Verified Computing in Homological Algebra

Arnaud Spiwack

▶ To cite this version:

Arnaud Spiwack. Verified Computing in Homological Algebra. Algebraic Topology [math.AT]. Ecole Polytechnique X, 2011. English. NNT : . pastel-00605836

HAL Id: pastel-00605836
https://pastel.archives-ouvertes.fr/pastel-00605836

Submitted on 4 Jul 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse de doctorat de l'ÉCOLE POLYTECHNIQUE

ParisTech

erified Computing in Homological Algebra

A Journey Exploring the Power and Limits of Dependent Type Theory

Arnaud Spiwack

Effectuée sous la direction de

Benjamin Werner
Thierry Coquand

Défendue le 25-03-2011 devant un jury composé de

Yves Bertot (président)
André Hirschowitz
Paul-André Melliès
Julio Rubio (rapporteur)
Claudio Sacerdoti Coen (rapporteur)
Bas Spitters
Benjamin Werner (directeur)

Verified Computing in Homological Algebra

A Journey Exploring the Power and Limits of Dependent Type Theory

Arnaud Spiwack

This thesis was defended the 25-03-2011 at École Polytechnique in Palaiseau, France.

It had been carried out under the guidance of

Thierry Coquand
Benjamin Werner

It was reviewed by

Julio Rubio
Claudio Sacerdoti Coen

And was defended before a jury composed of

Yves Bertot (president)
André Hirschowitz
Paul-André Melliès
Julio Rubio
Claudio Sacerdoti Coen
Bas Spitters
Benjamin Werner

Jabberwocky

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

'Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!'

He took his vorpal sword in hand:
Long time the manxome foe he sought—
So rested he by the Tumtum tree,
And stood awhile in thought.

And as in uffish thought he stood,
The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!

One, two! One, two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with its head
He went galumphing back.

'And hast thou slain the Jabberwock?
Come to my arms, my beamish boy!
O frabjous day! Callooh! Callay!'
He chortled in his joy.

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

Lewis Caroll, *Through the Looking-Glass and What Alice Found There*

The best book on programming for the layman is "Alice in Wonderland"; but that's because it's the best book on anything for the layman.

Alan Perlis

Acknowledgement

It is no simple task to reflect on the last few years of one's life. Especially when these pretty intense years have seen me, and my life, change in more way than one. There are many people who have been involved, one way or another, into the making of this thesis. More than I can gather on this page. These many people I would like to thank, though, as this episode of my life comes to an end. If you do not appear here but think you might have: please accept my dearest thanks.

I cannot thank enough my advisers, Thierry Coquand and Benjamin Werner, who taught me a lot throughout these years. They helped me through this thesis with a lot of patience, and I must confess that I probably did not make that easy every day.

Besides Thierry and Benjamin, when I had a question to ask – I often had – I would turn to Bruno Barras and Hugo Herbelin. They have all my gratitude and apologies for the time they spent helping me.

I also need to mention that this manuscript has been proof-read, several times, by Assia Mahboubi. She does not think she deserves praise. She certainly does.

Julio Rubio and Claudio Sacerdoti-Coen have accepted to referee this thesis. Tank you both. Yves Bertot, André Hirschowitz and Bas Spitters have also accepted to be member of my jury, I am grateful for it.

I like to believe that the story of this thesis has started before the beginning. When I was being taught computing science, my approach was already being shaped. Three teacher in particular were, I believe, especially influencial: Hubert Comon-Lundh, Jean Goubault-Larrecq and Jean Vuillemin.

Finally, I would like to thank those whose help was more indirect, but essential nonetheless. Florent Kirchner and Stéphane Lengrand have been friends more or less since I made my first steps in research. As more exeperience young researchers, they helped me understand this world were I was lost, at first. I had many collaborations with Matthieu Sozeau, not to mention friendship. Lisa Allali, Bruno Bernardo, Denis Cousineau and Elie Soubiran have been my *compagnons de route*: we started a thesis at the same time, in the same team, and shared office with one another for a while. Later came Vincent Siles and he who became my favorite trolling partner: Mathieu Boespflug. In Sweden I also shared office with Alexandre Buisse, Nils Anders Danielsson, Ulf Norell and Andrés Sicard. And had many fruitful discussion with Andreas Abel and Peter Dybjer.

Dear reader, if you enjoy reading my thesis, you have all these people to thank for it.

Contents

Contents 2

0 Introduction 5
- 0.1 Effective homology . . . 6
- 0.2 Coq . . . 9
- 0.3 Conventions & notations . . . 10
- 0.4 Premises . . . 11

I Mathematics to compute 13

1 A theory of sets 15
- 1.1 Bishopian toolkit . . . 16
- 1.2 Categories . . . 18
- 1.3 Choice! . . . 21

2 Homological algebra in type theory 25
- 2.1 Homology . . . 25
- 2.2 Abelian categories will not fit . . . 26
- 2.3 Preabelian categories . . . 29
- 2.4 Graded objects and chain complexes . . . 30
- 2.5 Kernels of matrices . . . 35

II Intermezzo: Down to Coq Implementation 39

3 Efficient computations 41
- 3.1 A brief history of $\mathbb{N}$ . . . 42
- 3.2 Going native . . . 45
- 3.3 Performance . . . 50
- 3.4 Related works . . . 52

4 A new tactic engine 53
- 4.1 Refinement . . . 56
- 4.2 Combining tactics . . . 59
- 4.3 About the implementation . . . 62
- 4.4 Further work . . . 62

III Faster, Higher, Stronger 65

5 On the category of sets 67
- 5.1 Topoi . . . 68
- 5.2 Quasitopoi . . . 71

5.3 Internal language 74

6 Sharing more code **83**
6.1 Categories & additive categories 83
6.2 Sets as categories and beyond 86
6.3 The category of graded objects is preabelian 90

Appendices **95**

A Coq development for Kernel computations **97**
A.1 Kernels are finite dimensional 97
A.2 Testing the program 99

B Strong subsets are classified **103**

Conclusion **107**

Bibliography **109**

Table of slogans **115**

Index **117**

Chapter Zero

Introduction

- Handbook for the recently diseased...
- Deceased.

Tim Burton & al., *Beetle Juice*

Humans' incredible ability to make mistakes is cosubstantial to our ability to interact with and communicate in a very noisy environment. It is necessary for us to be able to ignore differences in voice, and even accents. It contrasts strongly with computers' stubborn preciseness. Which is at the source of their ability to deal with extremely complex computations as if it was child play. In that sense, computers are really superhuman. But human are also supercomputer.

Communication between humans and computers is made hard because of the friction between the impreciseness and robustness of humans and the exactness and fragility of computers. However we can supplement one another in the areas we fail to be efficient at. It has become a banality that computers can help us in this communication process. In particular, we can write type systems to prevent evidently bad programs from being written. We could also mention many programmer-aiding tools like abstract interpretation or model-checking. They all have one thing in common: they do not do what we expect them to do. And there is a good reason for this: we expect them to solve undecidable problems. Computers cannot do that – though they can try to some extent – but humans somewhat can, by providing what computers lack: insight. In other words, humans can help the computer help them communicate with it.

Hence, to go further than fully automated tools, we need to lay out an interface between the human and the computer to allow humans to participate in the computing process. Mathematics is a good candidate to that purpose. Indeed, mathematics is designed to reflect humans' insight, yet they are expressible as rules to be applied and checked mechanically – which can be understood by a computer. Therefore, if mathematics is not, properly speaking, a language, they are a communication construct which is common to humans and computers. From that point of view, problems computers are supposed to help solving are expressed as a mathematical statement, and the human then provides a proof of it. This proof can be very fine grain, making direct use of simple rules in the style of formal logic systems. Or it can be

very coarse grain where the computer does most of the work and the human only gives a few hints. Or anything in between.

As mathematical statement to solve become more elaborate, proofs get typically more and more tedious. And "tedious" is the job of computers. So we might consider desirable that the computer would help us help it help us communicate with it.

Of course we can go on with this forever, there is some fixed point here. Pursuing this idea leads – among other things – to dependent type theories, where the activities of writing programs, mathematical statement and proofs are equated to some degrees.

In this manuscript we describe how to rebuild part of the theory of *effective homology* in the proof assistant Coq from this perspective. We have two main motivation for this. First, effective homology is a constructive theory which, in addition, has a mostly functional implementation as the computer algebra tool Kenzo [49] – two things that Coq is made to deal with. Hence an implementation in Coq ought to be quite feasible. This is a good opportunity to try and understand and discuss its limitations.

Incidentally, Kenzo happens to be a library for Common Lisp. Hence follows the proposed scheme.

The second motivation is actually twofold. First, most computer algebra tools come packaged with their own programing language. These programing language are usually fairly *ad hoc* and poorly designed. We could hope that, instead, computer algebra tools could be embedded as libraries for independently designed programming languages. Also, there is the issue of trust. Not only are computer algebra tools often made of very complex and subtle algorithms. But also, they do not necessarily guarantee anything, sometimes producing wrong results with no warning – independently of the correction of the implementation. To increase trust we can write a simple program to check the correction of the result. Even better we can prove this result-checking program correct by many a method. When the level of trust is considered acceptable then it is reasonable to accept the result of the tool as a proof argument. We can raise the trust even further by proving the computer algebra tool itself correct increasing the trust *a priori*: we know that the tool will work as a proof argument in identified cases. At the intersection lie dependent type theories. They pass as fair programming languages in which computer algebra tools can be embedded and they can be used to prove the correctness of the said tools to any degree. Additionally, as tool written and certified in a depend type theory can be used as part of proofs *inside* the type theory (this is often referred to as *reflexive proofs*) providing a new automation tool.

0.1 Effective homology

Homology is a tool in topological algebra. It consists in associating, to well behaved topological spaces, abelian groups embodying certain properties of their n-dimensional structure. The group representing dimension n is called the n-th homology group. The simplest case is for space which do not have an n-th dimensional structure, then the n-th homology group is trivial. For instance the third homology group of a sphere. The n-th homology group is also trivial in other cases, for instance the second homology group of a ball is trivial, this is because from the point of view of homology, the ball can

be turned into a point. Hence the two dimensional structure of the ball is irrelevant.

When homology groups are non-trivial they give an account of the shape of the n-dimensional structure of the space. In particular they count the number of "holes" in the space. The number of holes depends on the $n+1$-st dimension as well. For instance a circle which is not the border of a disk counts as a 1-dimensional hole, and a sphere which is not the border of a ball is a 2-dimensional hole.

Homology has been introduced as a variant of the more topological homotopy groups. Homotopy groups describe the topological structure of spaces more finely than homology groups. However, it is much more difficult to deduce useful information about homotopy group than homology groups. Algebraic topologist say that homology groups are easier to "compute", though they do not mean that information results of a computation a computer could do.

The theory of effective homology, on the other hand, is a constructive theory of homology. It gives a way to actually compute information about homology groups. Effective homology is built in such a way to reflect upon programming constructs. On the programming side, there is the tool Kenzo [49]. Kenzo computes useful descriptions of homology groups, so that the information is readily available.

More precisely, the so called *structure theorem for finitely generated abelian groups* (a.k.a. fundamental theorem of finitely generated abelian groups) states that finitely presented abelian groups are (up to isomorphism) of the form:

$$\mathbb{Z}^n \oplus \mathbb{Z}/_{q_1\mathbb{Z}} \oplus \mathbb{Z}/_{q_2\mathbb{Z}} \oplus \dots \oplus \mathbb{Z}/_{q_t\mathbb{Z}}$$

Where the q_i are powers of (not necessarily distinct) prime numbers. The numbers n (called the rank) and t and the coefficients q_i are unique (up to permutation of the indices). Kenzo computes this particular form for homology groups. In the case of homology group, the number n is the number of holes mentioned above.

Typically, Kenzo takes as input the description of a particular kind of space called a simplicial set

```
> (setf torus (crts-prdc (sphere 1) (sphere 1)))
[K33 Simplicial-Set]
```

(here the *torus* is defined as the cartesian product $\mathrm{S}^1 \times \mathrm{S}^1$ of two circles)

and computes the required homology groups.

```
> (homology torus 0 4)

Homology in dimension 0 :

Component Z


Homology in dimension 1 :

Component Z

Component Z


Homology in dimension 2 :

Component Z


Homology in dimension 3 :
```

Which reads as "the torus has homology groups $\mathbb{Z}$ in dimensions 0 and 2, $\mathbb{Z}^2$ in dimension 1 and the trivial group in dimension 3". The torus is a 2-dimensional space, it is no surprise that the third homology group is trivial. The first homology group is $\mathbb{Z}^2$ meaning it has two holes, this is because it is hollow: the void inside the torus's tube counts as a hole.

Kenzo can also take as input other topological objects, like simplicial groups – a variant of simplicial sets. Or directly a chain-complex – the principal object of study of homological algebra – which is devoid of topological characteristics: it is simply a collection of abelian groups with homomorphisms between them. Kenzo can also be used to compute homotopy groups in some cases.

As far as this manuscript is concerned, homological algebra is only a problem of abelian groups and chain-complexes. The precise definitions involved are given in Chapter 2. For everything that does not appear in this manuscript, the curious reader can refer to Allen Hatcher's *Algebraic Topology* [30] which is a comprehensive introduction to homological algebra – and more generally to algebraic topology.

The work presented here consists in giving a good mathematical presentation – with adequate computational properties – of the theory concerning chain-complexes and homology groups suitable for, and implemented in, Coq. We shall make sure that this presentation in flexible enough to incorporate *homology with coefficient* (that is, where every occurrence of "abelian group" is replaced by "module" over a specified ring). The next step would be to consider finite presentations to be able to complete the same sort of computations as Kenzo, albeit only from chain-complexes. An even further step will be to incorporate topology into the picture.

0.2 Coq

As most system based on some flavour of dependent type theory, Coq can be seen either as a mathematical system or as a programming language. As a mathematical system it is constructive, and has a primitive notion of computation but none of a set. As a programming language, it is a variant of ML with dependent types. This distinction does not really matter here, as this manuscript adopts both views simultaneously.

In a sense, though, both views correspond to different styles. When **forall** A:**Type**, A→A is seen as the type of function from A to A for an arbitrary A, we tend to write its inhabitants as:

```
Definition id : forall A:Type, A→A := fun x : A ⇒ x.
```

or, more concisely, but equivalently:

```
Definition id (A:Type) (x:A) : A := x.
```

On the other hand, when seen as the statement that "A implies A, for any proposition A", then we would write a proof of $\mathrm{forall}\, A : \Omega, A \to A$ interactively as follows (to the right is the feedback reported by Coq).

```
Lemma id : forall A:Prop, A→A.     ⊢ forall A:Prop, A→A
  intros A x.                      A:Prop, x:A ⊢ A
  apply x.                         Proof completed.
Qed.
```

In fact both methods do precisely the same thing. And it make perfect sense, when needed, to program using the interactive system (the instructions are called *tactics*) or to prove a statement using the programming style.

To make matter worse (or, rather, more fun), there is a way to mix both styles which is labelled with the keyword **Program** [50]:

```
Program Fixpoint fact (n:nat) : { p:nat | p ≥ 0 } :=
  match n with
  | 0 ⇒ 1                       2 obligations remaining
  | S n' ⇒ n*(fact n')
  end.
Next Obligation.                ⊢ forall n : nat, 0 = n → 1 ≥ 0
  intros n _.                   n:nat ⊢ 1 ≥ 0
  apply le_n.                   Proof completed.
Qed.                            1 obligation remaining
```

For computing the value of an expression, Coq uses a so called *strong reduction*. That is, a function like **fun** x⇒1+1 reduces to **fun** x⇒2 whereas in usual programming languages, **fun** x⇒1+1 would be a value, and not compute its body until it is passed an argument. Strong reduction is not really useful for programming purposes, however, from a logical side, it corresponds to *cut elimination*.

Incidentally, it has an influence on typing. Coq has dependent types, that is types with bits of programs in them. For instance one can have a type family A : nat → **Type**, then A (1+1) is a type, and so is A 2. These two examples are

a priori distinct types, however there is a typing rule of Coq, called *conversion* makes them equal. Specifically, the conversion states that two types which have the same *normal form* are considered the same. Hence typing is sensitive to the details of the reduction, and strong reduction means more typable expressions than a weaker one.

0.3 Conventions & notations

The mathematics, in this manuscript, are presented in an informal adaptation of the type theory of Coq. We shall consider it more closely momentarily. But before that we would like to raise a small issue. The first thing we will do is to build a set theory atop our type theory. Most keywords, and many notations, of type theories and set theories conflict. Though we can usually get away with the overloading of terms, sometimes it can highly obscure the discussion. The archetypal such sentence would be along the line of: "f is not a function, it is only a function". The reader should agree that the meaning of this sentence is fairly hard to grasp. In such situation we usually keep the name of the set theoretical notion, and change the name of the type theoretical one. In the case at hand the type theoretical notion of function will be called a *map* while set theory retains the use of the word *function* and the above sentence now reads: "f is not a function, it is only a mapping" (Section 1.1 will shed light on the particular sentence).

To choose new names we try to exploit mathematical or English synonyms of the word we are substituting.

Back to our type theory. It features a dependent product $\prod_{x\,:\,\mathrm{A}}\mathrm{B}$ whose inhabitants are maps; in the event B does not depend on x (which is written, generically, $\prod_{_\,:\,\mathrm{A}}\mathrm{B}$), then we may write $\mathrm{A} \rightarrowtail \mathrm{B}$ instead. We also need a dependent sum $\sum_{x\,:\,\mathrm{A}}\mathrm{B}$ whose inhabitants are pairs. We also suppose a type product $\mathrm{A} \times \mathrm{B}$ (equivalent to $\sum_{_\,:\,\mathrm{A}}\mathrm{B}$, though typically not implemented as such) and a type sum $\mathrm{A} + \mathrm{B}$ whose inhabitants are of the form $\iota_1\, a$ or $\iota_2\, b$.

In addition, we give ourselves a type **Type** of types. So that the system stays consistent, we have to assume that **Type** is predicative. In particular **Type** is not its own type. We shall alleviate the burden of thinking about these matters by ignoring this fact and promising that all the types we write can be stratified (which is, incidentally, also what Coq does).

Our type theory also has an impredicative type Ω of propositions. The type of Ω is **Type**. A proposition is simply the type of its proofs: it is built out of the above constructions. However, we will follow conventional logic notations when writing a proposition. Hence, we write $\forall x{:}\mathrm{A}.\,\mathrm{P}$ for $\prod_{x\,:\,\mathrm{A}}\mathrm{P}$, $\mathrm{A}\wedge\mathrm{B}$ for $\mathrm{A} \times \mathrm{B}$ and $\mathrm{A}\vee\mathrm{B}$ for $\mathrm{A} + \mathrm{B}$. The case of the dependent sum is a bit trickier: it can be seen as a subtype of a type A, which we write $\sum_{x\,:\,\mathrm{A}}\mathrm{P}\,x$ and like an existential quantification in which case we write $\exists x{:}\mathrm{A}.\,\mathrm{P}\,x$ the former being of type **Type** and the latter of type Ω.

We also use types $\mathbb{N}$ and $\mathbb{Z}$ of natural numbers and integers respectively, together with the usual arithmetic operations, and a type $\mathbb{B}$ of boolean, whose inhabitants are true and false.

We shall define new types using labelled products (so-called *records*) which

we write:

$$\begin{array}{l}\mathrm{Name}\\ \left\Vert \begin{array}{l} l_1 : \mathrm{A}_1 \\ l_2 : \mathrm{A}_2 \\ l_3 : \mathrm{A}_3 \end{array}\right.\end{array}$$

This reads "Name is defined as a product type with 3 components each of which is named l_i and has type A_i". Also note that A_2 can mention l_1 (and A_3 can mention both l_1 and l_2), so Name is equivalent to $\sum_{l_1 \,:\, \mathrm{A}_1}\sum_{l_2 \,:\, \mathrm{A}_2}\mathrm{A}_3$ (or, more briefly: $\sum_{\substack{l_1 \,:\, \mathrm{A}_1\\ l_2 \,:\, \mathrm{A}_2}}\mathrm{A}_3$).

Finally, we can define maps – *e.g.* the identity – with the syntax $\lambda x.\, x$. Alternatively, we can define a function f by case:

$$\begin{array}{lll} f : \mathrm{A} + \mathrm{B} \to \mathrm{B} + \mathrm{A} & & \\ f \quad \iota_1\, a & = & \iota_2\, a \\ f \quad \iota_2\, b & = & \iota_1\, b \end{array}$$

What exactly is covered by a type theory is very sensitive to small details, especially in the typing of dependent product and the allowed reductions. We shall purposely not specify these details to allow more freedom in what we can express in the manuscript. However, as a rule, the material in this manuscript has been verified in Coq (and in particular can typed in Coq), unless we specify otherwise. Hence, apart from these corner case, we are working in an informal version of Coq's type theory, whose details can be found at [4].

0.4 Premises

The work presented in this manuscript was done under some additional constraint. The main goal being to test dependent type theories in general, and Coq in particular, we did not want to just try and verify as much as possible. Instead we wanted to try and write proofs in a *satisfactory way*. This is of course subjective, let us make this claim more precise.

First, writing proof in a computer checked environment means that difficulties coming from the programming world spill over to toe proof world. In particular we want to share proofs as much as possible, avoiding duplication. This may be considered even more important than in traditional programing, as proofs may be longer and more tedious than bare programs. Also we want to rely on abstractions a lot. Abstraction is both part of traditional mathematics and traditional programming, but arguably is of even greater importance for the latter. In pen and paper mathematics, painful details can be simply omitted, the purpose of abstraction is more often to generalise results to a larger class of objects. In programming, however details must be dealt with, abstraction is principally a mean to get them contained so that they do not contaminate all the code. In this manuscript abstraction serves as both a means to generalise (*i.e.* share code) and a mean to avoid boilerplate. This makes abstraction a somewhat more central issue than in traditional mathematics.

Also we do not want to refrain to use dependent types. Dependent types are tricky to use and many legitimately shy away from them. This leads to a textual separation of programs (using ML style types) and proofs which we would like to avoid as much as possible. Indeed, dependent type like constructs are quite natural, they are common in the mathematical discourse. However, if in pen and paper mathematics we can use them as a figure of speech, in computer checked mathematics there is no such thing as a figure of speech. Another, arguably more important, reason is that dependent type theories advertise a style of programming where mathematical statements are included inside types by means of dependent types. One of our main motivation is to test how much it can be done in practice.

We shall generally view these rich mathematically enhanced types as actual mathematical statements, and programs as proofs thereof. This forces to consider the computational efficiency of proofs. Efficiency is the last of the premises which guide us in this work. Under the scrutiny of efficiency, mathematics take a somewhat different flavour and we shall rediscover many concepts with a new eye.

Part One

Mathematics to compute

Chapter One

A theory of sets

> I have often been impressed by the cleverness of my own first solutions; invariably the joy of the subsequent discovery how to streamline the argument was tempered by a feeling of regret that my cleverness was unnecessary after all. It is a genuine sacrifice to part from one's ingenuities, no matter how contorted. Also, many a programmer derives a major part of his professional excitement from not quite understanding what he is doing, from the daring risks he takes and from the struggle to find the bugs he should not have introduced in the first place.
>
> Edgster Dijkstra, *The threats to computing science (EWD 898)*

HE traditional body of mathematics is built upon Zermelo-Fraenkel set theory (ZF). This is, however, unsuitable for our ambition to write programs in a mathematical style. It is not a matter of constructiveness. Indeed there are constructive flavours of ZF [6]. It is rather than the meaning of mathematical objects in ZF is too alien to what is required for programming. It sums up in two points. First, functions are encoded as relations, which precludes any computations, we would rather have them be some description of computations (maps). Also, as equality is prescribed *a priori*, data has to be encoded such as to conform to equality, whereas programming implies choosing the representation of the data for efficiency purposes.

A solution to this mismatch is found in Errett Bishop's seminal book *Foundation of Constructive Analysis* [13]. Where sets are defined as

> The totality of all mathematical objects constructed in accord with certain requirements is called a *set*. The requirements of the construction, which vary with the set under consideration, determine the set. Thus the integers are a set, the rationnal numbers are a set, and the collection of all sequences each of whose terms is an integer is a set. Each set A will be endowed with a relation $=$ of equality. This relation is a matter of convention, except that it must be an *equivalence relation*.

This definition, though purposely informal, prefigures the distinction that Intentional Type Theory makes between *types* and *sets*: types are a description of elements, sets are more complex structures bearing a notion of equality.

1.1 Bishopian toolkit

Set

In term of our type theory, a set A would be defined as:

Set
$$\left\|\begin{array}{ll} \mathrm{El}\,A & : \mathbf{Type} \\ - =_A - & : \mathrm{El}\,A \rightarrowtail \mathrm{El}\,A \rightarrowtail \Omega \\ _ & : \mathrm{Equivalence}\,(- =_A -) \end{array}\right.$$

More often than not we shall use A instead of $\mathrm{El}\,A$ and omit the index of the equality when it is clear from context.

Let us dwell on this definition for a moment. Given a relation $\mathcal{R}$ on a set A, it is fairly straightforward how to build the quotient $A/_{\mathcal{R}}$: it amounts to replacing $- =_A -$ with $\mathcal{R}$. This is an important feature programming-wise: there is no need to change the representation of the data to change which data are equal. For instance, if one is interested in considering lists up to reordering of their elements (a.k.a. multisets), they can be just a lists. Considered up to reordering of their elements. This is essentially, by the way, what is done when Haskell's list monad is used as a non-determinism monad.

In zf, on the other hand, as equality cannot change, it is always the representation of data which much change to conform to it. In the case of quotienting, equivalence classes might look like a particularly odd encoding to a programmer.

On a side note, programmers are often particularly wary of what cannot be embedded in their programming languages. In particular, they usually try to avoid relying on equalities which are not decidable. Many even loathe equalities which are not *the* structural equality. These are limitations which will not apply to us. Though having a decidable – or structural – equality of course has benefits in many situations. This new found liberty might be a strong argument in favour of any system which can cope with general sets in this sense.

In the community of dependent type theories, sets in the sense of Bishop are often known as setoids. They also appear in other guises in some areas of mathematics. They can be seen as Ω-enriched groupoids, and, equivalently, as 0-groupoids. To categorically minded people, they may also remind the topos theoretic presentation of set theory.

Function

Functions from set A to set B are supposed to be maps from A to B. But, much for the same reasons that all functions from A to B cannot necessarily be lifted to be functions from a quotient of A to B, not all maps are eligible to be functions. Only those maps which *respect equality* are:

Function A B
|| $f : \mathrm{A} \rightarrowtail \mathrm{B}$
|| $_ : \forall a_1 {=_\mathrm{A}} a_2.\, f\, a_1 =_\mathrm{B} f\, a_2$

We will usually use functions as their inner map.

Functions are exactly why a structural equality is desirable: if the equality on A is structural (more generally, if it coincides with Leibniz equality), all maps can be trivially lifted as functions, otherwise the programmer bears the burden of proof. Again, this lifting issue is not an unknown problem to mathematicians, not only in the case of quotients but also in the case of algebraic structures, where the appropriate morphisms are those which respects the structure. A customary move when playing with morphisms is to move to a categorical abstraction in which everything is, by construction, morphism. We shall do it for abelian groups, however, desirable though it is, we could not achieve it in a satisfactory way in the case of sets (see Chapter 5 for details).

It is time to define our first set construction, namely the set of functions from A B (written $\mathrm{A} \longrightarrow \mathrm{B}$). Its equality is the so-called *extensional* equality:

$\mathrm{A} \longrightarrow \mathrm{B}$
|| $\mathrm{El}\,(\mathrm{A} \longrightarrow \mathrm{B}) =$ **Function** A B
|| $f = g \qquad = \forall a.\, f\, a = g\, a$

The set $\mathrm{A} \longrightarrow \mathrm{B} \longrightarrow \mathrm{C}$ (to be read as $\mathrm{A} \longrightarrow (\mathrm{B} \longrightarrow \mathrm{C})$) will be used to encode functions from A and B to C. Equivalently, we could have chosen the set $(\mathrm{A} \times \mathrm{B}) \longrightarrow \mathrm{C}$ – to be defined below – but the former is more customary in type theory.

Subset

A function $f{:}\mathrm{A} \longrightarrow \mathrm{B}$ is said to be injective when $\forall x\, y : \mathrm{A}.\, f\, x = f\, y {\to} x = y$. A subset of B is defined as being any such injection, like subobjects in a category are defined to be any monomorphism.

Part

Given a set A, a set of functions of particular interest is the set $\mathrm{A} \longrightarrow \Omega$ of predicate over A, where Ω is considered up to equivalence:

Ω
|| $\mathrm{El}\,\Omega \ = \Omega$
|| $p = q = p \iff q$

A predicate P can be seen as a subset of A. To emphasise this point of view, we shall sometimes say *part* instead of predicate, and write $x \,\varepsilon\, \mathrm{P}$ for $\mathrm{P}\, x$. Any part P defines a subset, called a strong subset:

$\{x : \mathrm{A} \mid x \,\varepsilon\, \mathrm{P}\}$
|| $\mathrm{El}\,\{x : \mathrm{A} \mid x \,\varepsilon\, \mathrm{P}\} = \sum_{x\,:\,\mathrm{A}} x \,\varepsilon\, \mathrm{P}$
|| $(x, _) = (y, _) \qquad = x =_\mathrm{A} y$

Given a function $f{:}\mathrm{A} \longrightarrow \mathrm{B}$, we can define its image $\mathfrak{Im}\, f{:}\mathrm{B} \longrightarrow \Omega$ as $b\ \varepsilon\ \mathfrak{Im}\, f{=}\exists a{:}\mathrm{A}.\, b = f\ a$.

