary syntactically equal to the original number. Indeed, `x` and `x'` can represent the same computation without being syntactically equal. Indeed, they can precisely be equal thanks to the two available properties of associativity and neutral element that we have, and this is precisely why we are building such proof automations. Said differently, even if we can prove $x = x'$ (if the function is correctly defined), we do not have $x \equiv x'$.

It would be a bad idea to force this function to produce an `Expr Γ x` as it would mix computations and proofs. Doing so would make this function compute (in an intermediate result) an `Expr Γ x'` –that's the computational part–, a proof of $x = x'$ into another intermediate result –that's the proof part–, and would use this proof to transform the `Expr Γ x'` into an `Expr Γ x`, that would finally be returned. However, after using the proof for converting the type of the expression, the proof would be lost as it is not returned. And in fact, what really interests us in this function is precisely this proof of $x = x'$, so we better return it. This proof is crucial because it will be the essential component for building the desired proof of $x = y$, which is the main problem that we are trying to solve.

More precisely, the tactic will work as follow. We have an expression $e_x$ encoding $x$, and an expression $e_y$ encoding $y$. The Idris expressions $x$ and $y$ are potentially open terms, so they make sense in a potentially

non empty context, due to variables that were universally quantified. We will normalize $e_x$ and this will give a new concrete number $x'$, a new expression $e_{x'}$ :`Expr Γ x'`, and a proof of $x = x'$. We will do the same with $e_y$ and we will get a new concrete number $y'$, an expression $e_{y'}$ :`Expr Γ y'`, and a proof of $y = y'$.
It is now enough to simply compare $e_{x'}$ and $e_{y'}$ using a standard syntactic equality test because these two expressions are now supposed to be in normal form. This syntactic equality can be defined like this :

```
eqExpr : (e : Expr Γ x) → (e′ : Expr Γ y) → Maybe (e = e′)
eqExpr (Plus x y) (Plus x′ y′) with (eqExpr x x′, eqExpr y y′)
  eqExpr (Plus x y) (Plus x y) | (Just Refl, Just Refl)
                                           = Just Refl
  eqExpr (Plus x y) (Plus x′ y′) | _ = Nothing
eqExpr (Var i) (Var j) with (decEq i j)
  eqExpr (Var i) (Var i) | (Yes Refl) = Just Refl
  eqExpr (Var i) (Var j) | _ = Nothing
eqExpr Zero Zero = Just Refl
eqExpr _ _ = Nothing
```

Now, if the two normalised expressions $e_{x'}$ and $e_{y'}$ are equal, then they necessary have the same type[1], and therefore $x' = y'$. By rewriting the two equalities $x = x'$ and $y = y'$ (that we obtained during the normalisations) in the new equality $x' = y'$, we can get a proof of $x = y$. This is what the function `buildProof` is doing.

```
buildProof : {x : Nat} → {y : Nat} → Expr Γ x′
  → Expr Γ y′ → (x = x′) → (y = y′) → Maybe (x = y)
buildProof ex′ ey′ lp rp with (eqExpr ex′ ey′)
  buildProof ex′ ex′ lp rp | Just Refl = ?MbuildProof
  buildProof ex′ ey′ lp rp | Nothing = Nothing
```

The argument of type `Expr Γ x′` is the normalised reflected left hand size of the equality, which represents the value `x′`. Before the normalisation, the reflected LHS was reflecting the value `x`. The `Expr Γ y′` is the normalised reflected right hand size, which now represents the

[1] We are working with the heterogeneous equality JMeq by default in Idris, but as always, the only way to have a proof of a:A = b:B is when $A \equiv B$.

value `y'`, but which was encoding `y` before the normalisation. The function also expects a proof of $x = x'$ and of $y = y'$, and we will be able to provide them because the normalisation function also produces the proof of equality between the original and the new concrete values.

**Note on metavariables :** In Idris, it is possible to put a metavariable `M`, noted with an interrogation mark `?M`, in place of a definition. This metavariable behaves as a placeholder that needs to be later defined or proven. Often, we will use metavariables as placeholders for proofs that will be done later, because we want to focus on algorithms and to put aside proofs, exactly as we did with provisional definitions (introduced in 1.3). What Idris does when it encounters such a metavariable `?M` is that it creates a name `M` with its type, but with no definitions associated. Obviously, the program is no longer total as long as the definition of the metavariable is not completed, and running it at this stage might not terminate or might not give an answer for some inputs. According to the Curry-Howard correspondence (see section 2.2), if this term was behaving as a proof term, then running the program is still possible, but all the logical properties obtained are now subject to the validity of this axiom. Said differently, the proofs are now made with this additional axiom, and if one wants to get rid of this axiom, one has to do the corresponding proof. The definition of a metavariable can be done in proof mode, or can be done traditionally as any other function.

As mentioned above, the proof-term that fills the hole represented by the metavariable `MbuildProof` is just a rewriting of the two equalities that we have :

```
MbuildProof = proof {
  intros; refine Just; rewrite sym p1; rewrite sym p2;
  exact Refl;
}
```

Finally, the main function which tries to prove the equality $x = y$ simply has to reduce the two reflected terms encoding the left and the right hand side, and to use the function `buildProof` in order to compose the two proofs that we just obtained :

```
testEq : Expr Γ x → Expr Γ y → Maybe (x = y)
testEq ex ey =
   let (x' ** (ex', px)) = reduce ex in
   let (y' ** (ey', py)) = reduce ey in
      buildProof ex' ey' px py
```

The only remaining part is to define the function `reduce`. To do that, we have to decide a canonical representation of associative natural numbers. We decide that the left associative form will be the canonical representation. Thus, the `reduce` function has to rewrite the reflected terms by rearranging the parentheses in order to transform the underlying concrete number in the form $(...((x1+x2)+x3)...+xn)$. To do so, one possibility is to define a new datatype which captures this property, and to write a function going from `Expr` to this new type. Thus it will be easier to be certain that we are effectively computing the normal form : forcing properties to hold by the shape of a datatype is a good usage of dependent types when, like here, it doesn't introduce more complications.

```
data LExpr : (Γ : Vect n Nat) → Nat → Type where
     LPlus : LExpr Γ x → (i : Fin n)
             → LExpr Γ (x + index i Γ)
     LZero : LExpr Γ Z
```

Figure 3.3: Reflected left associative numbers

This datatype has only two constructors. In fact, it combines the previous `Var` and `Plus` constructors so that it becomes impossible to write an expression which isn't left associative, because the constructor `LPlus` is only recursive on its first argument.

As part of the normalization, we write a function `expr_l` which converts an `Expr Γ x` to a `LExpr Γ x'` and which produces a proof of $x = x'$. This function will therefore use the two available properties multiple times, while rewriting the term until the fully left associative desired form is obtained.

```
expr_l : Expr Γ x
```

```
          → (x' ** (LExpr Γ x', x = x'))
expr_l Zero = (_ ** (LZero, Refl))
expr_l (Var i) = (_ ** (LPlus LZero i, Refl))
expr_l (Plus ex ey) =
  let (x' ** (ex', px)) = expr_l ex in
  let (y' ** (ey', py)) = expr_l ey in
  let (res ** (normRes, Pres)) = plusLExpr ex' ey' in
    (res ** (normRes, rewrite px in (rewrite py in Pres)))
      where
      plusLExpr : {Γ : Vect n Nat} → {x, y : Nat}
            → LExpr Γ x → LExpr Γ y
            → (z ** (LExpr Γ z, x+y=z))
      plusLExpr {x=x} ex LZero =
        (_ ** (ex, rewrite (plusZeroRightNeutral x) in Refl))
      plusLExpr ex (LPlus e i) =
        let (xRec ** (rec, prfRec)) = plusLExpr ex e in
            (_ ** (LPlus rec i, ?MplusLExpr))
```

In the case of an addition `Plus ex ey`, the function `expr_l` does the job of normalisation recursively on `ex` and on `ey`, and then it uses the sub-function `plusLExpr` to normalise the addition of these two –already normalised– terms. This sub-function `plusLExpr` has two kind of simplifications to do. When the second argument is an `LZero`, it simply returns its first arguments along with the justification for this rewriting, which obviously uses `plusZeroRightNeutral`. However, when the second argument is an `LPlus e i`, it continues recursively by computing `plusLExpr ex e`, and it finally adds `i` to it. That had the effect of moving the parenthesis on the left, and the correctness of this treatment is going to be justified by the use of `plusAssociative` in the proof that corresponds to the meta variable `MplusLExpr` .

This metavariable `MplusLExpr` –that expresses the equality between the old and the new index– requires us to prove the goal :
`x1 + (x2 + index i Γ) = xrec + index i Γ`
in a context where we've got, amongst other things :
`prfRec : x1 + x2 = xrec.`

By using the property of associativity on the goal, we now need to prove `(x1 + x2) + index i Γ = xrec + index i Γ`, which can be done by rewriting the proof `prfRec` obtained recursively.

```
MplusLExpr = proof {
  intros
  rewrite (sym (plusAssociative x1 x2 (index i Γ)));
  rewrite prfRec;
  exact Refl;
}
```

It is really important to understand that the kernel of the automatic construction of the desired proof of $x = y$ happens precisely in these usage of `plusZeroRightNeutral` and `plusAssociative` that are nested by chaining recursive calls in the definition of `expr_l` and of its meta-variable `MplusLExpr`. These proofs replace the arithmetical proofs that we were doing previously by hand in section 1.3 with the lemma `adc_lemma_2`.

Using this new datatype `LExpr` has changed the representation of our encoded natural numbers, so we need to convert back an `LExpr Γ x` to an `Expr Γ x`. The function `l_expr` performs this easy task :

```
l_expr : LExpr Γ x → Expr Γ x
l_expr LZero = Zero
l_expr (LPlus x i) = Plus (l_expr x) (Var i)
```

We notice that in order to transform the expression into its left associative equivalent representation, we've effectively needed to know where the variables and the $Z$ constants are : the functions *expr_l* and *l_expr* are doing different treatments for these different possibilities.

We can now define the normalisation function, which is just the composition of the two previous functions `expr_l` and `l_expr`:

```
  reduce : Expr Γ x → (x' ** (Expr Γ x', x = x'))
  reduce e =
     let (x' ** (e', prf)) = expr_l e in
```

```
          (x' ** (l_expr e', prf))
```

At the moment, what we've got is not exactly a real tactic, in the sense that we only have a function which produces a value of type `Maybe` $(x = y)$. A real tactic would be a wrapper of this function that would fail properly with an error message when the two terms are not equal. However, here, when $x \neq y$, the function `testEq` will simply produce the value `Nothing`.

## 3.4 Usage of the “tactic"

It is now time to see how to use this minimalist “tactic". Let's go back to the example 1. We had defined the expression $e1$ representing the value $((x+y)+(x+z))$ and we now have to produce the encoding for $(x+((y+x)+z))$, still in the context $[x,y,z]$ of three abstracted variables.

```
e1 : (x, y, z : Nat)
    → Expr [x, y, z] ((x+y) + (x+z))
e1 x y z = Plus (Plus (Var FZ)
                      (Var (FS FZ)))
                (Plus (Var FZ)
                      (Var (FS (FS FZ))))

e2 : (x, y, z : Nat)
     → Expr [x, y, z] (x + ((y + x) + z))
e2 x y z = Plus (Var FZ)
                (Plus (Plus (Var (FS FZ))
                            (Var FZ))
                      (Var (FS (FS FZ))))
```

Figure 3.4: Two test expressions

The numbers denoted by the expressions $e1$ and $e2$ are equal, and we can generate a proof of this fact by using `testEq`.

```
e1_e2_testEq : (x, y, z : Nat)
    → Maybe (((x + y) + (x + z)) = (x + ((y + x) + z)))
```

```
e1_e2_testEq x y z = testEq (e1 x y z) (e2 x y z)
```

We can evaluate this term, which produces `Just` and a proof of equality between the two underlying concrete values :

```
#\x => \y => \z => e1_e2_testEq x y z

\x => \y => \z => Just (replace (sym (replace (sym (replace
(sym (plusAssociative x 0 y)) (replace (replace (sym
(plusZeroRightNeutral x)) Refl) Refl))) (replace (sym
(replace (sym (plusAssociative x 0 z)) (replace (replace
(sym (plusZeroRightNeutral x)) Refl) Refl))) (replace (sym
(plusAssociative (x+y) x z)) [...]
: (x : Nat) → (y : Nat) → (z : Nat)
  → Maybe ((x + y) + (x + z)
             = x + ((y + x) + z))
```

Figure 3.5: Automatically generated proof (truncated)

And we effectively get the proof of equality we wanted. As expected, this proof uses the properties of associativity (`plusAssociative`) and the property of neutrality of $Z$ for $+$ (`plusZeroRightNeutral`).

## 3.5 Construction of the reflected terms

For the moment, even if what we have is perfectly usable and works, we had to create the reflected terms $e1$ and $e2$ by hand, which is easy but time consuming. We have replaced the (potentially hard) problem of proving something by the simpler problem of building some encodings. This is already a huge simplification, because as it can be seen in the definitions of $e1$ and $e2$, the reflected terms completely follow the structure of the expression to encode : there's absolutely no creativity needed for this task, unlike the proving activity. The way to create the encodings is in fact so systematic that, of course, we would like to automatize it in order to get a real and completely automatic tactic.

However, we can also note that even when done by hand, there is no room for making mistakes in this simple task of encoding : we simply can't generate a wrong encoding : if $e1$ and $e2$ are not respectively reflecting $((x+y)+(x+z))$ and $(x+((y+x)+z))$ then these definitions won't typecheck because the expected and real index won't match.

Still, we want to build an automatic way of constructing these reflected terms because what we have currently is not convenient enough for being used regularly. We want to program an automatic way of going from the concrete values (of type `Nat`), to the reflected terms (of type `Expr`). The only way to do that is to inspect the abstract syntax tree of the concrete value. By using Idris's reflection mechanism, we can tag a function with the keyword "%reflection", which means that this function runs on syntax instead of values. We will use this possibility to write a function of reflection that does the job of producing the reflected terms for the user of the tactic :

```
%reflection
total
reflectNat : {n:Nat} → (Γ : Vect n Nat) → (x:Nat)
      → (m ** (Γ' : Vect m Nat ** (Expr (Γ ++ Γ') x)))
```

This function will reflect the natural number $x$ in the context of $\Gamma$, which contains $n$ already abstracted variables. This function will compute an extension –of arbitrary size $m$– to the context, called $\Gamma'$. This extension will contain the variables used in $x$ that were not already present in $\Gamma$. It will also produce the reflected term, which is expressed in the complete context $\Gamma + + \Gamma'$, and which is, of course, indexed over the concrete value $x$.

If $x$ is the natural number `Z`, then we don't have any variable to add to $\Gamma$, so the extension will be the empty vector, and the reflected expression is simply `Zero`.

```
reflectNat {n=n} G Z =
    (Z ** ([] ** (Zero {n=n+0} {G=G++[]})))
```

Figure 3.6: Reflecting natural numbers - pattern for zero

If the natural number to reflect is an addition $x+y$, then we will start by reflecting $x$ in the context of $\Gamma$. That will give us an extension $\Gamma'$ and an expression `ex` of type `Expr (Γ ++ Γ') x`. We continue by reflecting $y$, but this time in the context $(\Gamma + + \Gamma')$ of $n + m$ already abstracted variables, because the reflection of $x$ has potentially abstracted some new variables, and we don't want to abstract them a second time. That will give us a second extension $\Gamma''$ and an expression `ey` of type `Expr ((Γ ++ Γ') ++ Γ'') y`. We now simply want to return the `Plus` of `ex` and `ey`, but we can't immediately, because these encodings aren't defined on the same context. The context in which `ex` makes sense is $(\Gamma + + \Gamma')$, but the context in which `ey` makes sense is $((\Gamma + + \Gamma') + + \Gamma'')$. We will therefore need a weakening function that takes a reflected expression $ex$, and returns the same expression, but expressed in an augmented context.

```
reflectNat Γ (x + y) =
  let (_ ** (Γ′ ** ex)) = (reflectNat Γ x) in
  let (_ ** (Γ″ ** ey)) = (reflectNat (Γ ++ Γ′) y) in
  let result = Plus (weaken Γ″ ex) ey in
      (_ ** ((Γ′ ++ Γ″) ** ?MreflectNat_1))
```

Figure 3.7: Reflecting natural numbers - pattern for a plus

The total extension that has been computed is $\Gamma' + + \Gamma''$, and the metavariable `MreflectNat_1` simply uses the associativity of the append operation to prove that the provided context $((\Gamma + + \Gamma') + + \Gamma'')$ is equal to the expected one $(\Gamma + + (\Gamma' + + \Gamma''))$.

The function `weaken` (Figure 3.8) is easy to write because we're adding the extension $\Gamma'$ on the right of $\Gamma$. For example, extending the context $[v, w, x]$ with $[y, z]$ produces the context $[v, w, x, y, z]$. The variables $v$, $w$ and $x$, which were respectively refereed to as the first, second and third variables in $\Gamma$, are still the first, second and third variables in the complete context. That means that a variable `Var i` will still be a `Var i` after the weakening : the position $i$ does not change by augmenting the context.

```
weaken : {n:Nat} → {m:Nat} → {Γ:Vect n Nat} → {x:Nat}
  → (Γ':Vect m Nat) → (Expr Γ x) → (Expr (Γ ++ Γ') x)
weaken Γ' Zero = Zero
weaken Γ' (Plus e1 e2) = Plus (weaken Γ' e1) (weaken Γ' e2)
weaken Γ' (Var i) = Var (convertFin _ i m)
```

Figure 3.8: Weakening function

The position `i` hasn't changed but however, the original `i` had type `Fin n`, but the new `i` must have type `Fin (n+m)`. This is why we've used `convertFin`, which returns the same element, but seen in a bigger `Fin`.

```
convertFin : (n:Nat) → (i:Fin n) → (x:Nat) → Fin (n+x)
```

We've treated the case of the constant `Z` and the case of the addition. We now have to deal with the last possibility of a variable. For encoding a variable, we must see if this variable is already present in the context $\Gamma$ of variables already abstracted. If it is already present there, then there is no extension to build, and the reflected term is simply `Var (convertFin n i Z)`, where `i` is the position of the variable in this context $\Gamma$. The conversion is needed because the original context is $\Gamma$, and the new one is $(\Gamma + + [\,])$, which aren't automatically unifiable. However, if this variable is missing, then we must add it, which means that we will return an extension containing this single variable. The original context $\Gamma$ had a size of $n$, and we've built an extension of size one, so the complete context has therefore size $(n+1)$. The variable that we've added is currently the last one of these $(n+1)$ variables. If we use a function[2] `lastElement' : (pn:Nat) → Fin (pn+1)` to construct the last element of a `Fin` of size $pn+1$, then the reflected term that we need to produce is simply `Var (lastElement' n)`.

[2] This function doesn't have the type $(n : Nat) \rightarrow Fin\ n$ because there is no last element of a `Fin` of size zero, as there is no element at all.

```
reflectNat Γ t with (isElement t Γ)
  | Just (i ** p) = let result =
              Var {Γ=Γ++[]} (convertFin n i Z) in
                  (Z ** ([] ** ?MreflectNat_2))
  | Nothing ?= (((S Z) ** ([t] **
                  Var {Γ=Γ++[t]} (lastElement' n))))
```

Figure 3.9: Reflecting natural numbers - pattern for a variable

`isElement` is a function which checks whether an element `x` belongs to a vector *Γ*, and if so, returns `Just` and a dependent pair containing the index of `x` in this vector, and a proof that `index i Γ = x`.

```
isElement : {n:Nat} → (x : a) → (Γ : Vect n a)
            → Maybe (i:Fin n ** (index i Γ = x))
```

The metavariable `MreflectNat_2` will require us to prove that the provided term of type `Expr (Γ ++ []) (index (convertFin n i Z) (Γ ++ []))` is transformable into a term of the expected type `Expr (Γ ++ []) t`. This is doable by using the proof `p` –returned by `isElement`– which says that `index i Γ = t`, together with the fact that `convertFin` does not change the index, but only converts its type.

