# Studies in Mechanical Verification of Mathematical Proofs

Mark Ruys

# Studies in Mechanical Verification of Mathematical Proofs

Cover design based on a Trojan labyrinth

CIP-Gegevens Koninklijke Bibliotheek, Den Haag

Ruys, Mark Pieter Jan

Studies in Mechanical Verification of Mathematical Proofs / Mark Pieter Jan Ruys.
- [S.l. : s.n.] (Wageningen : Ponsen en Looijen)
Proefschrift Nijmegen. - Met lit. opg. - Met samenvatting in het Nederlands.
ISBN 90-9012644-9
Trefw.: informatica.

# Studies in Mechanical Verification of Mathematical Proofs

een wetenschappelijke proeve op het gebied van de natuurwetenschappen, wiskunde en informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen
op dinsdag 29 juni 1999, des namiddags om 3.30 uur precies

door

Mark Pieter Jan Ruys

geboren 24 maart 1966 te Rochester, N.Y., U.S.A.

druk: Ponsen & Looijen, Wageningen

Promotor: Prof.dr. H.P. Barendregt

Co-promotor: Dr. E. Barendsen

Manuscriptcommissie: Prof.dr. J.W. Klop (Vrije Universiteit Amsterdam)
Dr. R. Pollack (University of Glasgow)
Dr. J.H. Geuvers (Technische Universiteit Eindhoven)

Voor Ton

# Acknowledgements

First and foremost my thanks must go to Henk Barendregt, who is an inexhaustible source of inspiration to me, always open for new ideas. On many occasions, Henk renewed my pleasure in doing research.

Furthermore, I would like to mention a number of people who played a particular rôle in the realization of my thesis. Erik Barendsen has been very important when it came down to actually finishing my thesis. Especially during this crucial period, Erik kept encouraging me. Also his reflections improved my thesis a lot. Gladly I thank Herman Geuvers. Particularly during the first years of my research, we had many discussions and shared a lot of ideas. Furthermore, without Randy Pollack there would have been no LEGO, and without LEGO, my thesis would not have been the same. I am grateful to Randy for building the proofchecker LEGO, and for the many significant comments I received from him. I also wish to mention the fruitful discussions I had with Gilles Barthe. A lot of ideas we developed are reflected in this thesis, especially in Chapter 5. I must also thank Jan Willem Klop, Peter Aczel, Hugo Elbers, John Harrison, Wil Dekkers, and Milena Stefanova who have in some way or another influenced my research.

Last but not least, I would like to thank my dearest Manon. More than once, a holiday was spoiled by rain. But again and again she left me space to continue my work. And let me mention our child Ilse. When she smiles, we lose all track of time.

Mark Ruys, May 1999

# Contents

**1 Introduction** **1**

1.1 Motivation . . . 1

1.2 Proof Assistants . . . 6

1.3 Type theory . . . 7

1.4 Description . . . 11

**2 Representing Logic** **13**

2.1 Relevant views on logic . . . 13

2.2 Logic formalized . . . 14

**3 Representing Mathematical Notions** **19**

3.1 Sets . . . 20

3.2 Functions . . . 29

3.3 Subsets . . . 35

3.4 Mathematical Structures . . . 39

3.5 Towards Analysis . . . 45

**4 Case Studies** **53**

4.1 Drinkers Principle . . . 53

4.2 There are infinitely many primes . . . 56

4.3 Binomial theorem . . . 61

4.4 Fundamental theorem of algebra . . . 62

**5 A Two-Level Approach** **71**

5.1 Introduction . . . 71

5.2 Mathematical Background . . . 72

5.3 The Naming Principles . . . 76

5.4 Congruence Types . . . 80

5.5 Formalization in LEGO . . . 83

5.6 Examples . . . 85

**6 Conclusions** **91**

6.1 Recommendations . . . 91

6.2 Lean proof checking . . . 95

6.3 Related work . . . 96

6.4 Results . . . 96

**A The LEGO System** **99**
A.1 Quick introduction to LEGO . . . . . . . . . . . . . . . . . . . . . . . . 99
A.2 Naming conventions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
A.3 The LEGO library . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
A.4 Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

**Bibliography** **107**

**Samenvatting** **115**

**Curriculum Vitae** **117**

# Chapter 1

# Introduction

This thesis is about proof checking in type theory. We will investigate the question how to mechanically verify mathematical proofs. The computer systems we consider are based on a type-theoretical framework. The method we follow is to develop a few representative case studies. This gives us the experience to draw conclusions and to give recommendations. In order to formalize the theorems presented in the case studies, we first have to develop a library of formalized mathematics. We will do this from scratch. During this development we will encounter several choices to make and problems to solve. One particular problem, namely equational reasoning, is studied in more depth and a method for dealing with it in a convenient way is presented in a separate chapter.

## 1.1 Motivation

Hardware and software systems belong to the most complex artifacts produced by the homo sapiens. These systems have appeared relatively late in the human enterprise, some 50 years ago. The rise of these systems marked the beginning of a new phase in the industrial revolution.

It is remarkable that one of the few possible classes (arguably the only class) of 'objects' more complex than IT (Information Technology) products, already has been around more than two millennia. We mean the timeless works of mathematics with its concepts, computations and proofs. One of the early high points in this realm, paying proper attention to all these three aspects, consists of the works of Archimedes (287-212 B.C.).

There appears to be a striking difference between the quality of IT systems and that of mathematics. The IT systems, notably the software, often contain bugs: they do not run as they are intended to. If one buys a disc with a program on it, then one usually sees that there is a warranty for the proper functioning of the disc, but not for that of the program itself! As embedded software is used in many vital components of industrial products, like rockets, airplanes, power plants and money transfer systems, the bugs in these systems have caused a lot of damage and even fatalities. It has been estimated that the bug in the Pentium chip a few years ago has caused a loss of 480 million US$ to the manufacturer. See (Peterson 1995) for other cases and documentation.

Mathematics on the other hand appears to be usually correct. True, also mathematical papers sometimes contain errors. These are either found by the author, by some expert reader (who possibly wants to use the result) or are left unnoticed. In some sense this is analogous to the situation with errors in IT systems. But there is an intrinsic reason to believe that present day software systems contain comparable more bugs than mathematical papers.

Before giving this reason, let us look at the cause of errors in software. The reason is simple. Programs are large, sometimes consisting of more than a million lines, and they have a refined action. A small oversight in one of the million lines is enough to introduce an error. The complexity of software is also the underlying cause of the 'second software crisis': there are simply not enough people capable of producing the relevant software needed for todays industry. But worse is the first and principal crisis: it is hard—if not impossible—to produce satisfactory software on time. Both crises are caused by the difficulty to instruct a computer to do simple tasks that humans seem to do without effort, let alone more difficult tasks. Since computers do these tasks much more efficiently than humans, the reward for this endeavor to build software has been considered worthwhile. The mentioned complexity is also the reason why testing IT systems is of limited value, since the number of cases to be checked is more than astronomical. An exhaustive test simply will take too long.

Back to mathematics. The statement that the work of mathematicians in general is more reliable than that of IT specialists is puzzling, since we also said that mathematics is more complex than IT. The reason for the reliability of mathematics is that this discipline has *proofs*. Proofs may be very complex, indeed more complex than software; but they are crystal clear. This clarity is a tool for the mathematician to purify his work from errors. The following metaphor may explain the situation better. Trying to find a proof is like searching for the correct path in a labyrinth to the exit. The search is complicated and so is the resulting path. But the fact that the path correctly leads to the exit can be verified without effort. This means that it is very easy to specify the correctness of a path: just follow the path and see it leads to the goal 'without cheating'.

Given this situation, the challenge is to design IT systems such that the fulfillment of their required properties can be warranted with the same degree of certainty as the validity of mathematical theorems.