Cartesian product

Set also have a have a cartesian product:

$$\begin{array}{l} \mathrm{A} \times \mathrm{B} \\ \left\| \begin{array}{ll} \mathrm{El}\,(\mathrm{A} \times \mathrm{B}) & = \mathrm{A} \times \mathrm{B} \\ (a_1, b_1) = (a_2, b_2) & = a_1 = a_2 \wedge b_1 = b_2 \end{array} \right. \end{array}$$

Set sum

And a sum, sometimes known as disjoint union whose carrier is the type theoretic $\mathrm{A} + \mathrm{B}$ and the equality is:

$$\begin{array}{llll} (\text{-} =_{\mathrm{A}+\mathrm{B}} \text{-}) : (\mathrm{A} + \mathrm{B}) \rightarrowtail (\mathrm{A} + \mathrm{B}) \rightarrowtail \Omega \\ (\text{-} =_{\mathrm{A}+\mathrm{B}} \text{-}) \quad \iota_1\, a \quad \iota_1\, a' & = & a = a' \\ (\text{-} =_{\mathrm{A}+\mathrm{B}} \text{-}) \quad \iota_2\, b \quad \iota_2\, b' & = & b = b' \\ (\text{-} =_{\mathrm{A}+\mathrm{B}} \text{-}) \quad _ \quad _ & = & \bot \end{array}$$

Numbers & booleans

The sets $\mathbb{N}$ of natural numbers, $\mathbb{Z}$ of integers and $\mathbb{B}$ of booleans are lifted canonically to sets in the obvious way. For instance the equality on $\mathbb{N}$ can be computed as

$$\begin{array}{lllll} (\text{-} =_{\mathbb{N}} \text{-}) : \mathbb{N} \rightarrowtail \mathbb{N} \rightarrowtail \Omega \\ (\text{-} =_{\mathbb{N}} \text{-}) & 0 & 0 & = & \top \\ (\text{-} =_{\mathbb{N}} \text{-}) & 0 & _ & = & \bot \\ (\text{-} =_{\mathbb{N}} \text{-}) & _ & 0 & = & \bot \\ (\text{-} =_{\mathbb{N}} \text{-}) & n & p & = & (n-1) {=_{\mathbb{N}}} (p-1) \end{array}$$

From now on, we shall consider these three as sets.

1.2 Categories

Homological algebra, in particular effective homology, fit very well in a categorical framework. From a programming perspective, categories will play the role of *abstraction barriers* hiding the unnecessary details of, say, the definition of abelian groups to the algorithms computing homology groups.

Category

Although Bishop does not give such a definition, there is a notion of category which follows naturally from his definition of sets and deserves to be called

Bishop categories.

$$
\begin{array}{l}
\textbf{Cat} \\
\left\| \begin{array}{ll}
\mathcal{O} & : \textbf{Type} \\
\mathrm{Hom} & : \mathcal{O} \rightarrowtail \mathcal{O} \rightarrowtail \textbf{Set} \\
1_{\mathrm{A}} & : \mathrm{Hom\,A\,A} \\
-;- & : \prod_{\mathrm{A\,B\,C}\,:\,\mathcal{O}} \mathrm{Hom\,A\,B} \longrightarrow \mathrm{Hom\,B\,C} \longrightarrow \mathrm{Hom\,A\,C} \\
_ & : f;(g;h) = (f;g);h \\
_ & : f;1 = 1;f = f
\end{array} \right.
\end{array}
$$

Notice the order of the composition: $f;g$ (often written simply fg) corresponds to the arguably more usual $g \circ f$.

The sets $(\mathrm{Hom\,A\,B})$ are called homsets, their elements are called either morphisms or arrows. We usually write $\mathrm{A} \xrightarrow{f} \mathrm{B}$ instead of $f{:}\mathrm{Hom\,A\,B}$.

An important feature of these Bishop categories is that the type $\mathcal{O}$ of objects is *not* a set. Consequently, the question whether two objects are equal or not does not make sense: *equality belongs to sets*. This contrasts with the usual definition of category where there objects are ZF-sets (often there is even a set of all objects). ZF-sets can be compared. However, in many areas of category theory, it is bad practice to compare objects or even outright banned. Hence Bishop categories are arguably a better definition of categories than those based on ZF.

Additionally, as categories are not sets, there is no need to restrict the category of sets to some small set of a sort. There is a category of all sets and functions. This is supported by the type theory of Coq, where we define categories at a higher sort than sets (in a sense, categories are bigger than sets).

Monomorphism & epimorphism

In a category $\mathcal{C}$, we say that an arrow $\mathrm{A} \xrightarrow{f} \mathrm{B}$ is a monomorphism if for any two $\mathrm{X} \xrightarrow{g} \mathrm{A}$ and $\mathrm{X} \xrightarrow{h} \mathrm{A}$ with $gf = hf$, then $g = h$. We also say that f is a mono, for short, or that f is monic. We also call subobject of A a mono $\mathrm{S} \xrightarrow{s} \mathrm{A}$.

In the category of sets, the monomorphisms are the injective functions. In particular subsets coincide with subobjects.

Dually, an epimorphism is an arrow $\mathrm{A} \xrightarrow{f} \mathrm{B}$ such that for any two $\mathrm{B} \xrightarrow{g} \mathrm{X}$ and $\mathrm{B} \xrightarrow{h} \mathrm{X}$ with $fg = fh$, we have $g = h$. We also say that f is an epi, or that f is epic.

In the category of sets, the epimorphisms are the surjective functions. While less obvious than for monos, it is not hard to prove.

It is easy to prove that surjective functions are epic. Conversely, given an epimorphism $\mathrm{A} \xrightarrow{f} \mathrm{B}$, let us consider two parts of B: $\top$ and $\mathfrak{Im}\,f$. By definition of $\mathfrak{Im}\,f$, $f;\top = f;\mathfrak{Im}\,f$. Since f is epic, $\top = \mathfrak{Im}\,f$, that is, f is surjective.

Initial & terminal objects

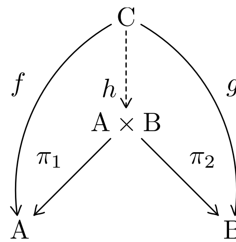
An initial object 0 of a category $\mathcal{C}$ is such that for any object A of $\mathcal{C}$, there is a unique arrow $0 \xrightarrow{0} \mathrm{A}$. There an initial object in the category of sets: the empty set.

Dually, a terminal object 1 is such that for any object A, of $\mathcal{C}$, there is a unique arrow $A \xrightarrow{!} 1$. The one element set is terminal in the category of sets.

Product

Given two objects A and B in a category $\mathcal{C}$, a product of A and B is an object $A \times B$ together with two arrows $A \times B \xrightarrow{\pi_1} A$ and $A \times B \xrightarrow{\pi_2} B$ such that for arrows $C \xrightarrow{f} A$ and $C \xrightarrow{g} A$, there is a unique arrow $C \xrightarrow{(f,g)} A \times B$ such that $(f, g)\pi_1 = f$ and $(f, g)\pi_2 = g$.

Phrases such as "there is only one such arrow" may benefit from a small clarification. They are to be understood relative to a given set, and mean, as in classical mathematics "for any two such arrows, they are equal". What changes from classical maths, however, is that two equal element of a set do not necessarily share the same representation. Hence, when we name an element, we choose, implicitely, a particular representation (presumably that which we believe will behave the best in programs).



This could be also stated as a product of A and B is a diagram composed of two arrows $A \times B \xrightarrow{\pi_1} A$ and $A \times B \xrightarrow{\pi_2} B$ which is *universal*. Universal definitions are common in the realm of categories. They feature an arrow such as (f, g) which has to be unique. The relevance of the uniqueness can be understood from the perspective of type theory as an extensional property. In the case of the product, the unicity of (f, g) is equivalent to the statement that for any $X \xrightarrow{x} A \times B$ we have $(x\pi_1, x\pi_2) = x$. This property is often called *surjective pairing*.

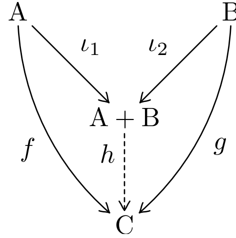
When a category $\mathcal{C}$ has a product $A \times B$ for every choice of A and B we say that $\mathcal{C}$ has *all* products. This terminology can be used with any suitable concept. We can also say that $\mathcal{C}$ has *enough* products when it does not necessarily have all products, but enough for the considered statement to make sense.

In particular the category of sets has all products: the cartesian products is a product in the categorical sense.

Coproduct

Dual to the product is the coproduct, also called sum. Namely, a coproduct of two objects A and B is an object $A+B$ together with two arrows $A \xrightarrow{\iota_1} A+B$ and $B \xrightarrow{\iota_2} A + B$ such that for any arrows $A \xrightarrow{f} C$ and $A \xrightarrow{g} C$, there is

a unique arrow $A + B \xrightarrow{[f,g]} C$ such that $\iota_1; [f, g] = f$ and $\iota_2; [f; g] = g$.



The category of sets has all coproducts, as the sum of two sets is a coproduct.

Equaliser & coequaliser

Given two arrows $A \xrightarrow{f} B$ and $A \xrightarrow{g} B$, an equaliser of f and g is an arrow $E \xrightarrow{e} A$ such that $ef = eg$. It needs to be a universal such arrow, that is for any $X \xrightarrow{x} A$ with $xf = xg$ there is a unique arrow $X \xrightarrow{x'} E$ such that $x'e = x$.

The category of sets has all equalisers: the equaliser of f and g is given by the strong subset $E = \{a : A \mid f\,a = g\,a\}$ with e the canonical injection into A, and for an x as above, x' is the lifting of x to E. Note that equalisers are necessarily monic.

Let $E \xrightarrow{e} A$ be the equaliser of $A \xrightarrow{f} B$ and $A \xrightarrow{g} B$. Let us consider two arrows $Y \xrightarrow{y} A$ and $Y \xrightarrow{z} A$ with $ye = ze$. As $yef = yeg$ there exist a unique arrow w such that $we = ye$. As both y and z qualify as such, they must be equal.

Dually, there is a notion of coequaliser: given two arrows $A \xrightarrow{f} B$ and $A \xrightarrow{g} B$, a coequaliser of f and g is an arrow $B \xrightarrow{q} Q$ such that $fq = gq$. Moreover, for any $B \xrightarrow{x} X$ such that $fx = gx$ there is a unique arrow x' such that $qx' = x$.

The category of sets has all coequalisers. It is given as $Q = B/\mathcal{R}$ with $\mathcal{R}$ the smallest relation such that for all a:A, $f\,a = g\,a$. The function q is the canonical projection onto Q and x' is the lifting of x to be of domain $B/\mathcal{R}$. Again, coequalisers must be epimorphisms.

1.3 Choice!

Principle of choice

If a function f:A $\longrightarrow$ B is surjective (*i.e.* $\forall b. \exists a. f\,a = b$), the constructive interpretation gives a map g from B to A such that $\forall b.\ f\,(g\,b) = b$, however this has no reason to be a function.

Indeed, supposing that all surjection had a preinverse, then let P be an arbitrary proposition. Consider $\mathbb{B}$ the set of booleans, and $\mathbb{S}$ the set where the base type is the booleans and where the equality is defined as

The terminology *preinverse* is borrowed from a categorical intuition: g is a preinverse of f if g followed by f is the identity. Dually, a *postinverse* is a function g such that f followed by g is the identity.

$$
\begin{array}{llll}
e : \mathbb{B} \rightarrowtail \mathbb{B} \rightarrowtail \Omega & & & \\
e & \text{true} & \text{false} & = \mathrm{P} \\
e & \text{false} & \text{true} & = \mathrm{P} \\
e & _ & _ & = \top
\end{array}
$$

Now the function

$$
\begin{aligned}
&f : \mathbb{B} \longrightarrow \mathbb{S}\\
&f \quad \text{true} \;=\; \text{true}\\
&f \quad \text{false} \;=\; \text{false}
\end{aligned}
$$

This is a variant of Diaconescu's argument[20], which proves that the principle of choice is stronger than the excluded middle in topoi.

is surjective. However, a preinverse function g would decide the truth of P, as $g\,\text{true} = g\,\text{false}$ implies $\neg\mathrm{P}$ and $\neg g\,\text{true} = g\,\text{false}$ implie P (equality on booleans is decidable). This is contradictory with the fact that provability is not decidable. The statement that all surjective functions have a preinverse is the *principle of choice*. Let us phrase it in a catchy slogan: *choice: surjections have a section*. Phrased this way – rather than referring to families of inhabited sets, the principle of choice can be read as property of a category, provided we read *epimorphisms* instead of *surjections*. *The principle of choice, hence, is not valid in the category of Bishop sets* (surjective functions are indeed the epimorphisms of the category of sets).

If f is injective, for any two equal b_1 and b_2, $g\,b_1 = g\,b_2$ follows from $f\,(g\,b_1) = f\,(g\,b_2)$. Hence g is a function.

Principle of unique choice

On the other hand, if f is also injective then g is indeed a function and actually an inverse of f.

This observation leads to *the principle of unique choice*. Functions which are both injective and surjective are called bijections. Hence we can state the the principle of unique choice as follows: *unique choice: bijections have an inverse*.

The principle of unique choice is, again, a property of a category (provided that "bijections" are replaced by "morphism which are both epic and monic").

The aforementioned principle of unique choice can be a liability rather than an asset. Indeed, the fact that a bijective function is tractable does not mean it has a tractable inverse. Typical examples of this often come from cryptography. For instance, let us consider a group G of order n and g a generator of G. The function from $\mathbb{Z}/_{n\mathbb{Z}}$ to G which maps p to g^p is bijective. However, there is no known efficient algorithm for its inverse, called *discrete logarithm*. A number of cryptographic protocols actually rely on the hope that there will never be an efficient algorithm for discrete logarithm – the most famous being the Diffie-Hellman key exchange protocol.

If the principle of unique choice is valid in our category of sets, an inverse function can be devised *automatically*. As mentioned, every known proof of bijectivity of exponentiation leads to intractable discrete logarithms. As a matter of fact, all the computational content of the inverse found using the principle of unique choice is contained in the proof of bijectivity (more precisely in the underlying proof of surjectivity).

Concretely, we are left with two choices if we want to control the computational complexity of our functions. Either we need to control the computational complexity of the proofs we write, in particular, when writing proofs of bijectivity, we devise specific inverses. Or we make arrangements so that the principle of unique choice is not valid.

The former option precludes from using mostly any form of automation, as they produce proofs whose complexity cannot be known in advance. Also, it is fairly restrictive in what can be proved, or, in other words, proofs of surjectivity which cannot be used to find an inverse, can still be used for other

purposes. Yet, we would be avoiding these. Therefore we shall choose the latter option: leaving aside the principle of unique choice.

As we have seen, the principle of unique choice is a direct consequence of the fact that given a proof of $\forall a{:}\mathrm{A}.\, \exists b{:}\mathrm{B}.\, \mathrm{P}\, a\, b$, we can extract a map $f{:}\mathrm{A} \rightarrowtail \mathrm{B}$ such that $\forall a{:}\mathrm{A}.\, \mathrm{P}\, a\, (f\, a)$. This property can be seen a reflection on the constructive nature of the proof, or as an internal *skolemisation*. It is a natural principle of systems such as the Calculus of Inductive Constructions which is a foundation of Coq. Fortunately for our application, Coq has a sort **Prop** which explicitly does not enjoy this property.

Now, our mathematics is constructive. This means that anyone inspecting a proof of bijectivity can extract an inverse from it. The important point, though, is that this cannot be done *inside* the system.

This leads to a distinction between computational and static parts of proofs. The computational parts – programs – cannot reflect on the content of static parts: they are computationally irrelevant. This does not mean that computational parts do not use static ones at all. In fact, static proofs in programs can be used for three purposes:

- Cutting branches: when some position in a program cannot be reached, it is sufficient to prove it, there is no need to provide computational code.
- Termination: a proof can assert that a particular map never loops on any input.
- Type coercion: when two types are provably identical, an element of the former can be used as one of the latter.

In this work we shall make use only of the first one of these usages. Indeed, in both other cases, as they are currently implemented in Coq, the computational content of the static proof is relevant. It would, hence, be unsuited for our goals. Not to mention that it is not really clear what it means for two types to be identical. All the maps defined here are *structurally recursive*, which makes them *obviously* terminating in the eyes of Coq.

In Coq, identity of types is taken care of by the notion of propositional equality. Yet the exact status of this equality is disputed. Additionally, from the point of view of this work, the fact that it gives a generic equality between elements of types is disturbing.

As a convention we will write x[:]A to signify that x is a proof a some static proposition A whereas the usual $x{:}\mathrm{A}$ will mean that x is a program of type A. With these notations, the definitions of sets and categories become:

Set

A	:	**Type**
- = -	:	$\mathrm{A} \rightarrowtail \mathrm{A} \rightarrowtail \Omega$
_	[:]	Equivalence (- = -)

and

Cat

$\mathcal{O}$	:	**Type**
Hom	:	$\mathcal{O} \longrightarrow \mathcal{O} \longrightarrow$ **Set**
1_{A}	:	Hom A A
-; -	:	$\forall \mathrm{A\,B\,C} : \mathcal{O}.\, \mathrm{Hom\,A\,B} \longrightarrow \mathrm{Hom\,B\,C} \longrightarrow \mathrm{Hom\,A\,C}$
_	[:]	$f; (g; h) = (f; g); h$
_	[:]	$f; 1 = 1; f = f$

The other constructions from Section 1.1 are unaltered.

As an aside, this presentation of mathematics without the principle of unique choice lacks criteria to determine that a given statement cannot be proved therein. In constructive mathematics, the standard approach is to reduce the decidability of a notoriously undecidable problem to the provability of the said statement. Similarly, the unprovable statement of our mathematics rarely subsume the full principle of unique choice, but we do not have an equivalent of the undecidable problems. To address this gap, we cannot have at the same time the principle of unique choice, classical logic and require that all function be recursive. On the other hand, any combination of two is consistent. Hence, if the conjunction of a statement A and the principle of excluded middle ($\forall P{:}\Omega.\ P \vee \neg P$) implies the decidability of some undecidable problem, then A is not provable.

While this is not entirely satisfactory, it should work fairly well. It was sufficient for the examples that showed up during the course of this work.

Let
$f : \mathbb{B} \longrightarrow \Omega$
$f \quad \text{true} \quad = \quad \top$
$f \quad \text{false} \quad = \quad \bot$
Since we are in classical logic, for any proposition P, $P = \top \vee P = \bot$, hence f is surjective (equality in the set Ω is logical equivalence). It is also injective, by case study on the possible arguments. However, if it had an inverse, it would decide the truth of propositions.

Chapter Two

Homological algebra in type theory

"What really is the point of trying to teach anything to anybody?"
This question seemed to provoke a murmur of sympathetic approval from up and down the table.
Richard continued,"What I mean is that if you really want to understand something, the best way is to try and explain it to someone else. That forces you to sort it out in your mind. And the more slow and dim-witted your pupil, the more you have to break things down into more and more simple ideas. And that's really the essence of programming. By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've learned something about it yourself."

— Douglas Adams, *Dirk Gently's Holistic Detective Agency*

ll of our formalisation of homological algebra is done in the vocabulary of category theory. This way, we abstract away many implementation details, making the proof hopefully easier. As importantly, it allows to generalise the proofs – which are, it is worth reminding, also to be viewed as program implementation – to any algebraic structure for which homology is definable. Most of the material in this chapter is presented as though we were working in the category of abelian groups, but, in fact, it applies to categories of modules and, hopefully, to fancier objects like the category of sheaves of abelian groups (though sheaves without the principle of choice remain to be scrutinised).

An early version of the work presented in this chapter previously appeared as [18].

2.1 Homology

Homology can be defined with any kind of modules over a fixed ring. But for the sake clarity, we will stick to abelian group (abelian groups are exactly the $\mathbb{Z}$-modules).

Let us consider an object C_*, called a chain complex, consisting in a family $(\mathrm{C}_n)_{n:\mathbb{Z}}$ of abelian groups together with a family $(d_n)_{n:\mathbb{Z}}$:$\mathrm{C}_n \longrightarrow \mathrm{C}_{n-1}$ of group morphisms with the property that for any n, $d_{n+1}; d_n = 0$.

In other words, that the image of d_{n+1} is a subgroup of the kernel of d_n. Since the C_n are abelian, we can take the quotient of the kernel of d_n by the

Otherwise we'd have to require that the image of d_{n+1} is normal.

image of d_{n+1}. This quotient is called the n-th homology group H_n of C_*.

The central problem of homological algebra is to find a good description for the homology groups H_n, usually a finite presentation. The chain complexes are typically derived from a well-behaved topological space, and knowledge of homology groups give information on homotopical properties of the space. For instance two spaces with the same homotopy type have the same homology groups.

We use the expression "composable pair of morphisms" casually as it makes phrasing smoother. However, in the context of Coq, morphism come with a type mentioning their source and target groups (or objects, in the case of categories). This is by no means necessary, but it is easier to manipulate in Coq, and probably better efficiency-wise too. Hence, in our Coq implementation a phrase like "for all composable pair of morphisms f and g", would rather look like "for all abelian groups A, B and C, and morphisms $\mathrm{A} \xrightarrow{f} \mathrm{B}$ and $\mathrm{B} \xrightarrow{g} \mathrm{C}$". In the Coq phrasing, composability is obvious from typing, hence has no particular status.

Connected pair

A pair f, g of composable (abelian) group morphisms is said to be connected – the terminology is ours – if their composition $f; g$ is the trivial morphism 0.

Homology

Let f, g be a connected pair of morphisms. Their homology is the quotient of the kernel of g by the image of f. An arguably more primitive notion than that of an image, is that of a cokernel: the cokernel of f is the quotient of the target group of f by its image. In term of kernel and cokernel: *The homology of f and g is the cokernel of f seen as morphism to the kernel of g.*

Exact pair

A pair f, g of connected morphisms is said to be exact if the image of f is equal to the the kernel of g. In term of homology, avoiding a reference to the image: *the pair f, g is exact if their homology is trivial* (*i.e.* the singleton group).

Exact pairs play a important role in homological algebra. To emphasise this, the homology of a connected pair is sometimes nicknamed "default of exactness".

Chain complex

A chain complex C_* – sometimes differential graded object – is the data of a family $(\mathrm{C}_n)_{n:\mathbb{Z}}$ of abelian group (dubbed "graded abelian group") together with a family $(d_n)_{n:\mathbb{Z}}:\mathrm{C}_n \longrightarrow \mathrm{C}_{n-1}$ of group morphisms such that the pairs d_{n+1}, d_n are connected. The d_n are called the differential morphisms (also, often, "boundary operators" due to the relation with a particular construction of topological space called cell-complex).

The homology H_n of the pair d_{n+1}, d_n is called the n-th homology group of C_*. Notice that the family $(\mathrm{H}_n)_{n:\mathbb{Z}}$ is itself a graded abelian group.

When abelian groups are replaces by modules, H_n is still called the n-th homology group, not the n-th homology module, for historical reasons.

2.2 Abelian categories will not fit

Homology comes in a variety of touches and feels: abelian groups and modules as presented in the previous section, but also, for the interested reader, cohomology of sheaves, for instance. It is useful to define and work with homology in a more abstract setting, in order to encompass as many variations as possible.

On a more pragmatic point of view, having an abstract approach to homology plays the equivalent of the programmatic "abstraction barriers". Even if we did not mean to reuse the code much – or, should we say, reuse the

proof – it allows to reason more purely on the homological problem, without being bothered by implementation details. On a similar tone, let us note that, contrary to what is usual practice in paper mathematics, we have a strong incentive not to represent groups and finitely presented groups in the same fashion. Hence, an abstract approach to homology help solve the unusual problem of defining homology in groups and homology in finitely presented groups.

The traditional approach, on this matter, is to introduce *abelian categories*.

Additive category

A category is called additive (sometimes preadditive) if all its homsets are endowed with an abelian group structure, and if the composition is bilinear. In other, fancier, words: an additive category is a category enriched in the category of abelian groups.

Though desirable, the notion of enriched category is not quite convenient in Coq, as it is. Hence we were not able to define additive categories this way. See Section 6.1 for further discussion.

Zero object

An object which is both initial and terminal is called a *zero* object. If a category has such an object, then there is a special kind of arrows named zero arrows, that is the arrows which "pass through" it:

$$A \longrightarrow 0 \longrightarrow B$$

By definition, there is only one such arrow for each pair of object A and B which we call *the* zero arrow from A to B and write 0. Also, if the category is additive, the zero arrow from A to B coincides with the neutral element of the abelian groups on the arrows from A to B

First notice that as there is only one arrow from 0 to B, it is the neutral element of the corresponding abelian group. Since composition is bilinear, in particular right linear, the precomposition with the unique arrow from A to 0 is the neutral element of the homset from A to B.

Kernel and Cokernel

In the presence of a zero object, we can also define a kernel (resp. cokernel) of an arrow $A \xrightarrow{f} B$ as an equaliser (resp. coequaliser) of f and 0. It is worth noticing that in an additive category, equalisers arise from kernels (and dualy, coequalisers from cokernels). Indeed a kernel of $f - g$ is also an equaliser of f and g. However, kernels and cokernels are usually considered more primitive objects than equalisers and coequalisers in linear algebra

Biproduct

In an additive category, we define a biproduct of two object A and B an object $A \oplus B$ together with injections $A \xrightarrow{\iota_1} A \oplus B$, $B \xrightarrow{\iota_2} A \oplus B$ and projections $A \oplus B \xrightarrow{\pi_1} A$, $A \oplus B \xrightarrow{\pi_2} B$ such that $\iota_1\pi_1 = 1_A$, $\iota_2\pi_2 = 1_B$ and $\pi_1\iota_1 + \pi_2\iota_2 = 1_{A\oplus B}$. $A \oplus B$ is both a product and a coproduct – hence the name "biproduct". It is noteworthy that a zero object is – in a sense that we will not make precise – a nullary biproduct (in particular it is neutral, up to isomorphism, for biproducts). Hence an additive category with all (binary) biproducts and a zero object has all biproducts (of arbitrary arity).

Normal monomorphism and epimorphism

A monomorphism (resp. epimorphism) is said to be normal if it is a kernel (resp. cokernel) of some arrow.

Abelian category

A category is said to be abelian if it is additive, has a zero element, all (binary) biproducts, has all kernels and cokernels, and all its mono and epi are normal.

The trouble, however, is that there are not enough such categories if we refuse the principle of unique choice. For instance the category of abelian groups is not abelian. This should not come at too much of a surprise, since, in abelian categories, morphisms which are both epic and monic are invertible, which sounds a lot like the principle of unique choice. Let us give, however, a more precise account of what does not work.

Suppose $A \xrightarrow{f} B$ both epic and monic. As f is epic, it is the a kernel of some arrow g. In particular $f;g = 0 = f;0$. Since f is epic, $g = 0$. The identity of B is a kernel of 0, hence of g. The universal property gives an inverse to f

- Morphisms between two abelian groups form an abelian group with respect to pointwise addition.
- Composition of abelian group morphism is bilinear.
- The one-element group is a zero object of the category of abelian groups.
- Every abelian group morphism $f{:}A \longrightarrow B$ has a kernel $\mathbf{Ker}f$ defined as the set $\{a : A \mid f\,a = 0\}$ with the group operations of A, and the canonical injection $\mathbf{ker}f$ into A.
- Every abelian group morphism $f{:}A \longrightarrow B$ has a cokernel $\mathbf{Coker}f$ defined as the set B quotiented by the equality $b_1 =_{\mathbf{Coker}f} b_2$ if and only if $\exists a{:}A.\, b_1 - b_2 = f\,a$ together with the group operations of B, and the canonical projection $\mathbf{coker}f$.
- For any two abelian groups A and B, there is a biproduct $A \oplus B$ defined as the set $A \times B$ together with component-wise group operations from A and B. The projections π_1 and π_2 are the projections of the set $A \times B$. The injections ι_1 and ι_2 are defined as $\lambda a.\,(a, 0)$ and $\lambda b.\,(b, 0)$ respectively.

However, monomorphisms of the category of abelian groups are not necessarily normal. That is they need to be a kernel of their cokernel. Let us remind the definition of a kernel: k is a kernel of h if

- $k; h = 0$,
- For any l such that $l; h = 0$, there is a unique u such that $u; k = l$.

Now, of course, $f; \mathbf{coker}f = 0$ for any arrow f, so the first condition holds in our situation. But we will not be able to prove that the universal property holds. Indeed, let us take a monomorphism f from group A to group B. To prove the universal property, the argument would go as follows: let g be a function from C to A such that $g; \mathbf{coker}f = 0$, this means that $\forall c{:}C.\, \exists b{:}B.\, g\,c = f\,b$. Which means there is a function v from C to the set $\{a : A \mid a \,\varepsilon\, \mathfrak{Im}\, f\}$ (which happens to be a group morphism when the image of f is endowed with the canonical structure of group). There is also, for the same reason, a function u from B to the set $\{a : A \mid a \,\varepsilon\, \mathfrak{Im}\, f\}$, which can also be viewed as a group morphism. u is a bijection. If we had the principle of unique choice, we could deduce an inverse u' to u, which would, again, be a

group morphism, as all (functional) inverses of group morphisms, and the universal property would be given by $v; u'$. However, this is not available and the proof fails.

Using, as in Section 1.3, that we can extend our mathematics with the excluded middle, we can show more precisely that being injective and surjective for an abelian group morphism does not imply that it has an inverse. Our slogan, here, will be: *unique choice is not valid in the category of abelian groups*.

The set $\mathbb{B}$ can be endowed with a structure of abelian group, using false as the neutral element and the "xor" as the composition. With the excluded middle available, the same can be done with Ω. The function
$f : \mathbb{B} \longrightarrow \Omega$
$f \quad \text{true} \quad = \quad \top$
$f \quad \text{false} \quad = \quad \bot$
Happens to be a group morphism. It is also both injective and surjective. As in Section 1.3, an inverse would decide the truth of propositions.

All this is not to say that abelian categories are of little use without the principle of unique choice. They may even have a important role to play in the setting of homological algebra computations. Our conjecture is that the category of finitely presented groups is an abelian categories. This result should also hold for categories of modules over a given ring, probably with the additional requirement that the ring has decidable equality. This result would be rather useful, as abelian categories have a variety of property which are interesting for computation as demonstrated, for instance, by Homalg [5]. The idea would be to reduce the computation of the homology of a connected pair of abelian group morphisms to the computation of the homology of a connected pair of finitely presented group morphisms where more computations are available. Effective homology features a tool for this sort of purposes called the basic perturbation lemma [48].

2.3 Preabelian categories

Preabelian category

A preabelian category is like an abelian category except it does not require that mono- and epimorphism are normal. In other words, a preabelian category is an additive category with a zero element, all (binary) biproducts, all kernels and cokernels. It is fairly straightforward to prove that any category of module is a preabelian category without making use of the principle of unique choice.

It is interesting, for or purposes, to notice that the theory of effective homology, upon which Kenzo [49] is based, does not seem to escape the framework of preabelian categories. As an illustration we shall briefly review the notion of *effective short exact sequence* of chain complexes.