As for the case where `t` was not in the original context (the `Nothing` case), we will need to prove that we can convert `(index (lastElement' n) (Γ ++ [t]))` into `t`. This is doable because we can prove and use the following lemma :

```
indexOfLastElem :  {T:Type} → {n:Nat} → (v:Vect n T)
→ (x:T) → index (lastElement' n) (v++[x]) = x.
```

That finishes the automatic reflection for this specific prover. The encoding of $((x + y) + (x + z))$ can now be automatically produced with `reflectNat [] ((x+y) + (x+z))`.

## 3.6 Summary

In this chapter, we've developed a very specific mechanism for automatically proving equalities over natural numbers, with universally quantified natural numbers and the addition. We've implemented this "tactic" in a *correct-by-construction* way, which means that no additional proofs are required after the definition of the functions : we produce the normal form and the proof at the same time, and the generation of the proof follows exactly the generation of the normalised term. The proof is somehow guided by the computation, and this way to build proofs is particularly well suited for programming languages with dependent types, as Idris, that are not equipped like proof assistants. We've also used a new kind of *type-safe reflection* mechanism, with embedded concrete values as index of dependent types. This index has played a central role in the construction of the proof.

We have also shown how we can use Idris reflection mechanism to build an automatic way of encoding the original Idris expressions into reflected terms, thus avoiding the need of building these encodings manually.

However, all this work was very specific : this prover can only be used for proving equalities over `Nat`, and only uses the two properties of associativity and of neutral element. There are other properties of interest (commutativity, distributivity of an operator on another one, etc) that appear in `Nat` and in various other datatypes. What we want now, is to deal with all these properties, for all these potential datatypes. The next chapter, which is the heart of this thesis, will develop such a generic collection of provers.

# Chapter 4

# Equivalences in Algebraic Structures

*Mathematics is the art of giving the same name to different things.*

— H. Poincaré

## 4.1 Generalising the problem

In the previous chapter, we've produced a tactic that proves equalities (on `Nat`), with the only two properties of associativity for $+$ and the property of simplification with the neutral element `Z`. Now that we have introduced the key ideas of our technique on this small example, and especially the type-safe reflection mechanism and the correct-by-construction approach, we will apply them for our more general main goal : proving equalities for many types and not only for `Nat`, and for various properties. Very often, the properties available on a given type are the ones of a well known algebraic structure : semi-groups, monoids, groups, rings, etc. An algebraic structure, in abstract algebra, is a set (called the carrier set, or the underlying set), equipped with operations and constants and that satisfy certain axioms. Thus, an algebraic structure is an abstraction, that can be instantiated, and where all the instances share some essential properties. The operations of these structures are often called sum and product, because the first and most natural instantiations of these structures are the set of numbers $\mathbb{N}$ and $\mathbb{R}$ equipped with their natural sums and products, but some instances can have nothing to do with numbers. These algebraic structures have been studied intensively, and many formal system fall in one of these structures. We can cite, for example, the natural numbers, their addition and the element 0 that form a monoid, refereed to as $(\mathbb{N},0,+)$. The requirement for being a monoid is that the addition $+$ must be associative, and that there must be a constant, often called 0, that must be the identity element of $+$. The natural numbers equipped with the standard addition and the constant 0 satisfy these conditions, and form therefore a monoid. The relative numbers, their addition, the unary negation, and the element 0[1] form a group. A group is a monoid (i.e. it must also satisfy all the axioms of a monoid) with inverses elements, i.e. with an element $-x$ for every element $x$, such that $x + (-x) = (-x) + x = 0$. The relative numbers form therefore a group, which happens to be commutative, because we also have the property that $x + y = y + x$ for all $x$ and $y$. If we add the multiplication, we obtain a ring, with the element one playing the role of multiplicative neutral element (also called multiplicative identity element). The exact axioms of a ring and all the other structures will be described in the next subsection, with the diagram 4.2. We see here that there are more sophisticated structures, like group, that inherit from more primitive structures (monoid here), which means that the most sophisticated structure has all the requirements of the most primitive one, plus a few extra ones.

Algebraic structures do not only describe numbers, and many other datatypes (or sets for mathematicians) are instances of these structures. For instance, lists with concatenation form a monoid, as the concatenation is associative (i.e. `∀ l1 l2 l3, l1++(l2++l2) = (l1++l2)++l3`) and the empty list `Nil` plays the role of a neutral element for the concatenation (i.e. `∀ l, l++Nil = Nil++l = l`). We can also cite the

[1] In ordinary maths there is no explicit notion of type, and therefore the element zero of the natural numbers is the same object as the element zero of relative numbers. However, in type-theory that is not the case, and these two objects are in fact distinct entities.

ring $M_n(R)$ of matrix of size $n * n$ over some ring R. What this means is that many common objects, like natural numbers, relative numbers, lists, and matrix have the same properties, or more precisely, a few of the same properties. These properties are the existence of neutral elements, the associativity or the commutativity of an operation, the existence of inverse elements, the distributivity of an operation (playing the role of a multiplication) over another one (playing the role of an addition), etc. Therefore, with some abstraction, we can solve the problem of generating proof of equality for many datatypes like natural numbers, lists, matrix, and many others *all at once* by implementing a very generic prover that works for any type that is an instance of an algebraic structure. In fact, that will be a hierarchy of provers –with one prover for each algebraic structure– that can prove equalities of terms in any instance of the corresponding structure. We will no longer only work with the properties of associativity and of right neutral element for natural numbers as it was the case in the previous chapter, but we will have available the properties of a given algebraic structure. These tactics will be usable on any type that satisfies these properties. The properties of a given algebraic algebraic structure will be expressed in a corresponding interface. This interface will extend the interface from which it naturally inherits (see the diagram4.2 for the complete hierarchy). For example, the Group interface will extend the Monoid one. We will end up with a hierarchy of interfaces and will write one tactic for each of these interface. All the tactics that we will implement (the ring prover, the group prover, etc) will be able to work on any type, being given that an implementation of the corresponding interface is provided.

## 4.2 Proving equivalences instead of equalities

Before developing a collection of tactics for proving equalities, we can realise that with some reasonable additional effort, we could produce a collection of tactics for proving equivalences. Therefore, why only prove equality when we could prove equivalence, which is more general?

The machinery would be very similar, and if we do the right abstraction from the start, we could gain one more degree of genericity, with the freedom of choosing the equivalence relation (which can of course be the usual equality). This is needed because Idris does not allow higher inductive types (HITs) that would allow the user to define constructors for equalities at the same time that the elements are being defined, thus extending the equality [3].

We want to give the user the possibility to use his own equivalence relation, as long as he is able to provide the properties of the algebraic structure he wants to use. Let's call `c` the *carrier* type, i.e. the type on which we want to prove equivalences. The equivalence relation on $c$ has the following profile `(≃) :  c → c → Type`[2], and has to be accompanied by the usual properties of reflexivity, symmetry and transitivity.

All of our tactics will require having a way of testing this equivalence between elements of the underlying set, that is to say, a way to test equivalence between constants. For this reason, we define a notion of `Set`[3], which only requires the definition of the equivalence relation – accompanied with the proofs that this is indeed an equivalence relation– and this equivalence test `set_eq`. All the algebraic structures will later extend this interface :

```
interface Set c where
    (≃) : c → c → Type
    refl : (x:c) → x ≃ x
    sym : {x:c} → {y:c} → (x ≃ y) → (y ≃ x)
    trans : {x:c} → {y:c} → {z:c}
            → (x ≃ y) → (y ≃ z) → (x ≃ z)
    set_eq : (x:c) → (y:c) → Maybe (x ≃ y)
```

Figure 4.1: Set

[2] This `Type` would be a `Prop` in systems –like Coq– that make a distinction between the world of computations and the world of logical statements.

[3] This notion of Set isn't a formalisation of sets and their elements, but is only a way to talk about the carrier type and an equivalence relation, sometimes called Setoid.

If one wants to prove propositional equalities, then one will simply instantiate `(≃)` with the built-in `(=)` during the implementation of the `Set` instance. We need `set_eq` in order to decide if two constants of the domain $c$ are considered as equivalent. Note that it only needs to produce a proof when the two elements are equivalent, but it doesn't produce a proof of dis-equivalence when they are different – instead, it simply produces the value `Nothing`–. That's quite natural, since we want to generate proof of equivalence, and not to generate counter examples for proving dis-equivalence, which is another problem.

Obviously, there is no tactic associated to `Set`, since we have no operations and no properties associated to this structure. Therefore, equivalences in a `Set` are just the "syntactic equivalences", and they can simply be proven with `refl`[4].

As it was the case in chapter 3, the kernel of our machinery will be a function `reduce` that will be composed of multiple functions composed together, that each do a part of the normalisation and that generate the proof of equivalence between old and new concrete values for this part of the normalisation. That was the two functions `expr_l` and `l_expr` on the smaller problem presented in chapter 3. For each of these functions, these proof obligations will be expressed as meta-variables, and their proofs will have to be done by hand, precisely because we do not have any tactic to automate them at the moment, as this is precisely what we are building. These small proofs done by hand are in fact the *proofs of correctness* of our machinery. And that's the compositions of all these small static proofs, done dynamically during the evaluations of the *reduce* function, that will produce the desired proof of $x = y$.

Working with equivalences instead of equalities will bring one complication. These proofs of correctness that we will have to produce by hand will be slightly more complicated, as we won't be able to use Idris' "rewrite" command, that enables rewriting a subterm of a term by another one, provided that the two subterms are equal. More precisely,

[4] `refl` should not be confused with `Refl`, the only constructor of `(=)`. But of course, when `(≃)` is instantiated with the equality `(=)`, the instantiation of `refl` will be `Refl`. Therefore, `refl` of the interface `Set` is a generalisation of the most specialised `Refl` for `(=)`.

if we need to prove $P\ x$, and have a proof of equivalence $pr : x \simeq x'$, we can not use `rewrite pr` in order to transform the goal into $P\ x'$, as we would normally do if the available proof would be an equality $x = x'$. This is a classical problem of working within a setoid, and this problem can be slightly mitigated when the programming language offers a good support for rewriting terms in setoids. However, Idris isn't equipped with any automation for setoids, and everything will have to be done by hand. For this purpose, we will use the following lemma :

```
eq_preserves_eq : {c:Type} → (Set c) → (x:c) → (y:c)
  → (c1:c) → (c2:c) → (x ≃ c1) → (y ≃ c2)
  → (c1 ≃ c2) → (x ≃ y)
```

This lemma says that the equivalence preserves the equivalence, which means that in order to prove $x \simeq y$, we can prove the (hopefully) simpler problem $c1 \simeq c2$, provided that $x \simeq c1$ and that $y \simeq c2$. This lemma will be intensively used for building the proofs of equivalence (between the old and new indices) that are required by the development of our machinery.

*Proof.* We have $x \simeq c1$ and we have $c1 \simeq c2$, therefore we have $x \simeq c2$ by the use of the axiom of transitivity (of the interface `Set`).

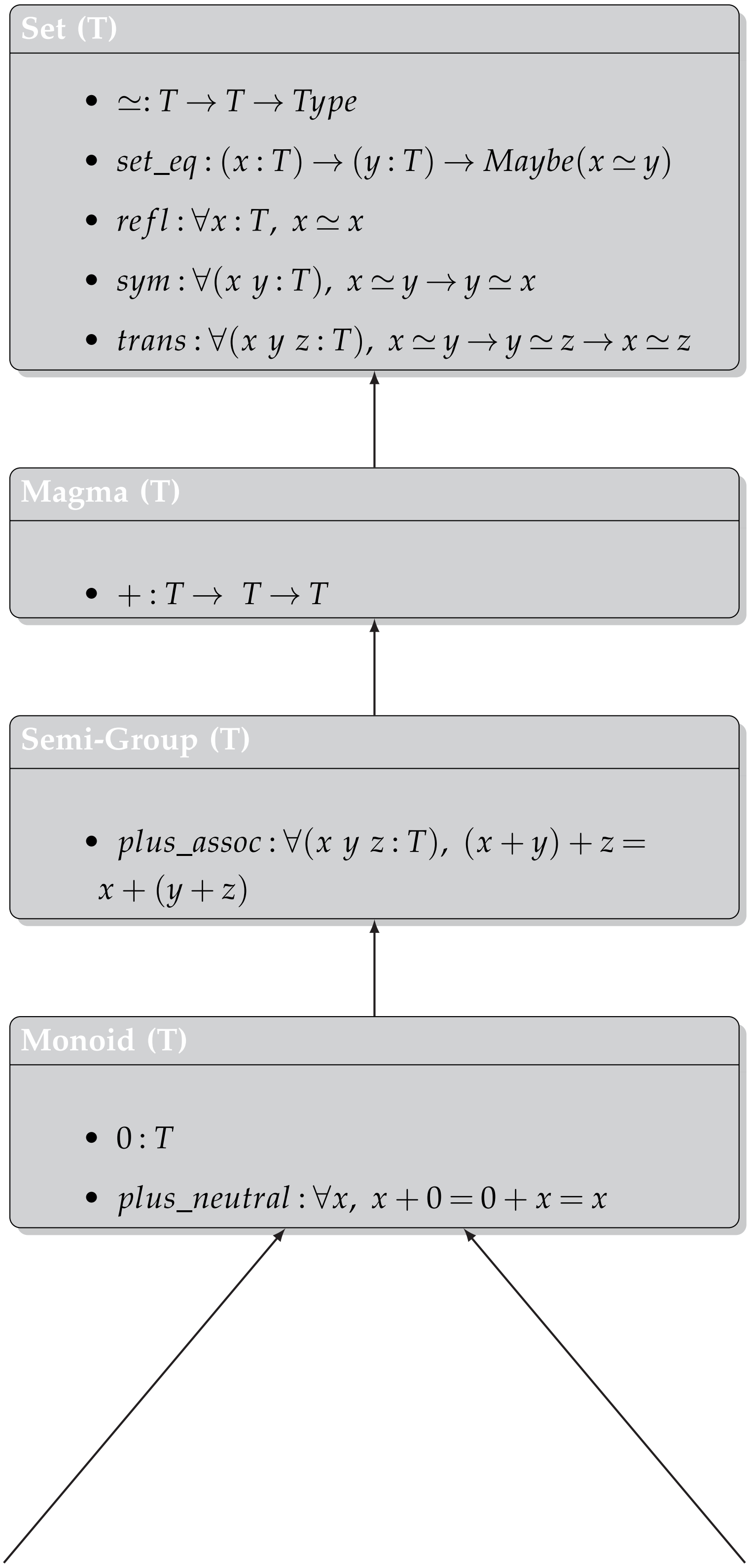
We also have $y \simeq c2$ and therefore we have $c2 \simeq y$ by the use of the symmetry axiom (of the interface `Set`).

Now that we have these fresh proofs of $x \simeq c2$ and of $c2 \simeq y$, we can use one last time the property of transitivity, in order to get a proof of $x \simeq y$. □

This lemma will be basically used as a replacement to the `rewrite` tactic that we can no longer use since we are dealing with equivalence and not equality.

## 4.3 The hierarchy

The complete hierarchy of structures that we will be dealing with is represented in the following diagram :



**Set (T)**

- $\simeq: T \to T \to Type$
- $set\_eq : (x : T) \to (y : T) \to Maybe(x \simeq y)$
- $refl : \forall x : T,\ x \simeq x$
- $sym : \forall (x\ y : T),\ x \simeq y \to y \simeq x$
- $trans : \forall (x\ y\ z : T),\ x \simeq y \to y \simeq z \to x \simeq z$

**Magma (T)**

- $+ : T \to\ T \to T$

**Semi-Group (T)**

- $plus\_assoc : \forall (x\ y\ z : T),\ (x + y) + z = x + (y + z)$

**Monoid (T)**

- $0 : T$
- $plus\_neutral : \forall x,\ x + 0 = 0 + x = x$

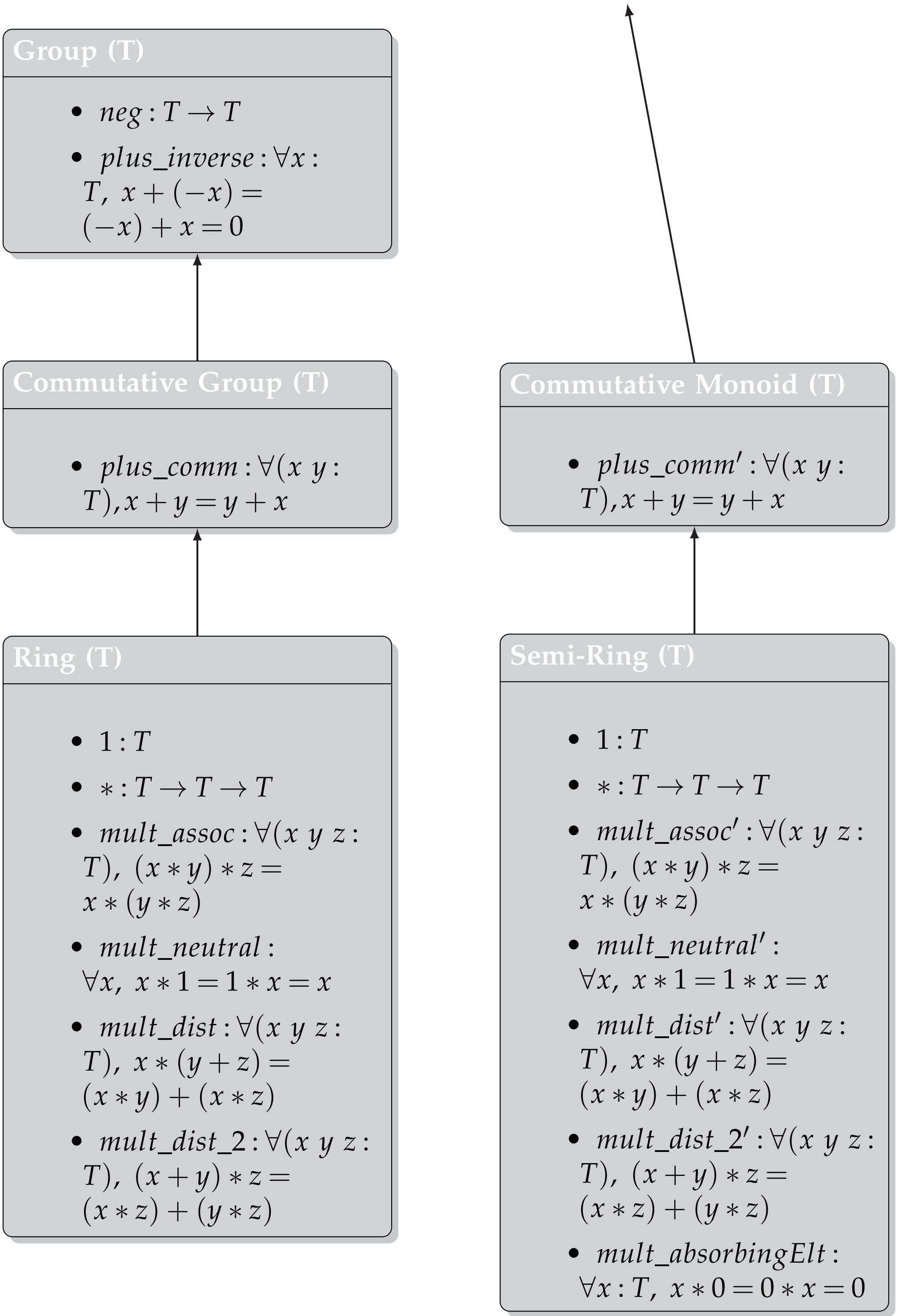

Group (T)
$neg : T \to T$
$plus\_inverse : \forall x : T,\ x + (-x) = (-x) + x = 0$
Commutative Group (T)
$plus\_comm : \forall (x\ y : T), x + y = y + x$
Commutative Monoid (T)
$plus\_comm' : \forall (x\ y : T), x + y = y + x$
Ring (T)
$1 : T$
$* : T \to T \to T$
$mult\_assoc : \forall (x\ y\ z : T),\ (x * y) * z = x * (y * z)$
$mult\_neutral : \forall x,\ x * 1 = 1 * x = x$
$mult\_dist : \forall (x\ y\ z : T),\ x * (y + z) = (x * y) + (x * z)$
$mult\_dist\_2 : \forall (x\ y\ z : T),\ (x + y) * z = (x * z) + (y * z)$
Semi-Ring (T)
$1 : T$
$* : T \to T \to T$
$mult\_assoc' : \forall (x\ y\ z : T),\ (x * y) * z = x * (y * z)$
$mult\_neutral' : \forall x,\ x * 1 = 1 * x = x$
$mult\_dist' : \forall (x\ y\ z : T),\ x * (y + z) = (x * y) + (x * z)$
$mult\_dist\_2' : \forall (x\ y\ z : T),\ (x + y) * z = (x * z) + (y * z)$
$mult\_absorbingElt : \forall x : T,\ x * 0 = 0 * x = 0$

Figure 4.2: Diagram of algebraic structures

At the root of the hierarchy, there is the Set structure with its equivalence relation $\simeq$ and the equivalence test `set_eq`. Then we find the `Magma`, `SemiGroup` and `Monoid` structures. The first one adds a $+$ operation and no properties about it, while the second one adds the property of associativity, and the third one further adds the property of neutral element for a distinguished element called 0.