## Formal methods in IT design

One possibility is to apply the method of proving to the construction of IT systems. This idea came up in the late 1960s under the name 'formal methods'. The idea was to have a language in which one could describe designs $d$ of IT systems together with the specification $S$ of the desired properties of the system. If one then could formally prove $S(d)$, this would give the highest possible warranty that a realization of the design $d$ has its required behavior. Proving out of the blue is usually impossible, so in the proof of $S(d)$ one may assume that components $d_i$ of $d$ satisfy a subspecification $S_i$, i.e. that $S_i(d_i)$ holds. These components $d_i$ either have to be constructed again, in which case the story is repeated; or are bought from another manufacturer, in which case the seller is responsible that $S_i(d_i)$ holds. Eventually this procedure leads to elementary components $d_e$ whose properties $S_e$ are warranted by the laws of nature.

Although the idea of using formal systems is in principle a good one, the existing technology is inadequate to deal with some huge problems. The dominating programming style is that of imperative languages (including variants like object oriented programming). This computational model is less apt to be treated by formal methods in a nice way. The reason is that a substantial program $d$ has a very large number of different components. This implies that the proof of $S(d)$ from $S_i(d_i)$ will be complex. This complexity is different from that of mathematical proofs. Proofs in mathematics are complex because of depth of abstraction. The intended proof of $S(d)$ from the $S_i(d_i)$ is complex because of their length consisting of rather trivial steps.

In order to state the mentioned problems, we have to be more explicit about what needs to be done. Following ideas of (Wupper and Meijer 1997) one can state this as follows. One wants a system with a given behavior. In order to construct such a system one writes down a formal specification $S$ of the behavior and a formal design $d$ of the system. The design will usually be built up from components $d_1, \ldots, d_n$, where each $d_i$ satisfies a subspecification $S_i$. Now one wants to be sure that the system obeys the required behavior. The best one can strive after is to prove formally that

$$S_1(d_1), \ldots, S_n(d_n) \vdash S(d) \qquad (+)$$

i.e. $S(d)$ formally follows from the assumptions $S_1(d_1), \cdots, S_n(d_n)$. At this point there are two problems

*1.* How do we get the correct specification $S$?

*2.* How can we warrant (+), knowing that the proof may be complex and hence error prone?

A satisfactory answer to problem 2 was given by (de Bruijn 1970). He designed a language Automath in which proofs can be represented precisely. A statement (+) being proved by $p$ now becomes

$$S_1(d_1), \cdots, S_n(d_n) \vdash_p S(d). \qquad (++)$$

Now problem 2 becomes

*2.1.* How do we get the so-called proof object $p$?

*2.2.* How can we warrant the correctness of (++)?

Problem 2.2 has been addressed satisfactorily by de Bruijn, as he succeeded to represent statements and proofs in such a way that (++) becomes verifiable by a small program. A proof checking algorithm with a small program is said to satisfy the *de Bruijn criterion*. The final reliability of (++) then can be checked by anyone who cares to inspect this program (or is willing to write a personal version of it).

So now we are left with problem 1 (how to obtain a formal specification $S$) and problem 2.1. (how to obtain a proof-object $p$). For software written in the current imperative style problem 1 is already somewhat awkward. Several specification languages have been proposed, e.g. VDM (Jones 1990). Now the formal specifications $S, S_1, \ldots, S_n$ in such a language is in principle of a manageable size. But the problem usually is whether a formal specification $S$ does express correctly the intended behavior of the system. Let us call this the specification problem. The other problem (2.1) of constructing a proof-object $P$ for (++) becomes the bottleneck for software written in an imperative style. In fact to achieve (++) for large imperative programs is very difficult. Most industrial software has not been verified this way.

In spite of this failure for software verification, the program to prove correctness of hardware has been successful, see (Goossens 1992, Rushby and von Henke 1993, Ruess, Shankar and Srivas 1996). This is so because hardware seen from the right level of abstraction is in general simpler than software. One can state stylistically:

$$\text{hardware} : \text{software} = \text{propositional logic} : \text{predicate logic}.$$

Since provability in predicate logic is undecidable, this also explains why we have to use proof objects for the formal verification of software. On the other hand provability in propositional logic is decidable and hence the verification of hardware can be done by (reliable) theorem provers. (Actually this is an over-simplification. Hardware is in fact more complex than the propositional logic level since repeated hardware patterns and also time considerations ask for predicates.)

This methodology of verified design has been used for more than 15 years (usually not in the style of de Bruijn, but with all kinds of proof generators). As a result, hardware nowadays is very reliable. The bug mentioned above in the arithmetic unit of the first Pentium chip was in fact not found in the hardware, but in the microcode (software provided by the manufacturer).

Since imperative software is very complex and has insufficient modular structure, there has not been developed a major activity producing corrections proofs of such programs. There is a notable exception. Communication protocols are small but highly important programs, often used as embedded software in telecommunication systems, remote control consoles and the like. The correctness of these programs is of vital importance for the proper functioning of these systems. Moreover it is expensive for the manufacturer to call back systems for a repair. Therefore considerable successful effort is being spent on proving the correctness of implementations of these communication protocols (Sellink 1996).

In the early 1990s functional programming started to come of age. For this model of computation each program component is built up in an surveyable modular way[1], i.e. is constructed from a limited number of parts such that the proportion of the whole is reflected in a clear way from those of the parts. Therefore it is justified to hope that for functional programming correctness proofs are feasible.

## Mathematics

In mathematics there has developed a strong culture for the right formulation of statements to be proved. Moreover, in many cases already simple statements are a challenge to be proved. For this reason our case studies come from mathematics. The goal of these case studies is not so much the construction of mechanically verified proofs, but rather a study of the technique one needs in order to comfortably formalize proofs.

Usually, proof-checking of existing mathematics does not have the property that the specification of the statement to be proved is problematic. On the other hand, the efficient formalization of proofs is a formidable challenge. One aims at proof-objects that are not only complete formalizations, but also should be 'feasible'.

[1]Functional programming languages have the very important property of *referential transparency*. Referential transparency forms the basis of modularity.

This means in the first place that it should be possible for a person that knows the intuitive proof to produce in a 'user-friendly way' the corresponding proof-object (possibly with the help of an interactive proof development system). It also means that the resulting proof-objects should not be too large.

We believe that producing case studies in the formalization of mathematics is of interest for several reasons.

1. There probably will be a positive spin-off for the quest for correctness of IT-systems, notably for software.

2. The highest possible degree of correctness of mathematics will be warranted. The role of refereeing by peers will change. The emphasis will shift from the correctness to the relevance of a result, since correctness already can be verified mechanically.

3. Complex mathematical notions can be represented exactly on a computer, even incompatible notions. In this way, systems of so called 'computer mathematics' (CM) may result in heavy libraries of verified theories, certified algorithms and user friendly tools to help the development of new theories. In short, experienced mathematicians may be helped by systems of CM in the same way as they are being helped by systems of Computer Algebra (CA) for the development of pure and applied mathematics. Also for students there is a benefit. Systems for CM already have proven to be instrumental for the craftsmanship of producing mathematical proofs. They help students in developing awareness of logical steps applied. Students even seem to like doing mathematics on a computer.

4. It is a challenge for logic and the foundations of mathematics to make feasible formalization possible. This point will be explained below.

5. Some proofs consists of a large number of cases of similar structure. Systems for CM can take over the elaborate part of generating all cases. An example is the proof of the four color theorem by (Appel and Haken 1976), which was carried out by means of a computer system.