Short exact sequence

A short exact sequence is an exact sequence of the shape.

$$\xrightarrow{0} \xrightarrow{i} \xrightarrow{j} \xrightarrow{0}$$

In other words an exact pair:

$$\xrightarrow{i} \xrightarrow{j}$$

with i mono, j epi.

Let f be a mono. We have $(\mathbf{ker}f)f = 0f$, hence $\mathbf{ker}f = 0$. Conversely, if f has a null kernel. Take $gf = hf$ that is $(g - h)f = 0$. Universality of kernels gives a unique u such that $g - h = u(\mathbf{ker}f) = 0$.
Dually, f is epic if and only if if has a null cokernel.

Splitting lemma

The splitting lemma is an important tool in classical homological algebra. It states, in particular, that in an abelian category, for a short exact sequence i, j, it is equivalent for i to have a postinverse and for j to have a preinverse.

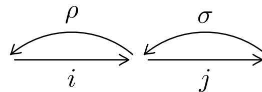
Let us consider i monic and epic. As it is monic, i followed by $\mathbf{coker}i$ is a short exact sequence. Since i is epic, $\mathbf{coker}i$ is trivial, hence 0 is a preinverse. The property of the splitting lemma gives us a postinverse to i. Since i is epic it is actually an inverse to i which realises the principle of unique choice.

However this property is stronger that the principle of unique choice in a preabelian category. We have seen, though, that the principle of unique choice does not hold in the (preabelian) category of abelian group, hence the splitting lemma does not hold, in general, in preabelian categories.

The category of abelian groups can be faithfully embedded in the category of chain-complexes of abelian groups, therefore there is no principle of unique choice or splitting lemma there either.

Effective short exact sequence

It is essentially to patch this difficulty, that Sergeart[48, p. 71] introduces the notion of effective short exact sequences of chain complexes. An effective short exact sequence is a diagram of the form:



with i and j chain-complex morphisms, ρ and σ graded module morphisms such that:

- $i; \rho = 1$
- $\rho; i + j; \sigma = 1$
- $\sigma; j = 1$
- the pair i, j is exact

In particular, in the category of graded modules, i is monic and j epic. As the category of chain-complex is a sub-category of that of graded modules, they are also, respectively, monic and epic as chain-complex morphisms. Hence, effective short exact sequences are indeed short exact sequences.

2.4 Graded objects and chain complexes

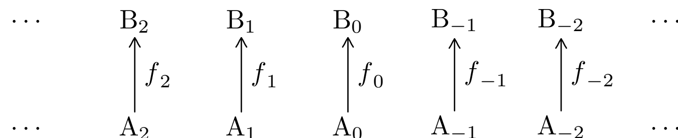
Preabelian categories are enough to represent the category of groups or other interesting categories. We also need a notion of chain complexes which makes sense inside a preabelian category.

Graded object

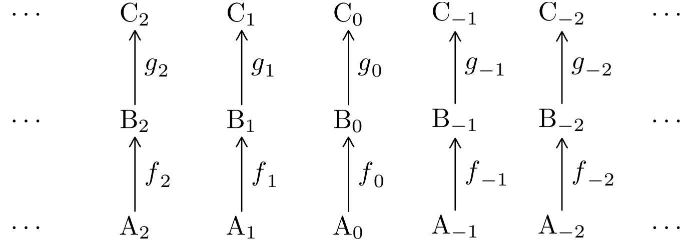
In an arbitrary category $\mathcal{C}$ (presumably preabelian for our purpose), we call *graded object* a $\mathbb{Z}$-indexed family $(A_n)_{n:\mathbb{Z}}$ of objects of $\mathcal{C}$.

Morphism of graded objects

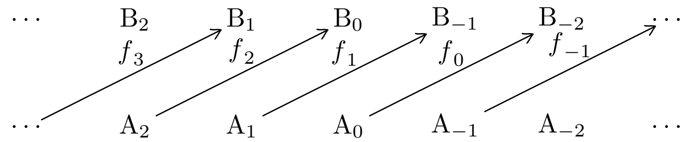
Given two graded object $(A_n)_{n:\mathbb{Z}}$ and $(B_n)_{n:\mathbb{Z}}$ we can define a notion of morphism between them. A first approach is to say that a morphism is a family $(f_n)_{n:\mathbb{Z}}$ with f_n an arrow from A_n to B_n.



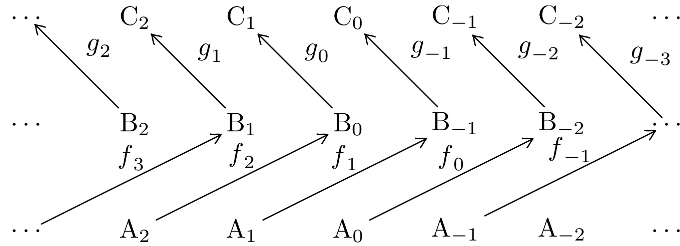
These morphisms compose naturally (by pasting the diagrams).



However, we will need some slightly richer kind of morphisms. We shall call a morphism of degree k between graded objects $(\mathrm{A}_n)_{n:\mathbb{Z}}$ and $(\mathrm{B}_n)_{n:\mathbb{Z}}$ a family $(f_n)_{n:\mathbb{Z}}$ where each f_n is an arrow from A_n to B_{n+k}. Here is, for example, a morphism f of degree -2:



We can still compose morphisms by pasting diagrams. The composite of a morphism of degree k with a morphism of degree l has degree $k+l$. As an illustration, here is the composition of f of degree -2 and g of degree 1, the composite has degree -1.



Similarily, the morphism of degree 0 which is identically the identity is a unit for the composition (notice that it has to be a degree 0 to be a unit).

This all seems quite categorical – even though we couldn't push the analogy to the point of real code-sharing in Coq (see Section 6.3). We shall see that we can also mimic the structure of preabelian category, provided the base category is preabelian.

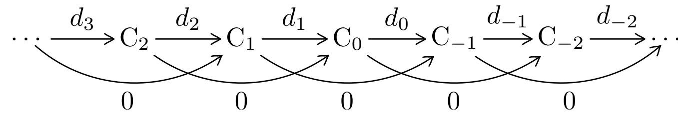
- The set of morphisms between two graded objects at a given degree has a natural group structure. That is the corresponding product group.
- For two graded objects $(\mathrm{A}_n)_{n:\mathbb{Z}}$ and $(\mathrm{B}_n)_{n:\mathbb{Z}}$, the graded object $(\mathrm{A}_n \oplus \mathrm{B}_n)_{n:\mathbb{Z}}$ has the properties of a biproduct
- The graded object $(0)_{n:\mathbb{Z}}$ behaves like a zero object.
- Similarily, the morphism obtained from $(f_n)_{n:\mathbb{Z}}$ by taking the kernel at each n has all the properties of a kernel. Notice that in this case we can choose the degree of the morphism – since we can always change the indices of the kernel object without compromising any property. We pick 0 as it will be convenient for the alignment of homology groups.

- Dually, the morphism obtained from $(f_n)_{n:\mathbb{Z}}$ by taking the cokernel at each n has all the properties of a cokernel. Here again we can pick the degree we wish, and take 0.

We can then define homology as in a preabelian category: let f and g be a connected pair of graded object morphisms, its homology is the cokernel of f as a morphism into the kernel of g. The homology of a pair of graded object morphisms is a graded object.

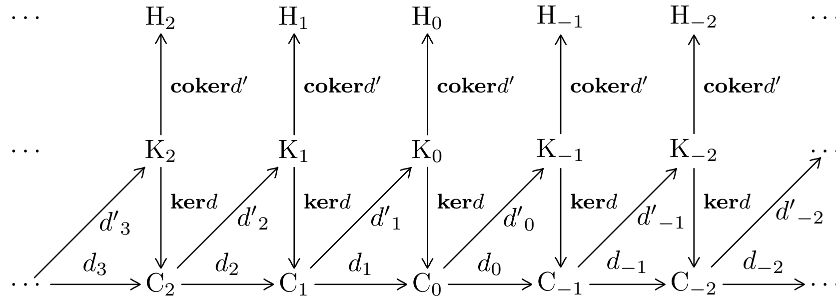
Chain complex

In this framework, a chain complex C_* can be defined as a pair of a graded object $(C_n)_{n:\mathbb{Z}}$, together with an endomorphism d of degree -1 connected with itself (the differential morphism).



The homology of d with itself is called the homology of the chain complex. It is a graded object – not a chain complex – $(H_n)_{n:\mathbb{Z}}$. H_n is called the n-th homology group of C_*.

The choice of degree 0 for both kernel and cokernel ensures that H_n is indeed the n-th homology group of C_*, that is the homology of the pair d_{n+1} followed by d_n – as illustrated by following diagram where d' is the embedding of d in the kernel of of d.



Coq voodoo

The definition of the category-like structure of graded objects deserves some technical comments. It is not straightforward to make Coq accept these definitions.

Let us see why, first. Given a preabelian category $\mathcal{A}$, we write $\mathcal{A}$ for the type of its objects and $\mathcal{A}$ A B for the homset from A to B. We define the graded object as the type Z → $\mathcal{A}$ (where Z is a binary representation of integers from the standard library of Coq). From there, there are two approaches to define the sets of morphisms.

These notations are made possible in Coq by the mechanism of implicit type coercions.

- We can define morphisms (without a degree) between the graded objects A and B to be the elements of type **forall** n:Z, $\mathcal{A}$ (A n) (B n), then we define a family of functors indexed by integer numbers, for a graded object

A, k↑A would be defined as as **fun** n ⇒ A (n+k) and for a morphism f, k↑f would be defined as **fun** n ⇒ f (n+k). Then a morphism of degree k between A and B is a morphism between A and k↑B.

Unfortunately, it does not work much further than that. Indeed let us consider two composable morphism f and g of respective degree k and l. Their composition f·g would be defined as **fun** n ⇒ (f n)·(k↑g n) whose target object would be **fun** n ⇒ C (n+k)+l where we would expect (k+l)↑C (*i.e.* **fun** n ⇒ n+(k+l)). They appear the same to the mathematician, as addition of integers is commutative. The Coq user, however, tends to shrieks and cries when presented with this situation. Indeed, Coq's way to identify terms is through the conversion rule. If two terms are convertible then they are the same, otherwise they are not. Associativity of addition is not understood by the conversion rule, as it is defined before addition. And, as a matter of fact, there is no way to define addition such that associativity is simulated by the conversion. To solve that, either one can give up on the decidability of conversion (like in extensional type theory [41]), or one allows to throw in new decidable conversion rules. The latter has two iconic approaches, the Calculus of Algebraic Constructions [14] – where functions can be defined as rewrite rules more sophisticated than pattern-matching (and more than necessary for ground terms) – and, more recently, the Calculus of Inductive Construction Modulo Theory [53] – where one can plug certified decision procedures in the conversion. Both these approaches are sufficient for our case.

- The other approach is to define directly morphisms of degree k from A to B as being functions in **forall** n, $\mathcal{A}$ (A n) (B n+k). At first sight, it does not seem of much help as the type of morphisms of degree k is the same as the one proposed in the functor-based encoding. It seems even worse, as we lose the categorical structure: we had a category (graded objects with morphisms) and a family of functors, we could use that setting abstractly, and reuse theorem about categories – if it had worked.

  There is actually hope in this direction, for that very reason: we abandon the idea of an encoding as a category with additional structure, this allows to "split the problem" differently. By choosing a more clever implementation of morphisms of degree k we will manage to have them behave fairly well.

As a matter of fact, it is not very plausible that we can write, in Coq, the (somewhat generalised) category of graded objects in any pleasant way at all. We give a partial solution in which the constructions of this category (in particular the composition of morphisms) are expressible and have the expected types. We actually encode morphisms of degree k as a record reporting some of the effort that conversion needed to do as proof obligations:

```
Record morphism (k:Z) (A B:Z→𝒜) := {
  shift : Z → Z ;
  maps : forall n:Z, 𝒜 (A n) (B (shift n)) ;
  homogeneity : forall n, shift n ≡ n+k
}.
```

Where ≡ denotes the propositional equality of Coq, usually written =, but we reserved the latter for set equality. Let us remark that propositional equality of Coq is stronger than convertibility. Indeed, if it coincides with convertibility on ground terms, it can make use of any available proof principle on free variables of its operands. In particular, **forall** x y z:Z, (x+y)+z≡x+(y+z) is provable.

We will write A ⟶ B :: k for morphism k A B (not to be confused with A → B which is the type of maps from A to B).

Now defining composition is straightforward:

```
Program Definition comp (k l:Z) (A B C:Z→𝒜)
                    (f:A⟶B::k) (g:B⟶C::l) : A⟶C::k+l := {|
  shift := fun n ⇒ shift g (shift f n) ;
  maps := fun n ⇒ (f n)·(g (shift f n))
|}.
```

We need only to prove that **forall** n, shift g (shift f n) ≡ (k+l)+n which is no problem.

However the story does not end here. If composition is now easy, equality (between two morphisms of the same type) which was easy in the first encodings has become hard. We shall need another trick to be able to define equality.

First let us introduce the following type:

```
Inductive eqopt {A B:Type} (F:A→B) (k l:A) :=
  | NotEq : eqopt F k l
  | Eq : (F k → F l) → eqopt F k l
.
```

In other words, an eqopt F k l is either NotEq or Eq f where f converts from F k to F l. Our intention is that NotEq means that k and l are not propositionally equal (*i.e.* ¬k≡l) and Eq f means that k and l are propositionally equal (*i.e.* k≡l) and, in addition, that f is the identity function (f≡**fun** x⇒x). Functions returning an eqopt F k l are, then, a variant of equality decision. We will use such a function for Z:

```
Definition Zeq {A:Type} (F:Z→ A) (k l:Z) : eqopt F k l.
Lemma Zeq_same : forall F k, Zeq F k k = Eq (fun x ⇒ x).
```

Now we can define equality.

```
Definition eq (k:Z) (A B:Z→𝒜) (f g:A⟶B::k) :=
  forall n,
    match Zeq B (shift f n) (shift g n) with
    | NotEq → False
    | Eq id → id (f n) = g n
    end
.
```

There are actually two other constructions that need the trick of using Zeq: addition of morphisms (for the same reason than equality: it acts on a pair of morphisms of the same degree) and cokernel. This means in practice that every addition or cokernel features a structural test of equality which always

succeeds (typing enforces it). This equality test will always be run on fairly short integers (no one is really interested in the millionth homology group of anything), nonetheless it is unsatisfying as this test is a useless computation. We can sum it up with the following slogan: *static properties need not be enforced by dynamic tests*.

There is a good reason in practice for Coq to require such tests, though. Types can be proved identical even if they are not, provided the context makes inconsistent hypotheses. The context is made of all the local hypotheses made by the function being defined, and the branches of pattern matching under scrutiny. Hence it does happen that some expressions are written in an inconsistent context; in this case we typically just want to prove that the context is inconsistent, though (remember, from Section 1.3, that this is the principal role of computationally irrelevant proofs in programs). Connor McBride coined the phrase *lies are locally true* to describe this situation. Dynamic tests required by Coq are meant to avoid making use of such a lie, which would lead to Coq's equivalent of `segfault`. This constraint is a consequence of strong reduction. In a weak reduction, every evaluation happens in the empty context, local lies are never evaluated.

In addition, these mechanisms come with rather severe limitations. If when working with morphisms of a concrete degree, things go fairly well – if d has type C⟶C::-1, then d·d has type C⟶C::-2 as expected. However, it is not possible to express arbitrary statement about morphisms with abstract degree. For instance, let f:A⟶B::k, g:B⟶C::l and h:C⟶D::p, then (f·g)·h has type A⟶D::(k+l)+p and f·(g·h) has type A⟶D::k+(l+p). These types are not convertible unless associativity of addition is supported by conversion. It is therefore not directly possible to express the associativity of composition, as equality has a type which supposes that both sides have the same type. There are in the literature examples of equality which works on non-convertible (yet equal) types; however they do not work well without some variant of the K axiom[52]. Such an approach could be worthwhile in the setting of graded objects, as K is valid on Z (and, more generally, on types with decidable propositional equality). We haven't conducted sufficient investigation to be able to say whether the it would solve this issue though, it seems to be enough for equality but the problem arises again with addition. Another solution could be to summon the Zeq trick again in the statement of the problem (**match** Zeq (**fun** n⇒A⟶D::n) ((k+l)+p) (k+(l+p)) **with** . . .). This would be verbose, and break any chance of abstraction, but, with some boilerplate, might work better with the rewriting mechanism of Coq.

2.5 Kernels of matrices

In this section we will present an example of a direct utilisation of the preabelian category framework to produce an effectively executable proof. We will prove that the kernel of a linear function between two finite dimensional *vector spaces* has finite dimension.

This might not be a very legitimate usage of the preabelian category abstraction, as the proof is not very different from its concrete counterpart. Additionally it does not seem to generalise to any other useful proof. On the other hand, a quite principled presentation arises from the approach. It is

meant to demonstrate the style of proofs preabelian categories suggest.

Field

Before we go on with the proof, let us consider the notion of field from the point of view of constructive logic. Fields typically come in two flavours: let $\mathbb{K}$ be the base set of a field

- Either it is given a function $\frac{1}{-}$:$\{a : \mathbb{K} \mid \neg a = 0\} \longrightarrow \mathbb{K}$ which verifies, for any a:$\mathbb{K}$ such that $\neg a = 0$, that $\frac{a}{a} = 1$.
- Or the property that $\forall a{:}\mathbb{K}.\ \exists a'{:}\mathbb{K}.\ aa' = 1$ holds.

From a constructive (*with* the principle of unique choice) point of view, the latter is strictly stronger than the former. More precisely, the second property is equivalent to the first in conjunction with the decidability of the equality on $\mathbb{K}$.

For the sake of completeness we should mention that the first property can also be split: it has a weaker version of the form $\forall a{:}\mathbb{K}.\ \neg a = 0 \rightarrow \exists a'{:}\mathbb{K}.\ aa' = 0$.

When we drop the unique choice it becomes a bit more complicated as the second property is split in two cases: as it is written it becomes rather weak since it does not allow to compute anything. But we can reformulate it to regain its original meaning, giving a function $\frac{1}{-}$:$\mathbb{K} \longrightarrow \{a = 0\} + \{a' : \mathbb{K} \mid aa' = 0\}$. Where the set $\{a = 0\}$, is the set of the proofs of $a = 0$ all considered equal. It is this last formulation that we will use for our proof.

$f\,1$ is either 0 in which case $f = 0$, or some non-zero k and $f\,x = x(f\,1) = kx$. In the latter case, let $g\,x = \frac{x}{k}$. We have that $f\,(g\,x) = g\,(f\,x) = x$, for any x.

A field $\mathbb{K}$ can also be seen as a $\mathbb{K}$-vector space. It verifies (both as a field and a vector space) that any arrow $\mathbb{K} \xrightarrow{f} \mathbb{K}$ is either 0 or has an inverse. The inverse of f plays the categorical-equivalent role of the division in the field. We shall remember that *endomorphisms of fields are null or automorphisms*.

Finite dimensional object

Let us fix, for the rest of this section, a preabelian category $\mathcal{A}$ and an object $\mathbb{K}$ of $\mathcal{A}$ such that any endomorphism of $\mathbb{K}$ is either 0 or an automorphism.

The usual definition of finite dimensionality for a vector space A states that A has a finite linearly independent generating family – the basis. An equivalent statement in the realm of categories would be that there is a bijective linear function b from some $\mathbb{K}^n$ – with n:$\mathbb{N}$ – to A. This b is to be understood as the function interpreting coordinates along the basis as a vector in A. Here $\mathbb{K}^n$ corresponds to finiteness, the fact that b is a monomorphism corresponds to linear independence, and the fact that it is epic corresponds means it is generating. Now, without the axiom of unique choice it the question is: is it the right definition, or do we need an inverse? An inverse to b gives a decomposition into coordinates. For our application, at least, this cannot be spared. Hence we shall say that A has finite dimension when there is an isomorphism between A and $\mathbb{K}^n$ for some n:$\mathbb{N}$. We shall adopt the terminology that the morphism from $\mathbb{K}^n$ is called the basis, and the reciprocal is called the decomposition along the basis.

Outer lemma

The core of the proof will consist in proving that if f is an arrow from $\mathbb{K}^n$ to $\mathbb{K}^p$ for some n and p, then $\mathbf{Ker}f$ is isomorphic to $\mathbb{K}^r$ for some r.

Let us first consider an arrow f in $\mathrm{A} \xrightarrow{f} \mathrm{B} \oplus \mathrm{C}$, it can be split into two components: $\mathrm{A} \xrightarrow{f\pi_1} \mathrm{B}$ and $\mathrm{A} \xrightarrow{f\pi_2} \mathrm{C}$. Note that f can be rebuilt from these two components ($\mathrm{B} \oplus \mathrm{C}$ is a product). Our first lemma is that the kernel of f is isomorphic to the intersection of those of the two components. We need to phrase this in the language of preabelian categories. Let us consider the arrow $\mathbf{Ker}(f\pi_2) \xrightarrow{\mathbf{ker}(f\pi_2)} \mathrm{A}$. Its composite $\mathbf{ker}(f\pi_2)f\pi_1$ can be read as the restriction of $f\pi_1$ to $\mathbf{Ker}(f\pi_2)$. Hence the lemma can be stated as: $\mathbf{Ker}(\mathbf{ker}(f\pi_2)f\pi_1)$ is isomorphic to $\mathbf{Ker}f$.

What this lemma tells us, is that we can restrict our attention to arrows of the shape $\mathbb{K}^n \xrightarrow{f} \mathbb{K}$. Indeed, if we can give a basis (and decomposition) to such an arrow, then to solve an arrow of the form $\mathbb{K}^n \xrightarrow{g} \mathbb{K}^p$ if p is 0, then full $\mathbb{K}^n$ is the kernel of g, which directly gives a basis, otherwise $\mathbb{K}^p$ is $\mathbb{K} \oplus \mathbb{K}^{p'}$, then $\mathbf{ker}(f\pi_2)f\pi_1$ is an arrow from $\mathbf{Ker}(f\pi_2)$ to $\mathbb{K}$. Since $f\pi_2$ is an arrow from $\mathbb{K}^n$ to $\mathbb{K}^{p'}$, by induction we get a basis of $\mathbf{Ker}(f\pi_2)$. Hence $\mathbf{ker}(f\pi_2)f\pi_1$ is an arrow of the shape $\mathbb{K}^m \longrightarrow \mathbb{K}$, which we can solve by assumption.

Let us write K for $\mathbf{Ker}f$ and K′ for $\mathbf{Ker}(\mathbf{ker}(f\pi_2)f\pi_1)$. To map K′ to K, consider the map $\mathrm{K}' \xrightarrow{\eta} \mathrm{A}$ defined as $\mathbf{ker}(\mathbf{ker}(f\pi_2)f\pi_1)\mathbf{ker}(f\pi_2)$. Since $\eta f = 0$, η can also be seen as an arrow from K′ to K. The other direction is realised by $\mathbf{ker}f$ which can be seen as an arrow to $\mathbf{Ker}(f\pi_2)$ (it's bigger than the kernel of f). It follows that $\mathbf{ker}f\mathbf{ker}(f\pi_2) = \mathbf{ker}f$ hence $\mathbf{ker}f\mathbf{ker}(f\pi_2)f\pi_1 = \mathbf{ker}ff\pi_1 = 0$, and $\mathbf{ker}f$ can be seen as an arrow from K to K′. It follows from the definitions that η together with $\mathbf{ker}f$ form an isomorphism.

Inner lemma

Let us consider an arrow $\mathrm{A} \oplus \mathrm{B} \xrightarrow{f} \mathrm{C}$. It can be split into two components $\mathrm{A} \xrightarrow{\iota_1 f} \mathrm{C}$ and $\mathrm{B} \xrightarrow{\iota_2 f} \mathrm{C}$. Here again, f can be entirely rebuilt from its two components ($\mathrm{A} \oplus \mathrm{B}$ is a coproduct). Now, let us suppose that $\iota_1 f$ has an inverse $\mathrm{C} \xrightarrow{g} \mathrm{A}$. Then the kernel of f is isomorphic to B. The following illustrates how g essentially performs a division. Let us pretend for a moment that $\mathbb{K} \oplus \mathbb{K}^n \xrightarrow{f} \mathbb{K}$, and that $\mathbb{K}$ is a field. Then $f(x_0, x_1, \ldots, x_n) = \lambda_0 x_0 + \lambda_1 x_1 + \ldots + \lambda_n x_n$ with λ_0 having an inverse. Then $f(x_0, x_1 \ldots, x_n) = 0$ if and only if $x_0 = \frac{\lambda_1}{\lambda_0}x_1 + \ldots + \frac{\lambda_n}{\lambda_0}x_n$. Hence $(1 - \frac{\lambda_1}{\lambda_0}, \ldots, 1 - \frac{\lambda_n}{\lambda_0})$ is a basis of the kernel of f.

If, on the other hand, $\iota_1 f = 0$, then $\mathbf{Ker}f$ is isomorphic to $\mathrm{A} \oplus \mathbf{Ker}(\iota_2 f)$.

The categorical proof goes about like the concrete proof showed in the text except that the basis is given as the morphism $\iota_2 - \iota_2 f g \iota_1$, the reader can convince himself that it corresponds to the basis proposed in the proof beside. The inverse function is $\mathbf{ker}f\pi_2$.

The proof that $\mathbf{Ker}f$ is isomorphic to $\mathrm{A} \oplus \mathbf{Ker}(\iota_2 f)$ is a a bit tedious, but rather straightforward. Let us sketch it briefly. $\mathrm{A} \oplus \mathbf{Ker}(\iota_2 f) \xrightarrow{\eta} \mathrm{A} \oplus \mathrm{B}$ where $\eta = 1 \times \mathbf{ker}(\iota_2 f) = \pi_1 1 \iota_1 + \pi_2 \mathbf{ker}(\iota_2 f)\iota_2$ can be seen as an arrow to $\mathbf{Ker}f$. The inverse arrow is $\mathbf{ker}f\pi_1\iota_1 + \mathbf{ker}f\pi_2\iota_2$ where $\mathbf{ker}f\pi_2$ is seen as an arrow to $\mathbf{Ker}(\iota_2 f)$.

The algorithm in motion

Let us paste the pieces together. Let $\mathbb{K}^n \xrightarrow{f} \mathbb{K}$. If $n = 0$, then the kernel is the zero object. Otherwise, $\mathbb{K} \oplus \mathbb{K}^{n'} \xrightarrow{f} \mathbb{K}$ which splits in two cases, either $\iota_1 f = 0$, then the kernel of f is isomorphic to $\mathbb{K}^{n'}$ – and we're done – or $\iota_1 f$ has an inverse and the kernel of f is isomorphic to $\mathbb{K} \oplus \mathbf{Ker}(\iota_2\, f)$ By induction, $\mathbf{Ker}(\iota_2\, f)$ is finite dimensional. And the problem is solved.

Remember that, as the philosophy of this work prescribes, the proof actually describes an algorithm which computes, given some $\mathrm{A} \xrightarrow{f} \mathrm{B}$, with A and B finite dimensional, a basis and a decomposition for $\mathbf{Ker}f$. Using the decomposition we can compute coordinates for the vectors of the basis of $\mathbf{Ker}f$.

As often, a f can be specified by a matrix (an element of $(\mathbb{K}^n)^m$) – this makes use of the bases of A and B. The implementation described here has been tested to compute bases of kernels of various matrix with coefficient in $\mathbb{K}$ – the field of rational numbers. The computation are fast up to size 30×30 and tractable up to size 50×50, approximately. This might be considered a disappointment, but we should stress that we have used a very naive

A skeleton of the Coq proof presented in this section can be found in Appendix A together with example runs of the proof seen as a program.

representation for matrices: lists of lists. In this example we were only mildly interested in performance. Otherwise we might have wanted to use a smarter representation, for instance replacing lists with binary lists [46, p. 119–122]. This would come at a cost, though, as the induction to solve arrows of the form $\mathbb{K}^n \xrightarrow{f} \mathbb{K}$ is structural on lists. Coq really likes structural recursion. We would need to devise a recursion principle to use, say, binary lists as linked lists. As mentioned in Section 1.3, using non-structural recursion in Coq has a cost on performance which is hard to quantify *a priori*. However, the outer induction (reducing the problem from $\mathbb{K}^n \longrightarrow \mathbb{K}^p$ to $\mathbb{K}^m \longrightarrow \mathbb{K}$) could be reduced to a logarithmic number of step instead of linear. Maybe more importantly, access in the matrices would be much faster, effectively reducing the complexity of the functions they represent. It is safe to say this would result in an overall improvement in time performance.

We might also want to represent matrices using some flavour or arrays – as it is more common in traditional programming languages. This may or may not be a good idea, as our algorithm constructs and destructs matrices – which is harder to do on arrays than on lists (binary lists being pointedly a compromise between arrays and lists). Though they are not part of historical Coq paraphernalia, some flavour of persistent array can be made available, as explained in Section 3.4. They might fit for this application. This leave us with one consideration: if we are to use arrays to represent products (or, in our case, biproducts) in category, we cannot restrict our attention to binary (plus nullary) products. All n-ary products can be generated using binary and nullary products. They can be coded in a variety of ways – including lists and binary lists mentioned earlier. However arrays are encodings of n-ary products which are atomic in that they cannot be described in terms of smaller – or bigger – products. This leads to a formulation of the phrase "a category with all products" in a way that is sometimes called *unbiased*. Further thoughts on this are presented in Section 5.3.

Part Two

Intermezzo: Down to Coq Implementation

Chapter Three

Efficient computations

> Five hundred carpenters and engineers were immediately set at work to prepare the greatest engine they had. It was a frame of wood raised three inches from the ground, about seven feet long, and four wide, moving upon twenty-two wheels. The shout I heard was upon the arrival of this engine, which, it seems, set out in four hours after my landing. It was brought parallel to me, as I lay. But the principal difficulty was to raise and place me in this vehicle. Eighty poles, each of one foot high, were erected for this purpose, and very strong cords, of the bigness of packthread, were fastened by hooks to many bandages, which the workmen had girt round my neck, my hands, my body, and my legs. Nine hundred of the strongest men were employed to draw up these cords, by many pulleys fastened on the poles; and thus, in less than three hours, I was raised and slung into the engine, and there tied fast
>
> Jonathan Swift, *Gulliver's travels*

OMPUTER algebra systems usually consist in rather hard computation. Kenzo – and homological algebra in general – is no exception. However, when it comes down to efficiency, dependent type theories rarely compete with traditional programming languages. As a matter of fact, the question of efficiency has been often dismissed in favour of program extraction: from a Coq program, one can extract an OCaml program with the same behaviour, but typically way more efficient. While this is perfectly reasonable for standalone program, there is a case where it cannot be accommodated: when computations are part of Coq proofs – process often known as reflexive tactics. The most famous such example is the proof of the four colour theorem [24], but there are various other reflexive tactics out there. Coq's standard library most notoriously provides a decision procedure for the theory of commutative ring[28].