Then, two structures inherit from `Monoid`. The most traditional one is the `Group` structure, which basically adds inverse elements, and from which the `CommutativeGroup` structure is built (also called Abelian group). Another one, less known, is the `CommutativeMonoid` structure (also called less commonly Abelian monoid), that directly inherits from `Monoid`, and which therefore is not equipped with inverse elements.

Now, if we extend `CommutativeGroup` with a product that is associative, distributive on the addition, and that admits a distinguished element called 1 as neutral element, we obtain the `Ring` structure. If we add the same things to the `CommutativeMonoid` structure, we obtain the intermediate structure of `PreSemiRing`, and if we extend the latter with the property[5] that 0 is an absorbing element for the product (sometimes called annihilator element), we obtain what is called a `SemiRing`. Both `Ring` and `SemiRing` structures can be extended with the commutativity of the product, which leads to `CommutativeRings` and `CommutativeSemiRings`.

The bottom of the hierarchy has two branches, that both inherits from the same structure (Monoid). This divergence comes from the fact that the left branch is equipped with negation and inverse elements (from the Group structure), while the right branch has skipped this structure, and directly inherits from Monoid. Many concrete structures are not equipped with negation and inverse elements, like natural numbers, lists, etc. Therefore leaving these operations and properties as optional is important for the generality of the provers. The `CommutativeMonoid` and `SemiRing` structures are very useful for advanced structures that have a commutative addition, or that are equipped with a product, but that do not have a notion of negation and inverse elements.

[5] This axiom isn't needed for the Ring structure because it follows from the other axioms.

### 4.3.1 Hierarchy of interfaces

We will develop a hierarchy of interfaces for all these algebraic structures. Each interface will contain the axioms of the corresponding abstract structure. At the usage, in order to invoke, for example, the group prover on a specific type T, we will have to provide an implementation of the interface Group T, which means that we will have to prove that T verifies all the properties to be a group.

The first structure, completely trivial, is the magma. A magma is a structure built on top of Set, that just adds a Plus operation, and no specific properties about it.

```
interface Set c => Magma c where
    Plus : c → c → c
```

Figure 4.3: Magma

This code means that a type c (for "carrier") is a Magma if it is already a Set (i.e. it is equipped with the equivalence relation $\simeq$ and the test set_eq), and if it has a Plus operation. In fact, there is an additional requirement that will apply for all operators (in this case, the Plus operation), which is that they need to be compatible with the equivalence relation, which is expressed by the following axiom for Plus :

```
Plus_preserves_equiv : {c1:c} → {c2:c}
    → {c1':c} → {c2':c} → (c1 ≃ c1') → (c2 ≃ c2')
    → ((Plus c1 c2) ≃ (Plus c1' c2'))
```

This requirement comes from the fact that we're dealing with arbitrary equivalence relation. The user is free to use his own equivalence relation, but it should be compatible with the operations that he is using. Said differently, the equivalence should be preserved by the Plus operation, and the user of our tactics will have to prove it. These requirements will be omitted from the interfaces that are showed in this text. As it was the case for Set, there is no tactic to write for Magma, because there is no property at all : all equivalences are again syntactic equivalences, and they can all be proven by refl.

A bit more interesting is the SemiGroup interface. A SemiGroup is a Magma (i.e. it still has a Plus operation), but moreover it has the property of associativity for this operation. We will have to write a prover for semi-groups, as the equivalence found in a semi-group are not only the syntactic ones, precisely because of the associativity property.

```
interface Magma c => SemiGroup c where
    Plus_assoc : (c1:c) → (c2:c) → (c3:c)
               → (Plus (Plus c1 c2) c3
                 ≃ Plus c1 (Plus c2 c3))
```

Figure 4.4: Semi-Group

A bit more sophisticated is the Monoid structure, which is a SemiGroup with the property of neutral element for a distinguished element called Zero.

```
interface SemiGroup c => Monoid c where
    Zero : c
    Plus_neutral_1 : (c1:c) → (Plus Zero c1 ≃ c1)
    Plus_neutral_2 : (c1:c) → (Plus c1 Zero ≃ c1)
```

Figure 4.5: Monoid

We continue the hierarchy with the more interesting structure of Group. A Group is a Monoid, with a new operation : the unary operation Neg. However, in order to give the user the flexibility of allowing the binary Minus, we also add this binary operator to the Group interface. We must have the property that Minus can always be simplified with the negation and the addition. That means that Minus is not a "primitive" operation of a group, since we can always rewrite $(a - b)$ into $(a + (-b))$. Letting the user using subtractions $(a - b)$ in the left and right sides of the equality he wishes to prove increases the usability of the prover, as he won't have to rewrite by hand the Minus operations by sums of negations before invoking the prover. Note that if he does not have a Minus operation in his concrete structure, he can define it to match directly the required property : $(a - b) \ := \ (a \ + \ (-b))$ when providing an implementation of the interface.

The second and main axiom of Group is the fact that any value c1 admits (Neg c1) as symmetric element (also called additive inverse), which means[6] that $c1 + (\texttt{Neg}\ c1) = 0 \wedge (\texttt{Neg}\ c1) + c1 = 0$. This notion of symmetric element can be expressed in Idris by the following definition :

```
-- This is a conjunctive predicate
hasSymmetric : (c:Type) → (p:Monoid c) → c → c → Type
hasSymmetric c p a b = (Plus a b ≃ Zero, Plus b a ≃ Zero)

interface Monoid c => Group c where
    Minus : c → c → c
    Neg : c → c
    Minus_simpl : (c1:c) → (c2:c)
                → Minus c1 c2 ≃ Plus c1 (Neg c2)
    Plus_inverse : (c1:c) → hasSymmetric c _ c1 (Neg c1)
```

Figure 4.6: Group

A Commutative Group is a group with the extra property of commutativity :

```
interface Group c => CommutativeGroup c where
    Plus_comm : (c1:c) → (c2:c)
                → ((Plus c1 c2) ≃ (Plus c2 c1))
```

Figure 4.7: Commutative Group

A Ring is a CommutativeGroup with a Mult operation (that preserves the equivalence), a distinguished element 1 that is a neutral ele-

[6] Note that we need both part of the conjunction, because Plus is not necessary commutative.

ment of Mult, together with the fact that Mult must be associative and distributive over Plus :

```
interface CommutativeGroup c => Ring c where
    One : c
    Mult : c → c -> c

    Mult_assoc : (c1:c) → (c2:c) → (c3:c)
        → (Mult (Mult c1 c2) c3)
          ≃ (Mult c1 (Mult c2 c3))
    Mult_dist : (c1:c) → (c2:c) → (c3:c)
        → (Mult c1 (Plus c2 c3))
          ≃ (Plus (Mult c1 c2) (Mult c1 c3))
    -- Needed because * is not necessary commutative
    Mult_dist_2 : (c1:c) → (c2:c) → (c3:c)
        → (Mult (Plus c1 c2) c3)
          ≃ (Plus (Mult c1 c3) (Mult c2 c3))
    Mult_neutral : (c1:c)
        → ((Mult c1 One) ≃ c1, (Mult One c1) ≃ c1)
```

Figure 4.8: Ring

We have not shown the interfaces for CommutativeMonoid and SemiRing because they are very similar to the CommutativeGroup and Ring interfaces.

All these interfaces will be used as predicates which classify types. In order to call a prover on a type c, the user of the system will have to satisfy the requirements expressed in the corresponding interface by providing an implementation of it for the type c, i.e. he will have to prove that the corresponding properties effectively hold for this type c. Note that the properties will either be obtained by implementation if the operations are real –computable– functions, or by axioms if the user is working within an axiomatised theory where the operations (Plus, Neg, etc) are defined as axioms.

As discussed in the section 3.1 on the smaller problem, the algorithm of normalisation will not directly use the concrete values of type c, but reflected terms instead, indexed over the concrete values. That still holds here.

### 4.3.2 Reflected terms

We need to define a datatype for reflecting terms in each algebraic structure. Each of these datatype is parametrised over a type `c`, which is the real type on which we want to prove equalities (the carrier type). It is also indexed over an implementation of the corresponding interface for `c` (we usually call it `p`, because it behaves as a proof telling that the structure `c` has the desired properties), and indexed over a context (a vector *Γ* of `n` elements of type `c`), and also indexed over a value of type `c`, which is precisely the concrete value being encoded. A magma is only equipped with one operation `Plus`. Thus, we only have three concepts to express in order to reflect terms in a `Magma` : constants, variables, and additions.

```
data ExprMa : Magma c → (Vect n c) → c → Type where
    ConstMa : (p : Magma c) → (Γ:Vect n c)
         → (c1:c)  → ExprMa p Γ c1
    PlusMa : {p : Magma c} → {Γ:Vect n c}
         → {c1:c} → {c2:c}
         → ExprMa p Γ c1 → ExprMa p Γ c2
         → ExprMa p Γ (Plus c1 c2)
    VarMa : (p:Magma c) → {Γ:Vect n c}
         → (i:Fin n) → ExprMa p Γ (index i Γ)
```

Figure 4.9: Reflected terms in a Magma

When we encode a constant `c1` in a context *Γ*, we use the constructor `ConstMa` to produce a term of type `ExprMa p Γ c1` : the index representing the concrete value is precisely this constant `c1`. If `e1` is an expression of type `ExprMa p Γ c1` (i.e. a term encoding the value `c1`), and `e2` is an expression of type `ExprMa p Γ c2` (i.e. a term encoding the value `c2`), then the term `PlusMa e1 e2` will have the type `ExprMa p Γ (Plus c1 c2)`, i.e. this term will encode the value `(Plus c1 c2)`, where `Plus` is the operation available in the implementation `p` of the `Magma` interface. Because the reflected terms embed their corresponding inputs, they are guaranteed to be faithful representations.

There are no additional operations in SemiGroup and Monoid, so the datatypes that reflect terms in these two structures will have exactly the same shape as the one for Magma that we have given above. However, the one for Group will introduce two new constructors for the Neg and Minus operations :

```
data ExprG :  Group c → (Vect n c) → c → Type where
    ConstG : (p : Group c) → (Γ:Vect n c)
         → (c1:c) → ExprG p Γ c1
    PlusG : {p : Group c} → {Γ:Vect n c}
         → {c1:c} → {c2:c}
         → ExprG p Γ c1 → ExprG p Γ c2
         → ExprG p Γ (Plus c1 c2)
    MinusG : {p : Group c} → {Γ:Vect n c}
         → {c1:c} → {c2:c}
         → ExprG p Γ c1 → ExprG p Γ c2
         → ExprG p Γ (Minus c1 c2)
    NegG : {p : Group c} → {Γ:Vect n c} → {c1:c}
         → ExprG p Γ c1 → ExprG p Γ (Neg c1)
    VarG : (p : Group c) → {Γ:Vect n c}
         → (i:Fin n) → ExprG p Γ (index i Γ)
```

Figure 4.10: Reflected terms in a Group

The index of type c (the real value encoded by an expression) is always expressed by using the lookup function index and the available operations in the implementation p, which for a group are Plus, Minus and Neg.

Again, because there is no additional operations in CommutativeGroup and CommutativeMonoid, the datatypes that reflect terms in these two structures will have exactly the same shape as the one for Group that we have showed above. However, the Ring structure has to have an additional construction for the Mult operation :

```
data ExprR : Ring c → (Vect n c) → c → Type where
    ConstR :(p:Ring c) → (Γ:Vect n c) → (c1:c)
         → ExprR p Γ c1
    PlusR : {p:Ring c} → {Γ:Vect n c}
         → {c1:c} → {c2:c}
         → ExprR p Γ c1 → ExprR p Γ c2
         → ExprR p Γ (Plus c1 c2)
    MultR : {p:Ring c} → {Γ:Vect n c}
         → {c1:c} → {c2:c}
         → ExprR p Γ c1 → ExprR p Γ c2
         → ExprR p Γ (Mult c1 c2)
    MinusR: {p:Ring c} → {Γ:Vect n c}
         → {c1:c} → {c2:c}
         → ExprR p Γ c1 → ExprR p Γ c2
         → ExprR p Γ (Minus c1 c2)
    NegR : {p:Ring c} → {Γ:Vect n c} → {c1:c}
         → ExprR p Γ c1 -> ExprR p Γ (Neg c1)
    VarR : (p:Ring c) → {Γ:Vect n c}
         → (i:Fin n) → ExprR p Γ (index i Γ)
```

Figure 4.11: Reflected terms in a Ring

### 4.3.3 A bit of notation

The equivalence we are trying to prove is `x ≃ y`, where `x` and `y` are terms (potentially open) of the type `c` in a potentially non empty context (due to variables that were universally quantified). The type `c` simulates a set with some properties (making it a `Monoid`, or a `Group`, etc). The fact that `c` fulfils the specification of an algebraic structure is expressed as an implementation of the corresponding interface, and this implementation will be denoted as `p`. The reflected term for `x` will be denoted `ex`, and this term will have the type `ExprG p Γ x`. The term `ex` is the encoding of `x` and its type is precisely indexed over the real value `x`, as the reflected terms embed the concrete values. We've got similar things for `y`, which is encoded by `ey`, and its type is indexed over the real value `y`. Running the normalisation procedure on `ex` will produce the normal form `ex'` of type `ExprG p Γ x'` and a proof `px` of `x ≃ x'` while running the normalisation procedure on `ey` will produce the normal form `ey'` of type `ExprG p Γ y'` and a proof `py` of `y ≃ y'`.

## 4.4 Deciding equivalence

Each algebraic structure will have a function for reducing the reflected terms into their normal forms. The fact that all of these algebraic structures admit a canonical representation for any element is in fact a very nice property that we are using in order to decide equivalences. Without this property, it would become a lot more complicated to decide the equivalence of two terms without brute-forcing a series of rewriting, that might even not terminate. These normalisation functions will compute the canonical representation of the reflected input term, by rewriting the given term multiple times, and at the same time, it will produce the proof of equality between the original real value (indexing the input term) and the produced real value (indexing the produced term). Thus, the type of the reduction function for group is :

```
groupReduce : {c:Type} → {n:Nat} → (p:Group c)
            → {Γ:Vect n c} → {x:c} → (ExprG p Γ x)
            → (x' ** (ExprG p Γ x', x ≃ x'))
```

This function has more work to do in structures with multiple axioms (like `Group`), than for the simpler ones (like `Monoid`). The details of these functions will be given in the next section. For the moment, we just assume that we've got such a function of normalisation for each structure. We will give details about the implementation for the `Group` structure, but the principle is exactly the same with the other structures.

We want to write the following function :

```
groupDecideEq : (p:Group c) → {Γ:Vect n c} → {x : c}
          → {y : c} → (ExprG p Γ x) → (ExprG p Γ y)
          → Maybe (x ≃ y)
```

The first thing this function has to do is to compute the normal form of the two expressions `ex` and `ey` respectively encoding `x` and `y`. Then, the only thing remaining to do will be to syntactically compare `ex'` and ey', and if they are equal then that will give us $x' = y'$, and we will be able to produce the desired proof of $x \simeq y$ by using the two proofs $px : x \simeq x'$ and $py : y \simeq y'$.

```
groupDecideEq p ex ey =
  let (x' ** (ex', px)) = groupReduce p ex in
  let (y' ** (ey', py)) = groupReduce p ey in
             buildProofGroup p ex' ey' px py
```

The syntactic test of equality between ex' and ey' and the composition of the two proofs is done in the auxiliary function buildProofGroup, similarly to what we've done for the small tactic on Nat in the previous section.

```
buildProofGroup : (p:Group c) → {Γ:Vect n c}
  → {x : c} → {y : c} → {x':c} → {y':c}
  → (ExprG p Γ x') → (ExprG p Γ y')
  → (x ≃ x') → (y ≃ y')
  → (Maybe (x ≃ y))
buildProofGroup p ex' ey' px py with (exprG_eq p ex' ey')
    buildProofGroup p ex' ex' px py | Just refl
                                     = ?MbuildProofGroup
    buildProofGroup p ex' ey' px py | Nothing = Nothing
```

We have used here a function exprG_eq to decide the syntactic equality between the normalised reflected terms ex' and ey'. This function plays the same role as eqExpr, presented on section 3.3 for the small prover on Nat. Also, the proof of the metavariable MbuildProofGroup corresponds exactly to the proof done for the metavariable MbuildProof.

## 4.5 Automatic reflection

As we did for the specific problem in chapter 3, we want to program an automatic reflection mechanism, in order to let the machine build automatically the encodings for us. This time, we want to write a function able to compute encodings for any type that behaves as one of our algebraic structure. We will show here the details for Ring, which is one of the most sophisticated structures. So, we've got a carrier type `(c:Type)`, an implementation `(p:Ring c)` which says that `c` behaves as a ring, a context $\Gamma$ of already abstracted variables, and an element `(x:c)` to encode, and we want to return an `ExprR p` $\Gamma$ `x`. We are doing this reflection for any type `c`, so we don't know which constants are inhabiting this type. For this reason, we will need to expect an extra argument `funReflectCst` of type `(ConstantsRingReflector c p)` which is a function able to do the reflection for constants only. This function will be provided for various datatypes, like `Nat`, `Z` and `List`. But if the user wants to use the reflection mechanism for a new datatype of his own, then he will only have to provide this function of encoding for the constants of his type. He won't have to rebuild the entire reflection mechanism. A `ConstantsRingReflector` for a given type `c` and a given implementation `p` is a function that takes any context $\Gamma$, any element `c1` of `c`, and which tries to produce an `(ExprR p` $\Gamma$ `c1)`.

```
ConstantsRingReflector : {c:Type} → (p:Ring c) → Type
ConstantsRingReflector {c=c} p =
    ({n:Nat} → (Γ:Vect n c) → (c1:c)
      → Maybe (ExprR p Γ c1))
```

If `c1` is indeed a constant, then it is supposed to return `Just` and a term reflecting this constant. If it was something else (a variable, a sum $a + b$, a product $a * b$, ...) then it simply returns `Nothing`.