### Obstacles

Although mathematics is regarded as an exact science, actually in a sense it is not exact at all. Namely, most proofs in mathematical journals are in fact merely sketches which should convince the reader that some asserted theorem does hold. All proof-steps which ought to be obvious for the reader are left out, leaving only those parts of the proof which form the essence. We call this *informal practice.* Informal practice makes proofs easier to grasp and hence makes it easier to see that these proofs are indeed correct. On the other hand, we have computers. In order to let a computer verify a proof, the program needs to obtain all details, and every tiny logical step of which the proof consists needs to be spelled out. A computer based proof checker needs a *fully formalized* proof as input.

Suppose the author of a proof has left out on purpose some side conditions or some exceptions for the sake of clarity. Usually, the reader will 'see' immediately that this omission is provable indeed. A computer system on the other hand still needs to verify these omissions in full detail. Usually, this is quite an involved job to do. Even worse, in some cases such an exception may not be provable at all without some (trivial) modifications in the original proof.

We also have to realize that proof checkers have a few inherent weaknesses. Firstly why should we believe that the proof checker accepts only valid (true) proofs? How can we be sure that it does not accept some proof that leads to a contradiction? Since a proof checker is just a piece of software that runs on some equipment, they are also vulnerable to the weaknesses of IT systems described in the beginning of this chapter. Following the *de Bruijn criterion*, we should keep the kernel of a proof checker as simple as possible. Then anyone could write his own proof checker fairly easily, and use it as an independent judge. The tools we use to generate proofs may be as complex as desirable, as long as the resulting proofs can be checked by a simple system.

On the other hand, we do not wish to work in a system whose proof theoretical strength is weak. Although we could encode the logic we wish to use, and introduce mathematical concepts axiomatically, this approach is only suitable for small case studies. Unless a proof checker provides very powerful tacticals, formalizing large bodies of mathematical text by means of a minimal verification system seems to be an extremely tedious job to do.

Secondly, it is very hard to be completely sure that a formalization exactly models the phenomenon we wish to describe. In order to believe that a given theorem $T$ we intend to prove is true indeed, we have to make sure that every formalized definition on which the formalization of the theorem $T$ directly or indirectly depends, exactly models our intention. This may seem easier than it is. We believe that this modeling problem is one of the most problematic aspects of mechanical proof verification.

## 1.2 Proof Assistants

Currently, a variety of computer systems is used in mathematics and the other exact sciences as an aid for research. Two of the most widespread categories are systems for numerical analysis and for computer algebra. A numerical analysis package is capable of making use of the raw processing power of a computer to quickly compute a possibly extremely large number of real values by approximations. Examples of these packages are LINALG (Johnson 1992) and ScaLAPACK (Choi, Dongarra, Pozo and Walker 1992).

Computer algebra systems are more sophisticated in the sense that they use an exact representation of symbols and formulas instead of approximated values. Examples of computer algebra systems are Maple (Char, Geddes, Gonnet, Leong, Monagan and Watt 1992) and Mathematica (Wolfram 1991). Because numerical analysis systems compute by approximations, they are vulnerable for rounding errors. Computer Algebra systems use a knowledge base of algorithms and rules to rewrite algebraic expressions into simpler ones. Although computer algebra systems do not use approximations, even these systems may deliver wrong solutions because most computer algebra systems do not check all side conditions. For example, taking the real valued square root of a negative number, or division by zero, is often undetected and accepted.

Proof assistant systems form a third category. Their emphasis is not on computing values, but on proving theorems. They are especially suited for checking results. Proof assistants might be divided into two flavors: *proof checking* and *proof generation*. In general, proof generation systems are limited because most logics are undecidable. When we use a proof generator to find a proof of some lemma, in most cases the checker will give up at a certain moment because it has no clue how to find a proof. Examples of these systems are systems based on resolution logic like Otter (McCune 1990), and the Boyer-Moore theorem prover Nqthm (Boyer and Moore 1988). The latter system will ask the user for a hint when it fails to generate a proof. So in an interactive way, the system can be used to formalize mathematics. (Shankar 1986, Shankar 1994) has used Nqthm to formalize Gödels second incompleteness theorem and the Church-Rosser theorem. Another very popular proof assistant system today is PVS (Owre, Rushby and Shankar 1992). PVS is a fairly user-friendly system, but less reliable.

As proof generation systems get more complex, they will be more vulnerable for implementation errors. Reliability of proof assistants is crucial. What is otherwise the point of formalizing mathematics in the first place? Proof verification systems have a far easier job to do. Namely instead of finding a valid proof, these systems only have to verify that a given proof follows the rules of logic. Consequently proof verification systems are easier to build and more likely to be correct. Examples of proof verification systems are HOL (Gordon 1991) and Isabelle (Nipkow and Paulson 1992).

As argued above, it is important to be able to independently check formalized proofs. Proof checkers which are based on type theory have proofs as first class citizens. This makes it easy to hand over these objects to other people using other proof checkers. Examples of proof checkers which actually produce proofs are Coq (Dowek et al. 1993), LEGO (Pollack 1994) and Alf (Magnusson and Nordström 1994). Another advantage of using type theory for formalizing proofs is that we are encouraged to build constructive proofs only. Although this is not compulsory, constructive proofs have the advantage that they often contain a computational content. So for example, suppose we have a constructive proof of a theorem

$$\forall x \exists y[\phi(x,y)] \ .$$

Then a constructive proof for this theorem will contain a function which assigns to every $x$ a $y$ such that $\phi(x,y)$ holds. We will use this fact to get a prime generator directly from a proof of Euclid's theorem of the infinity of the set of prime numbers.

## 1.3 Type theory

The common view that set theory as formulated in first order predicate logic with equality can serve as a foundation of mathematics is an misconception. Mathematics consists of defining, reasoning and computing. In ordinary logic, definitions are usually seen as abbreviations in an auxiliary language that denote their full expansion in the official language. For example in number theory one has the definitions:

$$\begin{aligned} \texttt{Prime}(x) &:= x > 1 \mathbin{\&} \forall y[(y < x) \to (y \mid x) \to (y = 1)] && (1.1) \\ y \mid x &:= \exists z[z * y = x] && (1.2) \end{aligned}$$

If these definitions are seen as abbreviations, then the statement expressing that there exists a prime twin:

$$\exists x[\texttt{Prime}(x) \mathbin{\&} \texttt{Prime}(x+2)]$$

stands for:

$$\begin{aligned}\exists x[\ &(x > 1 \;\&\; \forall y[(y < x) \to (\exists z[z * y = x]) \to (y = 1)]) \;\&\\ &(x + 2 > 1 \;\&\; \forall y[(y < x + 2) \to (\exists z[z * y = x + 2]) \to (y = 1)])\ ]\end{aligned}$$

For more complicated statements the official translation becomes infeasible. Therefore a better view on definitions like (1.1) and (1.2) is that the defined expressions become part of the official language and := stands for an axiomatic extension or a reduction discussed below.

The way reasoning is captured in first order predicate logic is not bad, except that natural deduction for intuitionistic logic or a sequent calculus for classical logic is superior to a Hilbert style formulation.

The main defect of the traditional foundational view is related to computations. If these are to be captured by the equality in first order logic, then a full proof of an algebraic equation like:

$$x^3 - y^3 \quad = \quad (x - y)(x^2 + xy + y^2)$$

becomes unreasonably long, in fact quadratic in the size of the statement itself. This is caused by the fact that in a chain of equations each time a heavy use is made of the congruence properties ($+$ and $*$ preserve equality). Statements like:

$$\lfloor\sqrt{1100}\rfloor \quad = \quad 33$$

$$\texttt{Prime}(1999)$$

are also extremely awkward to be proved in arithmetic as formalized in predicate logic with equation.

### 1.3.1 Flavors

In this section we will briefly present a few lambda calculi. For the reader who wants a full description of the systems and their properties, we will give some references.