A major improvement was achieved with the introduction of a dedicated virtual machine [27] allowing Coq programs to compare with (bytecode compiled) OCaml ones. Relatedly, there are preliminary works to leverage OCaml's native compilation in order to improve computation speed more.

Still, it does not solve everything. Indeed, Coq didn't provide any primitive data-structure, every type is to be encoded using the constructs allowed by the system (primarily, inductive definitions). So we went and added some. More precisely, we shall present in this chapter, how we extended Coq – in particular Coq's virtual machine – with integers in order to use the arithmetic abilities of the processor.

3.1 A brief history of $\mathbb{N}$

In this section we shall present various encoding of the type of natural numbers present in the standard library of Coq. The variety of these reflects the history of Coq, and how, as time passed, it has been seen increasingly important to compromise simplicity for the sake of efficiency.

Peano numbers

The simplest way to define natural numbers inside Coq's theory is to define them as Peano numbers:

```
Inductive nat : Type :=
  | O : nat
  | S (n:nat) : nat
```

nat is a so called *inductive type*, its definition is read: "nat is the smallest type which has an element O and for any element n, has an element S n".

Operations on inductive types are defined by recursion on their structure. For instance, addition on nat is defined as follows.

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O    ⇒ m
  | S n' ⇒ S (plus n' m)
 end
```

Addition on nat is an operational version of Peano's axiomatic addition. Let us give as an example, the computation of $3 + 2$:

```
plus (S (S (S O))) (S (S O))  ⇝  S (plus (S (S O)) (S (S O)))
                              ⇝  S (S (plus (S O) (S (S O))))
                              ⇝  S (S (S (plus O (S (S O)))))
                              ⇝  S (S (S (S (S O))))
```

The computation of a sum is linear (in its first argument), which is hardly acceptable. Our slogan will be: *Peano numbers: simple, yet intractable*.

Binary numbers

A binary representation of natural numbers allows logarithmic operations.

```
Inductive positive : Type :=
  | xI (p:positive) : positive
  | xO (p:positive) : positive
  | xH : positive

Inductive N : Type :=
  | N0 : N
  | Npos (p:positive) : N
```

An element of N is either 0 (N0) or a string of 1-s (xI) and 0-s (xO) starting with a 1 (xH). Defining operations for these numbers is not as simple and direct than with Peano numbers. We will not show a definition here, we will content with a run of the addition of $3 + 2$ (where Pplus is the addition over positive):

```
Nplus (Npos (xI xH)) (Npos (xO xH))  ⇝  Npos (Pplus (xI xH) (xO xH))
                                     ⇝  Npos(xI (Pplus xH xH))
                                     ⇝  Npos(xI (xO xH))
```

The improvement both in computation time and space usage is really worth additional hard work. In programs which do not do a lot of number crunching, this representation will probably be sufficient. An issue, though, is that these numbers can only be read from the least significant bit to the most significant. Some operations on natural numbers work better if the number is read the other way. An archetypal example is comparison: even if we store the length of the number beside, we still need access to the most significant bit, which is linear in the length (*i.e.* logarithmic in the value) of the operands.

As a side note, remark that this encoding of natural numbers gives rise, for free, to an encoding of the integers which appears in the library of Coq as the following type:

```
Inductive Z :=
  | Z0 : Z
  | Zpos (p:positive) : Z
  | Zneg (p:positive) : Z
```

Numbers as trees

A natural approach to deal with the order in which the bits can be read is to represent the numbers as binary trees, reading the digits from the leaves. To benefit from this construction we will need to keep track of the height of the tree, for that we need a previously existing kind of natural numbers. A good choice is Peano integers: the height of the trees is typically very small (filling a 4 Gb memory with digits requires 2^{35} bits, such a number has height 35 as a binary tree). And either way the overhead, in memory, of the tree itself is bigger than that of the Peano number representing its height.

We are ready to provide an implementation for such numbers:

The code for word showed in this section is a blatant, yet shameless, simplification of the code which can be found in the standard library of Coq.

```
Fixpoint word h :=
  match h with
  | O → bool
  | S k → (word k * word k)%type
  end.

Definition bigN := { h & word h }.
```

A bigN is defined as the pair of a Peano number h and a number of height h.

Note that we have lost the property that every number has a single representation, as a number can always be seen as a number of a larger height. This makes the basic operations more difficult to write down than with the earlier representations. Also, as the trees are perfectly balanced, there is easily a lot of trailing 0-s in front of the number ($2^{(h+2)}$ is represented as the pair of 1 and 0 seen as numbers of height $h+1$, 1 is, hence, represented with h additional 0-s). We shall modify our type word to take this remark into account:

```
Fixpoint word h :=
  match h with
  | O → bool
  | S k → option (word k * word k)
  end.
```

option A is either Some a with a:A or None. Here, None stands for 0 at any height. The operations will be programmed to favour None over the more classical representation of 0.

Another perk of this binary tree representation is that it is well-suited for Karatsuba multiplication [32]: a number n is represented as $n_h 2^h + n_l$ where h is the height of n. Let $p = p_h 2^h + p_l$ be another number of height h,

$$np = (n_h 2^h + n_l)(p_h 2^h + p_l) = (n_h p_h)2^{2h} + (n_h p_l + n_l p_h)2^h + n_l p_l$$

This gives a naive algorithm – in $O(n^2)$ as usual – which require four recursive multiplications. However, if we write a for $n_h p_h$ and b for $n_l p_l$, then

$$(n_h + n_l)(p_h + p_l) - a - b = n_h p_l + n_l p_h$$

Meaning we can compute np with three recursive multiplications (though at the cost of some extra additions). This leads to an algorithm in $O(n^{\log 3})$.

The idea of representing numbers as binary trees had been in the air for a while. To the best of our knowledge, its first written appearance is in a work from Edwin Brady & al. [16, Chapter 5] where they nicely nicknamed the approach: *every number has at most two digits*.

More digits

We are still very far from the performances of mainstream programming languages. Especially as far as memory is concerned: every single bit of our natural numbers are stored on a whole computer word (usually 32 or 64 bits).

The large overhead puts a lot of pressure on the garbage collector and the computer cache.

To adress this problem we can take the binary tree representation (or the list representation, but binary trees are generally better) on a bigger base than 2. We then implement the digits with a more compact representation and more efficient arithmetic operations. As computer scientists ought to, we shall only consider bases which are powers of 2.

A first approach has been successfully used to formally verify the primality of rather large prime numbers[29]. It consists in representing digits of base 2^8 as an enumerated datatype:

```
Inductive w8 : Type :=
  | W0
  | W1
  | W2
     ⋮
  | W255
```

This representiation is a very significant improvement over the former ones both in term of speed and memory (even though the packing is still not very good as long as values of w8 are stored on machine words). Arithmetic operations on w8 are implemented by case analysis which, for binary operations, involves a squared number of cases (2^{16} in this particular case). The file for w8 – which is, understandably, generated by a program – is 100 MB large. Would we try to move to base 2^{16}, we would end up with a 1 GB file. We cannot drive this approach much further. Also we are really underusing the processor which typically has very fast arithmetical operations.

These 100 MB represents definitions of arithmetic operations modulo 2^8 as well as operations handling carries, plus the proofs of their correction.

3.2 Going native

In order to use the processor arithmetic directly, a first possibility is to extend the theory underlying the Coq logic with:

- one primitive type int
- the constructors 0, 1, 2, . . . , 2^{n-1} of type int
- the basic primitive functions over the type int such as +, ∗, . . .
- the corresponding reduction rules for each primitive function.

It is also necessary to give it an equational theory, for instance, Peano theory together with a lemma stating that $(2^n - 1) + 1 \equiv 0$ where $(\equiv)$ is the propositional equality of Coq (or, equivalently, Leibniz equality). However, this approach has some drawbacks:

- It adds a large amount of new constructions to the theory. This goes against the so-called de Bruijn's principle which states that keeping the theory and its implementation as small as possible highly contributes to the trust one has in a system. Furthermore, on a more practical side, it will have a deep impact in the implementation, since the terms will have to be extended with new syntactic categories (primitive types and primitive functions).

- It adds a lot of new reductions, not only for ground arithmetical terms – like 18+24 – but also for theorems. For example, if we consider the theorem n_plus_zero that states that **forall** n:int. n + 0 ≡ n, n_plus_zero 7 should reduce to refl 7 where refl represents the reflexivity of equality. As a matter of fact, every proof of 7≡7 needs to reduce to refl 7 if we want to preserve an adequate notion of cut elimination for Coq. An alternative, probably more realistic, solution would be to drop cut elimination and pose the theory as axioms.

For these reasons, we have taken an alternative approach. Efficient evaluation in Coq, as provided by the virtual machine, uses a compilation step. Before evaluating a term, it transforms it into another representation that is more suitable for performing reduction. The idea is to introduce the native machine integers not as part of the theory of Coq but only in this compilation phase. So, the type int of machine integers is defined using the standard commands as a type with a single constructor that contains n digits:

Definition bit := Bool.
Inductive int : **Type** := In ($d_{n-1} \ldots d_1 d_0$: bit) : int.

We relate the machine numbers int with the relative numbers Z with the two functions toZ : int → Z and its inverse toI : Z → int and we prove that they satisfy the following two properties:

forall i: int, toI (toZ i) = i
forall z : Z, toZ (toI z) = z mod 2^n

Now, it is straightforward to define the primitive functions of int as the image of the corresponding function of Z. For example, addition for int is defined as follows:

Definition $i_1 +_{int} i_2$:= toI (toZ $i_1 +_Z$ toZ i_2)

So that addition verifies directly that

$i_1 +_{int} i_2 \equiv$ toI (toZ $i_1 +_Z$ toZ i_2)

Actually, since propositional equality is decidable on type int, it suffices that the context on which the proof depends is non-contradictory.

Actually, any closed definition verifying this property in the empty context would be suitable for $+_{int}$. Indeed, propositional equality of closed terms in the empty context coincides with convertibility. For instance it implies that $3 +_{int} 2$ is convertible to 5. It is hence safe to add the rule $3 +_{int} 2 \rightsquigarrow 5$ to the convertibility test.

Our job is to ensure that this new rewrite rule is prefered over a least efficient choice and that it is implemented so as to leverage the arithmetical capabilities of the processor. To that end, we shall modify the compiler in such a way that it treats the type int as real machine integers. The main difficulty is that Coq requires strong reduction. This is not the case of traditional functional languages where only weak reduction is needed (no reduction under binders).

3.2.1 Coq's virtual machine

Strong reduction by symbolic weak reduction

Before being compiled into the virtual machine, Coq terms to be normalised are compiled into an intermediate calculus called the symbolic calculus [27] The terms of the symbolic calculus are an extension of those of Coq, hence compilation is the identity (formally type annotations are erased, but it does not justify to perform an actual compilation pass in practice).

On the rewriting side, the symbolic calculus does not quite behave like Coq. It has a weak (no reduction under binders) call-by-value reduction. As such it has a notion of values, we will write them with v-s to distinguish them from non-value terms. This is where the extensions to the grammar of terms kick in: the symbolic calculus has a weak reduction, still we want to use it as part of strong reduction, hence it will need a notion of neutral term as it will have to deal with free variables. In the symbolic calculus lingo, this is called an accumulator, written $[k]$. Here k is typically a neutral term of the form $x\ v_1\ \ldots\ v_n$ where x is a (free) variable. It can also be a pattern matching: matching over a neutral term is neutral.

In addition to the standard rules like β-reduction, the symbolic calculus has a few rules to work with accumulators. For instance, the case of application is as follows:

$$[k]\ v \rightsquigarrow [k\ v]$$

To employ the symbolic calculus as a mean of computing normal forms of Coq terms, we use a variant of normalisation by evaluation [12]. An evaluator of the symbolic calculus gives us a function $\mathcal{V}$ which computes the value of a term. To actually normalise, we need another function $\mathcal{R}$ which reifies the value, such that $\mathcal{N}(t) = \mathcal{R}(\mathcal{V}(t))$ performs normalisation on t. Writing such a function $\mathcal{R}$ is rather straightforward, here is, for instance, the case of the abstraction (which happens to be the one of interest):

$$\mathcal{R}(\lambda x.b) = \lambda y.\mathcal{N}((\lambda x.b)\ [y])$$

where y is a fresh variable.

To sum up, the normal form of a term can be obtained by recursively computing its symbolic weak normal form and reading back the resulting value. The efficiency of the process clearly depends on the efficiency of the weak evaluation.

Compiling the symbolic calculus

To implement the weak evaluation efficiently, it is compiled into an abstract machine – a variant of the ZAM [39]. A state of the abstract machine is given by a triple (e, c, s) where c is code to be evaluated, s a stack, and e an environment – both s and e are lists. To give a hint of their behaviour, let us give two rules as an example:

$$\begin{array}{rcl} (e,\ \mathsf{ACCESS}(i); c,\ s) & \rightsquigarrow & (e,\ c,\ e[i] :: s) \\ (e,\ \mathsf{GRAB}; c,\ v :: s) & \rightsquigarrow & (v :: e,\ c,\ s) \end{array}$$

These reductions correspond to a variable and a λ-abstraction, provided that binders and variables are encoded using de Bruijn indices. For some

more flavour, here are the detailed compilation of a variable and λ-abstraction of the symbolic calculus presented with de Bruijn indices:

$$\begin{aligned} [\![i]\!] &= \mathsf{ACCESS}(i) \\ [\![\lambda t]\!] &= \mathsf{CLOSURE}(\mathsf{GRAB}; [\![t]\!]; \mathsf{RETURN}) \end{aligned}$$

Even if the few rules shown here are a bit simplified, everything works *mutatis mutandis* like OCaml's implementation of the ZAM except for accumulators. Accumulators are dealt with by cleverness as described in [27].

3.2.2 Adding machine integers

Extending the symbolic calculus

To implement our new reduction rules for integer arithmetic efficiently, we shall extend the symbolic calculus with a notion of integers. Integers in this sense – noted m for *machine* integers – are fixed size binary words (they support both arithmetic modulo and carries). Symbolic calculus's integers will act as counterparts to the $\mathsf{In}\ d_{n-1}\ \ldots\ d_0$, as defined previously. Of course, as the symbolic calculus deals with open terms it won't always be possible to compile an $\mathsf{In}\ d_{n-1}\ \ldots\ d_0$ into an integer. To alleviate the verbosity of this section, we shall write p instead of $\mathsf{In}\ d_{n-1}\ \ldots\ d_0$, whether open or close.

When close we shall write $\langle p \rangle$ the integer corresponding to p. Conversely we will write $\rangle m \langle$ for the p corresponding to m. These two conversions need to be reflected in the symbolic calculus. Integers are introduced with:

$$p \rightsquigarrow \langle p \rangle$$

They also need to be morphed back in case a pattern matching is needed:

$$\textbf{match}\ \mathrm{m}\ \textbf{with} \ldots \rightsquigarrow \textbf{match}\ \rangle m \langle\ \textbf{with} \ldots$$

Notice that closed p-s of the form $\mathsf{In}\ v_{n-1}\ \ldots\ v_0$ used to be values in the symbolic calculus, but in this new-and-extended version, they ought not to be. Machine integer, on the other hand, are values. As such they need to be handled by the function $\mathcal{R}$:

$$\mathcal{R}(m) = \rangle m \langle$$

With this in mind we can handle arithmetic operations (by which we mean *addition*). We are given an addition (+) coming from Coq, it has a counterpart $+_M$ which operates on machine integers. It comes equipped with the following rewrite rule:

$$m_1 + m_2 \rightsquigarrow m_1 +_M m_2$$

Notice that the symbolic calculus comes with a deterministic rewrite strategy and this rule conflicts with the normal rule on terms of the form $v_1 + v_2$ (presumably a β-reduction). We need to just forbid the latter rule when both v_1 and v_2 are machine integers.

Compiling the extensions

To implement these extensions to the symbolic calculus we must modify the virtual machine accordingly. First, it needs a notion of integer to compile symbolic calculus's ones. We'll write them m as well, so that the compilation of an integer will be simply:

$$[\![m]\!] = \mathsf{PUSH}(m)$$

Where PUSH has the straightforward semantics of pushing m to the stack:

$$(e,\ \mathsf{PUSH}(m); c,\ s) \rightsquigarrow (e,\ c,\ m :: s)$$

The rules to construct and destruct integers are reflected explicitly in the virtual machine, they are named OFINT and TOINT:

$$\begin{aligned}(e, \mathsf{OFINT}; c, d_0 :: \ldots :: d_{n-1} :: s) &\rightsquigarrow (e, c, \langle \mathsf{In}\ d_0 \ldots\ d_{n-1}\rangle :: s)\\ (e, \mathsf{TOINT}; c, m :: s) &\rightsquigarrow (e, c, \rangle m \langle :: s)\end{aligned}$$

Of course, it will happen that among they arguments of OFINT there will be an accumulator, in which case OFINT cannot build an m, it will build the corresponding p instead. Dually, TOINT might get a p rather than an m, in which case it will behave as the identity.

On the compilation side, OFINT is used to compile p-s so that machine integers are used as soon as soon as possible (this reflects the fact that, in the symbolic calculus, close p-s are no values):

$$[\![\mathsf{In}\ t_0\ \ldots\ t_{n-1}]\!] = [\![t_{n-1}]\!]; \ldots; [\![t_0]\!]; \mathsf{OFINT}$$

The instruction TOINT is used in the compilation of pattern matching. We shall cast a veil on the precise rule to preserve the sanity of the reader. Suffices to say that a TOINT is inserted before any pattern matching over type int to ensure they are well formed.

Finally, addition also has a dedicated instruction ADD, which performs addition over machine integers, presumably using the processor's own addition procedure.

$$(e,\ \mathsf{ADD}; c,\ m_1 :: m_2 :: s) \rightsquigarrow (e,\ c,\ (m_1 +_M m_2) :: s)$$

Here again, the ADD instruction will not always have machine integers as operands. When the case arises, ADD falls back to the normal behaviour or addition as specified by the original Coq term. Compilation of addition follows:

$$[\![t_1 + t_2]\!] = [\![t_2]\!]; [\![t_1]\!]; \mathsf{ADD}$$

It deserves notice that our new instructions rely on the ability to discern m-s from p-s. It so happens that there is an easy solution to that problem. Values of the form p are represented as *blocks*, that is *pointers*. When the size of machine integers are well-chosen, they can be implemented without pointers. The garbage collector also needs to distinguish between pointer and non-pointer values. For this reason, values of the virtual machine have a

special bit marking its status: for a pointer it is set to 0, otherwise it is set to 1. Our instructions match on this bit to test the form of their operands.

As a consequence, Coq being supposed to work on 32 bit machines and higher, the size of our machine integers is 31 (that is the maximum size available after reserving the bit used for garbage collection). In theory we could use any size but any bigger than 31 would require *boxing* integers, and they would require another trick to test which representation they use. Bigger integers mean some performance gains, boxed integers mean performance losses, but preliminary tests suggests that the loss would be largely outweighed by the gains. There is, also, an option in-between, which consists in providing unboxed 63 bit integers on 64 bit computers and boxing them on 32 bit ones.

We have developed a reasonable library of primitive functions for the type int. It contains the usual functions (addition, multiplication, square root, comparison, logical functions, shifts) but also some iterators. Functions like

```
Definition foldi (A:Type) (F:A→A) (a:A) (n_s n_e:int) :=
 if n_s ≤= n_e then
   (fix aux (i:int) (ai:A) {
       if i = n_s then F i ai else aux (i-1) (F i ai)
    }) n_e a
 else a.
```

that computes $\mathsf{F}\, n_s\, (\mathsf{F}\, n_{s+1}\, (\ldots\, (\mathsf{F}\, n_e\, a) \ldots))$ cannot be defined on top of our library. Because of the definition of int, this is not structurally recursive so Coq cannot establish that it always terminates. If they were not primitive, there would be no way to perform recursion on integers efficiently.

3.3 Performance

The figures reported in the Figure 3.1 are reproduced from the CoqPrime original benchmarks

Natural numbers implemented with these machine integers are very efficient indeed, but there is still some way to go before we can claim comparable performance with mainstream programming language.

The benchmark in Figure 3.1 is done with the CoqPrime library [54, 29], whose purpose is to verify, in Coq, the primality of large numbers. In the case of Figure 3.1, the verification procedure is run on some prime Mersenne numbers with different implementation of the type of natural numbers. The date and author of the discovery of each prime Mersenne is provided for the reader's entertainment. There are four implementations of natural numbers considered: N is the implementation presented in Section 3.1, w8 corresponds the "two digit" implementation rooted in the enumerated type w8, int corresponds to the "two digit" implementation rooted with machine integers, and Big_int is the implementation of unbounded numbers provided by OCaml (the tests are then run after extraction).

Each step is a formidable improvement over the previous one, as expected. It is worth noticing also that N uses significantly more space than w8 rooted numbers which, in turn, uses much more than int based ones. Even though it does not usually receives as much attention as time, space is still a resource which can be scarce. It is even more so in garbage collected language such as Coq, as more space usage means more pressure on the garbage collector, which can be felt strongly on the time performance.

Mersenne number		Test times (s)				
Value	# digits	N	w8	int	Big_int	Discovery
$2^{127}-1$	(39)	0.73	0.04	0.01	0.00	1876 by Lucas
$2^{521}-1$	(157)	53.00	1.85	0.10	0.00	1952 by Robinson
$2^{607}-1$	(183)	84.00	2.78	0.14	0.00	1952 by Robinson
$2^{1279}-1$	(386)	827.00	20.21	1.10	0.02	1952 by Robinson
$2^{2203}-1$	(664)	4421.00	89.00	4.50	0.08	1952 by Robinson
$2^{2281}-1$	(687)	4964.00	97.59	5.00	0.09	1952 by Robinson
$2^{3217}-1$	(969)	14680.00	237.65	11.70	0.22	1957 by Riesel
$2^{4253}-1$	(1281)	35198.00	494.09	24.90	0.60	1961 by Hurwitz
$2^{4423}-1$	(1332)	39766.00	563.00	27.40	0.67	1961 by Hurwitz
$2^{9689}-1$	(2917)		5304.00	214.00	5.89	1963 by Gillies
$2^{9941}-1$	(2993)		5650.63	229.00	6.32	1963 by Gillies
$2^{11213}-1$	(3376)		76707.00	308.00	11.25	1963 by Gillies
$2^{19937}-1$	(6002)		34653.12	1405.00	45.75	1971 by Tuckerman
$2^{21701}-1$	(6533)		43746.21	1736.00	58.56	1978 by Noll & Nickel
$2^{23209}-1$	(6987)		41210.56	2020.00	88.43	1979 by Noll & Nickel
$2^{44497}-1$	(13395)		282784.09	11246.00	476.75	1979 by Nelson & Slowinski

Figure 3.1: Compared performance on verifying prime Mersenne numbers

We should comment the difference between our int rooted numbers and the performance of OCaml's Big_int – which, the benchmarks suggest, is about 20 times faster. There are a number of factors which can explain the difference. The first source of inefficiency is the extra tests done by Coq's virtual machine to determine whether its dealing with close terms or open ones. This is confirmed by tests showing that extracting int to OCaml's 31 bit performs noticeably better. This inefficiency might be aggravated by the fact that these tests have been implemented a bit naively, resulting in many unnecessary tests and jumps. Another difference is that Big_int is programmed in C, which means in particular that it benefits from native code performance even in OCaml's virtual machine. This could be partially addressed if native compilation was available for Coq which might be in a near future [15]. Finally, Big_int implements numbers as arrays, it might be that this is simply more efficient than binary trees.

There is even faster than OCaml's Big_int: the tool of choice for numerical computations is the C library called GMP [2]. The secret to GMP's performance is to perform computation in place. Adapting Coq to allow in place computation is no small endeavour. Maybe it would be possible to extend Coq with something like Clean's uniqueness types [19] which could be seen, on the logical side, as a flavour of implication.

On a related note, it has been observed in a number of applications, such as the kernel computations of Section 2.5, that implementations of rational numbers using the machine numbers do not offer, in the present state, a very significant improvement over the historical representation using the numbers as list of bits (N). This suggests that Coq would benefit from more clever implementations of rational numbers.

3.4 Related works

The methodology presented in this chapter is in no way restricted to (fixed size) integers – if we except the discriminating use of the garbage collector bit. And as a matter of fact, it has been used to add arrays to Coq [8]. As mentioned above, Coq cannot really cope with true arrays, so actually these are *persistent arrays* as described in [17]. These arrays present a functional interface but use imperative computation internally. In particular, they use a real array so that if persistence is never used during a computation they perform much like real arrays – though their is some overhead due to internal bookkeeping. On the other hand, they perform rather badly if persistence is used heavily; though they would cope quite well with mild use of persistence (typically if it is used only as a backtracking mechanism).

Machine integers and arrays have been used successfully to formally verify traces of SAT solvers [8]. This work, despite its success, has emphasised strong limitations of our minimalistic approach to Coq's enrichment. The fact is that, in our approach, integers are in a compact form only during conversion, which means that they are in "Coq form" both for storing and type checking. In applications which use explicitly a lot of numbers it proves to be really painful. For this reason a work has been initiating by Benjamin Grégoire taking the hard road to additional primitive types. It has proven to be a long implementation work with a lot of details to take care of (for instance the fact that the type of arrays is covariant on its parameter, hence being acceptable for the positivity condition). Properties of primitive types are axiomatised and are not given any reduction rules. This means that some program can be written but not run (though they can hopefully be transformed such as to be rendered executable).

Still, it might be the future of efficient computation in Coq. It has been brought to our attention that in order to have floating point computations in Coq (presumably in the style of MPFR [1]), it was quite unrealistic to hope to give a Coq type which would perform identically to the underlying C library. This would tilt the balance more in the direction of the hard path. It may be an interesting experiment to see if Coq could be given a Foreign Function Interface to be able to plug such extensions dynamically rather than harcoding them in the kernel of Coq.

Chapter Four

A new tactic engine

> Oh! it is absurd to have a hard and fast rule about what one should read and what one shouldn't. More than half of modern culture depends on what one shouldn't read.
>
> Oscar Wilde, *The importance of being earnest*

When programming with dependent types, it often becomes necessary to actually use Coq tactics to write programs. The **Program** constructs recently added to Coq [50] make it rarer. However, traditional Coq tactics often prove unadapted to the task. The same issues actually arise in proofs about dependently typed programs as well – where tactics are the rule rather than the exception.

To give an idea of what we are complaining about, let us consider a Coq implementation of categories:

```
Record Category := {
  Obj : Type;
  Hom : Obj → Obj → Set;
  id : forall A:Obj, Hom A A;
  comp : forall A B C:Obj, Hom A B → Hom B C → Hom A C;
  ⋮
}
```

Now let us suppose we have a property P over categories, and we want to prove the following goal:

⊢ **exists** c:Category, P c

The traditional way to prove this goal, in Coq, is to use the tactic (exists b) for a well chosen category D which transforms the goal into

⊢ P b

Now, typically, this is done by exiting the proof, defining a suitable b (which will be global even if it is only useful for the proof at hand), reenter the proof, provide b, go on. This can be fairly irritating, when possible at all (what if b depends on some local context?). There is an alternative – more

recent – tactic to handle proofs of existential statement. It does not require the user to provide a witness for c, it replace it by an existential variable. In our case, using the tactic eexists produces the following goal:

$\vdash$ P ?c

which is very fine for automated deduction, notably less for interactive proving. It somehow expects that ?c will be filled in later by unification with some lemma.

Arguably what would be needed for our more or less imaginary case, is a tactic which would produce *two* goals rather than a single one:

$\vdash$ Category (?c)
$\vdash$ P ?c

Solving the first goal would provide the witness for the second one. We could then solve the first goal in successive stages. For example, we could say that we want to consider a categories with sets (Set) as objects and relations as morphism (**fun** A B $\Rightarrow$ A $\longrightarrow$ B $\longrightarrow \Omega$) which would produce new goals:

$\vdash$ **forall** A:Set, A $\longrightarrow$ A $\longrightarrow \Omega$ (?id)
$\vdash$ **forall** A B C:Set, (A $\longrightarrow$ B $\longrightarrow \Omega$) $\rightarrow$
(B $\longrightarrow$ C $\longrightarrow \Omega$) $\rightarrow$
(A $\longrightarrow$ C $\longrightarrow \Omega$) (?comp)
$\vdots$

Identity would then be provided – it is the equality relation on A – then it would be the turn of composition. In a third stage we are left with the various properties of categories. And finally we can prove the category we have just defined verifies P.

Now of course in real life sized programs this sort of things can arise in a variety of guises, and with larger records than categories, sometimes trying to work one's way around some really tricky dependent types. In such cases, a facility such as argued for here become even more desirable.

Historical type of tactics

There is a good reason for no one yet to have provided such a feature to Coq: it is not possible. More honestly, the type which represents tactics internally is not compatible with such a capability. The type of tactics was actually inherited from one of the earliest proof assistant: LCF [26]. It has the following shape:

```
type tactic = goal → (goal list * validation)
and validation = term list → term
```

A tactic reduces a proof of a goal to a proof of a number of subgoals. It also returns a validation which given a proof of each subgoals constructs a proof of the initial goal. This operates under the assumption that all goals are independent from one another. Which was all very well, until the usage of existential variables was deemed necessary – for automation purposes among other things.

When existential variables appear in goals, goal sharing an existential variable are no longer independent (acting on one can change the other by modifying the content of the existential variable). In a sense, existential variables in proofs behave much like references in programs – or maybe more accurately, shared memory between parallel processes – and as such they need to have an address space shared between the goals, this changes the type of tactics to:

```
type tactic =
        goal * evar_map → (goal list * evar_map * validation)
```

Where the evar_map component represents this shared space. Tactics may update existential variables, the new evar_map is then fed to the next tactic to be run. There is a single evar_map shared among all goals (past, present and future) of a proof.

Existential variable contexts

Existential variables are given a type (as a matter of fact, a sequent) when it is created. It can later be instantiated with a term which might, in turn, contain existential variables – as it was the case in our example above: we gave a partial instantiation of the existential variable ?c which specified only its objects and morphisms.

Objects of type evar_map associates to each existential variable appearing in the corresponding proof its type and, when relevant, the term with which it is instantiated. It also keeps track of unsolved (higher-order) unification problems.

Changing the type of tactics

This refinement of the type of tactics does not suffice to allow for our feature. Indeed, if it allows goal to share existential variables, it cannot instantiate an existential variable when solving a goal – if only because the validation is executed when the goal is fully solved, hence the instantiation would happen too late. Conversely, it would not be possible to solve a goal by the instantiation of an existential variable.

It was, therefore necessary to go and change everything. Which was quite of a problem because the type of tactics is hardly abstract at all. Hence modifying it impacts code throughout the sources of Coq. It can be observed that most tactics are implemented using combinators, which could minimise the issue. However, it so happens that tactics use explicitly their first argument to inspect the goal. As we will see in the rest of this chapter it may not be desirable that the type of tactics be of the form goal * s → t. For instance it is useful, in order to design automated proof search strategies, to be able to define tactics acting simultaneously on multiple goals.

Multiple goal tactics have been implemented, independently, in Matita [9].

The work presented in this chapter consists, with the above motivations in mind, to design an abstract type for tactics such as to support changing its implementation in the future without impacting thousands of lines of code. This is accompanied by a new implementation for the type of tactics (and of proof in progress) which will be briefly described. It provides support for the feature which motivated this work, as well as multiple goal tactics and logic programming style backtracking. This chapter is an enhanced version of [51].

4.1 Refinement

A perk of having an unrestricted treatment, in proofs and tactics, of existential variables, is that a term with existential variables is isomorphic to a partial proof derivation. This remark seems to originate from Bengt Nordström [10] and has been studied in more details by César Muñoz [44], though in situations where is no existential variable in goals. This suggests that we need only one atomic tactic – which we call *refinement*. Whereas formerly, there was a need of a fairly large set of core tactics mimicking the derivation rules of the system. It is noteworthy that refinement is also at the heart of the language Agda2 [45], as the main tool both for constructing proof and programming.

The idea to use a single refinement tactic is, *mutatis mutandi*, equivalent to what Lengrand*&al.* propose in the setting of Pure Type Sequent Calculus [38].

Coq features a tactic called refine which performs a restricted version of refinement: it does not deal with existential variables at all. Also it is not meant as a primitive tactic. Still, Coq's refine tactic is a fragment of the refinement we shall use.

To sum up, at the most atomic level, we are given an intuitionistic sequent (a goal), and to advance in our proof, we provide a term with existential variables, which we read as a partial proof derivation, and we get back a list of new sequents (subgoals).

Goal sensitive

More often than not we want our partial proofs to depend on the sequent it is supposed apply to. For instance in the sequent $x{:}A \vdash A$, the term x is a proof. Here x is to be interpreted in the context of the sequent. Other reasons to depend on the goal include reporting useful errors and automated procedure which would build a partial proof depending on what the goal looks like.

A value which can only be interpreted in the context of a goal will be called *goal sensitive*. Goal sensitive values are represented as the type:

```
type α sensitive
```

which will stay abstract, meaning that only a few primitives know of its concrete representation. Its concrete representation can be something like a function taking as argument a goal and a context for existential variables, and returning a value of type α. Among the needed primitives, we need access to the content of the goal. This is provided, mostly, by the following two primitives:

```
val concl : types sensitive
val hyps : env sensitive
```

to access, respectively, the right-hand formula and the context of the goal under scrutiny. Contexts – represented by type env – are represented as association lists.

Now, of course, we need a way to build on these primitives to make new goal sensitive values; for instance the number of hypotheses of the sequent is expected to be an int sensitive. For that purpose we introduce a monad on the goal sensitive values:

```
val return : α → α sensitive
val (>-) : α sensitive → (α → β sensitive) → β sensitive
```

```
(* evar is the type of existential variables *)
type goal = evar
(* We use a reference to an evar_map because we need to be
able to modify it in later primitives *)
type α sensitive = goal → env → evar_map ref → α

let return x _ _ _ = x
let (>-) s k goal env rdefs =
  k (s goal env rdefs) goal env rdefs

let concl goal _ rdefs =
  Evd.evar_concl (Evd.find goal !rdefs)
let hyps goal _ rdefs =
  Evd.evar_hyps (Evd.find goal !rdefs)
```

Figure 4.1: Implementation of some primitives for goal sensitive values

where (return x) is to be understood as the same as x but pretending to depend on a goal and (>-) "propagates" the goal under consideration.

Figure 4.1 gives an implementation of these types and primitives. To illustrate, here is the code that computes the number of hypotheses:

```
let hcount =
    hyps >- fun hs →
    return (List.length hs)
```

Refining goals

With this material, we are ready to disclose and discuss the type of the refinement tactic:

```
val refine : refinable → subgoals sensitive
```

Let us take it apart bit by bit – starting from the right (sensitive) and finishing with the left (refinable). First it returns a goal sensitive value, as expected. This value has type subgoals which is simply a list of goals, except that it is made *private*, which is an OCaml keyword to say that anyone can use an element of that type as a list of goals, but only functions local to the current module can actually construct a value of type subgoals. This restriction exists to prevent accidentally writing "rogue" tactics, which are not defined in terms of refine, and may have an incorrect behaviour.

Let us make this a little more precise by stating explicitly that values of type subgoals sensitive are to be used as tactics. Indeed α sensitive are not only elements of type α defined in terms of an unknown goal, they also depend on a sufficiently rich context to express the state of a proof (mainly definitions and typing information of existential variables around, plus the current partial proofs of goals which are also dealt with in terms of existential variables), which they carry around and modify when needed, much in the spirit of the State monad in Haskell [3].

The last part of this is the type refinable of the argument of refine. Elements of type refinable are essentially terms. In fact, they are precisely terms carrying

around which of the existential variable in mention are "new" – as opposed to those who are shared with other terms. Definition of elements of type refinable are handled by a module which reads as:

```
module Refinable : sig
  type handle

  val make : (handle → term sensitive) → refinable sensitive
  val mkEvar : handle → env → types → term sensitive
end
```

It has actually a few more functions defined (half a dozen in total), but these two suffice for this discussion. First we notice the (abstract) type handle which is a sort of registration machine: whenever we create a new existential variable (through mkEvar) the handle is notified. The job of the handle is to keep track of all these existential variables. The function make says that if, in the context of a handle, we can build a term (actually a goal sensitive term), then we can get a goal sensitive refinable. Note that this refinable needs to be goal sensitive, because it modifies the existential variable context – the "state" in the α sensitive monad – as it introduces new existential variables. Finally mkEvar registers a new existential variable to a handle, given enough type information, and returns the corresponding term.

To put all this in practice here is the definition of a simple introduction tactic. That is a tactic which given the name x, takes a goal of the form $\Gamma \vdash A{\to}B$ and turns it into $\Gamma,x{:}A \vdash B$.

```
let intro x =
  concl >- fun c →
  let (_,a,b) = destProd c in
  hyps >- fun hp →
  let new_hyps = (x, a)::hp in
  Refinable.make (fun h →
    Refinable.mkEvar h new_hyps b >- fun hp →
    return (mkNamedLambda (x,a,e))
  ) >- fun r →
  refine r
```

Where destProd decomposes a formula of the shape $A{\to}B$, and mkNamedLambda (x,a,e) builds the term $\lambda x{:}A.\, e$.

To conclude this section, let us linger on the fact that all tactics *must* be defined in terms of refine. This allows to delegate all the bookkeeping of tactics to the refinement procedure. This is at least good for maintenance purposes. More importantly, this allows to control much more easily the correctness of tactics. It will not prevent from writing bogus tactics, but refine can be used to verify that the partial proof given by your favourite tactic is valid before applying it. Hence enforcing that one is warned as soon as possible when tactics go wrong. In the former implementation, proof terms were verified only when they were complete (when the command Qed was entered). When some tactic produced an incorrect subterm, it was often hard to track down. Each tactic was responsible for ensuring the correctness of the proof it generated.

On the other hand, as proof checking is done at the end of the proof, there is some incentive not to check the terms in the course of the proof as proof checking can be costly. This is how tactics like exact_no_check were created (this particular one gives a complete proof term for the current goal without checking that it is actually a proof of this goal). Even there, basing everything on refinement can help: we can have a flag to the refinement procedure prescribing whether or not to check the proof term. This can hopefully prevent some code duplication.

The next step would be to incorporate the refinement procedure into the kernel of Coq, so that it would be trusted. In this case it would have to verify, in all situation, that no ill-typed term is ever produced – hence no *no check* flag. As a consequence, there would be no more need for an extra validation step at the end of the proof as interactively built proof would be correct by construction.

4.2 Combining tactics

4.2.1 An abstract approach

So far, we haven't given a way to combine existing tactics into one; in other words our tactics can have a single refinement step. While this is not a limitation in expressiveness, this lacks some amount of flexibility. The most typical tactic combinators in Coq are the composition – t1;t2 applies t1 and then applies t2 to all generated subgoals – and the alternative – t1||t2 tries to apply t1, if it fails, it applies t2 instead. We will introduce, on the OCaml side, two combinators (<*>) and (<+>), respectively, to represent them.

We could actually make them act on the subgoals sensitive type. However this leaves little space for improvement. Let us take a small detour to see why we shall use a dedicated type for tactics that combine.

Formerly, the type of a proof in progress was a tree representing the tactics that were used in its course. This does not allow for goals that are solved by side effect, which we want to introduce. As a matter of fact it does not deal very cleanly with side effects at all, as the order in which the goals are solved does not appear in the proof. We propose a new implementation, where a proof is described by an existential variable context, goals being themselves described as particular existential variables. The state of the current proof, which we call a *view*, is one such context, together with some of its open goals, said to be *under focus*, lined up in a list – so they can be addressed by their position.

In Coq, the type for existential variable contexts is called evar_map.

```
type proofview
```

Following our policy, the type is abstract. Executing a tactic on a view returns another view. Now values of type subgoals sensitive act as tactics, for instance by applying them to one particular goal or to all the goals simultaneously. We could imagine other kinds of manipulation of views. For instance changing the order of its goals, which can be part of a tactic (for instance the destruct and induction tactics yield goals in a different order). There is no reason to restrict ourselves to tactics which can be encoded as subgoals sensitive-s. Therefore we shall consider a new type to represent tactics,

```
type α tactic = proofview → (α*proofview)

let tclUNIT a view = (a,view)

let (>=) t k view =
  let (a,view') = t view in
  k a view'

let (<*>) t1 t2 view =
  t2 (snd (t1 view))

let (<+>) t1 t2 view =
  try t1 view
  with _ → t2 view
```

Figure 4.2: A first implementation of tactics

which can be seen as being functions from views to views.

```
type tactic
```

which is, again, abstract.

Another feature we might want to add is the ability for tactics to communicate some information to the tactic that follows. We can imagine a tactic which never fails, but "returns" a boolean informing whether it progressed, or an introduction tactic which chooses a name for the new hypothesis and passes it to the following tactics. To reach that goal, we enrich the type of tactics with a type parameter:

```
type α tactic
```

which represents the "return type" of a tactic. As a matter of fact we can install a monad on the type of tactics:

```
val tclUNIT : α → α tactic
val (>=) : α tactic → (α → β tactic) → β tactic
```

It can be seen as a state monad with a proof view as the state. Now the composition

```
val (<*>) : α tactic → β tactic → β tactic
```

can be viewed as a special case of `(>=)`. Finally the alternative

```
val (<+>) : α tactic → α tactic → α tactic
```

has to be implemented using some kind of exception mechanism. We propose, in Figure 4.2, an implementation of these primitives.

We also need a primitive internalising `subgoals sensitive` as tactics:

```
val tclSENSITIVE : subgoals sensitive → unit tactic
```

As expected, it produces a tactic which returns unit, as return values are novelties of the tactic level. As a convention, we decide that tclSENSITIVE t applies t to every goal under focus. Of course we also have primitives to manipulate focus, for instance:

As the order matters, because of side effects, we specify that t is applied from the last goal to the first one. Also, goals that are closed by side effect before being considered are ignored

```
val tclFOCUS : int → int → α tactic → α tactic
```

which focuses on a range of goals, applies the tactic argument, and then unfocuses back, effectively splicing the produced goals in place of the range it originally focused on. This effectively gives the ability to choose a particular goal and to apply a tactic to it, restoring the traditional approach of Coq.

4.2.2 Leveraging the abstraction barrier

With all this abstraction done, it becomes easy to change the underlying type of tactics to support new features. As a conclusion to this section, we shall describe briefly how to support backtracking in tactics. More explicitly, by backtracking, we mean the property that (a<+>b)>=c would be equivalent to (a>=c)<+>(b>=c). Which is not the case with the implementation we sketched earlier. Indeed it has the property that if a succeed, then (a<+>b)>=c is equivalent to a>=c.

There are many ways to implement backtracking. For the prototype, we have used a two-continuation type, much in the spirit of [35], except that it does not behave as a monad transformer. We give tactics the following type:

```
type α nb_tactic  = proofview → α*proofview
type ρ fk = exn → ρ
type (α,ρ) sk = α → ρ fk → ρ
type α tactic = { go :
      ρ. (α, ρ nb_tactic) sk → ρ fk → ρ nb_tactic
}
```

This deserves some explanation. Values of type α nb_tactics are non-backtracking tactics, as presented at the beginning of this section. Functions of type ρ fk are failure continuations, they are passed an exception, that with which the previous tactic failed. (α,ρ) sk are success continuation, they are passed an element of type α which is the result of the previous tactic – which succeeded – and a failure continuation to know where to go if it fails. Actual tactics are wrapped inside a record because we want to universally quantify over the type argument ρ, something which, in OCaml, is only supported via records.

Now to see how it supports backtracking, we will show the definition of (>=) and (<+>).

```
let (>=) t k = { go = fun sk fk view →
      t.go (fun a fk' → (k a).go sk fk') fk view
    }
```

This reads: "t>=k executes t, if it succeeds and returns a it then executes (k a) with its success continuation sk otherwise it executes its failure continuation fk".

```
let <+> t1 t2 = { go = fun sk fk view →
      t1.go sk (fun _ → t2.go sk fk view) view
    }
```

This reads: "`t1<+>t2` executes `t1` and continues with its success continuation. If it fails then the error is ignored and we go on with `t2` with the current success and failure continuations."

The failure continuations act as backtracking stacks which are propagated by the atomic tactics. In particular if we have a tactic of the form `(a<+>b)<+>c` where `a` is atomic, its success continuation `sk` is passed the following failure continuation:

```
fun _ → b.go sk (fun _ → c.go sk fk)
```

If it fails it then tries `b` then `sk`, if it fails again it then tries `c` then `sk`. This is precisely what we expected. Also note that `(<+>)` is also associative.

With this implementation we have strayed far from simple functions from a goal to a list of goals. However we have given an API which is stable under changes of implementation (and which abstract away the complexity of the underlying implementation: working with double continuation values is fairly destructive to ones brain cells). As a matter of fact there might be a better suited way to support backtracking for our tactics, it won't be a problem to experiment in the future thanks to the abstraction layer.

4.3 About the implementation

The API we proposed in this chapter, which is part of the development branch of Coq, is actually composed of 25 primitives for the α sensitive part (including the Refinable module) and 17 for the α tactic part, at the time when this article has was written. This represents less than 800 lines of code. This should be compared to the roughly 80 primitives (plus a few additional primitives scattered throughout the code, since the type of tactics was not abstract) and around 2000 lines of code for the legacy core tactic machinery of Coq (the proof manipulation part has been shortened even more, through better code sharing with other parts of the code base).

It would not have been feasible, though, to port all the code base to this new API at this stage, hence we have built a compatibility layer which includes tactics with a similar type than earlier as a sub-case of subgoals sensitive. This layer breaks the abstraction a bit, but is still fairly maintainable. The trouble is that it didn't allow us to eliminate much of the old code for now.

4.4 Further work

At the moment of this writing, the code described in this chapter is part of the development branch of Coq. It is undergoing quite a bit of work to repair the inevitable bugs and incompatibilities introduced by such a wide change to the code base. It has revealed a few inefficiencies in some functions of Coq which were innocuous with the former implementation of tactics. Some of them were really serious, a few are still being investigated.

The next move will be to permit the tactics available to the user to be extended with the capabilities of the new machinery. For the moment this is only possible by providing an entirely new tactic language, but it would be desirable, at least in a first time, to provide a few tactics to the Ltac [4] and Ssreflect [25] users. This amounts mostly to implement Ltac tacticals in terms

of the new tactics. This has no reason to be problematic, but it will certainly represent quite some hard work. This is an important milestone because it is only then that the new proof engine will enable new features.

When these are done, we will be left with much more endearing tasks with real bit of science in them. The first would be the design of a new tactic language really drawing on the capabilities of the new proof engine. It is imaginable to follow Moggi [43] and provide a language in the style of ML which makes no explicit mention of its monadic semantics. As for the details, it is hard to tell. Some are interested in a set of tactics which is more modular than Ltac, typically with primitives to control the behaviour of automated tactics. An example that has been given is an auto tactic whose unification procedure can be provided by the user. The motto here would be *fewer tactics in smaller chunks* – in a spirit somehow similar to Ssreflect.

Another point to be addressed concerns rewriting. Rewriting tactics are somewhat parametrised with rewriting strategies. These strategies resemble quite closely proof search strategies (see [34] for a discussion on this remark). Actually, they share more than these aspects: in Coq both implementations are almost identical. There are two implementations of backtracking using evil two-continuation monads; as if one was not enough. This code should be shared. However it is entirely unclear how. The monad transformer approach of [35] cannot help here as, at least in the case of the proof engine, the tactic monad cannot be split into an inner state monad and an outer backtracking monad transformer because our monad backtracks on the state of the proof as well as on the value, whereas in the monad transformer view it is not possible to backtrack on the state. It would be interesting to be able to come up with a representation of goals which could serve not only as a "proof to be completed" but also as a "rewrite to be completed".

Part Three

Faster, Higher, Stronger

Chapter Five

On the category of sets

The visions dancing in my mind
The early dawn, the shades of time
Twilight crawling through my windowpane
Am I awake or do I dream?
The strangest pictures I have seen
Night is day and twilight's gone away

With your head held high and your scarlet lies
You came down to me from the open skies
It's either real or it's a dream
There's nothing that is in between...

Electric Light Orchestra, *Twilight*

chematically, the verified development of homology described in Chapter 2 has three layers: most homology theory is developed in the framework of preabelian categories, above it some work is done in categories of graded objects, and below these there are sets. As we alluded to in Section 1.1, it would be somewhat desirable to insert an intermediate abstraction layer between sets and preabelian categories, if only to avoid the need of explicitly manipulating functions as records.

The approach with sought was to give a good abstraction of the category of sets, in which we can define groups and show that they form a preabelian category. Such a categorical abstraction would give us a general enough set of combinators to avoid any explicit proof that the functions we define do respect the equality of their domain: they would do so by construction.

In a sense, this would amount to defining a programming language for sets and functions. And it does raise classical design issues. Indeed, as we target intensive computations (computer algebra in general requires a lot of speed) we should avoid as much as possible to lose efficiency. Also a programming language tries to include convenience features like labelled sums or record in order to make programs easier to read and maintain.

We shall, in a first time, study the properties of the category of sets, from a point of view which makes apparent its programming language aspect, and in particular the type theoretic aspect. We shall, then, conclude the chapter with a discussion on the problematic of efficiency and convenience.

5.1 Topoi

Topoi are a particular kind of category which have been introduced as a mean to give an alternate description of set theory. They have had some success in this matter. In fact, if their basic constructions make them suitable as a *definition* of sets and functions, there is a wide variety of topoi. Some topoi enjoy the principle of choice, some do not. Some topoi verify the continuum hypothesis, others do not. In a sense, they allow to vary what one means by *set*. For the interested reader, categories of sheaves are topoi, meaning that sheaves can be considered as sets of sorts.

Defining topoi will also be the occasion to visit some more advanced concepts in category theory.

Functor

Categories have their own notion of morphisms, called *functors*. A functor, or *covariant* functor, is a mapping between objects that preserves the structure. Given two categories $\mathcal{B}$ and $\mathcal{C}$:

$$\begin{array}{l} \mathrm{Functor}\, \mathcal{B}\, \mathcal{C} \\ \quad \mathrm{F}_o \;:\; \mathcal{B} \rightarrowtail \mathcal{C} \\ \quad \mathrm{F}_h \;:\; \prod_{\mathrm{A}\,\mathrm{B}\,:\,\mathcal{B}} \mathrm{Hom}_{\mathcal{B}}\,\mathrm{A}\,\mathrm{B} \longrightarrow \mathrm{Hom}_{\mathcal{C}}\,(\mathrm{F}_o\,\mathrm{A})\,(\mathrm{F}_o\,\mathrm{B}) \\ \quad _ \;\; [:]\; \mathrm{F}_h\, 1_{\mathrm{A}} = 1_{\mathrm{F}_o\,\mathrm{A}} \\ \quad _ \;\; [:]\; \mathrm{F}_h\,(f;g) = (\mathrm{F}_h\, f);(\mathrm{F}_h\, g) \end{array}$$

Notice that the preservation of arrows is *computational* – it is, as a matter of fact a *function* – contrary to the preservation properties we have encountered so far. For a functor F, we shall usually write F for both F_o and F_h.

The commutation of $(\text{-} \times \mathrm{A})$ with identity and composition are then a matter of applying the equational theory of products. For instance, $(\pi_1 1, \pi_2 1) = (1\pi_1, 1\pi_2) = 1$.

Let us consider the mapping $(\text{-} \times \mathrm{A})$ (in a category where it makes sense) which maps each B to its product with A. It can be lifted to a functor, by mapping each $\mathrm{B} \xrightarrow{f} \mathrm{C}$ to the arrow $\mathrm{B} \times \mathrm{A} \xrightarrow{(\pi_1 f, \pi_2 1)} \mathrm{C} \times \mathrm{A}$. From now on we will refer freely to $(\text{-} \times \mathrm{A})$ as a functor.

Another important functor we shall consider has the map $(\mathrm{Hom}\,\mathrm{A}\ \text{-}){:}\mathcal{C} \rightarrowtail \mathcal{S}$ – where $\mathcal{S}$ is the category of sets – for some object $\mathrm{A}{:}\mathcal{C}$. For an arrow $\mathrm{B} \xrightarrow{f} \mathrm{C}$, $\mathrm{Hom}\,\mathrm{A}\,\mathrm{B} \xrightarrow{\mathrm{Hom}\,\mathrm{A}\,f} \mathrm{Hom}\,\mathrm{B}\,\mathrm{C}$ is defined as the postcomposition with f: $(\mathrm{Hom}\,\mathrm{A}\,f)\,g = gf$.

Contravariant functor

We shall also need a variant of functors said to be *contravariant*. Contrary to covariant ones, contravariant functors reverse the direction of arrows:

$$\begin{array}{l} \mathrm{CoFunctor}\, \mathcal{B}\, \mathcal{C} \\ \quad \mathrm{F}_o \;:\; \mathcal{B} \rightarrowtail \mathcal{C} \\ \quad \mathrm{F}_h \;:\; \prod_{\mathrm{A}\,\mathrm{B}\,:\,\mathcal{B}} \mathrm{Hom}_{\mathcal{B}}\,\mathrm{A}\,\mathrm{B} \longrightarrow \mathrm{Hom}_{\mathcal{C}}\,(\mathrm{F}_o\,\mathrm{B})\,(\mathrm{F}_o\,\mathrm{A}) \\ \quad _ \;\; [:]\; \forall \mathrm{A}{:}\mathcal{B}.\, \mathrm{F}_h\, 1_{\mathrm{A}} = 1_{\mathrm{F}_o\,\mathrm{A}} \\ \quad _ \;\; [:]\; \mathrm{F}_h\,(f;g) = (\mathrm{F}_h\, g);(\mathrm{F}_h\, f) \end{array}$$

The archetypal contravariant functor is $(\mathrm{Hom}\ \text{-}\ \mathrm{A})$. In this case $(\mathrm{Hom}\,f\,\mathrm{A})$ is given by precomposition: $(\mathrm{Hom}\,f\,\mathrm{A})\,g = fg$.

Functors, either covariant or contravariant, compose: given a functor F from $\mathcal{B}$ $\mathcal{C}$ and a functor G from $\mathcal{C}$ to $\mathcal{D}$, there is a functor $(\mathrm{G}\,(\mathrm{F}\ \text{-}))$ $\mathcal{B}$ to $\mathcal{D}$ obtained by composing both their object and arrow components. The composition of two covariant functors is covariant, so is the composition of two contravariant functors. The composition of a covariant functor and a contravariant one is contravariant.

Adjunction

Given two categories $\mathcal{B}$ and $\mathcal{C}$ and two (covariant) functors F:Functor $\mathcal{B}\,\mathcal{C}$ and G:Functor $\mathcal{C}\,\mathcal{B}$, we shall say that F and G are adjoints if they are equipped, for every B:$\mathcal{B}$ and C:$\mathcal{C}$ with an isomorphism $(\varphi_{\mathrm{B,C}}, \varphi^{-1}_{\mathrm{B,C}})$ between the sets $(\mathrm{Hom}_{\mathcal{C}}\,(\mathrm{F\,B})\,\mathrm{C})$ and $(\mathrm{Hom}_{\mathcal{B}}\,\mathrm{B}\,(\mathrm{G\,C}))$ which is natural, that is such that for every $\mathrm{B}' \xrightarrow{h} \mathrm{B}$ the following diagram commutes:

$$\begin{array}{ccc}
\mathrm{Hom}_{\mathcal{C}}\,(\mathrm{F\,B})\,\mathrm{C} & \xrightarrow{\varphi_{\mathrm{B,C}}} & \mathrm{Hom}_{\mathcal{B}}\,\mathrm{B}\,(\mathrm{G\,C}) \\
\Big\downarrow{\scriptstyle \mathrm{Hom}\,(\mathrm{F}\,h)\,\mathrm{C}} & & \Big\downarrow{\scriptstyle \mathrm{Hom}\,h\,(\mathrm{G\,C})} \\
\mathrm{Hom}_{\mathcal{C}}\,(\mathrm{F\,B}')\,\mathrm{C} & \xrightarrow{\varphi_{\mathrm{B}',\mathrm{C}}} & \mathrm{Hom}_{\mathcal{B}}\,\mathrm{B}'\,(\mathrm{G\,C})
\end{array}$$

and for every $\mathrm{C} \xrightarrow{k} \mathrm{C}'$ the following diagram commutes:

$$\begin{array}{ccc}
\mathrm{Hom}_{\mathcal{C}}\,(\mathrm{F\,B})\,\mathrm{C} & \xrightarrow{\varphi_{\mathrm{B,C}}} & \mathrm{Hom}_{\mathcal{B}}\,\mathrm{B}\,(\mathrm{G\,C}) \\
\Big\downarrow{\scriptstyle \mathrm{Hom}\,(\mathrm{F\,B})\,k} & & \Big\downarrow{\scriptstyle \mathrm{Hom}\,\mathrm{B}\,(\mathrm{G}\,k)} \\
\mathrm{Hom}_{\mathcal{C}}\,(\mathrm{F\,B})\,\mathrm{C}' & \xrightarrow{\varphi_{\mathrm{B,C}'}} & \mathrm{Hom}_{\mathcal{B}}\,\mathrm{B}\,(\mathrm{G\,C}')
\end{array}$$

Put in other words, the isomorphism $(\varphi_{\mathrm{B,C}}, \varphi^{-1}_{\mathrm{B,C}})$ is defined the *same way* for every B and C. In the realm of type theory, a corresponding property is that of parametricity: an OCaml function of type α list $\rightarrow$ α list is parametric in that it cannot be defined differently for various instantiations of α.

In such an adjunction, F is said to be the left adjoint and G the right adjoint.

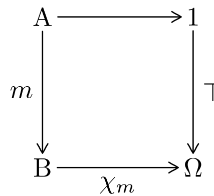
Cartesian closed category

A cartesian closed category is a category $\mathcal{C}$ with a terminal object 1 and all products $\text{-} \times \text{-}$ such that for every object A of $\mathcal{C}$, the product functor $\text{-} \times \mathrm{A}$ has a right adjoint -^{A} called the exponential functor.

The exponential is hence such that $\mathrm{Hom}\,(\mathrm{B}\times\mathrm{A})\,\mathrm{C}$ is isomorphic to $\mathrm{Hom}\,\mathrm{B}\,(\mathrm{C}^{\mathrm{A}})$. In the category of sets – which is cartesian closed – this exponential object is the set of functions and the isomorphism is given by currying.

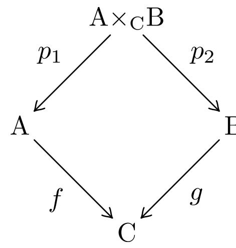
Subobject classifier

In a category with a terminal element 1, a *subobject classifier*, is an object Ω and an arrow $1 \xrightarrow{\top} \Omega$ such that for every monomorphism $A \xrightarrow{m} B$ there is an arrow $B \xrightarrow{\chi_m} \Omega$ such that the following diagram is a pullback:

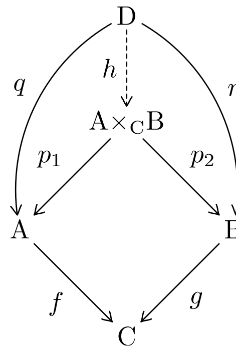


Pullback

Given two arrows $A \xrightarrow{f} C$ and $B \xrightarrow{g} C$ a *pullback* of f and g is a square – sometimes called a *prepullback*:



which is universal, that is such that for any object D and arrows $D \xrightarrow{q} A$ and $D \xrightarrow{r} B$ with $qf = rg$, there is a unique arrow $D \xrightarrow{h} A{\times_C}B$ such that $q = hp_1$ and $r = hp_2$:



In the category of sets, pullbacks are given by fibre products $\{(a, b) : A \times B \mid f\, a = g\, b\}$. As this suggests, in a category with enough products and equalisers, a pullback can be constructed as the equaliser of $A \times B \xrightarrow{\pi_1 f} C$ and $A \times B \xrightarrow{\pi_2 g} C$.

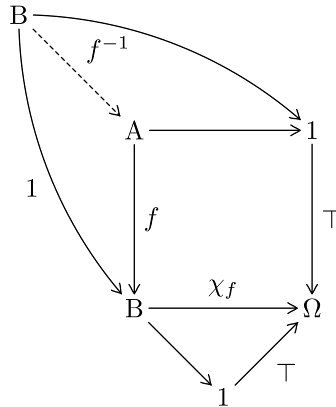
From now on, we shall suppose given a category $\mathcal{C}$ with all pullbacks.

Topos

A topos is a cartesian closed category with a subobject classifier. It is a fairly robust notion. Many of the constructions of topoi can be retrieved from others.

For instance the exponential objects can be reconstructed from a power object functor – giving a definition much like that which is traditional in ZF set theory.

Unfortunately, topoi suffer from the same problem as abelian categories – described in Section 2.3: the principle of unique choice is always valid in a topos. The proof of that is contained in a single commutative diagram:



The diagram is read starting from the square: since m is monic there is such a pullback. Then, writing ! for the unique arrow from any object to 1, $f!\top = !\top = f\chi_f$, which, since f is epic, lets the lower triangle commutes. The upper triangle is a prepullback diagram (since $1!\top = !\top$), hence there is a unique arrow f^{-1} such that $f^{-1}f = 1$, that is a preinverse to f. Since f is monic, it is actually an inverse.

where f is an arrow which is both epic and monic.

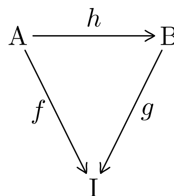
This ruins any chance for our category of sets to be a topos. Fortunately, as it was the case with the abelian categories, there is another – weaker – paradigm which has been less studied but is not unheard of at all where our sets fit.

5.2 Quasitopoi

Just as preabelian categories were our unique-choice free version of abelian categories, quasitopoi have a topos feel but need not verify the principle of unique-choice.

Slice category

Let I be an object of $\mathcal{C}$, the slice category $\mathcal{C}/_{\mathrm{I}}$ is defined as the category where objects are arrows $\mathrm{A} \xrightarrow{f} \mathrm{I}$. Morphisms between two objects $\mathrm{A} \xrightarrow{f} \mathrm{I}$ and $\mathrm{B} \xrightarrow{g} \mathrm{I}$ are given by morphisms $\mathrm{A} \xrightarrow{h} \mathrm{B}$ which make the following diagram commute:



which we will also write as $\begin{smallmatrix}\mathrm{A}\\ \downarrow f\\ \mathrm{I}\end{smallmatrix} \xrightarrow{h} \begin{smallmatrix}\mathrm{B}\\ \downarrow g\\ \mathrm{I}\end{smallmatrix}$

To motivate, and illustrate, these slice categories, we shall pretend that $\mathcal{C}$ is the category of sets. In this case the category $\mathcal{C}/_{\mathrm{I}}$ can be seen as the category

of *families indexed by* I. Indeed, a family $(\mathrm{A}_i)_{i:\mathrm{I}}$ can be, equivalently, seen as the set $\sum_{i\,:\,\mathrm{I}} \mathrm{A}_i$ of pairs (i, A_i). When a family is given in the latter form, the first projection $\mathrm{A} \xrightarrow{\pi_1} \mathrm{I}$ is sufficient to recover the former. It is given (up to isomorphism) by $(\{a : \mathrm{A} \mid \pi_1\, a = i\})_{i:\mathrm{I}}$. Hence, we can see the objects of $\mathcal{C}/_\mathrm{I}$ as families indexed by I given as a set of pairs and its first projection. We shall say that a family is in functional form if it is given as $(\mathrm{A}_i)_{i:\mathrm{I}}$, and that it is in set form if it is given as a set and a projection.

We can let this family intuition guide us through a process of finding out more about slice categories. First we need to figure how the morphisms of $\mathcal{C}/_\mathrm{I}$ fit the picture. A reasonable notion of morphism between families $(\mathrm{A}_i)_{i:\mathrm{I}}$ and $(\mathrm{B}_i)_{i:\mathrm{I}}$ would be a family of functions $(\mathrm{A}_i)_{i:\mathrm{I}} \xrightarrow{h_i} \mathrm{B}_i$. For set forms, it corresponds indeed to a function h which preserves the first projection. Indeed, given a functional-form morphism, we can construct such an h as $\lambda(i,a){:}\mathrm{A}.\,(i, h_i\, a)$.Conversly, for $\overset{\mathrm{A}}{\underset{\mathrm{I}}{\downarrow}} f \xrightarrow{h} \overset{\mathrm{B}}{\underset{\mathrm{I}}{\downarrow}} g$ a set-form morphism, h_i is given by the restriction of h to the set $\{a : \mathrm{A} \mid f\, a = i\}$ which has codomain $\{b : \mathrm{B} \mid g\, b = i\}$ since $g\,(h\, a) = f\, a$.

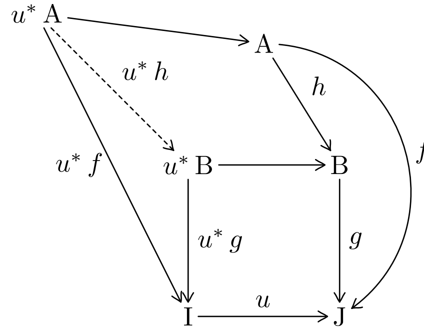
Base change

Given a function u from I to J and a family $(\mathrm{A}_j)_{j:\mathrm{J}}$, precomposing with u makes $(\mathrm{A}_j)_{j:\mathrm{J}}$ into a family indexed by I: $(\mathrm{A}_{u\,i})_{i:\mathrm{I}}$. This precomposition has a counterpart in the set forms. The fibre product $\{(i,a) : \mathrm{I} \times \sum_{j\,:\,\mathrm{J}} \mathrm{A}_j \mid u\, i = \pi_1\, a\}$ happens to be isomorphic to the set $\sum_{i\,:\,\mathrm{I}} \mathrm{A}_{u\,i}$, even though it packs some useless information – which seems to be a curse of the set form. Fibre products are the same as pullbacks in the category of set. This motivates the following definition: given an arrow $\mathrm{I} \xrightarrow{u} \mathrm{J}$ and an object $\mathrm{A} \xrightarrow{f} \mathrm{J}$ in $\mathcal{C}/_\mathrm{J}$ we define the object $u^*\,\mathrm{A} \xrightarrow{u^*\,f} \mathrm{I}$ of $\mathcal{C}/_\mathrm{I}$ as the following pullback:

$$\begin{array}{ccc} u^*\,\mathrm{A} & \longrightarrow & \mathrm{A} \\ {\scriptstyle u^*\,f}\downarrow & & \downarrow{\scriptstyle f} \\ \mathrm{I} & \xrightarrow[u]{} & \mathrm{J} \end{array}$$

In functional form, precomposition can be applied to family of morphisms, making it a functor. Likewise, the map u^* can be lifted to a functor by exploiting the universal property of pullbacks. Given an arrow $\overset{\mathrm{A}}{\underset{\mathrm{J}}{\downarrow}} f \xrightarrow{h} \overset{\mathrm{B}}{\underset{\mathrm{J}}{\downarrow}} g$ in $\mathcal{C}/_\mathrm{J}$, the arrow $\overset{\mathrm{A}}{\underset{\mathrm{I}}{\downarrow}} u^*\,f \xrightarrow{u^*\,h} \overset{\mathrm{B}}{\underset{\mathrm{I}}{\downarrow}} u^*\,g$ of $\mathcal{C}/_\mathrm{I}$ is computed with the following

diagram:



In the diagram besides, there are two copy of the pullback above. One with f and one with g. As $hg = f$, the square with h is a prepullback diagram. The universal property of the pullback of g and u gives $u^* h$.

the functor u^* is called a *base change* functor.

Locally cartesian closed category

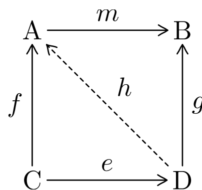
A locally cartesian close (LCC) category is a category with all pullbacks and such every base change functor u^* have a right adjoint Π_u.

Let us suppose $\mathcal{C}$ has a terminal object 1, consider the arrow $\mathrm{I} \xrightarrow{!} 1$. The functor $!^*$ from $\mathcal{C}/1$ to $\mathcal{C}/\mathrm{I}$ essentially allows to see any object A of $\mathcal{C}$ as the family $(\mathrm{A})_{i:\mathrm{I}}$. The right adjoint $\Pi_!$ gives an isomorphism between $(\mathrm{Hom}\,(!^*\,1)\,\mathrm{F})$ and $(\mathrm{Hom}\,1\,(\Pi_!\,\mathrm{F}))$. Following the functional form, $(\mathrm{Hom}\,(!^*\,1)\,\mathrm{F})$ is the set of functions $1 \xrightarrow{f_i} \mathrm{F}_i$ that is the set of all families $(f_i)_{i:\mathrm{I}}$ with $f_i{:}\mathrm{F}_i$. It follows from the isomorphism that $(\Pi_!\,\mathrm{F})$ is the *internalisation* of this set, hence $(\Pi_!\,\mathrm{F})$ is the *dependent product* $\prod_{i\,:\,\mathrm{I}}\mathrm{F}_i$.

Though it is not easy to convey an intuition for the general case, in the category of sets, for some $\mathrm{I} \xrightarrow{u} \mathrm{J}$, Π_u is realised by the family $\left(\prod_{i\,:\,u^{-1}\,j}\mathrm{F}_i\right)_{j:\mathrm{J}}$. Which can be seen as some kind of partial dependent product. For instance, consider $\left(\mathrm{F}_{(i,j)}\right)_{(i,j):\mathrm{I}\times\mathrm{J}}$, then Π_{π_2} is, up to isomorphism, the family $\left(\prod_{i\,:\,\mathrm{I}}\mathrm{F}_{(i,j)}\right)_{j:\mathrm{J}}$.

Strong monomorphism

A mono $\mathrm{A} \xrightarrow{m} \mathrm{B}$ is strong if for each epi $\mathrm{C} \xrightarrow{e} \mathrm{D}$ and all arrows $\mathrm{C} \xrightarrow{f} \mathrm{A}$ and $\mathrm{D} \xrightarrow{g} \mathrm{B}$, there is an $\mathrm{D} \xrightarrow{h} \mathrm{A}$ – necessarily unique – such that the diagram commutes:



In the category of set, any arrow $\mathrm{X} \xrightarrow{a} \mathrm{Y}$ can be lifted to an epi $\mathrm{X} \xrightarrow{a'} \{y : \mathrm{Y} \mid y \mathrel{\varepsilon} \mathfrak{Im}\,a\}$. Strong monos, are these arrows m where m' is invertible – for general monos, m' would only be a bijection. Indeed, in the above diagram, set e to be m', f to be the identity and g to be the canonical injection from $\{x : \mathrm{A} \mid x \mathrel{\varepsilon} \mathfrak{Im}\,m\}$ to A. Then h gives an inverse to m'. In other words, in the

Supposing m is the equaliser of x and y. In the above square, $fmx = fmy$ hence, $egx = egy$. Since e is epic, $gx = gy$. Then the universal property of equalisers gives h.

category of sets, strong monos are precisely what we dubbed strong subset in Section 1.1.

Also any equaliser is strong. In particular, all monomorphism in a topos are strong. It is not true of sets, however, since strong monos which are also epic are invertible (by setting e to m and taking f and g to be the identity). Which means that a category where all monos are strong verifies the principle of unique choice.

Quasitopos

A quasitopos is a category with all finite limits (*i.e.* a terminal object, all products and all equalisers), all finite colimits (*i.e.* an initial object, all coproducts and all coequalisers) which is LCC and with a classifier of strong monos. That is an object Ω and an arrow $1 \xrightarrow{\top} \Omega$ such that for each strong monomorphism $A \xrightarrow{m} B$, there is an arrow $B \xrightarrow{\chi_m} \Omega$ and the following square is a pullback:

$$\begin{array}{ccc} A & \longrightarrow & 1 \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle \top} \\ B & \xrightarrow[\chi_m]{} & \Omega \end{array}$$

The fact that the category of sets, without unique choice, is a quasitopos has been first noticed by Eduardo Dubuc and Luis Español in private conversation. We have certified in Coq that the set Ω is a classifier of strong mono. A skeleton of the Coq proof can be found in Appendix B.

We have given each of the required constructions for the category of sets, hence we can state the theorem that *the category of sets is a quasitopos*. Notice that we use critically that the type theory supports an impredicative type of propositions.

It turns out, for the interested reader, that sheaves in a quasitopos are in turn a quasitopos. Quasitopoi should be able to serve as an additional abstraction barrier our the development of homological algebra. Indeed, quoting Jacques Penon [47]: *quasitopoi are almost topoi*. As a matter of fact, a quasitopoi where the principle of unique choice is valid would be, in fact, a topos. Which strongly suggest that quasitopoi make an effective replacement of topoi for mathematics without the principle of unique choice.

In a quasitopos, every morphism f can be factored as an epimorphism f' followed by a strong mono ι_f ($f = f'\iota_f$) This is proved for topoi in [40, p. 184–185], a proof for quasitopoi is obtained by replacing every occurrence of *mono* by *strong mono*. When f is monic, f' is also monic. The principle of unique choice gives an inverse to f', then f is strong because ι_f is.

5.3 Internal language

As we have seen in the previous sections, quasitopoi mirror a kind of dependent type theory (of extensional flavour, as in [41]) This statement can be made somewhat more precise by writing down the appropriate type theory and then stating that it is *the internal language of quasitopoi*.

Being able to program conveniently in the language of quasitopoi is arguably a requirement for the relevance of this abstraction of the theory of sets. The set of combinators of quasitopoi gives a binder-free version of the type theory of quasitopoi. While there have been propositions of programming language without binders, like John Backus's FP [11]. We argue that such a language is hardly practical at all. For instance, the definition of Ackerman's function in a variant of FP might look like:

rec s (rec (1:1) (2; 2))

This is very concise, yet we would prefer something in the style of:

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\mathrm{ack} : \mathbb{N} \longrightarrow \mathbb{N} \longrightarrow \mathbb{N}} \\
\mathrm{ack} & 0 & n & = & n+1 \\
\mathrm{ack} & m & 0 & = & \mathrm{ack}\,(m-1)\,1 \\
\mathrm{ack} & m & n & = & \mathrm{ack}\,(m-1)\,(\mathrm{ack}\,m\,(n-1))
\end{array}
$$

This, however, poses two hard problems. First, representing object with binders in type theories such as that of Coq is an unsolved question. Although there are propositions which involve little, if any, modification of the language (see for instance [22] and [31]). Second, supposing we have a way to represent binders, the representation of recursive functions would be far from easy. Traditional dependent type theories have recursion combinators, which would make Ackerman's function very much like its binder-free version. Modern systems implementing type theories allow for recursive expressions which are checked for termination. Such a checking procedure is no light task to implement.

Also, there is an issue which is specific to quasitopoi – more precisely to LCC categories. In LCC categories, families of sets have a primary role (this is how the dependent product is characterised). However these families are manipulated as their graph rather than as functions. It does not pose any problem in traditional mathematics, if only because functions are *encoded* as graph anyway. In type theory, not only is it contrary to standard practice, but also, as we have already mentioned, the encoding of families in set form packs some information which is useless – which we would like to avoid in a computer. Our slogan is that *in* LCC *categories, families are in the wrong direction*. William Lawvere, in [36], defines a possible alternative – with families in the right direction – which he calls *hyperdoctrines*. We have yet to study whether it can be leveraged to give an appropriate variant of quasitopoi.

From now on, we shall suppose that we have two languages: a *host language*, Coq for instance, and an internal language defined *inside* the host language.

Multicategory

We would like to discuss in a bit more depth what constructions can be thought desirable for an internal language. Or first discussion will be about what multiple argument functions should be like. We shall look at it mostly from an efficiency perspective. The two usual answers are:

- single argument functions of a product: $\mathrm{A} \times \mathrm{B} \longrightarrow \mathrm{C}$
- single argument functions returning a function: $\mathrm{A} \longrightarrow \mathrm{B} \longrightarrow \mathrm{C}$

The first answer is favoured by ZF practitioners, as well as many category theorists, SML programmers and most imperative programmers. The second is preferred by λ-calculus folks and OCaml, Haskell, Agda and Coq programmers. Truth is, they are equally valid, and equally incomplete. Most of the time it is just a matter of convention, as both encodings are possible. In programming languages, though, the convention chosen by the designers usually fix the convention of the users as it leads to optimisation choices as well, and other convention would not perform as well in the said setting.

It gets interestingly worse when we implement programming languages inside a host one as in this section. The most archetypal case comes from monads in functional languages. In a functional language, a monad is given by a type family T:**Type** $\rightarrowtail$ **Type** together with a map:

$$\mathrm{return} \colon \prod_{\mathrm{A} \,:\, \mathbf{Type}} \mathrm{A} \rightarrowtail \mathrm{T\,A}$$

and a map:

$$\mathrm{bind} \colon \prod_{\mathrm{A\,B} \,:\, \mathbf{Type}} \mathrm{T\,A} \rightarrowtail (\mathrm{A} \rightarrowtail \mathrm{T\,B}) \rightarrowtail \mathrm{T\,B}$$

These maps must verify some axioms, for a more formal definition of monads, refer to Section 6.3. From the perspective of category theory, these are the main ingredient of the internal language of a category called a Kleisli category where morphisms are of the form $\mathrm{A} \rightarrowtail \mathrm{T\,B}$: return plays the role of identity and bind of composition. The encoding of choice for multiple argument functions is something of the form $\mathrm{A} \rightarrowtail \mathrm{B} \rightarrowtail \mathrm{T\,C}$ which is by no mean a single argument morphism of the Kleisli category.

Another example of an encoding of multiple argument functions which cannot be seen as a single argument function comes from the category of sets:

Function$_2$ $\mathrm{A\,B\,C}$
|| $f : \mathrm{A} \rightarrowtail \mathrm{B} \rightarrowtail \mathrm{C}$
|| $_ : \forall a_1 = a_2. \forall b_1 = b_2. f\, a_1\, b_1 = f\, a_2\, b_2$

This can be easily generalised to n arguments. Actually, such a scheme plays a central role in current implementation of the rewrite tactic of Coq. These play much better in Coq than functions of the form $\mathrm{A} \longrightarrow \mathrm{B} \longrightarrow \mathrm{C}$, and slightly better than functions of the form $\mathrm{A} \times \mathrm{B} \longrightarrow \mathrm{C}$.

We could make the assumption that in the internal language of quasitopoi, multiple argument functions are encoded as functions of a product, then using transformation inspired by compilers to make them into a more friendly form before they get executed. The transformation would probably have to be defined for each quasitopos, which is not a problem of code reuse as, presumably, it would be fairly different for each one. However it might still involve significantly more boiler plate and be conceptually less clear than simply having one define what is an n-ary function for the said quasitopos.

Fortunately enough, there exist a flavour of categories with multiple argument arrows called *multicategory*. The gist is that instead of having a hom-set for each pair of object, there is a hom-set ($\mathrm{Hom}\,\Gamma\,\mathrm{B}$) for every list of object Γ and object B. Hence two argument arrows are in hom-sets of the form ($\mathrm{Hom}\,[\mathrm{A},\mathrm{B}]\,\mathrm{C}$). A nice perk of multicategories is that they feature a primitive notion of *elements* of an object A – that is arrows of ($\mathrm{Hom}\,[\,]\,\mathrm{A}$). This is much desirable as it prevents from the need of using a function to choose an element in A (*e.g.* a group object has a neutral element which is usually represented as an arrow of ($\mathrm{Hom}\,1\,\mathrm{A}$)), functions being often a slow construct in programming languages it is preferable to avoid them when possible.

As an example, an abelian group in the category of sets could be defined as the data of a set A together with an addition:

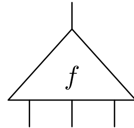
$$_ + _ : \mathrm{Hom}\,[\mathrm{A},\mathrm{A}]\,\mathrm{A}$$

and a neutral element:

$0 : \mathrm{Hom}\,[]\,\mathrm{A}$

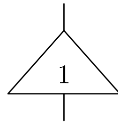
such that a few properties are verified.

It is natural to represent arrows in multicategories as triangle shaped boxes rather than simple arrows for categories:

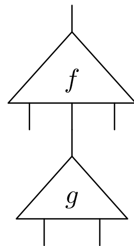


The top wire corresponding to the codomain of f, and the wires at the bottom to its domains.

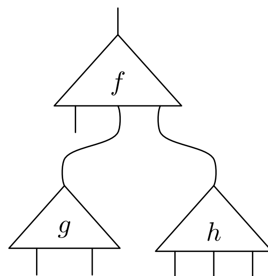
More precisely, a multicategory is the data of a type $\mathcal{O}$ of objects together with a set $(\mathrm{Hom}\,\mathrm{GG}\,\mathrm{A})$ of arrows for every list of objects GG and object A. Additionally it has an identity arrow $(\mathrm{Hom}\,[\mathrm{A}]\,\mathrm{A})$ for every A.



For every arrows $[\mathrm{A}_1, \ldots, \mathrm{A}_k] \xrightarrow{f} \mathrm{B}_i$ and $[\mathrm{B}_1, \ldots, \mathrm{B}_n] \xrightarrow{g} \mathrm{C}$, there is a composite arrow $[\mathrm{B}_1, \ldots, \mathrm{B}_{i-1}, \mathrm{A}_1, \ldots, \mathrm{A}_k, \mathrm{B}_{i+1}, \ldots, \mathrm{B}_n] \xrightarrow{f;g} \mathrm{C}$.



They follow the same associativity laws as standard categories. In addition they verify a last law which justifies our diagrammatic representation. This law states that when composing two arrows g and h with two different input wires of an arrow f, the order does not matter. Which means, in particular, that the following diagram makes sense:



Universal constructs of categories can be ported to universal categories rather straightforwardly. Here are the simplest examples (we write $[\Gamma, A]$ for the list extending Γ with A, ignoring the order of elements for simplicity):

- A terminal object 1 is such that for any context Γ, there exists a unique arrow $\Gamma \xrightarrow{!} 1$.
- An initial object 0 is such that for any context Γ and any object A, there exists a unique arrow $[\Gamma, 0] \xrightarrow{0} A$.
- A product of A and B is the data of an object $A \times B$ and two arrows $A \times B \xrightarrow{\pi_1} A$ and $A \times B \xrightarrow{\pi_2} B$ such that for any pair of arrows $\Gamma \xrightarrow{f} A$ and $\Gamma \xrightarrow{g} B$ there exists a unique arrow $\Gamma \xrightarrow{(f,g)} A \times B$ with $(f, g)\pi_1 = f$ and $(f, g)\pi_2 = g$.
- A coproduct of A and B is the data of an object $A + B$ and two arrows $A \xrightarrow{\iota_1} A + B$ and $B \xrightarrow{\iota_2} A + B$ such that for any pair of arrows $[\Gamma, A] \xrightarrow{f} C$ and $[\Gamma, B] \xrightarrow{g} C$, there exists a unique arrow $[\Gamma, A + B] \xrightarrow{[f,g]} C$ with $\iota_1[f, g] = f$ and $\iota_2[f; g] = g$.

It is possible, in principle, to go on up to the definition of quasitopoi, such as to allow for a primitive notion of multiple argument function. However, it is not quite clear how to even start doing this in Coq. Indeed, as early as the definition of composition, one would start to write down painful expressions. Most likely, it won't even be possible to state that composition is associative because of *associativity nightmare* – like in Section 2.4. Even supposing that the associativity issue is dealt with, it is not clear that there is any practical way of dealing with multicategories in Coq. Hopefully there is. Until it is found, we will probably be stuck with ordinary categories.

Algebraic datatypes

Another piece missing – this one coming directly from traditional programming languages – is algebraic datatypes. Indeed programming language usually allow the definition of new datatypes using sums and products in a fashion which is not customary to mathematics. There are two principal distinctive features to algebraic datatypes:

- The n-ary products (resp. sums) are typically not defined as iterations of binary products as this would typically imply useless indirections – memory calls being somewhat costly.
- Sums are generally labelled (by mean of so called *type constructors*). Products are often labelled too, when they are given as records (the labelled being usually called *projections* in this case)

Both are really useful tools, yet as soon as we go into generic programming, type level programming or categories we have no means of using them at all. The programmer reverts instantly to unlabelled binary sums and products. If we are to really program in an internal language, it sounds like an issue that has to be addressed.

For example, consider a variant of binary lists [46, p. 122–125]. They can be defined as the following data type in Coq:

```
Inductive NZBL (A:Type) :=
  | One (a:A)
  | Two (a a':A)
  | TwicePlusOne (a:A) (NZBL (A*A))
  | TwicePlusTwo (a a':A) (NZBL (A*A))
```

This definition – which may be relevant, for instance, for the kernel computations in Section 2.5 – uses a 4-ary labeled sum and ternary product for the TwicePlusTwo case. Quasitopoi, as we have presented here so far, do not support either labels or n-ary sums and products. We shall discuss here how they can be given support.

Tuples are usually represented in memory as adjacent memory cells, which plays well with the processor's cache, allowing to fetch the entire tuple with a single memory call (this is the *raison d'être* of structures in the C language). If we were not able the TwicePlusTwo case would have to be defined as a product of the form $A \times (A \times \mathrm{NZBL}\,(A \times A))$, resulting in a sort of memory pileup: rather than fetching both head elements and the recursive list at the same time, we could only fetch the first element in a first memory access, and with a second memory access, we would get the second element and the recursive list. The cost of memory access is not negligible, and should preferably be avoided. This is precisely why n-ary products are featured in most programming languages.

The memory representation tuples is tantamount to that of arrays presented in Section 3.4 – even though, because of the *update* operation, fetching the array actually needs *two* memory call rather than one. Also, tuples have fixed length, and each component may have a different type. There are much like a *flat* version of a heterogeneous list:

$$
\begin{array}{lcl}
\mathrm{HList} : \mathrm{List}\,\textbf{Type} & \rightarrowtail & \textbf{Type} \\
\mathrm{HList} \quad [] & = & 0 \\
\mathrm{HList} \quad [A, GG] & = & A \times (\mathrm{HList}\,GG)
\end{array}
$$

Where 0 is the element of the one element type, and $[A, \Gamma]$ is the list starting with head A and tail GG. Chapter 3, arrays are *defined* as lists and *compiled* as actual arrays, this suggest that it would be possible to do something similar for tuples.

An obvious difficulty arises: it is much harder to state associativity properties. In the category of sets, $A \times B$ is isomorphic to $B \times A$ and $1 \times A$ is isomorphic to A. When n-ary products are all defined primitively – Leinster [37] calls this an *unbiased* product – there are much more primitive isomorphisms: every two trees of products with the same leaves are isomorphic (see [37, p. 66]) for more details). It is left to be seen whether Coq can cope with such a fancy associativity principle painlessly.

Variants – as programmers call the elements of sum types – are usually represented in memory as the pair of a tag indicating which component of the sum their value belongs to and their value. Apart from the fact that it is a customary encoding in mathematics, the motivation for the representation of variants is that they are typically used by running a different piece of code

depending on which component they belong to. The emphasised ability is to be able to branch as rapidly as possible. The matter is usually made a bit more complicated as sums and products are merged to select a branch at the same time as fetching data – as it is also advertised in theoretical works, like Girard's ludic [23] – but we shall focus on separate type constructors for products and sums. In our example, suppose we want to compute the length of lists:

```
Fixpoint length (A:Type) (l:NZBL A) :=
 match l with
 | One _ ⇒ 1
 | Two _ _ ⇒ 2
 | TwicePlusOne _ l ⇒ 2*(length (A*A) l)+1
 | TwicePlusTwo _ _ l ⇒ 2*(length (A*A) l) +2
```

If we only had access to binary sums, we would have to do perform a memory access to check whether we are in the case One, if not then a memory access to check whether we are in the case Two, *etc*. An 4-ary sum, means that we have only one memory access to perform, which is preferable.

Provided we have an efficient representation for tags, dependent sums allows for the definition of variants as pairs. Specifically, supposing we have an efficient implementation for a finite type Tag, a sum indexed by Tag has a type of the form $\sum_{t\,:\,\mathrm{Tag}} A_t$. In Coq, this gives precisely the expected representation.

Incidentally, this suggests a solution for labelled sums as well – which are a convenience feature rather than an efficiency-related one. Provided the type Tag is viewed as a type of labels. To define all the sort of labelled sums intended, the type Tag must support case analysis; and, of course, their must be as many such types as can be conceived. In [7], this is addressed by having a special syntactic construction for labels, and a particular form of types which consist in *enumerations* of labels and support case analysis. To be able to define categories with unbiased sums, though, this isn't sufficient: the enumeration types need to be of a special type **Enum** themselves, so that they can be quantified upon. With all these, the statement that a category $\mathcal{C}$ has all unbiased sums would read along these lines: for every L:**Enum** and every L-index family of objects of $\mathcal{C}$ $(A_l)_{l:L}$, there is an object S and a family of injection $\iota{:}\prod_{l\,:\,L} \mathrm{Hom}\, A_l\, S$.

The material used for the sums leads to a dual definition of unbiased product: an object P with a family of projection $\pi{:}\prod_{l\,:\,L} \mathrm{Hom}\, P\, A_l$. Interestingly enough, it requires no support for labelled product in the host languages, as the object P can be implemented as an unlabelled product with π basically translating labels into positions – which is what compilers usually do under the hood anyway. The universal property of products, however, suggests a purely syntactical approach to records: it states that for any object X and family of arrows $X \xrightarrow{f_l} A_l$, there is an arrow $X \xrightarrow{\mathsf{record}_{l:L}\, f_l} P$ with the expected properties. In the host language it would amount to a map converting from $\prod_{l\,:\,L} A_l$ to P.

Code reuse

Somewhat ironically, at the point where we manage to write code in the internal language of some category, the biggest question left is one of code sharing. Indeed, it would happen that the same program unit need to be written both in the host language and in the internal language. If there is not a way to write the code in one (preferably the host language) and reuse it in the other, this would be a serious issue.

We expect, somewhat, to be able to parse the relevant programs from the host language, and then reinterpret it in the internal language. Fortunately, implementations of dependently typed language usually provide tools for parsing existing code. This leads to believe that modest extensions of the existing capabilities may be sufficient to reuse code from the host language into the internal language.

Chapter Six

Sharing more code

This is the butterfly of the storms.
See the wings, slightly more ragged than those of the common fritillary. In reality, thanks to the fractal nature of the universe, this means that those ragged edges are infinite – in the same way that the edge of any rugged coastline, when measured to the ultimate microscopic level, is infinitely long – or, if not infinite, then at least so close to it that Infinity can be seen on a clear day.

Terry Pratchett, *Interesting Times*

During the course of this manuscript we have noticed many resemblances between various structures. Categories resemble sets but with more fine-grain structure, additive categories are just like normal categories replacing sets and function by abelian groups and group homomorphisms, categories of graded objects are categories with more homsets.

Code sharing being one of our main focuses, we want to explore in this chapter how these resemblances can be made explicit by unifying structures. Constructions on sets can be generalised to categories, additive categories and categories can be seen as instances of a more general notion, categories of graded objects can be preabelian.

This led us into fields of mathematics which are rather young, and not always well studied. Our point of type theoretic point of view brings a fresh look to these mathematics and emphasises constraints that have been overlooked by mathematicians.

The notions used in this chapter are rather technical. We shall often refer to [37] which surveys the state of the art on which the chapter builds. Most of the definitions appearing of the chapter are discussed there, albeit from a quite different perspective.

6.1 Categories & additive categories

The definition of category gives a particular role to sets and functions, as homsets are sets and composition is a (binary) function. In additive categories,

though, the homsets are endowed with an abelian group structure and composition is a group morphism. The approach taken in Chapter 2 is to consider additive as a standard category with additional group structure. However this arguably falls short when defining functors between additive categories: at the very least one wants these functors to respect the group structure of the homsets. Yet, the arrow component of functors is, by definition, a function rather than a group morphism.

This implies working with a special kind of functors which have the additional property of respecting the group structure of homsets. Which entails proving that usual properties of functors – such as the fact they compose – pass on to group preserving functors, and proving slight variants of a number of properties of functors.

Enriched category

There need not be duplications between categories and additive categories. One might want to simply substitute the words *set* and *function* in the definition of categories with *abelian group* and *group morphism*. Or anything well behaved for that matter, for instance *topological space* and *continuous functions*.

There is a notion of enriched categories [33] which play precisely this role. More formally, given a multi-category $\mathcal{V}$, a $\mathcal{V}$-enriched category is defined as:

$$
\begin{array}{l}
\mathrm{EnrichedCategory}\ \mathcal{V} \\
\left\|\ \begin{array}{ll}
\mathcal{O} & : \textbf{Type} \\
\mathrm{Hom} & : \mathcal{O} \rightarrowtail \mathcal{O} \rightarrowtail \mathcal{O}_{\mathcal{V}} \\
1_{\mathrm{A}} & : \mathrm{Hom}_{\mathcal{V}}\ []\ (\mathrm{Hom\ A\ A}) \\
-;- & : \prod\limits_{\mathrm{A\ B\ C}\ :\ \mathcal{O}} \mathrm{Hom}_{\mathcal{V}}\ [\mathrm{Hom\ A\ B}, \mathrm{Hom\ B\ C}]\ (\mathrm{Hom\ A\ C}) \\
_ & : f;(g;h) = (f;g);h \\
_ & : 1;f = f;1 = f
\end{array}\right.
\end{array}
$$

Homsets are objects of $\mathcal{V}$, the identity arrow of A is an element (that is, a nullary endomorphism) of (Hom A A) in $\mathcal{V}$ and composition is a binary morphism in $\mathcal{V}$. Associative properties are just as usual.

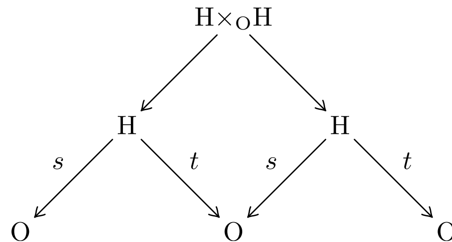
Obviously, standard categories are enriched in sets and functions. This poses a bootstrapping issue, as categories – multicategories actually – are needed for the definition of enriched categories which in turn allow to define standard categories. There does not seem to be a scheme around this in the literature. Thus we are bound, for the time being, to duplicate some properties between standard categories and enriched one. This may be seen as acceptable for two reasons: the first is that properties are to appear in at most two versions, as properties of enriched categories are written once and for all; the second is that there is a minimal need for properties of standard categories, only these needed to bootstrap enriched categories, as further properties can be inherited from the categories enriched in sets and functions.

In our case, where there are only two kinds of categories so far, it does not seem too helpful to make use of enriched categories. It would perform little, if any, code sharing while requiring potentially large a code.

Internal category

An alternate – non-equivalent – approach to enriched categories is that of *internal categories*. Roughly speaking, it consists in taking a category $\mathcal{V}$ and defining how to endow an object of $\mathcal{V}$ with categorical structure.

Formally, let $\mathcal{V}$ be a category with pullbacks. A category C in $\mathcal{V}$ is the data of two objects O and H together with a pair of morphisms $H \xrightarrow{s} O$ and $H \xrightarrow{t} O$. The object O contains objects of C and H all arrows of C, the morphism s associates to each arrow its source and t its target. Additionally, C has a morphism $O \xrightarrow{e} H$ giving an identity arrow to each of its object and a morphism $H{\times_O}H \xrightarrow{c} H$ performing composition where $H{\times_O}H$ is the pullback in the following diagram:



These various morphisms verify properties about the sources and targets of identity and composite arrows, and associativity properties all expressed as commutative diagrams.

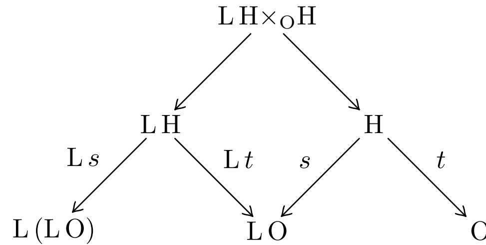
Hence, a category in the category of sets has a set of object and a set of arrows. Therefore, standard categories *are not* categories in that of sets, and they should not be as that would prevent us from defining the category of sets or that of abelian groups. Categories in the category of abelian groups have a group of objects and one of arrows. This is very different from a category enriched in abelian groups. As a consequence, internal categories are inappropriate for our applications.

In ZF, everything is set. In particular the objects of a category form a set, as well as the arrows. In this particular case, (small enough) categories are categories in the category of (large enough) sets.

Incidentally, just as it was the case with LCC categories in Section 5.2, we can say that *internal categories have dependencies in the wrong direction*, as source and target are a property of arrows rather than being indexes for homsets. Also, they seem to require stronger properties of the base category: it need to have pullbacks rather than simply being a multicategory.

The main point of internal categories is that they are easy to generalise, and the type and properties of composition remain obvious. For instance, we can give an internal version of multicategories. In the category of sets and functions, let L be the list functor. An internal multicategory is the data of an object set O, an arrow object H, a source function $H \xrightarrow{s} L\,O$, a target function $H \xrightarrow{t} O$, an identity function $O \xrightarrow{e} H$ and a composition $L\,H{\times_O}H \xrightarrow{c} H$

where $\mathrm{L\,H}\times_{\mathrm{O}}\mathrm{H}$ is the pullback in the following diagram:



The source and target assignment as well as associativity properties are given by the same diagrams as for categories, with L inserted when required (admittedly, the source assignment of composition requires some flattening of lists of lists of objects into lists of objects). Notice, that the composition is a bit different than that we gave in Section 5.3 as requires to give a n arrows on the left to compose with an n-ary arrow.

6.2 Sets as categories and beyond

A set S can be viewed as a category, with elements of S as objects, and for two elements a and b, the set $\{a =_{\mathrm{S}} b\}$ of all proofs of $(a =_{\mathrm{S}} b)$ – all considered equal – as $(\mathrm{Hom}\, a\, b)$. Composition is given by transitivity, and identities by reflexivity. The associativity laws of categories are trivial as there is at most one element in each $\{a =_{\mathrm{S}} b\}$.

Groupoid

A groupoid is a category equipped with an operation

$$_^{-1} \colon \prod_{\mathrm{A\,B}\,:\,\mathcal{O}} \mathrm{Hom\,A\,B} \longrightarrow \mathrm{Hom\,B\,A}$$

such that (f, f^{-1}) is an isomorphism, that is, $f f^{-1} = 1$ and $f^{-1} f = 1$.

Symmetry of the equality of a set makes it a groupoid. Here again, isomorphism law is trivial. Sets can be viewed as these groupoids which are degenerate in the sense that each homset has at most one element.

Functors between two sets so seen are exactly functions: the object component corresponds to the computation, the arrow component corresponds to the preservation of equality, the equational laws are trivial.

This means that the definition of sets and functions can be inherited from those of categories and functors. However, at this point, this might not be worth the effort as categories and sets do not share many properties.

Natural transformation

Of the properties of sets one that was a foremost importance was that there is a set of functions from A to B. It can be inherited from a statement about functors: there is a category of functors from $\mathcal{B}$ to $\mathcal{C}$. The morphisms of this category are called natural transformations.

Given two functors F and G from $\mathcal{B}$ to $\mathcal{C}$, a natural function from F to G is the data, for each object A of $\mathcal{B}$ of an arrow $\mathrm{F\,A} \xrightarrow{\varphi_{\mathrm{A}}} \mathrm{G\,A}$ such that the following diagram commute for any $\mathrm{A} \xrightarrow{f} \mathrm{B}$:

$$\begin{array}{ccc} \mathrm{F\,A} & \xrightarrow{\varphi_{\mathrm{A}}} & \mathrm{G\,A} \\ {\scriptstyle \mathrm{F}\,f}\downarrow & & \downarrow{\scriptstyle \mathrm{G}\,f} \\ \mathrm{F\,B} & \xrightarrow{\varphi_{\mathrm{B}}} & \mathrm{G\,B} \end{array}$$

Which can be read as φ_{A} is defined the same way for each A; or, in type theoretic jargon, φ_{A} is parametric in A.

If $\mathcal{B}$ and $\mathcal{C}$ are sets, then a natural transformation from F to G is the statement that $\forall a.\, \mathrm{F}\,a = \mathrm{G}\,a$. Thus, the category where objects are functors from $\mathcal{B}$ to $\mathcal{C}$ and arrows are natural transformation is the corresponding set of function $\mathcal{B} \longrightarrow \mathcal{C}$.

Bicategory

However, in general, functors between two categories equipped with natural transformation do not constitute a set as there are more than one natural transformation between two functors. For this reason, there is no category of categories and functors: the homsets need to be categories rather than sets.

There is a notion of *bicategory* [37, p. 26–32] where homsets are categories. There is a bicategory of categories. The detailed definition, however, is fairly long and would not belong here. We shall merely sketch it.

A bicategory is given by a type $\mathcal{O}$ of objects and for each pair A and B of objects, a *category* ($\mathrm{Hom\,A\,B}$) of morphisms – the morphisms of ($\mathrm{Hom\,A\,B}$) are called 2-morphisms. There are identity morphisms 1_{A} and a composition $(\text{-}\,;\,\text{-})$. Composition is required to be a (two argument) functor – just as it is a function in standard categories. Associativity laws only hold up to isomorphism (*e.g.* there is an isomorphism between $(f;g);h$ and $f;(g;h)$). The associativity isomorphisms are required to verify some extra laws (usually called *coherence laws*), amounting to the fact that all diagrams drawn out of them commute. For two parallel morphisms $\mathrm{A} \xrightarrow{f} \mathrm{B}$ and $\mathrm{A} \xrightarrow{g} \mathrm{B}$, we write $f \xRightarrow{\alpha} g$ for a 2-morphism between these.

The main peculiarity of bicategories is that there is two distinct ways of composing 2-morphisms. The first is given by the composition of the category ($\mathrm{Hom\,A\,B}$): for any three parallel morphisms

$\mathrm{A} \xrightarrow{f} \mathrm{B}$, $\mathrm{A} \xrightarrow{g} \mathrm{B}$ and $\mathrm{A} \xrightarrow{h} \mathrm{B}$

and any two 2-morphisms

$f \xRightarrow{\alpha} g$ and $g \xRightarrow{\beta} h$

there is a composite

$f \xRightarrow{\alpha\bullet\beta} h$

The second one is of different nature and is obtained by functoriality of the composition (-; -) of morphisms: given two pairs of parallel morphisms

$$\mathrm{A} \xrightarrow{f} \mathrm{B} \text{ and } \mathrm{A} \xrightarrow{g} \mathrm{B}, \text{ and } \mathrm{B} \xrightarrow{h} \mathrm{C} \text{ and } \mathrm{B} \xrightarrow{j} \mathrm{C}$$

each of them equipped with a 2-morphism

$$f \xRightarrow{\alpha} g \text{ and } h \xRightarrow{\beta} j$$

we shall construct a composite 2-morphism

$$f; h \xRightarrow{\alpha \blacklozenge \beta} g; j$$

Since (-; -) is a functor, we can construct the two following 2-morphism:

$$f; h \xRightarrow{\alpha; h} g; h \text{ and } g; h \xRightarrow{g; \alpha} g; j$$

Then we can define

$$\alpha \blacklozenge \beta = ((\alpha; h) \bullet (g; \beta))$$

When the $(\mathrm{Hom}\,\mathrm{A}\,\mathrm{B})$ are degenerate – that is, are sets – the bicategory is a category (and conversely, all categories are degenerate bicategories). Hence the category of sets is inherited from the bicategory of categories.

ω-category

Categories can be obtained from bicategories; just like sets can be obtained from groupoids. Bicategories have there own version of functors, which are actual functors when acting on categories – hence functions when acting on sets. We can then define a notion of tricategory such that there is a tricategory of all bicategory. The $(\mathrm{Hom}\,\mathrm{A}\,\mathrm{B})$ of tricategories are bicategories, which verify a very long list of properties. Also, tricategories feature three different compositions for their 3-morphisms. We shall call all these kind of categories *higher categories*, or n-categories (where 1-category means category, 2-category means bicategory, and so on) when focusing on a particular kind.

Somewhat we would like to say that bicategories are enriched in categories (see Section 6.1) and that tricategories are enriched in bicategories. However, it does not quite make sense as, precisely, there is no category of categories. Actually, even in ZF, where one can consider a category of (small) categories, the notion of enrichment needed here is, apparently, really hard (to the point that there is apparently no reference in the literature on the subject).

The usual approach to defining higher categories is an internal way: certain objects of certain categories are defined as being bicategories. The internal approach does not suit or objectives, in particular because it implies that all higher categories are sets. Also, there are various non-equivalent definitions of n-categories. On the positive side, some of these definitions focus on all possible ways to compose k-morphisms in an unbiased way (see Section 5.3).

In a non-internal perspective, it seems more reasonable to start and define a notion of ω-category, which would have n-morphisms for every natural number n, and then take an n-category to be degenerate from rank $n+1$ on – that is there is at most one $(n+1)$-morphism in each $(n+1)$-homsets.

$$
\begin{array}{l}
\mathrm{CatLike}\;\mathcal{O}\;\mathrm{Hom}\;1\;(\text{-};\text{-}) \\
\quad \text{-} \Rightarrow \text{-} : \prod_{A\,B : \mathcal{O}} \mathrm{Hom}\,A\,B \rightarrowtail \mathrm{Hom}\,A\,B \rightarrowtail \mathbf{Type} \\
\quad \mathrm{id} : \prod_{\substack{A\,B : \mathcal{O} \\ f : \mathrm{Hom}\,A\,B}} f \Rightarrow f \\
\quad \text{-} \bullet \text{-} : \prod_{\substack{A\,B : \mathcal{O} \\ f\,g\,h : \mathrm{Hom}\,A\,B}} (f \Rightarrow g) \rightarrowtail (g \Rightarrow h) \rightarrowtail (f \Rightarrow h) \\
\quad \mathrm{next} : \prod_{A\,B : \mathcal{O}} \mathrm{CatLike}\,(\mathrm{Hom}\,A\,B)\,(\text{-} \Rightarrow \text{-})\,\mathrm{id}\,(\text{-} \bullet \text{-}) \\
\quad \lambda : \prod_{\substack{A\,B : \mathcal{O} \\ f : \mathrm{Hom}\,A\,B}} 1; f \Rightarrow f \\
\quad \rho : \prod_{\substack{A\,B : \mathcal{O} \\ f : \mathrm{Hom}\,A\,B}} f; 1 \Rightarrow f \\
\quad \alpha : \prod_{\substack{A\,B\,C\,D : \mathcal{O} \\ f : \mathrm{Hom}\,A\,B \\ g : \mathrm{Hom}\,B\,C \\ h : \mathrm{Hom}\,C\,D}} f; (g; h) \Rightarrow (f; g); h \\
\quad \text{-} \blacklozenge \text{-} : \prod_{\substack{A\,B\,C : \mathcal{O} \\ f\,f' : \mathrm{Hom}\,A\,B \\ g\,g' : \mathrm{Hom}\,A\,B}} (f \Rightarrow g) \rightarrowtail (f' \Rightarrow g') \rightarrowtail (f; g \Rightarrow f'; g')
\end{array}
$$

Figure 6.1: A first approach to ω-categories

In addition, it is useful and meaningful to consider (-1)-categories and (-2)-categories. A (-1)-category is a proposition: its objects, or 0-morphism, are its proofs all considered equal, hence propositions are degenerate from rank 0. A (-2)-category is the proposition $\top$: its imaginary (-1)-morphisms must be all equal as the unique 0-morphism between them would be the proof of $\top$.

It is not obvious how to define ω-categories in type theory. As an infinitely deep structure, it would make use of co-induction. Maybe we want to define ω-categories directly, or we might want something more general and then constrain it down to ω-categories.

In the direction of the latter approach, there is a tentative definition due to Peter Hancock – in doubt of it deserving a name, we shall just call it CatLike:

Peter Hancock suggested this definition in private discussions, there is no published reference.

We recognize both products 2-morphisms. The second one, $(\text{-} \blacklozenge \text{-})$, also ensures the functoriality of the 1-morphism composition $(\text{-};\text{-})$. The third product of 3-morphism is not explicit here, and presumably cannot be retrieved as it is equivalent to the functoriality of $(\text{-} \blacklozenge \text{-})$, which is not required. Requiring that $(\text{-} \blacklozenge \text{-})$ is a functor would involve the third product of 3-morphisms, which in turn would not be a functor. Hence there is necessarily structure missing in this definition – there is no easy way around.

Also, instead of having isomorphisms between $f; (g; h)$ and $(f; g); h$, there is simply a morphism from the former to the latter. It is hard to judge at this point whether it captures new interesting mathematical objects – though it

certainly bears a striking resemblance with rewrite systems.

However, any definition of ω-category would presumably be stronger than that of CatLike. Hence CatLike makes a good object of study. Let us see how it works with the simplest ω-categories.

There is an ω-category $\top_s$ corresponding to $\top$ (with ! the sole proof of $\top$):

$$
\begin{array}{l}
\top_s : \mathrm{CatLike}\,\top\,(\lambda__.\,\top)\,!\,(\lambda__.\,!) \\
\quad \text{-} \Rightarrow \text{-} = \lambda__.\,\top \\
\quad \mathrm{id} \quad = \lambda___.\,! \\
\quad \text{-} \bullet \text{-} \quad = \lambda_______.\,! \\
\quad \mathrm{next} \quad = \top_s \\
\quad \lambda \quad = \lambda___.\,! \\
\quad \rho \quad = \lambda___.\,! \\
\quad \alpha \quad = \lambda_______.\,! \\
\quad \text{-} \blacklozenge \text{-} \quad = \lambda_________.\,!
\end{array}
$$

Hence T_s is define in term of itself.

Then, given a proposition, P, there is a CatLike Ω_s defined in term of $\top_s$. Likewise, given a set A we can define a corresponding CatLike $\mathbf{Set}_s$ using Ω_s. The definitions are straightforward, but the real goal is to be able to state something like *a proposition is an ω-category enriched in* T and *a set is an ω-category enriched in propositions*. We are lacking a good definition of enrichment for ω-categories.

To conclude on a side remark, if we manage to render ω-category workable, the inevitable question will be to be able to work in the internal language of such an ω-category.

6.3 The category of graded objects is preabelian

In Section 2.4, we have seen a slight generalisation of categories, where instead of having a homset (Hom A B) for each pair of object A and B, homsets are additionally indexed by an integer called the degree. In Section 5.3, we have defined multicategories, a variant of categories where homsets are indexed by a list of domains and a codomain. Multicategory are intended to abstract the notion of multiple argument functions. We might want to add a degree on top of that.

It so happens that there is a unified framework, albeit young, to deal with these multicategories, and degrees and such extension. It is dubbed *generalised multicategory*.

Monad

A monad is a functor T from a category $\mathcal{C}$ to itself, such that for each object A of $\mathcal{C}$, there is an arrow $A \xrightarrow{\eta_A} T\,A$ called the unit of T, and an arrow $T\,(T\,A) \xrightarrow{\mu_A} T\,A$ called the composition of T (both η and μ must be natural in A). Additionally they verify associativity laws, which amounts to say that any diagram built only of η and μ are commutative.

The functor L of the category of sets, associating to each set A the set of lists of elements of A, is a monad: $\eta_A\,a = [a]$ and $\mu_A\,[[a^1_1, \ldots, a^1_{k_1}], \ldots, [a^n_1, \ldots, a^n_{k_n}]] = [a^1_1, \ldots, a^1_{k_1}, \ldots, a^n_1, \ldots, a^n_{k_n}]$.

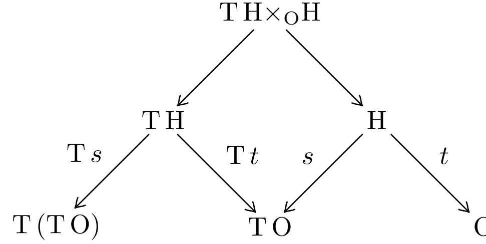
The functor - $\times \mathbb{Z}$ is also a monad: $\eta_{\mathrm{A}}\, a = (a, 0)$ and $\mu_{\mathrm{A}}\, ((a, n), p) = (a, (n + p))$. Actually, for any monad T of the category of sets, the composite functor $\mathrm{T}\,(\text{-} \times \mathbb{Z})$ is also a monad.

Let T be a monad of the category of sets. There is a function $f_{\mathrm{A}}{:}\mathbb{Z} \longrightarrow \mathrm{A} \longrightarrow \mathrm{A} \times \mathbb{Z}$ defined as $f_{\mathrm{A}}\, n\, a = (a, n)$. As T is a functor, we deduce a function $\mathrm{T}\,(f_{\mathrm{A}}\, n){:}\mathrm{T}\,\mathrm{A} \longrightarrow \mathrm{T}\,(\mathrm{A} \times \mathbb{Z})$, and then $\lambda(a, n).\, \mathrm{T}\,(f_{\mathrm{A}}\, n) : (\mathrm{T}\,\mathrm{A}) \times \mathbb{Z} \longrightarrow \mathrm{T}\,(\mathrm{A} \times \mathbb{Z})$. Using this function, we can turn an element of $(\mathrm{T}\,((\mathrm{T}\,(\mathrm{A} \times \mathbb{Z})) \times \mathbb{Z}))$ into one of $(\mathrm{T}\,(\mathrm{T}\,((\mathrm{A} \times \mathbb{Z}) \times \mathbb{Z})))$, from which we can deduce the composition of $\mathrm{T}\,(\text{-} \times \mathbb{Z})$.

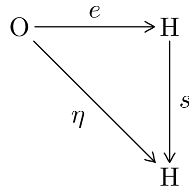
Generalised multicategory

The idea of generalised multicategory stems from these remarks. The generalisations of categories considered in this section have all something to do with some monad. A unified way to handle them might be to work with an arbitrary monad.

The usual definition is given in an internal way. Given a monad T (with additional properties) of a category $\mathcal{C}$, a T-category in $\mathcal{C}$ is the data of an object O of objects, an object H of arrows, a source morphism $\mathrm{H} \xrightarrow{s} \mathrm{T}\,\mathrm{O}$ and a target morphism $\mathrm{H} \xrightarrow{t} \mathrm{O}$, together with an identity arrow assignment $\mathrm{O} \xrightarrow{e} \mathrm{H}$ and a composition $\mathrm{T}\,\mathrm{H}{\times_{\mathrm{O}}}\mathrm{H} \xrightarrow{c} \mathrm{H}$ where $\mathrm{T}\,\mathrm{H}{\times_{\mathrm{O}}}\mathrm{H}$ is the pullback in the following diagram:



The source of the identity arrows uses the unit η of T to be defined:



Similarly, the source of composite arrow is given by the composition of the monad. See [37, Chapter 4] for a complete definition.

There are no non-internal definition known to the author. We can outline some ideas. As we are using monads, we need some category from which to pick the type of objects from. Let us suppose that we have a set of objects, and a monad T on the category of sets. Homsets would be of type $\mathrm{T}\,\mathcal{O} \longrightarrow \mathcal{O} \longrightarrow$ **Set**. As in the internal definition, the identity of A would have type $\mathrm{Hom}\,(\eta_{\mathcal{O}}\,\mathrm{A})\,\mathrm{A}$. The difficult part would be to handle composition, as one has to single out one of the source objects to compose on. We would need T to have some sort of derivative [42] $\partial\mathrm{T}$ equipped with a natural transformation $(\partial\mathrm{T}\,\mathcal{A}) \times \mathcal{A} \xrightarrow{\text{-}\,[\text{-}]} \mathrm{T}\,\mathcal{A}$ with appropriate properties. We could then define the composition as having type:

$$\text{-};\text{-}: \prod_{\substack{\Gamma : \mathrm{T}\,\mathcal{O} \\ \Delta : \partial\mathrm{T}\,\mathcal{O} \\ \mathrm{B}\,\mathrm{C} : \mathcal{O}}} \mathrm{Hom}\,\Gamma\,\mathrm{B} \longrightarrow \mathrm{Hom}\,(\Delta\,[\mathrm{B}])\,\mathrm{C} \longrightarrow \mathrm{Hom}\,(\mu\,((\partial\mathrm{T}\,\eta\,\Delta)\,[\Gamma]))\,\mathrm{C}$$

Which amounts precisely to say that compositions plugs an arrow with target B into a hanging B.

Such a definition plays better with type theory than an internal definition, however it is not clear that it leads somewhere. It requires the objects to form a set. This constraint could be relaxed, for instance we could assume that each type is naturally endowed with an ω-categorical structure. It is debatable, however, that types should have any structure canonically attached to them at all.

On the other hand, suppose we have a suitable definition of generalised multicategory worked out, and that we can even define a preabelian T-category. Hopefully, $\mathrm{T}\,(\text{-} \times \mathbb{Z})$ would also have the required properties to define $(\mathrm{T}\,(\text{-} \times \mathbb{Z}))$-categories. We could then state the following theorem: *let $\mathcal{A}$ be an abelian T-category, then the $(\mathrm{T}\,(\text{-} \times \mathbb{Z}))$-category of graded objects of $\mathcal{A}$ is preabelian*.

Graded category

As we have no complete definition of generalised multicategory, we shall restrict ourselves to categories with a single source object. Given a monoid M, we can define an M-category as a category where for each element d of M, called the degree, and objects A and B, there is a homset $\mathrm{Hom}_d\,\mathrm{A}\,\mathrm{B}$. Identity would be in $\mathrm{Hom}_e\,\mathrm{A}\,\mathrm{B}$ (with e the neutral element of M), and composition would have type:

$$\text{-};\text{-}: \prod_{\substack{d_1\, d_2\, :\, \mathrm{M} \\ \mathrm{A}\,\mathrm{B}\,\mathrm{C}\, :\, \mathcal{O}}} \mathrm{Hom}_{d_1}\,\mathrm{A}\,\mathrm{B} \longrightarrow \mathrm{Hom}_{d_2}\,\mathrm{B}\,\mathrm{C} \longrightarrow \mathrm{Hom}_{d_1 d_2}\,\mathrm{A}\,\mathrm{C}$$

An abelian M-category is defined straightforwardly. Each homset has an abelian group structure, composition is bilinear, for each construction where the degree is left free – like kernels and cokernels, and the injections and projections of the biproducts – we constrain it to be e.

Graded objects, second take

Given an M-category $\mathcal{C}$, we can define the category of its graded objects to be an $\mathrm{M} \times \mathbb{Z}$-category with objects the $(\mathrm{A}_n)_{n:\mathbb{Z}}$, with $\mathrm{A}_n{:}\mathcal{O}_\mathcal{C}$ and morphisms in $(\mathrm{Hom}_{(d,k)}\,((\mathrm{A}_n)_{n:\mathbb{Z}})\,((\mathrm{B}_n)_{n:\mathbb{Z}}))$ are families of morphisms $(f_n)_{n:\mathbb{Z}}$ with each f_n taken in $(\mathrm{Hom}_d\,\mathrm{A}_n\,\mathrm{B}_{(n+k)})$.

When $\mathcal{C}$ is preabelian, the preabelian structure can be extended to the graded objects like in Section 2.4. Hence the theorem: *The $\mathrm{M} \times \mathbb{Z}$-category of graded objects of a preabelian M-category is preabelian*.

This theorem appears to be a contribution of this work. It seems to be nowhere to be found in the literature. It is presumably of little relevance to pen-and-paper mathematics. However, it is fundamental in our presentation of effective homology.

Graded categories in Coq

Unfortunately, it is not directly possible to transpose the definition of graded categories into Coq. The main issue is the representation of set-indexed

families – of which $(\mathrm{Hom}_d\,\mathrm{A}\,\mathrm{B})_{d:\mathrm{M}}$ is. Such a family is given by a map F:X→Set (where X is the index set), it must be endowed with a function c : **forall**(x y:X), x = y → F x ⟶ F y As such, the composition id·f – where f is in Hom d A B – is not in Hom d A B but in Hom (e∗d) A , which is isomorphic, so we cannot state id·f=f, however, if l is a proof that e∗d=d, then we can have c l (id·f) = f.

The function c must verify properties which make F a functor from X seen as a category to the category of set. That is c (refl x) = id, c (trans p q) = (c p)·(c q) and (c p)·(c (sym p)) = id.

While possible this approach would make mathematical statement cluttered with explicit coercions. It could be arranged in such a way that the statement are almost as easy to read as statement without coercions. However every statement would need to come with an explicit proof of equality between degrees. In order to make this approach tractable we would need a way to automate the creation of the equality proof.

It is, otherwise, possible to tweak the definition of graded categories to make id·f=f typable. Instead of having a monoid defined in term of a binary and a nullary operations, we can define an unbiased monoid as a set M together with a function p : List M ⟶ M, where List M is the set of lists of elements of M. Also p needs to verify that **forall** d, p [d]=d together with an associativity law.

Then we would have

```
id:forall A B, Hom (p []) A B
```

And, more interestingly, we can define composition to be of type

```
comp : forall d1 d2 A B C,
        Hom (p d1) A B ⟶ Hom (p d2) B C ⟶ Hom (p (d1++d2)) A B
```

Where d1++d2 is the concatenation of both lists. What we have done here is, in essence, is to report some of the work to the syntactic level which is handled by conversion.

Now, for f:Hom (p [d]) A B id·f has type Hom (p []++[d]) A B which is convertible to Hom (p [d]) A B, making id·f=f typable.

However this is about as far as we can get in this direction as, for instance, 1·1 has type f:Hom (p []) A B, hence we can state 1·1=1 but it is *not* an instance of the above as p [d] does not unify with p [].

Graded categories are an indispensable tool to our approach of homological algebra if we want to be able to share proofs between standard preabelian categories and graded objects. However it seems very hard, if possible at all, to deal with them in Coq. It is likely, but time will tell, that this emphasises a weakness of intentional type theories. Extensional type theories do not have such issues but they assign a canonical set structure to every type, which does not fit well with the approach we have taken throughout this manuscript. Other options include giving up the categorical abstraction altogether, the work in [21] shows that it can be done in practice. However the author would argue that if dependent type theories want to be seen as an improvement on existing programming languages, the presentation of homological algebra designed in this manuscript should be implementable roughly as it is.

Appendices

Appendix A

Coq development for Kernel computations

A.1 Kernels are finite dimensional

```
Section split_left.