We can now present the type of the reflection function for `Ring`.

```
reflectRingTerm : {c:Type} → {p:Ring c}
  → {n:Nat} → (Γ : Vect n c)
  → (funReflectCst:ConstantsRingReflector {c=c} p)
  → (x:c)
  → (n' ** (Γ':Vect n' c
         ** (ExprR {c=c} {n=n+n'} p (Γ ++ Γ') x)))
```

This function will work very similarly to the reflection function for `Nat` that we've written in section 3.5. Here, there are simply more constructions to deal with, because `c` is also equipped with negations, subtractions and multiplications, and not only with additions. We show one of these new pattern on figure 4.12.

```
reflectRingTerm {p} {n=n} Γ funReflectCst (a*b) =
   let (n′ ** (Γ′ ** a′))
     = reflectRingTerm Γ funReflectCst a in
   let (n′′ ** (Γ″ ** b′))
     = reflectRingTerm (Γ ++ Γ′) funReflectCst b in
   let this = MultR (weakenR Γ″ a′) b′ in
     ((n′ + n′′) ** ((Γ′++Γ″) **
       (convertVectInExprR (plusAssociative n n′ n′′)
           (vectAppendAssociative Γ Γ′ Γ″) this)))
```

Figure 4.12: Automatic reflection - case of a multiplication

We have to do the same for additions, negations and subtractions (that we omit here), and then the final case to consider is for constants and variable. Since we've got the function `funReflectCst` –which tries to do the reflection for constants only–, we can easily finish the definition of `reflectRingTerm`. If `funReflectCst` returns something, then we know the input is a constant, and we got its encoding. Otherwise, we just have to treat it as a variable.

```
reflectRingTerm {p} {n=n} Γ funReflectCst t =
  case funReflectCst Γ t of
    -- funReflectCst decides that ′t′ is a constant
    Just this => Z ** ([] ** this)
    -- Otherwise it should be considered as a variable
    Nothing => case (isElement t Γ) of
      Just (i ** pr) =>
          let res = VarR p (RealVariable i) in
              (Z ** ([ ] ** res))
      Nothing =>
        let res = VarR p
                  (RealVariable (lastElement′ n)) in
            (S Z ** ([t] ** res))
```

Figure 4.13: Automatic reflection - case of constants and variables

We have intentionally omitted showing a few conversions of type in order to keep this part of the definition more readable.

## 4.6 Normalisations functions and re-usability of the provers

The only thing left to describe about the implementation is the normalisation functions for the various algebraic structures. We will also explain how we're reusing the provers, with the most specialised structures that inherit from the simplest ones. What we describe in this section is the construction of all the normalisation functions (`monoidReduce`, `groupReduce`, etc) that we've skipped in section 4.4. These functions take an expression in input, and compute the normal form of the given expression and the proof of equivalence between the underlying concrete values.

The construction of the desired proof is always done bit by bit, because every rewriting of the reflected term is immediately justified by a simple accompanying proof which shows the equivalence of the previous and new index. This is the *correct-by-construction* style in which we will write the normalisation.

```
smallRewriting : (p:algebraic structure on c)
   → (Γ:Vect n c) → {c1:c}
   → (Expr p Γ c1) → (c2 ** (Expr p Γ c2, c1 ≃ c2))
smallRewriting p Γ e = (new value **
                        (new expression, justification))
```

The idea will be to write each simplification as a small rewriting, and composing these small rewritings will lead to the computation of the normal form and to the proof of equivalence. Before going into the implementation details of these rewritings, we first need to decide a normal form shape.

### 4.6.1 Normal form shape

It is noticeable that all the traditional algebraic structures that we're dealing with admit normal forms : for every term `t`, there exists a term `(norm t)` such that, for any other term `t'` equivalent to `t`, their normal form will be the same, and reciprocally, when the normal form `norm t` and `norm t'` are the same, then the two original terms are equivalent. The existence of normal forms for algebraic structures is rarely (if not never) mentioned in the literature, but this is something crucial to us, because we precisely use this property for deciding the equivalence. More precisely, we will use the fact[7,8] that $\forall t\ t',\ norm\ (reflect\ t) \equiv norm\ (reflect\ t') \rightarrow t \simeq t'$ for building a proof of $t \simeq t'$ (when possible) after having computed $norm\ (reflect\ t)$ and $norm\ (reflect\ t')$ and having compared them with a syntactic equality test.

Let's describe the normal form shape first before describing the implementation of the normalisation functions. For the sake of completeness, let's consider the case of a ring, that has all the operations and all the properties about these operations. The normalisation function takes in input an expression expressed with sums, products, constants (zero, one...) and variables that belong to an ordered set $\mathcal{V}$ of variables. In short, the normalisation function takes in input a polynomial of multiple variables. In output, it must produce a normal form representing the same polynomial. Therefore, we have to decide a canonical representation of polynomials. Many canonical representations can be used. We decide to stick with classical mathematical conventions. The first one, is that the polynomial will be completely developed, i.e. the distributivity of $*$ over $+$ will be applied until it can't be applied any more. The advantage is the simplicity, as factorising would be significantly more complicated. Because the polynomial is completely developed, at the toplevel, it is a sum :

[7] which is in fact the correctness lemma of the machinery, see sections 4.7.1 and 4.8.1

[8] We will discuss in the subsection 4.7.2 that the normalisation function also needs to be complete for being useful, and completeness is the implication in the other direction.

$$P = \sum_{i=1}^{a} (\prod_{j=1}^{b} Monomial_i^j)$$

where

$$Monomial_i^j = C_i^j * \prod_{k=1}^{c} Var_{i,k}^j$$

with $C_i^j$ a constant, and $Var_{i,k}^j$ one of the variable that belong to $\mathcal{V}$.

We use the most common and most flexible definition of a *monomial*, where the product of variables is preceded by a constant[9] .

One could be surprised by the fact that it's a sum of product of monomials, and not directly a sum of monomials. The reason is the following. A monomial (for our normal form) is a product of a constant $C_i^j$ (like 4) and of a product of variables (like $x * y * z$). For example, $5 * (x * (y * z))$ is a monomial. Now let's consider the term $(5 * (x * (y * z))) * (4 * (z * z))$. This term is not a monomial, but we could be tempted to simplify it into the monomial $20 * (x * (y * (z * (z * z))))$. However, that would assume that the product is always commutative, i.e. that we can re-organize the subterms of a product as we want. This is the case in a commutative ring, but this does not hold for any ring. Therefore, after the full development, the polynomial is a sum of *product of monomials*, and not directly a sum of monomials. The only rearrangement that can be done with the multiplication is to check if two constants are consecutive in a product, and if so, to replace them by the constant that represents the product of them.

However, because $+$ is always a commutative operator in a ring, the different products of monomials themselves can be rearranged in different ways in this sum. That will be done at the level of the commutative group prover if we can provide an ordering for product of monomials.

We start by defining an order between monomials, and in a second time, we will use this order in order to build an order between product of monomials. We decide the following order between monomials,

[9] In maths, a monomial can refer to a product of variables or to a product of variables multiplied by a constant, called the coefficient of the monomial.

that we name `isBefore_mon`[10]. Given two monomials $Monomial_i^j$ and $Monomial_{i'}^{j'}$ we need to decide which one comes first, i.e. we need to decide which of these two generic terms comes first :

$$Monomial_i^j = C_i^j * \prod_{k=1}^{c} Var_{i,k}^j$$

and

$$Monomial_{i'}^{j'} = C_{i'}^{j'} * \prod_{k=1}^{c'} Var_{i',k}^{j'}$$

We will decide the ordering of these two product by looking at the variables, simply ignoring the constant, and so we need to unfold the first variable of the products :

$$Monomial_i^j = C_i^j * (Var_{i,1}^j * \prod_{k=2}^{c} Var_{i,k}^j)$$

and

$$Monomial_{i'}^{j'} = C_{i'}^{j'} * (Var_{i',1}^{j'} * \prod_{k=2}^{c'} Var_{i',k}^{j'})$$

The term $Var_{i,1}^j$ contains one of the variables of the ordered set $\mathcal{V}$. Let's call this variable $v_e$ where $e$ is the position of this variable in the ordered set : $Var_{i,1}^j = v_e$ with $v_e \in \mathcal{V}$. Now let's call $v_{e'}$ the variable denoted by $Var_{i',1}^{j'}$. We will decide which terms comes first by using the order on the variables, on these two variables. If $e<e'$ then we decide that $Monomial_i^j$ comes before $Monomial_{i'}^{j'}$. However, if $e>e'$, then we decide that $Monomial_{i'}^{j'}$ comes before $Monomial_i^j$. The last case is when $e = e'$, i.e. when both monomials start with the same variable, and in this case we continue by inspecting the remaining variables. If we can't continue because one of the two monomials has less variables than the other, then we decide that the one with fewer variables comes first (exactly like the word "house" comes before the word "housemate" with the lexicographic order).

[10] which can be found on line 71 of the file commutativeGroup_reduce.idr, where this function takes two monomials in input, and decides if the first comes before the second

We can now build the order on *product of monomials* that we needed. We name it `isBefore`[11]. Given two product of monomials $Prod_i$ and $Prod_{i'}$ we need to decide which one comes first, i.e. we need to decide which of these two generic terms comes first :

$$Prod_i = (\prod_{j=1}^{b} Monomial_i^j)$$

and

$$Prod_{i'} = (\prod_{j=1}^{b'} Monomial_{i'}^j)$$

We will obviously use the order `isBefore_mon` on the first monomials of these two products. If it says that $Monomial_i^1$ comes before $Monomial_{i'}^1$, then we decide that $Prod_i$ comes before $Prod_{i'}$. Conversely, if that's $Monomial_{i'}^1$ that comes first, then we decide that $Prod_{i'}$ comes first. However, if $Monomial_i^1$ and $Monomial_{i'}^1$ have exactly the same position[12] in the order, then we continue by inspecting the remaining monomials with a recursive call on `isBefore`. As previously, if we can't continue because one of the two products has less monomials than the other, then the one with fewer monomials will come first.

A few additional conventions had to be decided about the normal form :

- The top-level sum of the polynomial will be written in the completely right-associative form :
  $prodMon_1 + (prodMon_2 + (prodMon_3 + (... + prodMon_a)))$
- The two levels of product that we have, namely the products of monomials (at the main level) and the products of variables (at

[11] which can be found on line 96 of the file commutativeGroup_reduce.idr, where this function takes two product of monomials in input, and decides if the first comes before the second

[12] We have an auxiliary function for this task, called `samePosition_mon`, that decides if two monomials have exactly the same position in the order. Note that even two different monomials can have the same position in the order because we do not compare the constants. Comparing the constants would be a bad idea as it would add the unnecessary requirement that the user must provide an order between the constants of the underlying set.

the monomial level), will also be written on the completely right-associative form.

- We always simplify as much as possible with constants. That includes doing the simplification for the addition with zero and for the multiplication with one, doing the computation between two constants that are adjacent in a sum or in a product, etc.
- We always replace the binary operator `Minus` by a sum and a `Neg` ($a - b = a + (-b)$) since the binary operator `Minus` is just allowed for the user's convenience, but internally, we replace them all by their complete form.
- We always simplify the sum of an expression $e$ and its inverse $-e$ into the constant zero.

### 4.6.2 Computing the normal form

Now that we have decided a shape for our polynomials in normal form, we need to implement its computation, while keeping an eye on the simplicity because the normalisation does not only produce a normalised term, but it also produces a proof of equivalence between the original form and the new one, and this is in fact the most important part. Our normalisation function will do the simplifications that are required to do according to a Knuth-Bendix [26] completion, but it is important to recall that our goal here is not to implement a generic Knuth-Bendix completion for any set of rules. Instead, what we want is to implement an algorithm that computes the result of such a completion, for each of our algebraic structures.

A naive implementation would be monolithic, but we are building a hierarchy of tactics for various algebraic structures. There will be a prover for each structure, and we want to reuse each of them as much as possible for building the others. For example, the semi-group prover will reorganize sums into their right-associative form, and the monoid level –that specifically needs to simplify sums with the constant zero– will reuse the monoid level for producing the right-associative form. Thus, even if our attention is on the most complicated structure that we're dealing with (ring), we will pay attention to take each step of the normalisation independently and we will implement them separately at the appropriate level on the most general way. This way, each simplification will not only work for the level where it is implemented, but also for all the levels where this treatment is required. The next subsections will describe the organisation of all the simplifications needed at the different levels of the hierarchy.

### 4.6.3 Normalization of terms in semi-groups

In a semigroup, we only have to deal with the property of associativity. As it was the case with the toy prover for `Nat` in section 3, we will have to rearrange the parenthesis on a systematic way, either left associative, or right associative. We'll use the complete right associative form. However, this is not the only thing that we have to do for dealing completely with the property of associativity. Indeed, if $x$, $y$ and $z$ denote variables, we don't only want to transform $x + ((y + 4) + (5 + z))$ into $(((x + y) + 4) + 5) + z$. We also want the constants 4 and 5 to be simplified together, because they are close to each other (and we have the right to do so because we can rearrange the parenthesis thanks to the associativity), and the result should be $((x + y) + 9) + z$ on this example. Three possible patterns need this treatment : $(x + const1) + (const2 + y)$, $(x + const1) + const2$ and $const1 + (const2 + x)$ where $const1$ and $const2$ denote constants. The simplification between the constants $const1$ and $const2$ will be denoted as $constResult$, and this computation can be done at the underneath magma level, with its normalisation function called `magmaReduce`. All he have to do here is to rewrite the first pattern into $(x + constResult) + y$, the second into $(x + constResult)$ and the third one into $(constResult + x)$. We show the code for the first pattern :

```
assoc : (p:SemiGroup c) → (Γ:Vect n c) → {c1:c}
        → (ExprSG p Γ c1)
        → (c2 ** (ExprSG p Γ c2, c1 ≃ c2))
assoc p Γ (PlusSG
             (PlusSG ex (ConstSG _ _ const1))
             (PlusSG (ConstSG _ _ const2) ey)) =
  let (r_ihx ** (e_ihx, p_ihx)) = (assoc p Γ ex) in
  let (r_ihy ** (e_ihy, p_ihy)) = (assoc p Γ ey) in
  let (r_3 ** (e_3, p_3))
    = magmaReduce (semiGroup_to_magma
        (PlusSG (ConstSG _ _ const1)
                (ConstSG _ _ const2))) in
  let e_3' = magma_to_semiGroup p e_3 in
    (_ ** ((PlusSG (PlusSG e_ihx e_3') e_ihy),
          ?Massoc1))
```

Figure 4.14: Computing with associativity in a Semi-Group, first pattern

The treatment is applied recursively on the sub-expressions `ex` and `ey`, which produces the new expressions `e_ihx` and `e_ihy` together with the new concrete values they represent (`r_ihx` and `r_ihy`) and the proofs (`p_ihx` and `p_ihy`) justifying these transformation on the concrete values. Note that the computation of the constant representing $const1 + const2$ is done by calling the underneath magma prover, which gives a new expression `e3`, even thought it could perfectly be done directly here : we simply wanted to show how a prover can be reused for building another prover on this very simple example. We see that we need functions for converting terms between the different levels in order to reuse the underneath prover : that's the role of the functions `semiGroup_to_magma` and `magma_to_semiGroup`.
A meta-variable `Massoc1` has been introduced by this definition. This metavariable corresponds to a proof obligation for the following property : $Plus\ (Plus\ x\ const1)\ (Plus\ const2\ y)\ \simeq\ Plus\ (Plus\ r\_ihx\ r_3)\ r\_ihy$ in a context where there is, amongst other things, $p\_ihx : x\ \simeq\ r\_ihx$, $p\_ihy : y\ \simeq\ r\_ihy$ and $p_3 : Plus\ const1\ const2 \simeq r\_3$. The left hand side of the equivalence to prove is the expected concrete value, and the right-hand side is the concrete value produced by the computation that we have

done. This lemma says that the treatment done by this function `assoc` on this pattern was sound because the previous and the new concrete values are equivalent. It can easily be proven with a few rewritings and the use of the associativity, available from the implementation `p` of the semi-group interface.

After this simplification of constants computable thanks to associativity, the normalization for semi-group also has to rearrange the brackets on a systematic way. The fully right or fully left associative forms can both be chosen for this purpose, and the implementation is very similar to the function `expr_l` previously described for the smaller example on section 3.3

### 4.6.4 From a semigroup prover to a monoid prover

In a monoid, we simply have to eliminate zeros, thanks to the two properties of left and right neutrality. The simplification of additions between a zero and another constant has already be done at the level of semigroup because nearby constants have been simplified, so we only have to simplify the addition between the constant zero and a variable. There are in fact two cases, since the variable can come first, or the zero can come first. We can write a function `elimZero` which rewrites these two patterns `x+Z` and `Z+x` into `x`.

The normalization of reflected expressions in a monoid is simply made of a call to the function reducing terms on a semigroup, followed by a call to elimZero that eliminates the remaining additions with zeros.

### 4.6.5 From a monoid prover to a group prover

The first thing that the reduction function of the group prover has to do is to transform every subtraction $(a - b)$ into $(a + (-b))$, and we are entitled to do so precisely because we have available the property `Minus_simpl` contained in the implementation of the interface. After that, the reduction continues with the propagation of the `Neg` operation inside the parenthesis and $-(a + b)$ is transformed into $(-b) + (-a)$.

Note that we have to be careful and not simplify it to $(-a) + (-b)$ as it would assume that we're having a commutative monoid. Then, because we've pushed some negations inside the parenthesis, we might have sequences of two or more consecutive negations. We simplify them by removing two consecutive negations every time that we find such a sequence. Once this is done, there is a last major step specific to groups to accomplish, which is the simplification of sums of symmetric elements.

This might be a direct sum of symmetric elements, like $-e1 + e2$ or $e1 + -e2$ and in these two cases, we need to simplify them when $e1$ can be reduced[13] to $e2$ , and if so the result should be $Zero$. We are entitled to do this treatment because of the property `Plus_inverse`.

The sum to simplify might be more complex, with two or three levels of $Plus$. In these cases, there might be some simplifications to do thanks to the associativity property. With two levels of sums, we will transform $(e1 + (-e2 + e3))$ and $(-e1 + (e2 + e3))$ into $e3$ when $e1$ can be reduced to $e2$. Also, $((e1 + e2) + -e3))$ and $((e1 + (-e2)) + e3)$ are simplified into $e1$ when $e2$ can be reduced to $e3$. Finally, in presence of three level of sums, we will transform $(a + b) + ((-c) + d)$ and $(a + (-b)) + (c + d)$ into $a + d$ when $b$ can be reduced to $c$.

The last remaining step is to call the monoid prover. Unfortunately, we can't directly write a function of conversion from group to monoid, because in a group we have the possibility to express negations and subtractions, that we do not have in a monoid. The idea is that we will encode negations as variables, and we will let the monoid prover deal with them as ordinary variables. In order to achieve this goal, we need the following datatype that will help us to distinguish between a real variable and the encoding of a negation :

```
data Variable : {c:Type} → {n:Nat}
              → (Vect n c) → c → Type where
    RealVariable : (Γ:Vect n c) → (i:Fin n)
         → Variable Γ (index i Γ)
```

[13] It is not enough to simply check their syntactic equality, as they might not be immediately equal but could perhaps be normalised to the same term. Thus, this step –which is part of the reduction function– uses the entire reduction function, but it calls it on a smaller term.

```
EncodingNegOfVar : (Γ:Vect n c) → (i:Fin n)
       → Variable Γ (Neg (index i Γ))
```

We only need to encode negations of variables, as negations of constants can be simplified into a constant. Also, there can't be a negation of something different than an atom (a variable or a constant), because of the treatments we've done beforehand, and particularly because we have pushed all the negations symbols inside the parenthesis.

The constructor for variables now takes a Variable as parameter, instead of taking directly an element of `(Fin n)` :

```
VarG : (p:Group c) → {Γ:Vect n c} → {val:c}
        → (Variable Γ val) → ExprG p Γ val
```

Thanks to this encoding, we can now transform an `ExprG` to an `ExprMo`. A constant `(ConstG p Γ c1)` will be transformed into the corresponding constant `(ConstMo p Γ c1)`, a `PlusG` into the corresponding `PlusMo`, a real variable into the same real variable, the negation of a constant into the resulting constant, and finally the negation of a variable `i` into a `(VarMo p (EncodingNegOfVar Γ i))`. Once this encoding is computed, the penultimate step is to call the monoid prover for finishing the normalisation. Finally, we interpret back the result into an `ExprG`, which implies converting the potentials `EncodingNegOfVar` into `Neg` of the corresponding variables.