#### Lambda Calculus

Because of the emphasis we lay on proof checkers which actually produce proof objects, the class of proof checkers we will study is based on the typed lambda calculus. The typed lambda calculus stems from the (untyped) lambda calculus originally introduced by (Church 1932,1933). The lambda calculus is a general theory of functions and logic, intended as a foundation of mathematics. The reader is referred to (Barendregt 1984) for an in-depth treatment of the lambda calculus.

#### Typed Lambda Calculus

The typed lambda calculus is introduced by (Curry 1934, Church 1940). Essentially, every lambda term has a type associated. In fact, only terms are allowed which can be typed. Typed lambda calculi are useful for several reasons. Firstly, types can be used to specify an algorithm. Then an inhabitant of a type, that is, a term which has the type, is an implementation of the algorithm. In a similar way, types can be seen as propositions. In that case inhabitants are proofs, and inhabited types are true propositions. Furthermore, types can be used to make compilations of terms more efficient.

Examples of typed lambda calculi are system F and the Calculus of Constructions (CC). The latter calculus is invented by (Coquand and Huet 1985, Coquand and Huet 1988) and is a lambda calculus with dependent types. In (Coquand 1985) it is shown that the system CC is consistent. In (Girard, Lafont and Taylor 1989), the reader finds a treatment of system F and the connection with (constructive) logic.

In (Barendregt 1992), various systems of typed lambda calculi are presented in a general framework of the form of a cube, the so-called $\lambda$-*cube*. These eight systems differ in expressiveness and proof strength. On one extreme we have simply typed lambda calculus $\lambda{\rightarrow}$, which can be viewed as a minimal propositional logic with implication alone. Then we have the polymorphic lambda calculus $\lambda 2$ which is essentially system $F$. A stronger system is $\lambda P2$ in which we also have predicates. On the other end of the cube we have the system called $\lambda$C which has much in common with higher-order predicate logic. The $\lambda$-cube is generalized into so-called Pure Type Systems (PTS's) independently by (Berardi 1989) and by (Terlouw 1989).

### Sigma types and type hierarchies

A weakness of CC is that it is not possible to directly form the product type $A \times B$, given two *types* $A$ and $B$. In (Luo 1990) a system called the Extended Calculus of Constructions (ECC) is presented. Essentially, it integrates the (impredicative) Calculus of Constructions and Martin-Löf's (predicative) type theory with universes (Martin-Löf 1972, Martin-Löf 1984). In ECC, product types are introduced by so-called $\Sigma$-types. Luo showed strong normalization for ECC.

### Inductive types

Another extension of the Calculus of Constructions are inductive types. In CC it is possible to define notions with a recursive nature like the natural numbers. The trick is to encode such a type in terms of a higher order impredicative definition. Inductive types enable us to formalize recursive definitions in a much more direct way. For example, to define the natural numbers, we write

$$\mu X[X, X \to X] \ ,$$

which stands for the smallest type consisting of one inhabitant and a function which transforms an inhabitant into another one. In fact, we use the following formalization:

$$\mathbb{N} \quad \equiv \quad \mu X : *.(0 : X, S^+ : X \to X)$$

So in this way, the natural numbers are defined as the smallest set $\mathbb{N}$ of type $*$, which consist of two constructors: a constant named 0 and unary function named $S^+$. Because $\mathbb{N}$ is defined as the *smallest* type, we obtain an induction principle $\varepsilon_{\mathbb{N}}$:

$$\varepsilon_{\mathbb{N}} \quad : \quad \forall \phi{:}\mathbb{N}{\to}*.\,(\phi\, 0) \to (\forall n{:}\mathbb{N}.\,(\phi\, n){\to}(\phi(S^+\, n))) \to (\forall n{:}\mathbb{N}.\,\phi\, n) \ ,$$

together with two accompanying rewriting rules:

$$\begin{array}{lll} \varepsilon_{\mathbb{N}}\, \phi\, f\, g\, 0 & \to_\iota & f \\ \varepsilon_{\mathbb{N}}\, \phi\, f\, g\, (S^+ n) & \to_\iota & g\, n\, (\varepsilon_{\mathbb{N}}\, \phi\, f\, g\, n) \end{array}$$

where $\phi : \mathbb{N} \rightarrow *$, $f : \phi\, 0$, $g : \forall n{:}\mathbb{N}.\,(\phi\, n) \rightarrow (\phi(S^+\, n))$, and $n : \mathbb{N}$. Note that we use the notion of $\iota$-reduction for rewriting inductive terms. We say that we have defined the natural numbers *inductively*. Now it is possible to define other terms *by recursion*. For example, we can define the addition by recursion on it's second argument:

$$\texttt{add} \quad \equiv \quad \lambda x{:}\mathbb{N}.\, \varepsilon_{\mathbb{N}}\, (\lambda y{:}\mathbb{N}.\, \mathbb{N})\, 0\, (\lambda y{:}\mathbb{N}\, \lambda h{:}\mathbb{N}.\, S^+ h)$$

Note that it is also possible to use inductive types instead of sigma types to define the product $A \times B$ of two given types $A$ and $B$.

$$\texttt{prod} \quad \equiv \quad \lambda A{:}*\, \lambda B{:}*\, \mu X{:}*.\, (\texttt{tuple} : A \rightarrow B \rightarrow X)$$

### Term rewriting

Besides $\beta$- and $\iota$-reduction, we also use in a few very specific cases *$\rho$-reduction*. $\rho$-reduction allows us to add arbitrary rewriting rules. So this is potentially very dangerous, as we could add a rule which rewrites true into falsum. But used with great care, in some circumstances it will be extremely useful. In Chapter 5 we implement an algorithm to automatically verify equations of a class of algebraic structures. This is done by carefully adding the rewrite-rules of the Knuth-Bendix completion of the algebraic structure.

### Delta reduction

As mentioned before, we wish to have definitions as part of our type theory. Then we are able to attach a name to a term. Definitions are used both for defining mathematical objects, as for naming lemmas and theorems. The mechanism used to unfold a definition is called $\delta$-reduction. The type checker we use is able to automatically unfold definitions when necessary.

### Summarizing

So we end up using a type system with the following properties:

- Calculus of constructions for higher order predicate logic.
- Conversion ($\beta$-reduction) for computations.
- $\delta$-reduction for definitions.
- Inductive types (via $\iota$-reduction) also for definitions and computations.
- $\rho$-reduction which enables us to add rewriting rules.
- Cumulative type hierarchies and sigma types.

The type checker which implements all these features is LEGO (Luo and Pollack 1992). LEGO is designed and implemented by Randy Pollack. It is coded in New Jersey Standard ML. LEGO is freely available from the Internet. It is remarkable that LEGO also implements another concept, namely argument synthesis. Argument synthesis is the ability of the type checker to fill in certain types in a term we left out.

## 1.4 Description

This thesis is divided into four parts. The first chapter is about logic. We will present different kinds of logic, and the connection with type theory.

Next we will develop a small library of mathematical concepts, like sets, functions, structures and some analysis. The main emphasis is on constructive definitions, although we define the real number system with a decidable equality relation.

The library is used in Chapter 4 where we present four case studies. The first one is to make clear how logical reasoning is done in an interactive way on a proof checker. Another case study is used to show the difficulties of equational reasoning. The last case study is the ambitious goal to fully formalize the Fundamental Theorem of Algebra. Its purpose is to get an understanding of the question whether it is possible to formalize large bodies of mathematical texts into type theory. As we will see, it turned out that the formalization could not be completed within reasonable time.

We spend a separate chapter to study equational reasoning. We propose a method which enables us to use the type checker to automatically verify equational reasoning.