  Variable (H:preabelian) (A B C:H).
  Variable f : H (A⊕B) C.

  Section split_left_when_isomorphism.

    Variable g : H C A.
    Variable i : isomorphism (ι1·f) g.

    Let η0 := ι2 - ι2·f·g·ι1.

    Let universalisable : η0·f = 0.

    Lemma split_left_when_isomorphism0 :
            isomorphism (uker f _ η0 universalisable) ((ker f)·π2).


   Definition split_left_when_isomorphism : { η : H B (Ker f)
                                & { μ | isomorphism η μ } }.

  End split_left_when_isomorphism.

  Section split_left_when_zero.

    Variable (z:ι1·f=0).

    (* There is an isomorphism between Ker f and
      A⊕Ker ι2·f.
      We need a function from Ker f to B which nullifies ι2·f.
      And a function from Ker ι2·f to A⊕B which nullifies f.
      Then we deduce from the second one a function from A⊕Ker ι2·f to
      A⊕B which also nullifies f, and then a function from
      A⊕Ker ι2·f into Ker f.
      We procede the other way around for the first one. *)
```

```
    Let η0 := (1:H A A) ⊗ ker (ι2·f).
    Let μ0 := ker f · π2.

    Let pti_f : π2·ι2·f = f.

    Let universalisable_η0 : η0·f = 0.

    Let universalisable_μ0 : μ0·(ι2·f) = 0.

    Let η1 := uker _ _ η0 universalisable_η0.
    Let μ1 := ker f ·π1·ι1 + (uker _ _ μ0 universalisable_μ0)·ι2.

    Lemma split_left_when_zero0 : isomorphism η1 μ1.

    Definition split_left_when_zero : { η : H A⊕(Ker (ι2·f)) (Ker f) &
                                { μ | isomorphism η μ }}.

  End split_left_when_zero.

End split_left.

Implicit Arguments split_left_when_isomorphism [ [H] [A] [B] [C] ].
Implicit Arguments split_left_when_zero [ [H] [A] [B] [C] ].


Section split_right.

  Variable (H:preabelian) (A B C:H).
  Variable f:H A B⊕C.

  Let η0 := ker (ker (f·π2) ·f·π1)· ker (f·π2).

  Let universalisable_η0 : η0·f = 0.

  Let η1 := uker f _ η0 universalisable_η0.

  Let μ0 := ker f.

  Let universalisable_μ0 : μ0·(f·π2)=0.

  Let μ1 := uker (f·π2) _ μ0 universalisable_μ0.

  Program Let μ2 := uker (ker (f·π2)·f·π1) _ μ1 _.

  Lemma split_right0 : isomorphism η1 μ2.

  Definition split_right : { η : H (Ker (ker (f·π2)·f·π1)) (Ker f) &
                        { μ | isomorphism η μ }}.

End split_right.
```

```
Implicit Arguments split_right [ [H] [A] [B] [C] ].


Fixpoint pow {H:preabelian} (K:H) (n:nat) :=
  match n with
  | 0%nat ⇒ Zero
  | S p ⇒ K⊕(K ^ p)
  end
where " K ^ n " := (pow K n).


Section effective_kernel.

  Variable (H:preabelian) (K:H).
  Variable (atomic : forall f:H K K, { g | isomorphism f g } + { f=0 }).

  Definition decide_kernel_to_atomic : forall {n:nat} (f:H (K^n) K),
                                { k : nat            &
                                { η : H (K^k) (Ker f) &
                                { μ | isomorphism η μ }}}.

  Definition decide_finite_dimention_kernel : forall (n m:nat) (f:H (K^n) (K^m)),
                                   { k : nat             &
                                   { η : H (K^k) (Ker f)  &
                                   { μ | isomorphism η μ  }}}.

End effective_kernel.
```

A.2 Testing the program

```
(* Λ injects fields into the category of their vector spaces. *)
(* arrow_of_matrix takes a matrix represented as an element of
    (𝕂^n)^m and returns a linear function from the vector
    space 𝕂^n to 𝕂^m *)
Definition arrow_of_matrix {K:DiscreteField} {n m:nat} (μ:((Λ K)^n)^m) :
                          C K ((Λ K)^n) ((Λ K)^m).


(* Λ_base_of_kernel wraps up decide_finite_dimention_kernel to produce
   the basis of the kernel of f in form of a matrix – that is k
   elements of 𝕂^n forming the basis of the kernel.
   C K is the preabelian category of vector space with scalars K. *)
Definition Λ_base_of_kernel {K:DiscreteField} {n m:nat}
                          (f:C K ((Λ K)^n) ((Λ K)^m)) : { k : nat & ((Λ K)^n)^k}.

(* Q_base_of_kernel is a specialised version of Λ_base_of_kernel to the rational
   which shows the rational numbers in reduced form. *)
Definition Q_base_of_kernel {n m:nat}
```

```
                    (f:C Q ((Λ Q)^n) ((Λ Q)^m)) : { k : nat & ((Λ Q)^n)^k}.
```

(1#3 is Coq's notation for the rational number $\frac{1}{3}$. *)*

```
Time Eval vm_compute in
  Q_base_of_kernel (arrow_of_matrix (
   [ [ 1#3 ,  0  ],
     [  0  , 1#3 ] ] : ((ΛQ^2)^2))).
(* Finished transaction in 0. secs (0.012001u,0.s)

   [0 / ⋆]

   Where ⋆ is the unique element of the trivial vector space.
   Homothetic linear functions are injective, hence have a
   0-dimensional kernel. *)

Time Eval vm_compute in
  Q_base_of_kernel (arrow_of_matrix (
   [ [  0 ,  2#1  ],
     [  4#1  ,  0 ] ] : ((ΛQ^2)^2)%cat)).
(* Finished transaction in 0. secs (0.016001u,0.s)

   [0 / ⋆]

   Rotations followed by scaling is injective too.*)

Time Eval vm_compute in
  Q_base_of_kernel (arrow_of_matrix (
   [ [  1#1 ,  1#2  ],
     [   0  ,  0 ] ] : ((ΛQ^2)^2)%cat)).
(* Finished transaction in 0. secs (0.028001u,0.s)

   [1 / [[ -1#2 , 1 ]]]

   Projectors, on the other hand, are not injective.*)


(* μ_n_m n m d s returns a matrix of size n × m. It is filled
 with rational numbers d, d + s, d + 2s, etc...
 We shall use it to produce non-trivial examples to test. *)
 Definition μ_n_m (n m:nat) (d s:Q) : ((ΛQ^n)^m) .


Time Eval vm_compute in
  Q_base_of_kernel (arrow_of_matrix (μ_n_m 5 30 (1#1) (1#1))).
(* Finished transaction in 2. secs (1.76411u,0.032002s)

   [3 /
    [[   1   , -2#1 , 1 , 0 , 0],
     [ 2#1 , -3#1 , 0 , 1 , 0],
```

```
  [ 3#1 , -4#1 , 0 , 0 , 1]]] *)

Time Eval vm_compute in
  Q_base_of_kernel (arrow_of_matrix (μ_n_m 10 10 (1#1) (1#1))).
(* Finished transaction in 1. secs (0.752047u,0.s)

  [8 /
  [[  1 , -2#1, 1, 0, 0, 0, 0, 0, 0, 0 ],
   [ 2#1, -3#1, 0, 1, 0, 0, 0, 0, 0, 0 ],
   [ 3#1, -4#1, 0, 0, 1, 0, 0, 0, 0, 0 ],
   [ 4#1, -5#1, 0, 0, 0, 1, 0, 0, 0, 0 ],
   [ 5#1, -6#1, 0, 0, 0, 0, 1, 0, 0, 0 ],
   [ 6#1, -7#1, 0, 0, 0, 0, 0, 1, 0, 0 ],
   [ 7#1, -8#1, 0, 0, 0, 0, 0, 0, 1, 0 ],
   [ 8#1, -9#1, 0, 0, 0, 0, 0, 0, 0, 1 ]]] *)

Time Eval vm_compute in
  Q_base_of_kernel (arrow_of_matrix (μ_n_m 20 20 (1#1) (1#1))).
(* Finished transaction in 8. secs (8.648541u,0.056004s)

  [18 /
  [[  1 , -2#1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [ 2#1, -3#1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [ 3#1, -4#1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [ 4#1, -5#1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [ 5#1, -6#1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [ 6#1, -7#1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0],
   [ 7#1, -8#1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0],
   [ 8#1, -9#1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0],
   [ 9#1,-10#1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0],
   [10#1,-11#1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0],
   [11#1,-12#1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0],
   [12#1,-13#1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0],
   [13#1,-14#1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0],
   [14#1,-15#1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0],
   [15#1,-16#1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0],
   [16#1,-17#1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0],
   [17#1,-18#1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0],
   [18#1,-19#1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]]] *)

Time Eval vm_compute in
  Q_base_of_kernel (arrow_of_matrix (μ_n_m 40 40 (1#1) (1#1))).
(* Finished transaction in 141. secs (139.556721u,0.528033s)

  [38 / ...] *)
```

Appendix B

Strong subsets are classified

```
(* subset P and { P } are binder-free notations for {x : X | P x}. *)
Definition subset (X:set) (P:X⟶Omega).
Implicit Arguments subset [X].
Notation "{ P }" := (subset P) : bishop_scope.

(* Definition of a strong mono *)
Definition strong_mono (C:category) (X Y : C) (m:C Y X)  : Prop :=
  mono m ∧
  forall (Z W : C) (e:C W Z) (q : epi e) (f:C Z X) (g:C W Y),
        e·f = g·m → exists h:C Z Y, h·m = f.
Implicit Arguments strong_mono [C X Y].

Section in_category_of_sets.

Let C := category_of_sets.

(* The canonical injection from { P } is a mono *)
Lemma ci_mono : forall (X:C) (P : C X Omega), mono C (canonical_injection P).

(* Definition of the function which is constantly equal to True *)
Definition top (X:set) : X ⟶ Omega.
Implicit Arguments top [X].

(* The following function is the function used to explecitely prove that
    the canonical injection from { P } is a strong mono.
    Namely, it is the witness of the existensial property *)
Definition diagonal (X Y:set) (P : Y ⟶ Omega) (f : X ⟶ Y) :
                 (forall x:X, P (f x)) → (X ⟶ { P }).

Lemma ci_strong_mono : forall (X:C) (P: C X Omega),
                         strong_mono ((canonical_injection P):C {P} X).

(* The terminal object of the category of sets. *)
Definition One : C.

(* This is the unique function from X to One *)
Definition one_function (X : C) : C X One.
Implicit Arguments one_function [X].
```

```
Lemma one_terminal : terminal One.

(* Xi m is the image of m. When m is a strong mono, it is used
   as its characteristic function. *)
Definition Xi (X Y : set) (m : X ⟶ Y) : Y ⟶ Omega.
Implicit Arguments Xi [X Y].

(* Defines the natural arrow between a subset m and the
   associated strong subset { Xi m }.*)
Definition expansion_arrow (X Y : C) (m : C X Y) : C X {Xi m }.
Implicit Arguments expansion_arrow [X Y].


(* epi <-> surjective. *)
Lemma epi_surjective : forall (X Y : C) (f : C X Y),
                       (forall y : Y, exists x : X, y = f x) ↔ epi f.

(* corollary: the expansion arrow is always epic. *)
Lemma expansion_arrow_epi : forall (X Y : C) (m : C X Y),
                            epi (expansion_arrow m).

(* if m is a strong mono with domain X, then X is isomorphic to { Xi m }.
   Basically means that all strong monos are (almost) canonical arrows.
   It is also stated that the isomorphic has a commutation property
   which will be used for the final result. *)
Lemma expansion_equiv2 : forall (X Y : C) (m : C X Y), strong_mono m →
    exists u : C { Xi m } X, exists v : C X { Xi m }, isomorphism u v ∧
                      u·m = (canonical_injection (Xi m)).

(* Defines the natural (cartesian) product object of two objects *)
Definition product (X Y : C) : C.

(* Intermediary definition to define a pullback object. *)
Definition pullback_part (X Y Z : C) (f : C X Z) (g : C Y Z) : product X Y ⟶ Omega.

(* The three following definitions define the pullback of two functions. *)
Definition set_pullback_obj (X Y Z : C) (f : C X Z) (g : C Y Z) : C :=
 { pullback_part X Y Z f g }.
Implicit Arguments set_pullback_obj [X Y Z].

Definition set_pullback_p1 (X Y Z : C) (f : C X Z) (g : C Y Z) : C (set_pullback_obj f g) X.
Implicit Arguments set_pullback_p1 [X Y Z].

Definition set_pullback_p2 (X Y Z : C) (f : C X Z) (g : C Y Z) : C (set_pullback_obj f g) Y.
Implicit Arguments set_pullback_p2 [X Y Z].

(* Explicit  definition of the existential arrow closing the universal diagram
    of the natural pullback. *)
Definition set_pullback_universal (X Y Z:C) (f  : C X Z) (g : C Y Z) (Q : C)
```

```
                    (q1 : C Q X) (q2 : C Q Y) (wp : q1·f = q2·g) :
                     C Q (set_pullback_obj f g).
Implicit Arguments set_pullback_universal [X Y Z f g Q q1 q2].

(* Proof that the natural pullback is actually a pullback. *)
Theorem sets_have_pullback : forall (X Y Z : C) (f : C X Z) (g : C Y Z),
             pullback f g (set_pullback_p1 f g) (set_pullback_p2 f g).

(* Subsets of the form { P } are classified with characteristic function
   P. This is proved by giving an isomorphism between { P } and the
   natural pullback of one_function and P. *)
Lemma subset_pullback : forall (X:C) (P:C X Omega),
    pullback (top:C _ _) P one_function (canonical_injection P).

(* Reduces the property of being a classifying pullback
   to being isomorphic to { P } with the appropriate commutation properties. *)
Lemma isomorphic_subset : forall (X Y:C) (P: C X Omega) (p2 : C Y X)
   (u : C { P } Y) (v: C Y { P }),
   u·p2 = canonical_injection P →
   isomorphism u v →
   pullback (top:C _ _) P one_function p2.

(* The following two theorems asserts that the strong monos of the
   category of sets are classified. *)
Theorem characteristic_function : forall (X Y : C) (m : C Y X), strong_mono m →
  (m·(Xi m) = one_function·top ∧
   pullback (top:C One Omega) (Xi m) (one_function:C Y One) m).

Definition el_function (X:set) (a:X) : One ⟶ X.
Implicit Arguments el_function [X].

Theorem Xi_unique : forall (X Y : C) (m:C Y X) (x : C X Omega), strong_mono m →
   pullback (top:C _ _) x one_function m → x = (Xi m).
```

Conclusion

In this manuscript we have studied dependent type theories through the spectrum of homological algebra. Specifically, we have developed some parts of the theory of effective homology in the style which is supposedly the most natural in type theory, keeping in mind two specific problematics from programming: efficiency and code sharing.

We have had the occasion to see that intentional type theories, though very powerful, are not always up to the task we would like them to perform. On the one hand, they support naturally efficient functional programs, provided we have a good way to rid the mathematical framework of the principle of unique choice (which is straightforward in Coq) and programs can be done at a fairly conceptual level allowing some code sharing. We needed only a minimalistic modification of the core theory to allow for truly efficient numerical computation. We also needed some more heavyweight work to simplify the handling of dependent types in programs and proofs, but this was part of the development environment rather than the theory. On the other hand, the limits of dependent types are reached easily, and many useful mathematical concepts that could provide more code sharing – like graded categories – better efficiency – like multicategories – or reduced boilerplate – like internal languages – have reasonable representations which cannot be implemented.

Modifying the theory to accommodate for these new structures is no simple work. Graded categories cannot be defined because the associativity of a generic monoid cannot be part of the conversion. Similarly, the categories of graded object need a dynamic check of a static property to work around the rigidity of the conversion. There are propositions in the literature addressing precisely this sort of issue, they are far from trivial but might make reasonable extensions of intentional type theories.

Higher dimensional constructions like multicategories or higher categories are on another level altogether: not only intentional type theories are most likely incapable of accommodating them without further modifications, but it is not clear how such a construction should be encoded and used. Hence, higher order constructions need to be better understood from a programming perspective before the question of what they would require as modifications to type theories can be asked.

Internal languages, lastly, raise the question of binders. Representing binders in very expressive type theories like Coq does not seem to be easy at all. There are works in the literature which indicate that it might be possible to do it without extending the language, though. It might be better nonetheless

to add specific support for binders in the core theory.

These questions, at least graded categories and multicategories, can be seen as being in the spirit of so called type-level programming. Where types become a subject of programming rather than just properties. Dependent type theories allow for well behaved type-level programming, though, as we have argued, not powerful enough to express certain idioms we would like to use. Were the above issues be solved, they would open way for new styles of programming which seem beyond the reach of traditional functional languages.

Bibliography

[1] GNU MPFR Library.
URL http://www.mpfr.org/

[2] GNU Multiple Precision Arithmetic Library.
URL http://gmplib.org/

[3] Haskell.
URL http://www.haskell.org/

[4] The Coq Proof Assistant.
URL http://coq.inria.fr/

[5] The Homalg project.
URL http://homalg.math.rwth-aachen.de/

[6] Peter Aczel & Michael Rathjen. Notes on constructive set theory, 2000.
URL http://www.ml.kva.se/preprints/meta/AczelMon_Sep_24_09_16_56.rdf.html

[7] Thorsten Altenkirch, Niels Anders Danielsson, Andres Löh & Nicolas Oury. ΠΣ: Dependent Types without the Sugar. *Functional and Logic*, (Sheard 2005), 2010.
URL http://www.springerlink.com/index/91W712G2806R575H.pdf

[8] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack & Laurent Théry. Extending Coq with Imperative Features and its Application to SAT Verification. *Interactive Theorem Proving*, pp. 83–98, 2010.
URL http://www.springerlink.com/index/U814181H51857784.pdf

[9] Andrea Asperti, Wilmer Ricotti, Claudio Sacerdoti & Enrico Tassi. A new type for tactics. *PLMMS'09*, (June):p. 22, 2009.

[10] Lennart Augustsson, Thierry Coquand & Bengt Nordström. A short description of another logical framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pp. 39–42. 1990.

[11] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, volume 21(8):pp. 613–641, 1978. ISSN 00010782. doi:10.1145/359576.359579.
URL http://portal.acm.org/citation.cfm?doid=359576.359579

[12] Ulrich Berger & Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda -calculus. In *Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pp. 203–211. IEEE Comput. Sco. Press, 1991. ISBN 0-8186-2230-X. doi:10.1109/LICS.1991.151645.
URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=151645`

[13] Errett Bishop. *Foundations of constructive analysis*. McGraw-Hill, 1967.

[14] Frédéric Blanqui, Jean-Pierre Jouannaud & Mitsuhiro Okada. The Calculus of Algebraic Constructions. In Paliath Narendran & Michael Rusinowitch (editors), *Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-66201-3.
URL `http://arxiv.org/abs/cs/0610063`

[15] Mathieu Boespflug. Conversion by Evaluation. In *Practical Aspects of Declarative Languages: 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010, Proceedings*, p. 58. Springer-Verlag New York Inc, 2010. ISBN 3642115020.

[16] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. Ph.D. thesis, 2005.

[17] Sylvain Conchon & Jean-Christophe Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 workshop on Workshop on ML - ML '07*, p. 37. ACM Press, New York, New York, USA, 2007. ISBN 9781595936769. doi:10.1145/1292535.1292541.
URL `http://portal.acm.org/citation.cfm?doid=1292535.1292541`

[18] Thierry Coquand & Arnaud Spiwack. Towards constructive homological algebra in type theory. In Manuel Kauers, Manfred Kerber, Robert Miner & Wolfgang Windsteiger (editors), *Towards Mechanized Mathematical Assistants*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73083-5. ISSN 0302-9743.
URL `http://www.springerlink.com/index/9264606874082662.pdf`

[19] Edsko de Vries, Rinus Plasmeijer & David Abrajamson. Uniqueness typing simplified. In Olaf Chitil, Zoltán Horváth & Viktória Zsók (editors), *Implementation and Application of Functional Languages*, volume 0 of *Lecture Notes in Computer Science*, pp. 201–218. Springer Berlin / Heidelberg, 2008. doi:10.1007/978-3-540-85373-2_12.
URL `http://www.springerlink.com/index/r3469703v3p34020.pdf`

[20] Radu Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, volume 51(1):pp. 176–178, 1975. ISSN 0002-9939.
URL `http://www.ams.org/journals/proc/1975-051-01/S0002-9939-1975-0373893-X/S0002-9939-1975-0373893-X.pdf`

[21] César Domínguez & Julio Rubio. Computing in Coq with Infinite Algebraic Data Structures. 2010.
URL `http://arxiv.org/abs/1004.4998`

[22] François Garillot & Benjamin Werner. Simple types in type theory: Deep and shallow encodings. In *Proceedings of the 20th international conference on Theorem proving in higher order logics*, pp. 368–382. Springer-Verlag, 2007. ISBN 3540745904.
URL http://portal.acm.org/citation.cfm?id=1792233.1792260

[23] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, volume 11(03):pp. 301–506, 2001. ISSN 0960-1295.
URL http://journals.cambridge.org/abstract_S096012950100336X

[24] Georges Gonthier. A computer-checked proof of the four colour theorem, 2004.
URL http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf

[25] Georges Gonthier & Assia Mahboubi. A Small Scale Reflection Extension for the Coq system, 2008.
URL http://hal.inria.fr/inria-00258384/en

[26] Michael Gordon, Robin Milner & Christopher Wadsworth. *Edinburgh LCF: A mechanised logic of computation*. Springer-Verlag, 1979. ISBN 0387097244.

[27] Benjamin Grégoire & Xavier Leroy. A compiled implementation of strong reduction. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pp. 235–246. ACM, 2002. ISBN 1581134878.
URL http://portal.acm.org/citation.cfm?id=583852.581501

[28] Benjamin Grégoire & Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In *Theorem Proving in Higher Order Logics*, pp. 98–113. Springer, 2005. ISSN 1521-3773.
URL http://www.springerlink.com/index/8rxwltpuh8ly4936.pdf

[29] Benjamin Grégoire & Laurent Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In *Automated Reasoning*, pp. 423–437. Springer, 2006.
URL http://www.springerlink.com/index/060m1340v752t4r5.pdf

[30] Allen Hatcher. *Algebraic Topology*. Cambrige University Press, 2002. ISBN 9780521795401. doi:10.2277/0521795400.
URL http://www.math.cornell.edu/~hatcher/AT/ATpage.html

[31] Douglas Howe. Higher-Order Abstract Syntax in Isabelle/HOL. *Interactive Theorem Proving*, pp. 481–484, 2010.
URL http://www.springerlink.com/index/EL63808617428182.pdf

[32] Anatolii Alexeevitch Karatsuba & Yuri Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akad. Nauk SSSR*, volume 145, p. 85. 1962.

[33] Gregory Maxwell Kelly. Basic concepts of enriched category theory. *Theory And Applications Of Categories*, (10), 1982.

[34] Claude Kirchner, Florent Kirchner & Hélène Kirchner. Strategic Computation and Deduction. 2008.
URL http://hal.inria.fr/inria-00433745/

[35] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman & Amr Sabry. Backtracking, interleaving, and terminating monad transformers. *ACM SIGPLAN Notices*, volume 40(9):p. 192, 2005. ISSN 03621340. doi:10.1145/1090189.1086390.
URL http://portal.acm.org/citation.cfm?doid=1090189.1086390

[36] F. William Lawvere. Adjointness in Foundations. *Dialectica*, volume 23(3-4):pp. 281–296, 1969. ISSN 1746-8361. doi:10.1111/j.1746-8361.1969.tb01194.x.
URL http://onlinelibrary.wiley.com/doi/10.1111/j.1746-8361.1969.tb01194.x/abstract

[37] Tom Leinster. Higher Operads, Higher Categories. 2003.
URL http://arxiv.org/abs/math.CT/0305049

[38] Stéphane Lengrand, Roy Dyckhoff & James Mckinna. A Focused Sequent Calculus Framework for Proof Search in Pure Type Systems. *Logical Methods in Computer Science*, 2010.

[39] Xavier Leroy. The Zinc Experiment: An Economical Implementation Of The ML Language, 1990.

[40] Saunders Mac Lane & Ieke Moerdijk. *Sheaves in geometry and logic: A first introduction to topos theory*. Springer, 1992. ISBN 978-0-387-97710-2.

[41] Per Martin-Löf. *Intuitionistic type theory*. 1984.
URL http://www.cs.cmu.edu/afs/cs/Web/People/crary/819-f09/Martin-Lof80.pdf

[42] Conor McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, 2001.

[43] Eugenio Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989.*, pp. 14–23. IEEE Comput. Soc. Press, 1989. ISBN 0-8186-1954-6. doi:10.1109/LICS.1989.39155.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=39155

[44] César Muñoz. A calculus of substitutions for incomplete-proof representation in type theory, 1997.
URL http://hal.inria.fr/inria-00073380/

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

[46] Chris Okasaki. *Purely functional data structures*. 1999. ISBN 0521663504.

[47] Jacques Penon. Sur les quasi-topos. *Cahiers de topologie et géométrie différentielle catégorique*, volume 18(2):pp. 181–218, 1977.

[48] Julio Rubio & Francis Sergeraert. *Constructive homological algebra and applications*. 2006.
URL http://map.disi.unige.it/summer_school/pdf/Sergeraert%_MAP_06.pdf

[49] Francis Sergeraert, Yvon Siret & Xavier Dousson. The Kenzo program.
URL http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/

[50] Matthieu Sozeau. Subset coercions in Coq. *Types for Proofs and Programs*, pp. 237–252, 2007.
URL http://www.springerlink.com/index/8p35448426n2n41k.pdf

[51] Arnaud Spiwack. An abstract type for constructing tactics in Coq. 2010.
URL http://hal.inria.fr/inria-00502500/

[52] Thomas Streicher. *Semantical investigations into intensional type theory*. Ludwig Maximilian Universität München, 1993.

[53] Pierre-Yves Strub. Coq Modulo Theory. In *Computer Science Logic*, pp. 529–543. Springer, 2010.
URL http://www.springerlink.com/index/J64X4228024756L1.pdf

[54] Laurent Théry & Benjamin Grégoire. CoqPrime.
URL http://coqprime.gforge.inria.fr/

Table of slogans

Equality belongs to sets, 19
Choice: surjections have a section, 22
Unique choice: bijections have an inverse, 22
Unique choice is not valid in the category of abelian groups, 29
Static properties need not be enforced by dynamic tests, 35
Lies are locally true, 35
Endomorphisms of fields are null or automorphisms, 36
Peano numbers: simple, yet intractable, 42
Every number has at most two digits, 44
Fewer tactics in smaller chunks, 63
Quasitopoi are almost topoi, 74
In LCC categories, families are in the wrong direction, 75
Internal categories have dependencies in the wrong direction, 85

Index

Abelian category, 28
Additive category, 27
Adjoint, 69
Adjunction, 69
Arrow, 18

Base change, 72
Bicategory, 87
Biproduct, 27
Boundary operator, 26

Cartesian closed category, 69
Cartesian product, 18
Category, 18
- abelian, 28
- additive, 27
- cartesian closed, 69
- enriched, 84
- graded, 92
- internal, 85
- LCC, 73
- locally cartesian closed, 73

Chain complex, 26, 32
Classifier
- subobject, 70

Coequaliser, 21
Cokernel, 27
Connected pair, 26
Contravariant functor, 68
Coproduct, 20
Covariant functor, 68

Differential morphism, 26, 32
Disjoint union, 18

Enriched category, 84
Epimorphism, 19
- normal, 28

Equaliser, 21
Exact pair, 26

Field, 36
Finite dimensional object, 36
Function, 16
Functor, 68
- contravariant, 68
- covariant, 68
- monad, 90

Generalised multicategory, 91
Goal sensitive, 56
Graded abelian group, 26
Graded category, 92
Graded module, 26
Graded object, 30
- morphism, 30

Groupoid, 86

Homology, 26
Homset, 18

Image, 17
Initial object, 19
Internal category, 85

Kernel, 27

Lcc category, 73
Locally cartesian closed category, 73

Monad, 90
Monomoprhism
- normal, 28

Monomorphism, 19
- strong, 73

Morphism, 18

Multicategory, 75

Natural transformation, 86
Numbers
  binary trees, 43
  list of bits, 43
  more digits, 44
  Peano, 42

Object, 18
ω-category, 88

Part, 17
Postinverse, 21
Preabelian category, 29
Predicate, 17
Preinverse, 21
Prepullback, 70
Principle of choice, 21
Principle of unique choice, 22
Product
  category, 20
  set, 18
Pullback, 70

Quasitopos, 74

Set, 16
Short exact sequence, 29
  effective, 30
Slice category, 71
Splitting lemma, 29
Strong monomorphism, 73
Subobject, 19
Subset, 17
  strong, 17
Sum
  category, 20
  set, 18

Terminal object, 19
Topos, 70

Unbiased, 78

Zero arrow, 27
Zero object, 27