### 4.6.6 From a group prover to a commutative group prover

A commutative group, compared to its underlying group, only has one extra axiom : the axiom of commutativity for the addition. The normalisation will therefore work as follow : it will start by calling the group prover. Here, the two algebraic structures have the same power of expressivity, and thus there's no specific encoding to design in order to convert an `ExprCG` into an `ExprG`. Calling the group prover will do all the needed simplifications, apart from the simplifications relative to the use of commutativity. That's why, afterwards, we need to order the *products of monomials* in the toplevel sum, following the rules presented in 4.6.1. The current implementation uses an insertion sort for ordering these products of monomials. As will be discussed in 4.7.5, if the efficiency of the commutative group prover (and the provers above) becomes an issue on large terms, it will be possible to replace this insertion sort by a more efficient one, like merge sort.

### 4.6.7 From a commutative group prover to a ring prover

The most important specific task of the ring reduction is to develop entirely the expression with the use of distributivity. After that, it needs to call the commutative group prover, but here again, we're losing some expressivity as it won't be possible to express a product at the commutative group level. We follow the same idea that we've implemented for the conversion going from group to monoid, and we will use some specific encoding. More precisely, all monomials (containing products), like $3 * (x * y)$ are encoded as variables before being passed to the commutative group prover. This is possible because we've fully developed the expression, which had the effect of ensuring that a product can't contain a sum. We add a constructor to the type `Variable` in order to encode these monomials.

## 4.7 Properties and results

### 4.7.1 Correctness

All the machinery that we have produced has been formally proven correct within Idris internal type theory. The proof of correctness is in fact already contained in our development, and no additional proofs are required in order to formally establish this correctness, as this proof of correctness is in fact what our tactics are producing. Indeed, in order to operate, all our tactics work by normalising the left and the right hand sides of the potential equivalence, but these normal forms are just a support for producing the proof of equivalence required by the user. The proof itself is what our tactics were expected to produce. This is rare enough to be mentioned, as usually, the result produced by a function is not the same object as its proof of correctness. The area of proof automation is here an exception.

Written in plain English, the correctness of our tactics says that when required to prove (in a potentially non empty context) an equivalence $x \simeq y$, the system either generates nothing, or it generates a valid proof of $x \simeq y$. Because our tactics return an inhabitant of the type `Maybe(x ≃ y)`, and because Idris' type-checker verifies the proof that has been generated, we have the guarantee that the proof cannot be invalid, assuming the internal kernel of Idris[14] is sound, which we expect to be[15].

### 4.7.2 Completeness

We've said in the previous subsection that *correctness* is the fact that our tactics either generates nothing or a valid proof of $x \simeq y$. Therefore, a tactic that would always return the value `Nothing` could be seen as correct : it never generates a false proof, because it simply never generates any proof. However, such a system, although "correct", would be completely useless, and this is why *completeness* matters. Being complete for a tactic means that whenever ask to generate a proof of $x \simeq y$, the tactic always returns a proof if $x \simeq y$ is provable.

Our tactics return a proof of $x \simeq y$ when the normalisation of the reflected left and right hand sides are syntactically the same entities, which is checked by the `expr_eq`. Therefore, completeness can be stated as the following meta-theorem :

```
completeness_1 : ∀ x y, x ≃ y →
  expr_eq (reduce (reflect x)) (reduce (reflect y))
           = Just refl
```

[14] By "internal kernel", we refer to the concrete implementation of Idris' underlying type theory.

[15] If Idris' internal type theory or its implementation would be unsound, not only the correctness of our tactics would be affected, but potentially any development could become unsafe, so the need for proofs and proof automations would disappear as nothing could be trusted anymore.

Another possibility to express it, is to say that if the result is `Nothing`, then the two concrete expressions $x$ and $y$ are not equivalent :

```
completeness_2 : ∀ x y,
  expr_eq (reduce (reflect x)) (reduce (reflect y))
           = Nothing → x ≄ y
```

`completeness_1` implies `completeness_2` because $A \rightarrow B$ implies $\neg B \rightarrow \neg A$ (here $A$ is $x \simeq y$, and $B$ is `expr_eq (reduce (reflect x)) (reduce (reflect y)) = Just refl`) and because the result is either `Nothing` or `Just refl`, so one possibility is the negation of the other.

**Remark.** The other direction `completeness_2 → completeness_1` also holds, but only if we place this meta-theoretic analysis in the context of a classical logic, as $(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$ can't be proven in constructive logic[16].

Encoding completeness and proving it inside the language is very difficult because the output of our normalisation function is just an `Expr`, and this type does not constrain the normalisation function to produce an expression that is indeed in normal form, as this is the type of the input as well. The type-safe reflection technique that we've developed (that made the building of the tactic and its proof of correctness easier) was the main focus on our approach, and the trade-off was to make the proof of completeness more difficult, or even impossible to do with a reasonable amount of work. However, it would theoretically have been possible to manipulate a specific datatype for expressions in normal forms. That would have added quite a lot of hassle, because our datatypes are already very generic and indexed by multiple values. Someone focusing on a mechanically verified proof of completeness would have certainly made a different choice here, by defining a specific datatype for normalised expressions, that would allow to obtain by construction the proof of completeness.

[16] Only a weaker version can be proven in constructive logic : $(\neg B \rightarrow \neg A) \rightarrow (\neg\neg A \rightarrow \neg\neg B)$.

Anyway, there is a reason for not needing that much a formal proof of completeness, which is that nothing can go wrong if a tactic is not complete. In fact, if a tactic is not complete, then we might realise it at some point when we will try to prove an equivalence that we know holds, but that the system can not prove. At this stage, nothing will be broken, and the system will still be sound. We would just have to do the proof by hand (as we did without automations), or to fix the prover for dealing with what is certainly a forgotten treatment in the normalisation function. As long as no such example are found, the system can be considered as total in practice without any risk. Developing the formal proof of completeness would be very complicated, and there is no point to do so because the guarantee offered by the completeness isn't something critical that must be ensured at all time. Moreover, if the proof of correctness *is* vital for us, the proof of completeness would not be of any practical interest inside the proof assistant.

However, we obviously believe that our tactics are complete, or a least complete-enough for being *useful*. We believe that they are complete for two reasons :

The first one is practical : we haven't seen any example where the proof should have been generated, but wasn't generated.

The second, which is the most important, is theoretical : we believe that the normalisation function that we have implemented effectively puts the polynomials in a canonical form. The normal form that we have chosen, which is fully developed (see section 4.6.1), makes it easy to believe this second point. Let $x$ and $y$ be the left and right side of the potential equality. Let's assume that $x \simeq y$. Now, let's consider the shape of `norm x` and `norm y`. After normalisations, we know[17] that we have got polynomials (of multiple variables) expressed as a sum of products of monomials, where each monomial is itself a product of a constant and of a product of variables (see 4.6.1 for more details). But since $x \simeq y$, then, by *identification*, the corresponding constants must be equal, which makes `norm x` and `norm y` being syntactically the same. We can visualize this argument more easily with one variable

[17] This is precisely this knowledge which is hard to prove inside the proof assistant, without using a specific datatype which enforces the normal form shape.

: if we take a polynomial of a variable $x$ of degree $n$, developing it completely will give, after some basic rearrangements and simplifications : $P = a_0 * x^0 + a_1 * x^1 + a_2 * x^2 + a_n * x^n$.
If $P$ is equal to another polynomial $P'$ of the same degree, and if we rewrite this second polynomial into : $P' = a'_0 * x^0 + a'_1 * x^1 + a'_2 * x^2 + a'_n * x^n$ then, since $P = P'$, by identification, each coefficient must be equal to its correspondent coefficient : $a_i = a'_i$ for each $i$ in $[0,\ n]$. Said differently, once normalised like this, a polynomial of a variable can only be equivalent to itself if we only consider normalised expressions. The same happens with our polynomials of multiple variables, but the identification works there between the coefficients of the monomials.

### 4.7.3 Termination

Termination is another important property because no one likes to wait for the output of a program that will never come because the program is stuck in an infinite loop. We use computer programs to automate tasks and we want the tasks to be effectively done after a finite amount of time. In order to gain guarantee about the termination, Idris is equipped with a totality checker. A function is total if it is defined for all possible inputs and if it is guarantee to always terminate, so *totality* is more than just *termination* as it also contains *coverage*. In dependently typed theories, totality checkers check the termination by looking at the size of the arguments in recursive calls. If all recursive calls always happen on a term strictly smaller than the original one, then we have the guarantee that the sequence of recursive calls will terminate at some point. Because it is difficult for the totality checker to automatically determine a good notion of size, this check is often implemented by simply checking that recursive calls happen on a strict subterm of the original argument for at least one argument $n$, and in this case the function is said to be a structural recursion on $n$. Therefore, *the semantic notion of termination is approximated* by a more restrictive syntactic notion. That will necessary lead to false positives, i.e. situations where the totality checker reports a function as potentially not terminating, when this function is in fact always terminating. This situation happened multiple times during the development of our hierachy of tactics. Here is an example that belongs to the group level : it is the function that pushes negations inside the parenthesis, following the rule $-(a+b) = (-b) + (-a)$ :

```
propagateNeg : {c:Type} → (p:Group c)
        → {g:Vect n c} → {c1:c} → (ExprG p g c1)
        → (c2 ** (ExprG p g c2, c1~=c2))
propagateNeg p (NegG (PlusG e1 e2)) =
  let (r_ih1 ** (e_ih1, p_ih1)) =
      (propagateNeg p (NegG e1)) in
  let (r_ih2 ** (e_ih2, p_ih2)) =
      (propagateNeg p (NegG e2)) in
    -- Careful : - (a + b) = (-b) + (-a) in a group
    -- and not (-a) + (-b) in general.
    ((Plus r_ih2 r_ih1)
      ** (PlusG e_ih2 e_ih1, ?MpropagateNeg_1))
propagateNeg p (NegG (NegG e)) =
  let (r_ih1 ** (e_ih1, p_ih1)) = propagateNeg p e in
    (r_ih1 ** (e_ih1, ?MpropagateNeg_2))
propagateNeg {c} p (NegG e) =
    -- Here ’e’ can only be a constant or a variable
    -- as we’ve treated Plus and Neg before
    (_ ** (NegG e, set_eq_undec_refl {c} _))
propagateNeg p (PlusG e1 e2) =
  let (r_ih1 ** (e_ih1, p_ih1)) =
      (propagateNeg p e1) in
  let (r_ih2 ** (e_ih2, p_ih2)) =
      (propagateNeg p e2) in ((Plus r_ih1 r_ih2)
           ** (PlusG e_ih1 e_ih2, ?MpropagateNeg_3))
propagateNeg {c} p e =
    (_ ** (e, set_eq_undec_refl {c} _))
```

This function is reported as non-total by Idris’ totality checker because in the first pattern (where the input is `(NegG (PlusG e1 e2))`), the recursive calls are made on the arguments `(NegG e1)` and `(NegG e2)` which are not subterms of `(NegG (PlusG e1 e2))`. In fact, this function will always terminate, because at some point, the negations will only be in front of atoms (constants and variables), and in this case the result is produced directly by returning the input expression (that's the third pattern).

Here is another example. In the normalisation functions, we've sometimes faced situations where we had to apply a treatment $f$ on the original expression $x_0$ until the point where applying $f$ again would give the same result. For example, at the ring level, we have a function `develop` which develops one time the products of sums, and then we use a function `develop_fix` to fully develop the polynomial by applying `develop` as many times as needed. A function $f\_fix$ that does this job is said to compute a fixpoint of $f$, i.e. it produces a solution to the equation $x = f\ x$. Here, the fixpoint that we compute is the fixpoint starting from the input value $x_0$[18] . We've needed these fixpoints when we had to apply a treatment on a specific pattern (or on a set of patterns), but with the possibility that the application of the treatment could create a new instance of the pattern that precisely needs to be rewritten. There is also the case of nested patterns where we can not treat them all at the same time, and we have to start the treatment by one of them, leaving the other one for later (i.e. for the next passes). The idea for writing these $f\_fix$ functions is to apply the one-pass treatment $f$ once, and then to inspect if some simplifications have been done by this application, i.e. to check if the current result has changed. If so, $f\_fix$ is applied again recursively. If not, the current result is returned. Below is the scheme (for rings) of how such fixpoints can be computed for a specific function $f$.

```
f_fix : {c:Type} → (p:Ring c)
        → {g:Vect n c} → {c1:c} → (ExprR p g c1)
        → (c2 ** (ExprR p g c2, c1~=c2))
f_fix p e =
  -- Apply the one-pass treatment f once
  let (r_1 ** (e_1, p_1)) = f p e in
    -- Look for syntactical equality : have we done
    -- some simplification in the last pass ?
```

[18] Formally, the result $x$ verifies $\exists n,\ f^n\ x_0 = x \wedge f^{n+1} x_0 = x$.

```
case exprR_eq p _ e e_1 of
  -- Previous and current terms are the same :
  -- we stop here
  Just pr => (r_1 ** (e_1, p_1))
  -- Previous and current are different :
  -- we apply the fixpoint again
  Nothing => let (r_ih1 ** (e_ih1, p_ih1)) =
                  f_fix p e_1 in
                    (r_ih1 ** (e_ih1, ?M_f_fix_1))
```

The termination of a function that follows the scheme of $f\_fix$ depends on what the one-pass treatment $f$ is doing. For example, if $f$ replaces all instances of a specific pattern in the original expression, then even if $f$ occasionally creates an instance of this pattern, we might know that overall, at some point, they will all be replaced and the function will terminate. Therefore, only a smart and fine-grained analysis can realise that functions such as `develop_fix` terminate. The syntactic and over-pessimistic totality checker cannot do automatically this kind of analysis. We could do some hard work for encoding a good notion of size, and proving that it decreases at every recursive call. However, as was the case for completeness, developing such a proof in the system is not worth the effort. So far, we haven't seen any example of a term that would make our normalisation function run into an infinite loop. If this situation happens at some point, we will simply fix the normalisation function. There is only one case where having a non-terminating function could be dangerous, and that's in the case where this function would be used inside the proof of correctness, i.e. for proving the desired equivalence $x \simeq y$ (or in some auxiliary proofs used in this proof). Because of the Curry-Howard correspondence, proofs are functions, but these functions absolutely need to be total in order to maintain the system sound. We make the following claims about termination :

- Even if some functions that produce computational content (i.e. not proofs) can't be tagged as total in the system, we always have good reasons for trusting that they will terminate, as it is the case for the function `develop_fix`. This reason can be a complicated semantic notion that would be hard to encode in the system in order to formally prove the termination, but that is theoretically doable : we have never written a function that we know could potentially never stop on stop inputs.

- The proof of $x \simeq y$ –that is being built automatically by our machinery– never uses a potentially non-terminating function as a proof, because that would make the system unsound. In fact, all the lemmas used in the proof $x \simeq y$ can be tagged as total. Therefore, if our tactics generate a proof of $x \simeq y$, we can be sure that this is a valid proof.

Finally, let's emphasize the fact that our machinery runs during the type-checking of the user's program, and not during its runtime. The reason is that our machinery produces something with no real computational content : it generates a proof that only has to be type-checked in order to obtain the guarantee that it conveys. Therefore, the termination of our tactics, even if important, can not affect the termination of the user's programs.

### 4.7.4 Results

The collection of provers that we've developed can prove equivalences in algebraic structures. These provers are generic in multiple ways :

1. They work for many algebraic structures : magma, semi-group, monoid, commutative monoid, group, commutative group, semi-ring and ring.

2. They work for any type $c$ that is an instance of one of the structures mentioned above.

3. They work for any equivalence relation $\simeq : c \rightarrow c \rightarrow Type$ and not only for the propositional equality.

These provers are necessary correct (as their proof of correctness is precisely their output, see section 4.7.1), and we have good reasons to expect them to be complete (see section 4.7.2)

These provers can be used to automatically prove the proofs obligations that appear naturally during the development and the certification of real-world applications. These proofs obligations are common when dependent types are being used. Let's go back to the concrete problem presented 1.3 with binary numbers and their addition, that led to two proof obligations `adc_lemma_1` and `adc_lemma_2`. The latter, which was the most complicated, required us to prove that :
$vLsb + ((vCarry0 + v) + v1) + ((vCarry0 + v) + v1)$
$= (c + (bit + (v + (v + 0)))) + (bit1 + (v1 + (v1 + 0)))$
In a context there is the following induction hypothesis
$ihn : (c + bit) + bit1 = vLsb + (vCarry0 + (vCarry0 + 0))$

We are now able to use the proof automation that we've developed in this chapter in order to automatically prove this goal. That will demonstrate on an example that our machinery can effectively be used to solve the initial problem of providing proofs for index mismatches. We are working with natural numbers and the addition, which is associative, commutative, and which admits 0 as neutral element. Therefore, we are trying to prove an equality in a commutative monoid. The first thing that we have to do in order to use the commutative monoid prover is to write an *implementation* of the `CommutativeMonoid` interface for the type `Nat`. That consists of providing proofs for all the required properties. Many of them are already provided in the standard library, so we can just pass these already made proofs. Once the implementation of the `CommutativeMonoid` is defined, we can use the corresponding normalisation procedure for normalising the hypothesis and the goal (both left and right hand sides). In this example, we can't simply call the prover with the goal that we have to prove, as the left and right hand sides of the goal are not using the same variables ($vLsb$ and $vCarry0$ only appear on the left hand side of the goal, while $c$, $bit$ and $bit1$ only appear on the right) and that's the hypothesis `ihn` that makes the link between these different variables. Let's assume the ordering of the variables decided by our machinery is $bit < bit1 < c < v < v0 < vCarry0 < vLsb$. We start by normalising the hypothesis, which gives : $ihn\_norm : bit + (bit1 + c) = vCarry0 + (vCarry0 + vLsb)$. We also normalise the goal, which gives : $v + (v + (v1 + (v1 + (vCarry0 + (vCarry0 + vLsb))))) =$

$bit + (bit1 + (c + (v + (v + (v1 + v1)))))$

We can rewrite the normalised hypothesis `ihn_norm` in the normalised goal, which gives the following formulae to prove : $v + (v + (v1 + (v1 + (bit + (bit1 + c))))) = bit + (bit1 + (c + (v + (v + (v1 + v1)))))$. At this stage, the prover can finish on its own by normalising the left and right sides of the current goal. This example has required a few more calls to the normalisation function for commutative monoid because we had to deal with an hypothesis that made the link between different variables, but even in this case, all the normalisations were done automatically, so that was effortless. We've demonstrated that our machinery can automatically prove equalities (and equivalences in general), which helps for the development and the certification of programs, especially the ones that intensively use dependent types and indices.

### 4.7.5 Complexity and performances

#### Worst-case complexity

Evaluating the worst-case complexity of our tactics requires us to analyse the complexity of the normalisation functions which is not an easy task, as it depends on the length of the input expression, but also on the actual content of the expression : if negations are used, on the proportion of constants being used, and even on the balance of the expression. For example, if the input expression of length $n$ is a sum $A + B$, we will often have recursive calls on the sub-terms $A$ and $B$. If the expression is completely unbalanced, with a length of 1 for $A$, and a length of $n - 1$ for $B$, then the recurrence relation is $T(n) = T(1) + T(n-1) + f(n)$. Let's assume that the cost of the base case is small compared to the cost of the work done outside of the recursive calls, i.e. $T(1) = O(f(n))$. We get $T(n) = T(n-1) + O(f(n))$. However, if the input expression was perfectly balanced, then the recurrence relation is $T(n) = 2 * (T(n/2)) + f(n)$. In order to see the difference between these two cases, let's assume the cost outside of the recursive calls is linear (i.e. $f(n) = n$). The first possibility $T(n) = T(n-1) + O(f(n))$ leads to a complexity of $\boldsymbol{O(n^2)}$. The second case needs the application of the master theorem that gives solution of the general recursion relation $T(n) = a * T(n/b) + O(n^d)$. The master theorem says that if $d < log_b a$ then $T(n) = O(n^{log_b a})$ : that's the case where the cost of the treatment done outside of the recursive calls is insignificant compared to the cost of the recursions. However, if $d = log_b a$ then $T = O(n^d.log_b n)$ : that's the case where the cost of the recursive calls and the cost of the treatment done outside of them both matter. The last case is when $d > log_b a$, and in this case $T(n) = O(n^d)$ : that's the case where the treatment done outside of the recursive calls is so expensive that it overwhelms the cost of the recursions.