# Chapter 2

# Representing Logic

Various logical systems may differ in their expressive power and in their proof-theoretic strength. The simplest logical system is first order minimal proposition logic. It does not possess predicates, nor connectives besides the implication alone, nor any quantifications. At the other end of the spectrum we have higher order predicate logic. The system that we will use in the following chapters is found in between. We will use *second order* logic with universal quantification and implication alone. The other logical connectives are defined impredicatively. Furthermore, in order to be able to express mathematical notions, we will also permit *predicates*. So we will work in second order predicate logic. Also we add inductive types. These are quite convenient for defining mathematical objects with a recursive structure.

In this chapter we will first give a brief overview of the history of logic. It is by no means complete and is only intended to give a context to the logical system we use. The reader is referred to (van Dalen 1978) for a more in-depth treatment of this topic. Next, we show how we can formalize these systems into type theory.

## 2.1 Relevant views on logic

The first works on logic are by Aristotle who wrote his *Organon* in the fourth century B.C. Only in 1854, G. Boole introduced the first logical system (Boole 1854). Then it was Frege who gave the first formal description of mathematical logic. In his *Begriffschrift* (Frege 1879), he presented a formal system for first order logic. Hilbert used what is called the axiomatic method (Hilbert and Bernays 1934, 1939). Brouwer revised the idea of provability in his intuitionistic logic. Following ideas attributed to R. Pollack, we can view logic from different angles.

#### Formalistic approach

We call a system formal if we can give a precise definition of its syntax and the derivation rules. Examples of formal systems are Peano arithmetic and predicate logic. Also Hilbert's *Grundlagen der Geometrie* (1899) is a fine example of the formalistic approach. In this work, he introduces geometric objects like points as given, without giving it any interpretation. Then he formulates the properties that these objects should obey. Formalists wish to axiomatize all of mathematics.

**Logicistic approach**

The main idea of the logicism is to identify mathematics and logic. Frege defined mathematical concepts in terms of logic. To do so, he gave a very precise description of logic and enriched the power of expression. In (Frege 1879), propositional logic is based on a abstract notion of truth values.

Mathematical logic as founded by Frege was elaborated by Whitehead and Russell. Their *Principia Mathematica* (Whitehead and Russell 1910–1913) can be viewed as the culmination of logicism.

**Intuitionistic approach**

The Dutch mathematician Brouwer approached logic from a completely new point of view (Brouwer 1907). He rejected the principle of boolean values which assigns to every sentence a truth value. In Brouwer's view, mathematical objects are created by mental constructions. Intuitionists believe that logic rests on mathematics. Where classical logic is descriptive by nature, constructive logic focuses attention on the dynamic interaction of the individual with the mathematical universe (Mines, Richman and Ruitenburg 1988, Chapter 1, Section 1). To prove a theorem, we have to give a construction of a proof.

In order to reason, intuitionists gave a new interpretation to the logical connectives. Let $P$ and $Q$ be propositions, $A$ a set and $\phi(x)$ a predicate.

- A proof of $P \mathbin{\&} Q$ consists of a proof of $P$ and a proof of $Q$, just as in classical mathematics.
- To prove $P \vee Q$, we must either prove $P$ or prove $Q$, whereas in classical logic it would be possible to prove $P \vee Q$ without proving $P$ or $Q$.
- A proof of $P \to Q$ consists of a construction which transforms any proof of $P$ into a proof of $Q$.
- The negation of $P$ is defined as $P \to \perp$, where $\perp$ is some contradiction. So to prove $\neg P$, we have to show how to transform a proof of $P$ into a proof of *absurdum*.
- A proof of $\forall x{:}A.\, \phi(x)$ consists of a construction that assigns to every $a : A$ a proof of $\phi(a)$.
- A proof of $\exists x{:}A.\, \phi(x)$ consists of a construction which gives a $a : A$ and a proof of $\phi(a)$.

The inductively defined logical connectives presented in Definition 2.2.3 fit perfectly well in the view-point of logic being part of mathematics.

## 2.2 Logic formalized

Let us switch focus and investigate how we can formalize the various logical paradigms in type theory.

## Axiomatic formalization

Following the formalistic tradition, we encode propositional logic axiomatically. Note that most logical connectives come with introduction and elimination rules.

2.2.1. Definition. (i) We introduce a type of propositions $\texttt{prop}$, and a type function $\texttt{T}$ assigning to each proposition the type of its proofs.

$$\begin{array}{lcl} \texttt{prop} & : & * \\ \texttt{T} & : & \texttt{prop} \to * \end{array}$$

(ii) We axiomatize the implication connective together with proof constructors for introduction and elimination as follows.

$$\begin{array}{lcl} \texttt{imp} & : & \texttt{prop} \to \texttt{prop} \to \texttt{prop} \\ \texttt{imp}_e & : & \Pi\alpha,\beta{:}\texttt{prop}.\,(\texttt{T}(\texttt{imp}\,\alpha\,\beta)) \to (\texttt{T}\alpha) \to (\texttt{T}\beta) \\ \texttt{imp}_i & : & \Pi\alpha,\beta{:}\texttt{prop}.\,((\texttt{T}\alpha) \to (\texttt{T}\beta)) \to \texttt{T}(\texttt{imp}\,\alpha\,\beta) \end{array}$$

(iii) In a similar way we introduce falsum and the negation.

$$\begin{array}{lcl} \texttt{fls} & : & \texttt{prop} \\ \texttt{fls}_e & : & (\texttt{T}\,\texttt{fls}) \to \Pi\alpha{:}\texttt{prop}.\,\texttt{T}\alpha \\ \texttt{not} & : & \texttt{prop} \to \texttt{prop} \\ & \equiv & \lambda\alpha{:}\texttt{prop}.\,\texttt{imp}\,\alpha\,\texttt{fls} \\ \texttt{not}_{DN} & : & \Pi\alpha : \texttt{prop}.(\texttt{T}(\texttt{not}\,(\texttt{not}\,\alpha)) \to \texttt{T}\alpha \end{array}$$

In a similar way, all the other connectives can be introduced axiomatically. The first Automath Translation AUT-68 (van Benthem Jutting 1994) used a system very similar to this one to encode classical minimal predicate logic.

Suppose we work in the type system $\lambda P2$ (one of the eight systems of the $\lambda$-cube, see Section 1.3.1). If we substitute the type $*$ for $\texttt{prop}$, and the identity for $T$, we have formalized second order predicate logic in a more straightforward way. Then we also should replace $(\texttt{imp}\,A\,B)$ by $(A{\to}B)$. This idea is called the *propositions-as-types* interpretation of (de Bruijn 1970, Howard 1980). This interpretation was used for the AUTO-QE system. See (Barendregt 1992) for more details on the propositions-as-types interpretation.

## Impredicative formalization

In the logicistic spirit we define in type theory the logical connectives $\neg$, &, $\vee$ and $\exists$ by second order definitions.

2.2.2. Definition. Define the logical connectives impredicatively.

$$\begin{array}{lcl}
\bot & : & * \\
 & \equiv & \Pi X{:}*.\, X \\
\neg & : & * \to * \\
 & \equiv & \lambda\alpha{:}*.\, \alpha \to \bot \\
\_ \vee \_ & : & * \to * \to * \\
 & \equiv & \lambda\alpha{:}*\, \lambda\beta{:}*\, \Pi X{:}*.\, (\alpha{\to}X){\to}(\beta{\to}X){\to}X \\
\_ \mathbin{\&} \_ & : & * \to * \to * \\
 & \equiv & \lambda\alpha{:}*\, \lambda\beta{:}*\, \Pi X{:}*.\, (\alpha{\to}\beta{\to}X){\to}X \\
\exists & : & \Pi T{:}*.\, (T{\to}*) \to * \\
 & \equiv & \lambda T{:}*\, \lambda\phi{:}T{\to}*\, \Pi X{:}*.\, (\Pi t{:}T.\, (\phi\, t){\to}X){\to}X
\end{array}$$

Other logical connectives can be defined in terms of the previous ones. For example logical equivalence can be defined as follows.

$$\begin{array}{lcl}
\mathtt{iff} & : & * \to * \to * \\
 & \equiv & \lambda\alpha, \beta{:}*.\, (\alpha{\to}\beta) \mathbin{\&} (\beta{\to}\alpha)
\end{array}$$

By definition, we get the introduction rules for the impredicatively defined connectives for free. The elimination rules are easily provable. Remark that if we use a type system with type hierarchies, we may raise the type of $T$ in the definition of $\exists$ from $*$ to $\Box$, or even higher.