Here $a = 2$, $b = 2$, and $d = 1$, which gives $d = log_b a$, so we are in the second case of the master theorem, and the solution is therefore $T(n) = O(n^d.log(n))$, i.e. $\boldsymbol{O(n.log(n))}$. We see here that we can get a significant difference in the theoretical complexity of a treatment depending on the balance of the expression. We will therefore have to make a rough approximation, abstracting over what the content of the expression is and how it is balanced. Our normalisation functions are split into small functions that each do a simple task, and these simple tasks often belong to these categories :

- Traversing an expression, while locally replacing a specific pattern by another one, like replacing $(a - b)$ by $(a + (-b))$. For the cases where both the recognition and the replacement of the pattern is done in constant time, this treatment correspond to the recurrence relation $T(n) = T(n-1) + O(1)$ in the non-balanced case, and to $T(n) = 2.T(n/2) + O(1)$ in the perfectly balanced case. The balancedness doesn't change the result of what is in fact a full exploration of the expression, and the cost is linear on the size : $T(n) = O(n)$.

- Applying recursively a treatment on the subterms $A$ and $B$ of a binary expression like $A + B$ or $A * B$, and continuing by doing a treatment of linear complexity. That gives either the relation $T(n) = T(n-1) + O(n)$ or $T(n) = 2.T(n/2) + O(n)$ depending on the balancedness, leading to respectively $O(n^2)$ and $O(n.log(n))$, as explained above. Shuffling parenthesis goes into this category, like transforming an expression into its full right-associative form.

- Reorganizing sub-terms of an expression, like the components of a sum within a commutative group or commutative monoid. The reorganization is implemented as a sort, and the complexity of this treatment therefore depends on the complexity of the sort itself. We've implemented an insertion sort (of worst-case complexity $O(n^2)$) instead of a more efficient merge-sort or quicksort for simplicity, as we don't only sort the subterms of an expression but we also generate a proof term at the same time, which makes the task harder. This sorting algorithm can be replaced latter on, giving room for improving the performances for all the commutative structures. However, this sorting algorithm is not a theoretical bottleneck as we have other treatments that are $O(n^2)$ in the worst case.

For the most interesting structures of commutative monoids, commutative groups, semi-rings and rings, the normalisation function composes these three kind of treatments, so the overall complexity of the provers for these structures is expressed by the sum $O(n) + O(n.log_2 n) + O(n^2)$, leading to a worst-case complexity of $O(n^2)$.

The best case complexity of the current implementation is also quadratic due to the insertion sort. However, the case where the left and right hand sides are exactly the same could be improved by first checking if the two expressions are exactly (i.e. syntactically) the same, and if so to answer immediately, and to only normalise them otherwise.

### Performances

In order to give an idea of what the performances are on a real and fairly complicated example, we go back again to the concrete problem presented in section 1.3 with binary numbers and their addition, that led to the proof obligation `adc_lemma_2`. As explained in the subsection 4.7.4, the automatic generation of this proof required 6 calls to the normalisation algorithm for terms in a commutative monoid. The first two calls were for normalising the left and right hand sides of the hypothesis. The next two calls where for normalising the left and right hand sides of the goal. After a rewriting of the normalised hypothesis into the normalised goal, a call to the prover finished the proof with two last normalisations. In a more conventional example that does not involve different variables linked via an hypothesis, only the last two calls would be needed. Therefore, this example required three times more normalisation calls, which implies that the time needed to produce the proof is roughly tripled compared to an example with some equivalent expressions (of roughly the same size) but without hypothesis to deal with. Still, in this example, our implementation generates and prints the proof in less than 8 seconds on a dual core i5 processor @ 2.4 Ghz. Even if that can be seen as slower than hoped, the tactic is perfectly usable. Also, if we only ask the system to generate the proof and not to print it (as printing it is completely useless : the goal of the proof is only to be checked by Idris' type-checker for ensuring the corresponding guarantee), the execution time goes under 4 seconds. Therefore, for many concrete examples that involve comparable expressions (with 7 to 10 variables and 8 to 10 use of operators), the generation of the proof will often take less than a second, which is felt as instantaneous by the user.

In the collection of tests that we have implemented for the various tactics, one of the most complicated one is the proof[19] (at the ring level) of $((((((3*x)*(y*2))*u)+(x*(y-y)))+(3*((x*y)*(5*g)))) = (((3*x)*(y*5))*g)+(3*((x*y)*(2*u))))$ where $x$,$y$,$u$ and $g$ are also relative numbers. The automatic generation of this proof with 4 variables and many simplifications to be performed is done instantly by our ring prover on the same machine.

In order to compare our performances with another implementation, Coq's ring tactic takes a bit less than a second to automatically produce the proof of *adc_lemma*_2 by using the same sequence of automating rewritings, so it is roughly 4 times more efficient on this test. The second test just above also leads to a proof generated instantly as it was the case for our own prover.

[19] The automatic generation of this proof can be found in the file Provers/ring_test.idr in the term `proof_expCr_expC2r`.

## 4.8 Alternative approaches

### 4.8.1 A naive approach

Without our type-safe reflection mechanism, the naive and traditional[20] way to go for this problem of automatic proof generation would be to have functions producing only computational content (i.e. the normalisation functions would only produce a normal form and no proof), and some external lemmas about them. If we no longer have our type `Expr` indexed over the concrete value of type `c`, then we would have to start by defining the function `reify:Expr → c` which interprets back an expression into its concrete value. As we did in our approach, there would still be a function for reflecting terms `reflect:c → Expr` that does the opposite job. Because `reflect` is a function based on *syntax*, it needs to use Idris' reflection mechanism. We would need to prove the correctness of these two functions, and it turns out that one lemma is enough for completely specifying the expected behaviour of these two functions at the same time : `reflect_and_reify_correct:∀ x:c, reify (reflect x) ≃ x`.

Then comes the normalisation function : `normalise:Expr → Expr`. Because this function is weakly typed compared to our approach (we no longer have an index giving a guarantee about the concrete values), we would need to provide a lemma of correctness about it after its definition. This lemma of correctness must say that the interpretation of the original (reflected) term is equivalent to the interpretation of the normalised (reflected) term :
`normalise_correct:∀ e:Expr, reify (normalise e) ≃ reify e`
We have avoided the need of such a complicated proof in our correct by construction approach enabled by our type-safe reflection mechanism.

Now, similarly to what we've done, we would need a syntactical equality test, checking whether two reflected expressions are syntactically equal. The difference here is that, as often with this kind of "traditional" approach, we would produce something that belongs to the world of computations, usually an uninformative boolean :

[20] By "traditional" we have in mind the proof engineering style developed in the Coq'Art [6] where functions are defined with weak types, and some lemmas proven afterwards complete their specification.

`beq_Expr:Expr → Expr → bool`
Because this boolean on its own is completely uninformative, we now need a proof of correctness for this function as well, which says that if this function decides that two given terms are syntactically equal, then their interpretation should be equivalent :

```
beq_Expr_correct : ∀ (e1 e2:Expr),
    beq_Expr e1 e2 = true → reify e1 ≃ reify e2
```

We did not have to do prove this lemma either with our approach.

We could now build the kernel of the tactic as the following decision procedure :

```
decideEq : c → c → bool
decideEq x y =
      let ex = reflect x in
      let ey = reflect y in
        beq_Expr (normalise ex) (normalise ey)
```

This function on its own would not be enough as the goal of a tactic is not to answer a simple "yes they are equivalent" or "no they are not equivalent", but to produce the proof of equivalence when appropriate. Therefore, we would need to add a correctness theorem, and this is precisely this correctness theorem that would be the used when calling the tactic. This correctness theorem for `decideEq` would be `decideEq_correct:∀ x y:c, decideEq x y = true → x ≃ y`. Scheme of proof[21] : we first need to obtain a proof `H0` of $reify\ (reflect\ x) \simeq reify\ (reflect\ y)$, which mostly involves using two times the lemma `normalise_correct` and one time `beq_Expr_correct` in the (unfolded) hypothesis. Then, we can build the desired proof of $x \simeq y$ by using two calls to the `reflect_and_reify_correct` lemma and the recently obtained proof `H0`. □

[21] I have developed a formal proof of it in Coq which can be found in the file others/traditionalApproach.v which is an axiomatic formalisation of how the tactics could be developed without our type-safe reflection mechanism.

Now, the tactic could be built by using this correctness theorem. When trying to prove a goal $a \simeq b$, the tactic would apply this theorem `decideEq_correct`, which means that $x$ will be unified to $a$ and $y$ to $b$. The desired proof is thus produced if the premise `decideEq a b = true` holds. But whether this premise holds or not can be easily tested by running the function `decideEq` on the two arguments $a$ and $b$.

Both with this “naive and traditional approach" and with our type-safe reflection, the activity of proving has effectively been replaced by the task of running a function, and a simple evaluation is now enough to produce the desired proof. The main advantage of our approach compared to this kind of naive approach is that we don't need external lemmas, and more generally that the proof of correctness is obtained a lot more easily. Also, as we will see in the next subsection, this kind of naive approach could not be followed exactly, and we will show how Coq's implementation of a ring prover –which almost follows this naive approach– has been adapted.

### 4.8.2 Coq's implementation

#### Coq's implementation

Coq is not equiped with a full hierarchy of provers comparable to what we've built for Idris, but it is equiped with a prover for rings and semirings. In Coq's latest implementation of the ring tactic, described in [22], they almost followed what has just been desribed as a “traditional" approach with the use of many auxiliary lemmas. The biggest difference between our work and Coq's Ring tactic is that they've implemented it in Ltac, when our entire development is carried inside Idris' type theory. Ltac is an untyped tactic language, in the sense that an Ltac "function" produces something which might not have a valid –and unique– type. Because of this, Ltac definitions can only be used in the context of goals. Applying a tactic defined with LTac might work, and then it makes progress to the current goal, or it might fail, and in this case the goal is kept unchanged. And anyway, the validity of the proof done is going to be checked during the final `QED` so no inconsistencies can be introduced with these definitions. Ltac functions can't be used in the statement of a lemma, and it is therefore not possible to reason about them. That means that it is not possible to write the lemma `reflect_and_reify_correct` of the "traditional" approach described in the previous subsection, because it is not even possible to state it as it uses a function defined in LTac. Because it is impossible to state this property, it would not be possible to finish the proof of the main theorem `decideEq_correct:∀ x y:c, decideEq x y = true → x ≃ y` that we had to do in the naive approach. Indeed, the last step of this proof was to apply two times –one for the LHS, one for the RHS– the fact that $\forall\ x : c,\ reify\ (reflect\ x)\ \simeq\ x$. One possibility would be to add this axiom, but this is particularly unsightly, and potentially harmful. Also, it would imply that anyone using the ring prover would have this axiom added to his development, and would be forced to believe in it. Some proofs would be done automatically for the user of the system, but at the overly-expensive price of adding some uncertainty. Instead, what they did for Coq's implementation of the ring prover was to replace the main theorem $\forall\ x\ y : c,\ decideEq\ x\ y\ =\ true\ \rightarrow\ x\ \simeq\ y$ by the following lemma[22], which is weaker, but still powerful enough to build the desired tactic :

```
f_correct:∀ (e1 e2 : Expr),
    beq_Expr (norm e1) (norm e2) = true
      → reify e1 ≃ reify e2
```

Now, the tactic works like this. When trying to prove the goal $a \simeq b$, it computes $(reflect\ a)$ and $(reflect\ b)$. It then tries to apply $(f\_correct\ (reflect\ a)\ (reflect\ b))$ to the goal.

- If it can unify the goal $a \simeq b$ with $reify\ (reflect\ a)\ \simeq\ reify\ (reflect\ b)$ then it only has to check the validity of the premise by running the decision procedure. More precisely, if `beq_Expr (norm (reflect a)) (norm (reflect b))` is evaluated to `true`, then it has built

[22] This lemma –that they call `setpolynomial_simplify_ok`– can be found in Coq's repository in the file *plugins/ring/Setoid_ring_normalize.v*.

the premise of `f_correct` and there's nothing left to prove. However, if the result was `false`, then it means that the automatic ring prover hasn't been able to prove this goal, because the left-hand side and the right-hand side don't reduce to the same thing. Assuming the ring prover is complete, that means that $a$ is not[23] equivalent to $b$, or at least, they aren't equivalent only with the properties of a ring.

- If it could not unify the goal $a \simeq b$ with $reify\ (reflect\ a) \ \simeq\ reify\ (reflect\ b)$, then it means that there is something wrong in the definitions of the functions `reflect` and `reify`, and it could print something like "The ring prover has failed unexpectedly. There's something wrong with the implementations of the `reflect` and `reify` functions". Nothing really bad would have happened, as no inconsistent proof would have been produced, nor no axioms added. That would just mean that the implementation of the ring prover is slightly broken, and that some goals that should be automatically provable aren't automatically proved. Thus, it would only decrease the completeness of the prover. That would be of course bad –because as discussed in 4.7.2, in the extreme case the prover never succeeds to generate a proof and is therefore completely useless–, but that would only limit the scope of usage of the prover.

### Differences with Coq's implementation

Coq's ring prover follows the traditional approach of defining functions, and latter on proving many auxiliary lemmas about them. This is particularly adapted to Coq, which has many facilities for the construction of proofs, and a powerful proof mode. However, this approach kind of duplicates the work : they first tell the machine *how* it works, and latter, *why* it works. This separation between the world of computations and the world of logic becomes smaller in our approach with a fine use of dependent types, that allows to

[23] Note that in this case, it hasn't produced a formal proof of dis-equivalence, which isn't the goal of a ring prover.

write more *specific types*, and thus to capture logical properties. The writing of functions can therefore be guided by these more precise types, and this is one of the main benefit of the approach we have followed. Moreover, we almost get the proof of correctness for free, because every little bit of rewriting done for the normalisation of the reflected terms is accompanied by the logical justification which tells why this rewriting can be done –and locally, this justification is always simple !–. In the end, all the proofs we had to do were systematically straightforward, since they only contained rewritings with the available hypothesis and the use of the properties of the corresponding algebraic structure.

Another difference with our implementation is that we have implemented a hierarchy of tactics for many algebraic structures, but Coq's implementation only deals with rings and semi rings. If someone wants to prove equalities in a commutative group that isn't a ring, he simply can not use their prover. A dedicated prover for commutative groups would be needed, and Coq currently does not have one. The worst is that such a prover for commutative group would do very similar treatments, which means that a lot of code and proofs could have been factorised. This is what we've obtained by taking this in account from the start. Our monoid prover uses the underneath semi-group prover, our group prover uses the underneath monoid prover, and so on. In this dimension, the level of re-usability that we have achieved is better than the re-usability of the ring tactic for Coq. However, their implementation is more efficient, as discussed earlier in 4.7.5.

## 4.9 Summary

In this chapter we have implemented a very generic solution to the original problem of index mismatch that appeared in chapter 1.3 when we tried to define functions that use dependent types. This solution takes the form of a hierarchy of provers for equivalence relations in algebraic structures. These provers are generic in multiple ways : they work for many algebraic structures (semi-group, monoid, commutative monoid, group, commutative group, ring and semi-ring), for any type that behaves as one of these structures, and for any equivalence relation on this type (and not only for the propositional equality). The implementation is modular and each prover reuses the prover of the structure from which it directly inherits.

These provers can automatically prove equivalence between terms, which discharges the user of providing proofs for all the proof obligations that appear naturally when using dependent types. Therefore, these provers enable the user to focus on the interesting and main proofs of his development that require specific knowledge and creativity, instead of wasting time and energy for proving routine lemmas that can be automated. We think that the human time is best used when it is devoted for proving things that can not be automated.

The *correct-by-construction* method that we have followed was the same as the one presented in chapter 3. It involved the design of a *type-safe reflection* mechanism where reflected terms were indexed over the concrete expressions, and from which we were able to pull out the proofs easily. The construction of the proof is done step by step, and it follows the construction of the normalised terms. Therefore, unlike other implementations –like Coq's implementation of a ring prover– we do not have the traditional duplication that happens when separating the computational content from the proof of correctness[24] . Instead, in our approach, the construction of the proof is guided by the construction of the normalised terms, which, in addition of avoiding redundancy, simplifies the proof generation considerably.

[24] This is particularly well suited here, because the normalised terms are not needed in output, and the proof of correctness is in fact what has to be produced. See section 4.7.1.

# Chapter 5

# Programming with Dependent Types

*He who hasn't hacked assembly language as a youth has no heart. He who does as an adult has no brain.*
— J. Moore

In the previous chapter, we've built a hierarchy of automatic provers for algebraic structures. Such automations make it easier to use dependent types for everyday programming, as discussed in chapter 1.

As dependent types have become more popular, new usages of them have been investigated, not only as a formal language for doing pure theorem proving, but also as a way to improve code safety without necessarily going as far as proving the full correctness. We explore in this chapter some of these usages, and we highlight some connections with the machinery that we have developed.

## 5.1 Using views to gain structural information

Dependent types have been shown [36, 33, 47, 32, 8, 35, 44, 10, 7] to be an effective tool for expressing logical links, and especially for building links between different representations of the same data.

In [36], an alternative representation of lists is studied, which gives an immediate access to the last element, while still preserving an almost immediate access to the first. Such a representation of an existing datatype is called a *view*, provided that it is possible to write a function that builds the view from the value of the original type (`List` here). This new view of lists is particularly useful for writing an algorithm of (right) rotation on lists, as it offers a direct access to the last element of the list, while preserving the natural order of the list, which therefore goes from the first to the penultimate. In this section, we use this example as a case study, and we adapt it to Idris.

These new lists are defined inductively, and they are indexed over the list that is being represented, denoted as a standard list. In Idris, this definition can be written like this :

```
data SnocView : {A:Type} → List A → Type where
     SNil : SnocView Nil
     Snoc : {A:Type} → (xs:List A) → (x:A)
            → SnocView (xs ++ [x])
```

It is important to see that the `Snoc` constructor encapsulates a standard list instead of being recursive. The value `SNil` represents an empty list, and `Snoc xs x` represents the list `xs` augmented with `x` at its end, which can be expressed as `xs ++ [x]`. This representation gives an immediate access to the last element as pattern matching a `SnocView` value gives it immediately when the underlying list is not empty. We also have an almost immediate access to the head of the list as this only requires to pattern match the value and its underlying list if we are in the case of a `Snoc`.

Given a standard list, we want to have an automatic way to build its `SnocView` representation. For this purpose, we write a function which builds the view from the original list, and it has the following type :

```
buildSnocView : {A:Type} → (l:List A) → SnocView l
```

The type of this function already expresses that the SnocView that is built effectively represents the input l, as it produces a value of type SnocView l.

The SnocView of an empty list Nil is Snil :

```
buildSnocView Nil = SNil
```

The SnocView of a non empty list h::t is built by computing recursively the SnocView of the sub-list t and using this result in a dependent pattern matching (see 1.2.4) that gives us some information about the input we are having :

```
buildSnocView (h::t) with (buildSnocView t)
    buildSnocView (h::Nil) | SNil = Snoc Nil h
    buildSnocView (h::(ys ++ [y])) | Snoc ys y
        = Snoc (h::ys) y
```

If the result of the recursive call buildSnocView t produces SNil, we know that t is Nil, and so we simply return Snoc Nil h with only one element, which is necessarily the last. However, if the result of the recursive call is Snoc ys y, then we deduce that the input t is ys ++ [y]. In this case, as y is the last element of the sub-list t, it is necessarily also the last element of the original input list h::t, and the function can therefore return Snoc (h::ys) y.

This buildSnocView function builds the SnocView of a list given in input, and thus provides a representation which is adapted for manipulating both the first and the last element of the structure. Following [36], if we now want to write a rotateRight function on lists, the view that we have created (and which can be automatically computed with buildSnocView) will be extremely useful. This rotateRight function takes a list in input and returns a list where the last element of the original list is now the first element, and all the other elements have been moved one step to the right. Building such a rotateRight function can easily be done by doing a dependent pattern-matching on the intermediate computation buildSnocView l.

```
rotateRight : {A:Type} → List A → List A
rotateRight l with (buildSnocView l)
    rotateRight Nil | SNil = Nil
    rotateRight (ys ++ [y]) | Snoc ys y = y::ys
```

If the `SnocView` is `SNil`, then the input is forced to be *Nil*, and its right rotation is simply `Nil`. If the `SnocView` is `Snoc ys y`, the input is forced to be `ys ++ [y]`, and in this case its right rotation can be expressed easily as `y::ys`.