### Propositions-as-Types

One of the advantages of the propositions-as-types paradigm is that it follows closely the intuitionistic tradition. Suppose we can prove in a logic $L$ the claim that $A$ follows from the assumptions $\Gamma$. Formally, we prove this by showing that

$$\Gamma \quad \vdash_L \quad A \ .$$

A formula $A$ in $L$ corresponds with a term $[A]$ of type $*$ in type theory. We call an inhabitant $p$ of $[A]$ a proof object. This proof object is a faithful encoding of a proof for $A$. So,

$$[\Gamma] \quad \vdash_\lambda \quad p : [A] \ .$$

When we extend our type theory with inductive types we do not need the second order logic anymore to define the logical connectives. Some logicians consider second order logic as non-constructive. There seems to be less resistance against inductive types. This is because inductive types are constructed from base elements. On the other hand, some stronger versions of the elimination principle of inductive types are more or less disputed.

2.2.3. Definition. Define the logical connectives inductively in first order minimal predicate logic with inductive types.

$$\begin{array}{lcl}
\bot & : & * \\
 & \equiv & \mu X{:}*.\,() \\
\neg & : & * \to * \\
 & \equiv & \lambda\alpha{:}*.\,\alpha \to \bot \\
\_ \vee \_ & : & * \to * \to * \\
 & \equiv & \lambda\alpha{:}*\,\lambda\beta{:}*\,\mu X{:}*.\,(\texttt{inl} : \alpha{\to}X, \texttt{inr} : \beta{\to}X) \\
\_ \,\&\, \_ & : & * \to * \to * \\
 & \equiv & \lambda\alpha{:}*\,\lambda\beta{:}*\,\mu X{:}*.\,(\texttt{pair} : \alpha{\to}\beta{\to}X) \\
\exists & : & \Pi T{:}*.\,(T{\to}*) \to * \\
 & \equiv & \lambda T{:}*\,\lambda\phi{:}T{\to}*\,\mu X{:}*.\,(\texttt{exintro} : \Pi t{:}T.\,(\phi\, t){\to}X)
\end{array}$$

As opposed to the impredicatively defined connectives, we get the elimination rule for free. This rule is namely precisely the elimination principle of the corresponding inductive type. The introduction rules we get for free as well from the constructors.

In second order predicate logic with inductive types, the characterization of the connectives defined in definitions 2.2.2 and 2.2.3 are provably equivalent. Remark that for reasons of consistency, the inductive definition of $\exists$ does not permit $T$ to be quantified over types higher than $*$. In some cases this can be rather inconvenient. This restriction does not hold for the impredicative version of $\exists$.

We summarize the different systems in the following table.

| | 1 Aut | Aut | LEGO | $*$ | Coq |
|---|---|---|---|---|---|
| $\to$, $\forall$ | Encoded | Propositions-as-types | Propositions-as-types | Propositions-as-types | Propositions-as-types |
| $\neg$, & | Encoded | Encoded | Second Order | Second Order | Inductive |
| $\vee$ | Encoded | Encoded | Second Order | Inductive | Inductive |
| $\exists$ | Encoded | Encoded | Second Order | Second Order | Inductive |

The column '1 Aut' stands for the first automath translation, which is purely an encoded logic. The other Automath systems (the so called second Automath translations, labeled as 'Aut') makes use of the propositions-as-types isomorphism. The type checker LEGO uses an impredicatively defined logic ('Second Order'). Our system, labeled '$*$', is similar except for the or-connective. As we will see in Section 3.3.4, we make use of both the inductive as the impredicative defined or-connective. The type checker Coq (Dowek et al. 1993) is purely based on inductively defined connectives.

## Classical logic

Of course, the law of excluded middle does not hold in intuitionistic logic. If we really need this law in type theory, we have to extend the context with a constant of one of the following types.

2.2.4. Definition. Define the laws of excluded middle and double negation.

$$\begin{array}{lcl}
\mathit{EM} & : & * \\
 & \equiv & \Pi\alpha{:}*.\,\alpha \vee \neg\alpha \\
\mathit{DN} & : & * \\
 & \equiv & \Pi\alpha{:}*.\,(\neg(\neg\alpha)) \to \alpha
\end{array}$$

2.2.5. Lemma. *The laws of double negation and excluded middle are equivalent in intuitionistic logic.*

Proof. *Well-known.*

2.2.6. Definition. We can encode classical propositional logic using inductive types as follows. For an explanation of the notation $\varepsilon_{\texttt{bool}} \ldots \Longrightarrow$, see Appendix A.1.3.

$$\begin{array}{lcllcl}
\texttt{bool} & : & * & & & \\
 & \equiv & \mu X{:}*.\,(\texttt{false} : X, \texttt{true} : X) & & & \\
\texttt{istrue} & : & \texttt{bool} \to * & & & \\
 & \equiv & \varepsilon_{\texttt{bool}} & \texttt{false} & \Longrightarrow & \bot \\
 & & & \texttt{true} & \Longrightarrow & (\bot \to \bot) \\
\texttt{neg} & : & \texttt{bool} \to \texttt{bool} & & & \\
 & \equiv & \varepsilon_{\texttt{bool}} & \texttt{false} & \Longrightarrow & \texttt{true} \\
 & & & \texttt{true} & \Longrightarrow & \texttt{false} \\
\texttt{or} & : & \texttt{bool} \to \texttt{bool} \to \texttt{bool} & & & \\
 & \equiv & \varepsilon_{\texttt{bool}} & \texttt{false} & \Longrightarrow & \lambda x{:}\texttt{bool}.\, x \\
 & & & \texttt{true} & \Longrightarrow & \lambda x{:}\texttt{bool}.\, \texttt{true}
\end{array}$$

Note that as expected,

$$\Pi x{:}\texttt{bool}.\,(\texttt{istrue}\,(\texttt{neg}\,(\texttt{neg}\,x))) \to (\texttt{istrue}\,x)$$

is provable by induction on $x$.

Using the classical truth values `bool`, it becomes straightforward to define functions and predicates by cases. For let $T$ be a type, $h_1, h_2 : T \to T$ and $\phi : T \to \texttt{bool}$. If we wish to define a function $f$ such that

$$\begin{array}{rcll}
f(x) & = & h_1(x) & \text{if } \phi(x) \\
 & & h_2(x) & \text{otherwise} \ ,
\end{array}$$

we can do this as follows:

$$\begin{array}{lcllcl}
\texttt{f} & : & T \to T \to T & & & \\
 & \equiv & \lambda x{:}T.\,(\varepsilon_{\texttt{bool}} & \texttt{true} & \Longrightarrow & (h_1\,x) \\
 & & & \texttt{false} & \Longrightarrow & (h_2\,x))\,(\phi\,x) \ .
\end{array}$$

This is not possible for the impredicatively defined or-connective. In that case we obtain an elimination principle into propositions, not into types. However, in Section 3.3.4 we will show how we can strengthen the or-connective in order to achieve case distinction.

# Chapter 3

# Representing Mathematical Notions

To be able to develop mathematical proofs in a proof development system, we have to define the standard notions in type theory. Some notions are of set-theoretic nature, like sets, subsets and functions. Some are algebraic, like monoids, or fields. Others have to do with analysis, of which the natural numbers and reals are examples.