If we evaluate `rotateRight [1, 2, 3, 4, 5]`, we get the expected output `[5, 1, 2, 3, 4]`.

With this technique of defining a view (`SnocView`) which carries the information that we need and an automatic way to compute this view from any input (the `buildSnocView` function), we have been able to gain some information about the shape of our inputs because of the strong link expressed (by a dependent type) between a list `l` and its view of type `SnocView l`. Append is not a constructor of `List`, so it would not have been possible to directly pattern-match a list against an append operation. Here, by using the more general dependent pattern matching on a well-chosen view, we've been able to gain the information we needed about the last element of the input list. Such a pattern-matching remain total because it explores exhaustively the possible outcome of computing the view.

In [36], such techniques are implemented for the development of parts of a domain-specific language for cryptographic protocols inspired by Cryptol (Galois, Inc. 2002). One of the most distinctive feature of Cryptol is its pattern matching on bit vectors, where a word is split into pieces, as in the following definition :

```
swab : [32] → [32]
swab [a b c d] = [b a c d];
```

Such a pattern-matching can be simulated by defining a view indexed over the complete bit vector :

```
data SplitView {A:Type} : {n:Nat} → (m:Nat)
```

```
                  → Vect (m * n) A → Type
```

A `SplitView n m v` represents the vector `v` (of size $m * n$) split into $m$ vectors of size $n$. This view has a single constructor :

```
    [_] : {n, m:Nat} → (xss : Vec m (Vec n A))
             → SplitView m (concat xss)
```

where `concat` is the simple following function :

```
concat : {A:Type} → {n,m:Nat} → Vect m (Vect n A)
           → Vect (m * n) A
concat [] = []
concat (xs::xss) = xs ++ concat xss
```

It is then enough to write a `buildSplitView` function that automatically builds the `SplitView` from the flat input vector, and then the `swab` function can be written straightforwardly :

```
swab : Word 32 → Word 32
swab xs with buildSplitView 8 4 xs
    swab _ | [a::b::c::d::Nil]
        = [b::a::c::d::Nil]
```

The `buildSplitView` function is not detailed, but it uses a `split` function, of type
`∀ (n,m:Nat), Vec (m*n) A → Vec m (Vec n A)` and it requires this lemma about it:

```
∀ {n,m:Nat} (xs:Vec (m*n) A),
  concat (split n m xs) = xs
```

Writing views for defining custom pattern matching principles in a dependently-typed language has originally been described by Wadler in [47] and later enhanced by McBride in [33] where a general high-level style of programming in dependently type theory is described. These theoretical ideas about a dependent pattern-matching which gains information about various terms simultaneously following their logical dependencies have lead to various implementation, first in Epigram [32], then in Agda [35] and more recently in Idris [8]. In these kind of usages, dependent types are not used as a way to build a complete formal proof of correctness (we haven't really proved the correctness of the `rotateRight` and `swab` functions after all), but instead as a way to express more easily some algorithms. These algorithms become so easy to write that the need for a formal proof becomes less important because a formal specification would match exactly the code that has been written. For example, we've written that the `rotateRight` of any list `ys ++ [y]` is simply `y::ys`, but a legitimate logical specification would be to state exactly that.

In [31], these techniques are also used for gaining structural informations about terms, this time not exactly for more easily writing an algorithm, but for proving more easily the termination of a first-order unification algorithm. Usually, unification algorithms are written using general recursion, and the proof of termination is done separately, relying on the "occur-check" to ensure that each substitution reduces the number of distinct variables remaining in the problem. As the usual representation of terms does not take in account the number of variables involved, this proof of termination is quite difficult. McBride presents a new representation of terms, indexed over the number of variables $n$, which enables writing a unification algorithm as a structural recursion in $n$. By expressing more structure, dependent types therefore make more recursion structural. Similar techniques have been used in [44, 10, 7] where indexed types and views enables getting more structural informations about terms.

In all these examples from the literature that we have mentioned, the use of dependent types bring more formal guarantees or enables to write more easily some algorithms. This new style of programming that uses the types to be guided to the solution is, we believe, the future of functional programming. However, in all of these formalisations, some work has to be done for proving routine lemmas that are needed for the definition of the function that builds the view. In that respect, the work that we have presented in chapter 4 on proof automation can help programming more naturally with dependent types, as motivated initially in 1.3. Building automations for all possible kind of goals encountered when programming with dependent types would go well beyond algebraic structures and beyond the aim of this thesis, but it would continue to go in the same direction.

## 5.2 Using indexed types to build structures on trusted ones

When programming with dependent types, it is possible to define a new concept as a data-type indexed over another one, the later playing the role as a (potentially partial) semantic for the former. As an example, let's explore briefly how one could define labelled binary trees as a type indexed over the list of the elements found in the tree :

```
data Tree : (A:Type) → List A → Type where
    EmptyTree : {A:Type} → Tree A []
    Node : {A:Type} → {l1:List A} → {l2:List A}
           → (left:Tree A l1)
           → (elem:A)
           → (right:Tree A l2)
           → Tree A (l1++[elem]++l2)
```

Such a representation makes it easier to statically check some operations on trees, as most of them will have an impact on the underlying list of values. It is indeed possible to express directly the effect of an operation on trees by stating how the underlying list is transformed, using functions on lists that are assumed to be correct. This way, we get the correctness of operations on trees relatively to a (computational) specification on lists.

For instance, if we want to write a function that inserts at the right location an element $x$ into a supposedly ordered tree, we can write this function with the following type :

```
insertTree : {A:Type} → (Aord : PartialOrder A)
             → (x:A)
             → {lindex:List A} → Tree A lindex
             → Tree A (insert Aord x lindex)
```

With such a type, if we assume that the `insert` function on `List` is correct, we immediately get that any implementation of `insertTree` that typechecks is correct too.

It is then possible to write the following function, which builds a binary search tree from a list of values :

```
buildSortedTree : {A:Type} → (Aord : PartialOrder A)
                  → (l:List A)
                  → Tree A (sortList Aord l)
```

Again, any implementation of `buildSortedTree` that typechecks will be correct if `sortList` is correct too.

This technique, which consists of relying on a trusted datatype and its associated operations to build a new one, can also be used for writing a new and cleverer version of the same datatype. The first representation can be simple and easy to reason about, and the second one can improve the performances on many operations, or use a more compact representation. In this case, *indexing the new type over the existing one* will give a complete semantics as both types represent the same concept.

In all these kind of applications, there is always many proof obligations coming along, as the term which is computed has very rarely the right indices without doing some rewriting. For this reason, having proof automations available can greatly improve the usability of such techniques, where an existing type is used as a way to express the specification of a new one.

## 5.3 Refinement types and restricted forms of dependent types

Using dependently typed theories to entirely verify the correctness of an application always requires an incredible amount of time and energy. Not only is writing a good and complete formal specification difficult, but the proving step that comes just after is even more difficult. Also, the text that constructs all the proofs is often between 5 to 10 times the volume of the code being verified. That's why we need as many proof automations as possible to make formal verification doable, and this thesis has presented a way to build a hierarchy of provers to help the activity of theorem proving for formal verification. There is also some ongoing research[37, 40, 43, 48, 49, 50] in alternative systems that offer a restricted form of dependent types, and which are therefore not as general as the languages and theories that we have exposed throughout this thesis. However, these systems with such limitations induce decidable logics, and much of the proofs are therefore constructed by the system itself without requiring some manual intervention.

One of these systems is the Logically Qualified Data Types system [40] (abbreviated to Liquid Types), where liquid types are a restricted version of refinement types in which type inference is decidable, meaning that the user never has to write manual type annotation.

Let's explain first this notion of *refinement types*, in the context of the Hindley-Milner type system. A refinement type has the form $\{e : T \mid b\}$ where $T$ is an Hindley-Milner type, and $b$ is a boolean expression which may contain the variable $e$ and the free variables of the program. This refinement type contains all the values $v$ of type $T$ that make the boolean predicate $b[v/e]$ evaluate to *true*. This construction is the exact equivalent to the construction of an intensional set in set theory : $\{x \in X \mid P(x)\}$, which builds a new set from the set $X$, which contains all the elements $x$ of $X$ that verify the condition $P(x)$. For example, the type $\{e : Nat \mid e > 0\}$

is a refinement of the type $Nat$, without the value 0. The notion of *subtyping* is essential in this system, as the notion of subset is essential in set theory with intensional sets. A refinement type $\tau_1 = \{e : T \mid b_1\}$ is a subtype of the refinement type $\tau_2 = \{e : T \mid b_2\}$ if the formulae $b_1 \rightarrow b_2$ is universally valid. Any original (Hindley-Milner) type $T$ can be converted to the refinement type $\{e : T \mid true\}$ without changing the terms that it contains, so any refinement $\tau = \{e : T \mid b_\tau\}$ of $T$ is a subtype of $T$, as $b_\tau \rightarrow true$ for all $b_\tau$.

A *liquid type* is a refinement type where the boolean predicate $b$ is a *conjunction of boolean predicates* from a set $\mathbb{Q}*$. This set $\mathbb{Q}*$ is different at each text location. Every set $\mathbb{Q}*$ is obtained from a set $\mathbb{Q}$ given by the programmer, which contains all the *logical qualifiers* of interest. These logical qualifiers are boolean predicates built using program variables, a special value variable $\nu$ which is distinct from the program variables, and a special placeholder variable $*$ that can be instantiated with any program variable. All the $\mathbb{Q}*$ sets are built by replacing the placeholder $*$ of all the logical qualifiers from $\mathbb{Q}$ by variables in scope at that location, and by only keeping the well-typed logical qualifiers. As in [37, 40], let's take the example of a set $\mathbb{Q}$ containing the logical qualifiers $\{0 \leq \nu, * \leq \nu, \nu < *, \nu < len *\}$. At a program location where the following variables are in scope : $[x : Nat, y : Nat, a : Array]$, the set $\mathbb{Q}*$ is : $\{0 \leq \nu, x \leq \nu, y \leq \nu, k \leq \nu, \nu < n, \nu < len\ a\}$. Note that ill-typed logical qualifiers are removed immediately after being rejected, and are therefore not part of $\mathbb{Q}*$.

Liquid types offer the possibility to get many of the benefits of doing formal certification, without paying much of the cost. The user only has to provide the pre- and post-conditions of a function as liquid types, without having to give any intermediate assertions like loop invariants which can be inferred by the system. Then, all the proof obligations are automatically extracted, and proved automatically by the system. The main restriction is the kind of logical formulae which can be proved in this way : they must belong to a decidable logic, which ensures that they are provable by an SMT prover, which is the internal proving machinery of Liquid Types systems.

Liquid Haskell (LH) [37, 45, 46] is a system which implements Liquid Types on top of the programming language Haskell, therefore providing a weak version of dependent types to Haskell. LH uses the haskell compiler GHC in order to first type-check the program in the Hindley-Milner sense, and then deals with the Liquid type annotation given by the programmer. From these pre and post-conditions, it generated proof obligations in the form of type constraints, which are then solved by an SMT solver.

Let's also mention the F* language[43], which, unlike Liquid Haskell is equipped with full dependent types, while still offering refinement types. In practice, this system looks like an extension of Ocaml with dependent types. This language has initially been designed for the development and the verification of cryptographic protocols[39].

Some other forms of restricted dependent types have been studied for programming with dependent types without all the burden of proof assistants with full dependent types. In Dependent ML (DML) [50], the ML language is extended with a restricted form of dependent types (that its creator has named DML-style dependent types) where the index terms are required to be taken from a type index language that is completely separate from the real (run time) values. This system therefore does not really implement the "types as values" paradigm advocated by dependent types, as there is still a distinction between the real values and the types. In practice, this system is useful if the properties that one want to verify can be encoded as natural numbers, or as some other simple expressions. For example, a library of *Vector* can easily be developed in this system, with the *append* function that has, as usual, a type specifying the size property of the output. By being a conservative extension to the ML type system, any program that typechecks in DML is already typable in ML. However, it might have a more precise type in this extended type system, bringing some parts of formal verification to ML programming.

Even though DML-style types and Liquid Types are only restricted forms of dependent types, they are often enough to express interesting properties about programs. They can be an interesting middle ground between not verifying anything (like in Haskell and Ocaml programming), and verifying every little property (like in Coq). The programming language Idris that has been used throughout this thesis can be seen itself as an intermediate between these weaker forms of dependent types, and a proof assistant like Coq. It is more powerful than Dependent ML and Liquid Haskell as it has support for *full dependent types*, but its emphasis is more on *programming* than on proving, unlike Coq. For instance, Idris does not enforce the termination check, which often makes the implementation of prototypes easier, as it does not require to make all recursions structural in the first instance. Still, we can learn from these simpler systems, and interfacing Idris with an SMT solver for discharging proof obligations that are decidable by an LH-style machinery could be an interesting thing to do, that would continue to go in the proof automation direction that this thesis has taken.

Often, ensuring some logical properties and some links between different data can be more important than doing a full proof of correctness, and these restricted forms of dependent types are particularly well suited for these tasks. When a proof of correctness is done, it is always done relatively to a formal specification. But the formal specification can be as complicated as the code, and the guarantee obtained by the proof becomes less clear. The next and final chapter of this thesis explores these issues of adequacy between formal specification and real requirements.

# Chapter 6

# Predicate Testing in Formal Certification

*Beware of bugs in the above code; I have only proved it correct, not tried it.*

— D. Knuth

In this thesis, we have automated the construction of proofs of equivalence in some algebraic structures, therefore discharging the user from some proof obligations and making formal certification and type-based verifications more accessible to programmers. The use of these type-based verifications helps to increase the confidence in critical software, which is the general goal of formal certification and of the introduction of proofs.

However, a formal proof is only a guarantee relative to a formal specification, and not necessary about the real requirements [38]. There is always a jump when going from an informal specification to a formal specification expressed in a logical theory. Thus, proving the correctness of a piece of software always makes the implicit assumption that there is adequacy between the formalised specification –written with logical statements and predicates– and the real requirements –often written in English–. Unfortunately, a huge part of the complexity lies precisely in the specification itself, and it is far from obvious that the formal specification says exactly and completely what it should say. Why should we trust more these predicates than the code that we've first refused to trust blindly, leading to the necessity of these proofs? In this thesis, we've automated the generation of some kind of proofs, and in this last chapter, the question is simply : how much can we trust formal proofs ?

As we will show, the proving activity has not replaced the testing activity but has only changed the object which requires to be tested. Instead of testing code, we now need to test predicates and logical definitions. We present recent ideas about how to conduct these tests inside the proof assistant on a few examples, and how to automate them as far as possible.

## 6.1 Believing in machine-checked proofs

One way to increase our confidence in software is to formally prove its correctness using a proof assistant. Proof assistants enable to write code, logical statements and proofs in the same language, and offer the guarantee that every proof will be automatically checked. Many of them are functional programming languages, like Coq [6], Idris [8] and Agda [34], and others, like the B-Method [1] belong to the imperative paradigm. These different paradigms are internally supported by different logic. As presented intensively in the previous chapters, systems like Coq, Idris and Agda are based on various constructive logics (CIC and different variants of ML, respectively) and are realisations of the Curry-Howard correspondence, while the B-Method is based on Hoare logic. These different foundations lead to different philosophies and different ways to implement and verify a software, but all of them greatly increase the confidence on the produced software. However, these guarantees tend to be too often considered as perfect, when they are in fact far from it. Knuth was saying "Beware of bugs in the above code; I have only proved it correct, not tried it". And indeed, having only a formal proof is not enough. When we prove the correctness of a function, we only gain the guarantee expressed by the proven lemma, and nothing more.

Say we want to implement a formally verified sorting function for list of elements of type $T$, where $T$ is ordered by a relation $\leq$. We can decide to define the sorting function with a "weak" type, like sort:List T → List T, and to use an external lemma to ensure the correctness of the function. Which property does this function has to respect? First, the output has to be sorted, so we need to define this notion of being sorted, here as an inductive predicate :

```
data isSorted : {T:Type} → (Order T)
                → (List T) → Type where
  NilIsSorted : (Tord : Order T) → isSorted Tord []
  SingletonIsSorted : (Tord : Order T) → (x:T)
                      → isSorted Tord [x]
  ConsSorted : {Tord : Order T} → (h1:T) → (h2:T)
       → (t:List T) → (isSorted Tord (h2::t))
       → (h1 ≤ h2) → (isSorted Tord (h1::(h2::t)))
```

The first and second constructor of this predicate say that [] and [x] are sorted according to any order, and for any x. The third one says that a list of two or more elements is sorted if h1 ≤ h2, and if the list deprived from its head h1 is also sorted. In order to express that the result of sort is sorted, we can prove the following lemma: sort_correct:∀ (T:Type) (Tord:Order T) (l:List T), isSorted Tord (sort l). The problem with this specification is that it does not say anything about the content of the output. The function sort could just return the empty list [ ] all the time, it would still be possible to prove this correctness lemma. Here, the problem is that the function is underspecified, and it is therefore possible to write a senseless implementation, which can be unfortunately proved to be "correct". Only a careful reader could realise that the lemma sort_correct forgets to mention that the input and output list should be in bijection, meaning that everything which was originally in the input list should still be in the output, and that nothing else has been added.

Another bug in the specification could have been to simply forget the third constructor `ConsSorted`. But things more nasty can happen. Imagine that this constructor would have been written with a typo, and that the condition `(h1` $\leq$ `h2)` would have been incorrectly written as `(h1` $\leq$ `h1)`. Any list would be seen as “sorted", just because of this single typo, and the algorithm could for example return its input unchanged. One could object that when doing the proof of correctness, we should realize that the proof is being done too easily, without having to use the essential property that the output is being built such that any element in the list is always lower or equal than its next element. The reality is quite different because many efforts are going in the direction of proof automation, which aims to let the machine automatically generate the proof for some kind of goals. As we've seen in the previous chapters, Coq has already a Ring prover [22] and many others automations, and we have implemented in this thesis a hierarchy of provers for algebraics structures for Idris [42]. There are even extensions to languages, such as Ltac [17] and Mtac [51] that aim to help the automation of tactics. The problem is that the machine is never going to find a proof “too easy", and will never report that something seems weird with the specification given by the user.

Thus, if we want to trust the proven software, we're now forced to believe that there is adequacy between the formal specification and the informal requirements. A switch has occurred. We used to have to trust code, but we now have to trust logical statements and predicates. However, if the specification does not obviously capture the informal requirements, why should we blindly believe in it, when we've first refused to blindly trust the code? Of course, with stronger systems, like dependently typed systems used throughout this thesis, we can make a formal specification and an implementation evolve together, which helps to build some confidence. However, the guarantee that we get is not an absolute one, and the aim of this chapter is to raise awareness on the adequacy concern, and to see how heterogeneous approaches, that mix both proofs and tests, can help to go a step forward in the certification process, in the context of proof assistants based on type theory. More precisely, we :

– Show some basic approaches to the problem of underspecification (section 6.2)

– Present a new way to test the predicate in the proof assistant, by automatically generating terms, and we completely automate these tests. We also show how we can go a step forward by replacing these tests about the predicate by some proofs. (section 6.3)

– We discuss possible directions for making dependently typed programming languages more adapted to the testing of specifications (section 6.4).

We continue to use Idris, but as before, all the ideas that we present can be applied to any programming language or proof assistant based on dependent type theory. In this chapter, we will use a running example about sortedness that can be found online at `https://github.com/FranckS/ProofsAndTests`. The ideas presented in this last chapter have been published in [41].