### 3.0.1 Validation contra verification

In general we see a trade-off between the complexity of definitions and the complexity of the proofs of properties concerning these definitions. If we choose compact and simple definitions, we often need to spend more effort in proving the desired propositions. But on the other hand, we could try to enhance our definitions in order to get simpler and more straightforward proofs. But then the definitions usually get more complex and less clear. For two reasons we have a strong preference for simple definitions as opposed to short proofs.

First, if we formalize mathematics, it is of great importance that all definitions are as clear and transparent as possible. We need to be sure that all formal definitions correspond precisely to the concepts and notions we have in mind: i.e. we have to *validate* the formal definitions. Gaining this insight is more delicate than formally *verifying* correctness of proofs. The latter can even be done by a machine. So we believe that we should focus on short and simple definitions, and accept that in some cases proofs may get more complex.

Second, theorem provers are primary designed for building proofs interactively in a more or less convenient way. In order to prove a proposition, the theorem prover helps us in finding an appropriate list of tactics. Using these tactics, the theorem prover produces the proof object which represents a proof of the proposition. On the other hand, these tactics were not designed for constructing *definitions* in an interactive way. Of course in type theory one could regard a definition as a proof of a proposition. So we could use the theorem prover to 'build' a definition, but this approach seems a bit awkward.

**Notation**

In this chapter, we will develop a notation for mathematics. This notation should be easily readable for humans. But it should be precise and formal in such a way that it can be canonically transformed into the syntax of the proof checker. We will call this notation *Pseudo LEGO* (see Appendix A.1.2).

Throughout this thesis, we will present formalizations of lemmas, definitions, and assumptions in a uniform manner. We have to be very careful to separate two levels of formality. Namely, we should distinguish between *informal mathematics* and mathematics formalized in *type theory.* The latter will generally be typeset in type writer font. Lemmas are introduced as follows.

LEMMA. Statement $S$ is true.

```
Name  :  S
      ≡  Proof
```

Note that `S` will often be of type `PROP`. If we are not interested in the actual proof-term `Proof` we leave it out and replace it by `⌞...⌟`. In that case the full proof with all the details can usually be found in the LEGO library (see Appendix A.3).

In type theory there is no essential distinction between lemmas and definitions. Both consist of a term, a corresponding type and a name attached to the term.

DEFINITION. Define *definiendum* as *definiens.*

```
definiendum  :  T
             ≡  definiens
```

Often `T` will be of type `SET`.

In some occasions we need to assume that a certain statement $S$ holds. We formalize this by a variable declaration which extends the current context of the type checker.

AXIOM. Assume $S$ is true.

```
x  :  S
```

As part of our pseudo LEGO, we allow the definition of lambda-terms as an infix operator. We indicate this by writing explicitly the place holders used. An example is to define the binary operator equality as _ = _.

## 3.1 Sets

When we want to represent mathematics the first question we encounter is how to deal with sets. As Bishop wrote, a *set* is defined by describing what must be done to construct an *element* of the set, and what must be done to show that two elements of the set are *equal* (Bishop 1967, chapter 3,paragraph 1). This leads us to the following approaches.

### 3.1.1 Sets as types

Terms live in types, and elements belong to a set. So it seems to be natural to define sets as types. The phrase ‘is an element of’ is formalized by ‘has type’. For equational reasoning we need an (polymorphic) equivalence relation over types. We will define Leibniz equality[1] for this purpose.

3.1.1. Definition. (i) We need the type of propositions, the type of sets, and a universe of types.

$$\begin{array}{lcl} \mathtt{PROP} & : & \mathtt{SET} \\ & \equiv & * \\ \mathtt{SET} & : & \mathtt{TYPE} \\ & \equiv & \Box_0 \\ \mathtt{TYPE} & \equiv & \Box_1 \end{array}$$

(ii) Define what it is for a relation to be an equivalence relation. Let $T{\mid}\mathtt{SET}$ be a type[2], $R{:}T{\to}T{\to}\mathtt{PROP}$ be a relation over $T$.

$$\begin{array}{lcl} \mathtt{reflexive} & : & \mathtt{PROP} \\ & \equiv & \Pi x{:}T.\, R\,x\,x \\ \mathtt{symmetric} & : & \mathtt{PROP} \\ & \equiv & \Pi x,y{:}T.\,(R\,x\,y){\to}(R\,y\,x) \\ \mathtt{transitive} & : & \mathtt{PROP} \\ & \equiv & \Pi x{\mid}T\,\Pi y{:}T\,\Pi z{\mid}T.\,(R\,x\,y){\to}(R\,y\,z){\to}(R\,x\,z) \\ \mathtt{equivalence} & : & \mathtt{PROP} \\ & \equiv & \mathtt{reflexive}\ \&\ \mathtt{symmetric}\ \&\ \mathtt{transitive} \end{array}$$

(iii) Define Leibniz equality impredicatively.

$$\begin{array}{lcl} \_ =_L \_ & : & \Pi A{\mid}\mathtt{SET}.\, A{\to}A{\to}\mathtt{PROP} \\ & \equiv & \lambda A{\mid}\mathtt{SET}\,\lambda x,y{:}A\,\Pi\phi{:}A{\to}\mathtt{PROP}.\,(\phi x){\to}(\phi y) \end{array}$$

(iv) Define the natural numbers by induction.

$$\begin{array}{lcl} \mathtt{nat} & : & \mathtt{SET} \\ & \equiv & \mu X{:}\mathtt{SET}.\,(\mathtt{zero} : X, \mathtt{S}^{\mathtt{N}} : X{\to}X) \end{array}$$

Note that in part ii and iii we make use of argument synthesis. This mechanism allows us to leave out the applicant in those cases when the type checker is able to reconstruct the term. So instead of writing '$=_L\ A\,x\,y$', we may drop $A$ and write '$=_L\ x\,y$'. In fact, pseudo LEGO even allows us to write '$x =_L y$'.

3.1.2. Lemma. *Obviously, Leibniz equality is an equivalence relation.*

$$\begin{array}{lcl} \mathtt{Refl_L} & : & \mathtt{reflexive} =_L \\ & \equiv & \llcorner\ldots\lrcorner \\ \mathtt{Sym_L} & : & \mathtt{symmetric} =_L \\ & \equiv & \llcorner\ldots\lrcorner \\ \mathtt{Trans_L} & : & \mathtt{transitive} =_L \\ & \equiv & \llcorner\ldots\lrcorner \end{array}$$

[1]Called after the German philosopher G.W. Leibniz (1646-1716) who investigated mathematical logic.

[2]See appendix A.1.1 for an explanation of the principle of argument synthesis and the $\mid$ notation.

At first sight it may seem better to define SET on the lowest type-level possible, namely as the type of propositions $*$. In order to enable the formation of the set of propositions and the set of predicates, we raise the type SET to $\Box_0$. Then we have that $*$ lives in SET. In Section 3.3.2 we will see that this choice is convenient.