## 6.2 Usual approaches to the adequacy problem

When confronted to this problem of adequacy between the intuitive notion and the formalised one, a first possibility is to formalise the notion multiple times, with different predicates, and to prove that they are equivalent. With our example, that means that we need to find another formalisation `isSorted'` of being sorted, and to prove the following lemma :

```
pred_equiv : ∀ (T:Type) (l:List T),
(isSorted l) ↔ (isSorted' l)
```

This approach aims at increasing the confidence in our formal definitions by assuming that if we've managed to define multiple times the same notion, then we have probably succeeded to define the notion we wanted. The biggest problem with this approach is to be able to find some alternative formalisations that are sufficiently different from the original one. Obviously, if the new formalisations are too similar to the original one (and in the worst case the new ones are just syntactical variants of the first one), then we won't gain any guarantee. The ideal would be to capture the same notion by using very different points of view, and we will show in section 6.3 an original approach for doing so.

In order to gain confidence in the formal specifications we write, another traditional approach is to test the predicate on some values. That consists in defining a few terms, usually by hand, for which we know if the predicate should hold or not, and to prove that the predicate effectively holds when it should, and that it does not when it should not. For example, with the predicate `isSorted` defined above, we can prove that it holds on the list `[3, 5, 7]` that we know sorted.

```
isSorted_test1 : isSorted natIsOrdered [3, 5, 7]
isSorted_test1 =
  let p1 : (3 <= 5) =
         tryDec (lowerEqDec natIsOrdered 3 5) in
  let p2 : (5 <= 7) =
         tryDec (lowerEqDec natIsOrdered 5 7) in
    ConsSorted 3 5 [7]
      (ConsSorted 5 7 [] (SingletonIsSorted _ 7) p2) p1
```

This test is a *test done by proof* : we show that the predicate holds on some specific value, here `[3, 5, 7]`, by doing the proof. We can go a step forward by removing the need of doing these specific proofs by hand, because in this case, the predicate `isSorted` can be decided : there exists an algorithm that produces a proof of `(isSorted l)` if appropriate, or a proof of `(not (isSorted l))` otherwise :

```
decideIsSorted : (Tord : Order T) → (l:List T)
                → Dec(isSorted Tord l)
decideIsSorted Tord [] = Yes (NilIsSorted Tord)
decideIsSorted Tord [x] = Yes (SingletonIsSorted Tord x)
decideIsSorted Tord (h1::(h2::t))
  with (lowerEqDec Tord h1 h2)
  | (Yes h1_lower_h2) with (decideIsSorted Tord (h2::t))
      | (Yes h2_tail_sorted) = Yes
          (ConsSorted h1 h2 t h2_tail_sorted h1_lower_h2)
      | (No h2_tail_not_sorted) = No [...]
  | (No h1_not_lower_h2) = No [...]
```

Now, in order to do *tests by proof*, we can simply run the decision procedure.

```
isSorted_test1' : Dec (isSorted natIsOrdered [3,5,7])
isSorted_test1' = decideIsSorted natIsOrdered [3,5,7]
```

And if we evaluate `isSorted_test1'`, the system will answer `Yes` and a proof of `isSorted [3, 5, 7]`, which means that the predicate has passed this test. With this technique, we can run semi-automatically a few tests on the predicate `isSorted`. It is semi-automatic in the sense that we still have to define by hand some terms that we know sorted or unsorted but we can let the machine produce the proof that the predicate holds or not on these specific values. This is not too bad –and this is in fact all of what is usually done, when it is actually done– but we would like to have a stronger guarantee, and not only that the predicate will coincide with our intuitive notion on a couple of tested terms.

## 6.3 Predicate testing by automatic generation of terms

We can operate an interesting change of point of view by generating the set of terms that we precisely wanted to describe with the predicate. It is often easier to generate examples of a notion than it is to precisely define it. With our example of sorted lists, that means that we need to define the same notion of being sorted, but this time with a definition example-based. Writing a function that produces all the sorted lists might be a bit more difficult than just giving a few examples, but this complicated function will have the main advantage of being so different from the predicate that if the two notions agree, then we will have gain a great confidence on the predicate. We decide to use coinduction and the type `Stream` (a coinductive version of `List`, potentially infinite) in order to generate –with what we call a generator– all the sorted lists of size *n*.

```
generateSortedList : (T:Type) → (recEnu:RecEnum T)
  → (Tord : Order T) → (n:Nat) → Stream (List T)
```

To do so, the type `T` needs to be recursively enumerable, which means that there must exist a computational map `Nat → Maybe T` with the condition that this map is surjective, which means that any value of type `T` should be hit at least once by the map :

```
map_is_surjective : (y:T) →
    (x:Nat ** (computableMap x = Just y))
```

We want to check that this function and the predicate coincide. Since the predicate `isSorted` is decidable with `decideIsSorted`, we can automatically check whether the generated sorted lists of size *n* are automatically sorted (according to `isSorted`) by running this decision procedure. But since there can be infinitely many generated sorted lists of size *n* when *n*>0, we will only check that the generator and the predicate coincide on a finite observation[1] of the resulting stream. The key point is that this observation can be arbitrary big, and the bigger it is, the better the guarantee is about `isSorted`.

We show how the observation is made on an example. Let `T` be a type that only contains three constant values `A`, `B` and `C`. Since `T` is finite (it's an enumeration), it is therefore obviously recursively enumerable. We define on it the strict order `A < B < C`. Let's automatically generate the first *m* sorted lists of `T`, of size *n*, by unfolding *m* times the result of `generateSortedList`.

```
testGenerator : (m:Nat) → (n:Nat)
               → Maybe(Vect m (List T))
testGenerator m n =
  let x = generateSortedList T TisRecEnu TisOrdered n
    in unfold_n_times x m
```

We can ask for the first 8 sorted lists of size 4 by evaluating `testGenerator 8 4`:

```
Just [[A, A, A, A], [A, A, A, B], [A, A, A, C],
      [A, A, B, B], [A, A, B, C], [A, A, C, C],
      [A, B, B, B], [A, B, B, C]]
      : Maybe (Vect 8 (List T))
```

Now, instead of simply generating the first *m* sorted lists, we run the decision procedure on all of these *m* tests in order to know if the predicate and the generator agree on this portion. The result will be a vector of *m* booleans.

```
testSorted : (m:Nat) → (n:Nat) → Vect m Bool
testSorted m n =
  let x = generateSortedList T TisRecEnu TisOrdered n in
  let y = Smap
    (\l => let res = decideIsSorted TisOrdered l in
```

[1] A finite observation of a stream, also called approximation at rank *m* of a stream, is a vector of size *m* that has the same *m* first elements than the stream.

```
              case res of
                Yes _ => True
                No _ => False) x in
       unfold_n_times_with_padding y m True
```

And we can inspect the result of running the first 8 tests of size 4 by evaluating `testSorted 8 4`.

```
[True, True, True, True, True, True, True, True]
 : Vect 8 Bool
```

When we want to test `isSorted` on a large number of tests, we might not want to inspect manually the result of each test. We can write a function

```
testSorted_result : (m:Nat) → (n:Nat) → Bool
```

that calls `testSorted m n` and does the boolean `And` on each element of the resulting vector. Now, we can for example test the predicate on the first 50 sorted lists of size 9 by running `testSorted_result 50 9` and if we do so we get the overall result `True` which means that the predicate agrees with the generator on all these 50 tests.

However, if the predicate `isSorted` has been incorrectly written, then the result of this test might inform us that there is something wrong with the formal specification. For example, if we've forgotten the third constructor `consSorted` in the definition of `isSorted`, then the result of `(testSorted 8 4)` will be `False`, which means that at least one of the produced list is not seen as sorted according to `isSorted`, and we will therefore know that this predicate does not capture our intuitive notion of sortedness.

In order to go a step forward, we can decide to replace the tests on the predicate by proofs. Instead of testing the predicate on a finite subset of all the generated sorted lists as we just did, we can try to prove that any of the automatically generated sorted list is provably sorted according to `isSorted`.

```
generated_implies_pred_holds : {T:Type}
   → (recEnu:RecEnum T) → (Tord : Order T)
   → (n:Nat)
   → (All (generateSortedList T recEnu Tord n)
          (\l => isSorted Tord l))
```

*Proof.* By induction on $n$. When $n$ is zero, there is only one sorted list generated, which is the empty list, and we know that the empty list is sorted thanks to the constructor `NilIsSorted`. When `n` is some successor `(S pn)`, we know by using recursively the lemma on the smaller value `pn` that all sorted lists of size `pn` are sorted according to `isSorted`. Since the stream of all sorted lists of size `(S pn)` has been made from the stream of all sorted lists of size `pn` by adding to all of them –on the head position– an element lower or equal to their respective current heads, we know that the property has been preserved at the higher rank. □

This lemma has the advantage of not requiring the predicate to be decidable, whereas this was needed when we automatically tested the predicate on a finite observation. However, one could object that this lemma is itself built by using a predicate, `All`, and that we can't necessary trust blindly such a specification. The answer is that no guarantee is perfect, and all we can do is to add some guarantees, but there is necessarily always something to trust. Moreover, this new kind of specifications and proofs –about the predicate itself– uses more primitive components like streams and the predicate `All`, and these components can be provided once and for all. If they are part of some standard library being used intensively, there is very low risk that they do not capture the desired semantics.

## 6.4 Summary

In this chapter, we've discussed the confidence that we can have in formal proofs, and we've presented a new way to test a predicate based on an automatic generation of terms that are expected to have the desired property. This adequacy between the predicate and the generator helps in gaining confidence in the predicate. The technique presented on section 6.3 was based on a finite observation of the terms generated in potentially infinite streams, processed by the decision procedure.

We haven't been able to find much work done in the direction of predicate testing in the environment of proof assistants, but we strongly believe that this aspect is crucial, as there is absolutely no point in proving the "correctness" of a function relatively to a bad specification. In this thesis, we've automated the construction of proofs of equality and equivalence which are rather primitive notions, so we never had to question the exactitude of our specifications. However, there is more than equality and equivalence in formal certification, and it is therefore legitimate to question the validity of formal proofs in general.

The machinery developed for the running example presented in this paper is extremely specific and it is not reasonable to believe that this work should and could be done for every formal specification. What it shows is that we really need to explore how proof assistants themselves could help to gain confidence about predicates and logical formulae. A possible direction could be to build execution engines for formal specifications written in dependent type theories. Such a system would take in input a predicate and would produce some of the terms that make this predicate hold. The ideal would be to have a query system where one could ask the system to try to look if some specific terms are captured by the predicate. Since the problem of finding proofs is undecidable in the general case in most logics (not only in higher-order logics, but even in first-order logic), such a system can't be complete and entirely automatic, and therefore the user would have to help the system at times.

Another possible direction is to equip proof assistants with many robust and generic concepts (like being sorted) once and for all. That would save the user from the error prone activity of writing such primitive logical properties. Equipping proof assistants with many generic and useful concepts already available, like bricks ready to be assembled, is another current challenge to make proofs assistants really usable.

# Chapter 7

# Conclusions

**Dependent types and their usability (chapters 1-2)** The starting point of this thesis was the need of proof automations to make dependently typed languages more usable. Dependent types aren't really usable without some abstractions that programming languages have to provide (like the dependent pattern matching of Idris), and without proof automations as the ones we have built throughout this thesis, because proof obligations arise naturally, and especially with dependent types, as initially motivated on the example with binary numbers indexed over their representation in section 1.3. If we want dependent types and formal verification techniques to become part of traditional functional programming, we need to build abstractions, tools and automations that will help the users with the most repetitive tasks.

**Reflection and Proof automation (chapters 3-4)** In order to develop some automations to address some of these issues, first for natural numbers and their additions in chapter 3, and later for an entire hierarchy of algebraic structures in chapter 4, we have developed a type-safe reflection technique which made the building of these tactics much easier. The proof of correctness of the tactics, which is what we really need (see 4.7.1) is easily pulled out from the index representing the real, concrete Idris value of the reflected expressions. The hierarchy of tactics that we have built with this technique is very generic and in multiple ways : they work for many algebraic structures (semi-group, monoid, commutative monoid, group, commutative group, ring and semi-ring), for any type that behaves as one of these structures, and for any equivalence relation on this type (and not only for the propositional equality). The implementation is modular and each prover reuses the prover of the structure from which it directly inherits, thus avoiding as much as possible code and proof duplication. An interesting thing is that our approach uses an indexed representation –and thus dependent types themselves– to build automations that are needed in the presence of dependent types (but not only). However, our implementation choices are not necessarily always the best. For instance, our hierarchy of provers is more general and more reusable than the Coq ring prover, but the later is more efficient (see 4.7.5). Also, we have focused on our type-safe reflection mechanism which enables to build the proof of equivalence easily, but making a formalised proof of completeness would be very difficult, as the shape of our formal form is not enforced by a specific type. If someone were to develop again this kind of proof automations for another language, that could be something interesting to take in account from the start. Note that this is not mutually exclusive with using our type-safe reflection mechanism, as both approaches are compatible (see section 3). Also, the work which has been done for this thesis has helped to improve Idris : it was one of the first big pieces of code written in it, so it has naturally led to discovering some bugs in its implementation. It may also provide inputs for the successor of Idris (see https://github.com/edwinb/Blodwen).

**Programming with dependent types, views and indices (chapter 5)** Using such indexed types is part of a broader and fairly new usage of dependent types, presented quickly in chapter 5, where dependent types are not only used as a formal logic for encoding full proofs of program correctness, but also for building the programs themselves. In this spirit, the types are meant to guide the programmer towards an implementation easily, and without much debugging work needed afterwards. For this, some precise types need to be defined. The views presented in the chapter 5 are part of these precise types that give access to some important information that could not be easily accessed otherwise. This style of programming with dependent types does not necessary goes as far as having a formal proof of correctness, but instead tries to gain confidence on an implementation by refining the types bit by bit, to the point where the implementation is almost trivial to write and thus easy to believe.

**The limits of formal certification (chapter 6)** These new possibilities brought by dependent types are not yet broadly used. In dependently typed systems, dependent types are more often seen as a way to reason formally (and they are indeed very good for this task), but are rarely used as a tool for helping the programming activity. Let's recall the conclusion of chapter 6 : a full proof of correctness mechanically verified is a nice thing, but is also often over-rated when it comes to software development (the situation is different if the goal is the formalisation of mathematics). Even for a reasonably sized application, certifying formally its correctness involve very complex formal specifications, and the question is why should we trust more these definitions than the actual code? In order to bring some guarantee, the formal specification is supposed to be simpler than the code, but when the formal specification is not trivial to follow (and the specification is rarely trivial for a real program), the guarantee carried by a formal proof is not perfect. That's the limit of formal specification, presented in the last chapter. This limit does not encourage us to stop working with types and formal proofs. Instead, we are encouraged to *not only* develop formal proof of correctness, but also to explore ways to express many links between data as often as possible, as done in chapter 5 on various examples. It also encourages us to mix different approaches, combining proofs, tests, and other formal methods. Programming with dependent types still has a long way to go before becoming mainstream in functional programming.

# Bibliography

[1] Jean-Raymond Abrial, Matthew K. O. Lee, David Neilson, P. N. Scharbach, and Ib Holm Sørensen. The b-method. In Søren Prehn and W. J. Toetenel, editors, *VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 21-25, 1991, Proceedings, Volume 2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer, 1991.

[2] Thorsten Altenkirch. Naïve type theory, 2017.

[3] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2018.

[4] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *J. Formalized Reasoning*, 7(2):1–89, 2014.

[5] HP Barendregt. Lambda calculi with types, handbook of logic in computer science (vol. 2): background: computational structures, 1993.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[7] Nicola Botta, Patrik Jansson, and Cezar Ionescu. Contributions to a computational theory of policy advice and avoidability. *Journal of Functional Programming*, 27, 2017.

[8] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[9] Edwin Brady. *Type-Driven Development with Idris*. Manning Publications, 2017.

[10] Edwin Brady and Kevin Hammond. A dependently typed framework for static analysis of program execution costs. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19-21, 2005, Revised Selected Papers*, volume 4015 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2005.

[11] Edwin Brady, James McKinna, and Kevin Hammond. Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types. In Marco T. Morazán, editor, *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007.*, volume 8 of *Trends in Functional Programming*, pages 159–176. Intellect, 2007.

[12] Jacques Carette and Russell O'Connor. Theory presentation combinators. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International*

*Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 202–215. Springer, 2012.

[13] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[14] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[15] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[16] P Crégut. Une procédure de décision reflexive pour un fragment de l'arithmétique de presburger. In Journées Francophones des Langages Applicatifs. Technical report, 2004.

[17] David Delahaye. A proof dedicated meta-language. *Electr. Notes Theor. Comput. Sci.*, 70(2):96–109, 2002.

[18] David Delahaye and Micaela Mayero. Field, une procédure de décision pour les nombres réels en coq. In Pierre Castéran, editor, *Journées francophones des langages applicatifs (JFLA'01), Pontarlier, France, Janvier, 2001*, Collection Didactique, pages 33–48. INRIA, 2001.

[19] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 139–145. Springer, 1993.

[20] Gottlob Frege. *The Foundations of Arithmetic, 1884, translated from the German by JL Austin*. Evanston, Ill.: Northwestern University Press, 1980.

[21] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and types*, volume 7. Cambridge University Press Cambridge, 1989.

[22] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005.

[23] A. Heyting. Die formalen regeln der intuitionistischen logik, english trans. by a. rocha in mancosu [1998, 311-327]. *Sitzber. Preuss. Akad. Wiss. (phys.-math. Klasse)*, pages 42–56, 1930.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[25] W.A. Howard. The formulae-as-types notion of construction. In J.R. Seldin and J.P. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, 1980.

[26] Gérard P. Huet. A complete proof of correctness of the knuth-bendix completion algorithm. *J. Comput. Syst. Sci.*, 23(1):11–21, 1981.

[27] Pepijn Kokke and Wouter Swierstra. Auto in agda - programming proof search using reflection. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 276–301. Springer, 2015.

[28] Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in agda. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2004.

[29] Zhaohui Luo. A unifying theory of dependent types: The schematic approach. In Anil Nerode and Michael A. Taitslin, editors, *Logical Foundations of Computer Science - Tver '92, Second International Symposium, Tver, Russia, July 20-24, 1992, Proceedings*, volume 620 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1992.

[30] Gregory Malecha, Adam Chlipala, and Thomas Braibant. Compositional computational reflection. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference ITP, 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 374–389. Springer, 2014.

[31] Conor McBride. First-order unification by structural recursion. *J. Funct. Program.*, 13(6):1061–1075, 2003.

[32] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

[33] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[34] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[35] Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

[36] Nicolas Oury and Wouter Swierstra. The power of pi. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 39–50. ACM, 2008.

[37] Ricardo Peña. An introduction to liquid haskell. In Alicia Villanueva, editor, *Proceedings XVI Jornadas sobre Programación y Lenguajes, PROLE 2016, Salamanca, Spain, 14-16th September 2016.*, volume 237 of *EPTCS*, pages 68–80, 2016.

[38] Robert Pollack. How to believe a machine-checked proof. *Twenty Five Years of Constructive Type Theory*, 36:205, 1998.

[39] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP):17:1–17:29, 2017.

[40] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008.

[41] Franck Slama. Automatic predicate testing in formal certification - you've only proven what you've said, not what you meant! In Bernhard K. Aichernig and Carlo A. Furia, editors, *Tests and Proofs - 10th International Conference, TAP 2016, Held as Part of STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings*, volume 9762 of *Lecture Notes in Computer Science*, pages 191–198. Springer, 2016.

[42] Franck Slama and Edwin Brady. Automatically proving equivalence by type-safe reflection. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, edit-

ors, *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, volume 10383 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2017.

[43] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[44] Wouter Swierstra. Sorted - verifying the problem of the dutch national flag in agda. *J. Funct. Program.*, 21(6):573–583, 2011.

[45] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: experience with refinement types in the real world. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 39–51. ACM, 2014.

[46] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014.

[47] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 307–313. ACM Press, 1987.

[48] Hongwei Xi. Applied type system: Extended abstract. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES*

*2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2003.

[49] Hongwei Xi. Dependent ML an approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

[50] Hongwei Xi. Applied type system: An approach to practical programming with theorem-proving. *CoRR*, abs/1703.08683, 2017.

[51] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: a monad for typed tactic programming in coq. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 87–100. ACM, 2013.