As long as we deal with atomic constructions, Leibniz equality is fine. So for example Leibniz equality is suitable for equality over inductive types like the natural numbers nat. Also it is suitable when we have a set with no structure, like the axiomatically introduced real numbers (see Section 3.5.4). Another example is the formalization of process algebras. In general one uses only atomic or inductive types to define the datastructures when formalizing process algebra. For this reason (Sellink 1996) was able to make use of an inductively defined equality. A complication of an *intensional* equality like inductively defined equality or Leibniz equality is that when we use the *sets as types* approach, elements of sets may have structure because they are constructed from other building blocks. In many such cases we need a specific *extensional* equality which identifies more elements than Leibniz equality does. Leibniz equality is in a sense minimal, because it identifies those elements which have exactly the same behavior. This will become problematic in some cases, for example when we construct the set of functions as follows.

$$\begin{array}{lcl} \texttt{fun} & : & \texttt{SET}{\rightarrow}\texttt{SET}{\rightarrow}\texttt{SET} \\ & \equiv & \lambda A, B{:}\texttt{SET}.\, A{\rightarrow}B \\ \_ =_{\texttt{fun}} \_ & : & \Pi A, B{\mid}\texttt{Set}.\, (\texttt{fun}\, A\, B){\rightarrow}(\texttt{fun}\, A\, B){\rightarrow}\texttt{PROP} \\ & \equiv & \lambda A, B{\mid}\texttt{Set}\, \lambda f, g{:}\texttt{fun}\, A\, B\, \Pi x{:}A.\, (f x) =_L (g x) \end{array}$$

The intended equality is pointwise equality of function results. Then

$$\begin{array}{lcll} A & : & \texttt{Set} & \text{let } A \text{ be a set} \\ f & : & \texttt{fun}\, A\, A & \text{let } f \text{ be a function over } A \\ z & : & \Pi x{:}A.\, (f x) =_L x & \text{assume f is the identity function over } A \\ g & : & \texttt{fun}\, A\, A & \text{define } g \text{ as the identify function over } A \\ & \equiv & \lambda x{:}A.\, x\ . & \end{array}$$

Now $z$ is also a proof of the statement $f =_{\texttt{fun}} g$, but $f =_L g$ is not provable without extending the context with the axiom of extensionality:

$$\texttt{Ext} \ : \ \Pi A, B{\mid}\texttt{SET}\, \Pi f, g{:}\texttt{fun}\, A\, B.\, (f =_{\texttt{fun}} g){\rightarrow}(f =_L g) \ .$$

The formation of quotient sets is even more troublesome. For example, there is no straightforward way to define the set of integers as the quotient of $\mathbb{N} \times \mathbb{N}$ by the relation

$$(x_1, x_2) \sim (y_1, y_2) \quad \equiv \quad x_1 + y_2 =_L x_2 + y_1 \ .$$

Because the type theory we work in does not have quotient types, sets-as-types is not suitable to formalize quotient sets.

3.1.3. Remark. Polymorphic equality can also be defined inductively.

$$\begin{array}{lcl} \texttt{Eq}_{\texttt{ind}} & : & \Pi\alpha{:}\texttt{SET}.\, \alpha{\rightarrow}\alpha{\rightarrow}\texttt{PROP} \\ & \equiv & \lambda\alpha{:}\texttt{SET}\, \lambda x{:}\alpha\, \mu P{:}\alpha{\rightarrow}\texttt{PROP}.\, (\texttt{Refl}_{\alpha,x} : P\, x) \end{array}$$

It is easy to show that $\texttt{Eq}_{\texttt{ind}}$ is equivalent to Leibniz equality. Inductive equality has a stronger elimination principle, which may be convenient in some rare occasions.

### 3.1.2 Setoids

In order to solve the problem of a fixed equivalence relation, we could relativize all quantifications of sets $A$ by a binary relation $R$ over $A$ and by a proof that $R$ is an equivalence relation. So if we wish to express 'for every set $A$, $\phi(A)$', we can formalize this roughly as

$$\Pi A{:}\texttt{SET}\,\Pi R{:}A{\to}A{\to}\texttt{PROP}.\,(\texttt{equivalence}\ R) \to \text{`}\phi_R(A)\text{'}$$

where $\phi_R$ results from $\phi$ by replacing the equality by $R$. It will be much more natural (with respect to Bishop's definition of sets) to package the type $A$, the relation $R$ and a proof of $\texttt{equivalence}\,R$ into one single type. For this, we need sigma types.

3.1.4. Definition. Define the type of setoids, notation $\texttt{Set}$, as a type $T$, together with a binary relation $R$ over $T$ and a proof that $R$ is an equivalence relation.

$$\begin{array}{lcl}
\texttt{EqRel} & : & \texttt{SET}{\to}\texttt{SET} \\
 & \equiv & \lambda T{:}\texttt{SET}\,\Sigma R{:}T{\to}T{\to}\texttt{PROP}.\,\texttt{equivalence}\,R \\
\texttt{Set} & : & \texttt{TYPE} \\
 & \equiv & \Sigma T{:}\texttt{SET}.\,\texttt{EqRel}\,T
\end{array}$$

Now we model a set $A$ by $T/R$ for some type $T : \texttt{SET}$ and equivalence relation $R$ over $T$. The equivalence classes of $T/R$ represent the elements of $A$. This approach is called *sets as setoids*.

3.1.5. Definition. To extract the various components of a formal set, we define

$$\begin{array}{lcll}
\texttt{el} & : & \texttt{Set}{\to}\texttt{SET} & \text{`the elements of } A\text{'} \\
 & \equiv & \lambda A{:}\texttt{Set}.\,\pi^3_1(A) & \\
\_=\_ & : & \Pi A{\mid}\texttt{Set}.\,(\texttt{el}\,A){\to}(\texttt{el}\,A){\to}\texttt{PROP} & \\
 & \equiv & \lambda A{\mid}\texttt{Set}.\,\pi^3_2(A) & \\
=_{\texttt{refl}} & : & \texttt{reflexive}\,(={\mid}A) & \text{`the proof that } A \text{ is reflexive'} \\
 & \equiv & \lambda A{\mid}\texttt{Set}.\,\texttt{out}\,^3_1\pi^3_3(A) & \\
=_{\texttt{sym}} & : & \texttt{symmetric}\,(={\mid}A) & \text{`the proof that } A \text{ is symmetric'} \\
 & \equiv & \lambda A{\mid}\texttt{Set}.\,\texttt{out}\,^3_2\pi^3_3(A) & \\
=_{\texttt{trans}} & : & \texttt{transitive}\,(={\mid}A) & \text{`the proof that } A \text{ is transitive'} \\
 & \equiv & \lambda A{\mid}\texttt{Set}.\,\texttt{out}\,^3_3\pi^3_3(A) &
\end{array}$$

The term $\pi^n_i$ for $1 \leq i \leq n \in \mathbb{N}$ is introduced as pseudo LEGO in Appendix A.1.2.

Some examples:

$$\begin{array}{lcll}
A & : & \texttt{Set} & \text{`let } A \text{ be a set'} \\
x & : & \texttt{el}\,A & \text{`let } x \text{ be an element of } A\text{'}
\end{array}$$

For the sake of readability we sometimes write '$\forall x{:}T$' instead of '$\Pi x{:}T$'. Again, argument synthesis appears to be quite convenient for the definition $=$ of equality of a setoid. We do not need to explicitly specify the set to which the equality belongs.

When we construct a setoid, we generally do this in two stages. First we construct a *type* $T : \texttt{SET}$. Next we define a relation over $T$, show that it is an equivalence relation, and combine them into a *setoid* $\texttt{Set}$.

3.1.6. Definition. Define the empty set, a singleton set and the set of natural numbers.

(i) First on the level of types.

$$\begin{array}{lcl}\texttt{EmptySET} & : & \texttt{SET} \\ & \equiv & \mu X{:}\texttt{SET}.\,() \\ \texttt{UnitSET} & : & \texttt{SET} \\ & \equiv & \mu X{:}\texttt{SET}.\,(\texttt{star} : X)\end{array}$$

(ii) By adding Leibniz equality[3], we obtain setoids.

$$\begin{array}{lcl}\texttt{EmptySet} & : & \texttt{Set} \\ & \equiv & <\texttt{EmptySET}, =_{L}{}_{\texttt{EmptySET}}, \llcorner\ldots\lrcorner> \\ \texttt{UnitSet} & : & \texttt{Set} \\ & \equiv & <\texttt{UnitSET}, =_{L}{}_{\texttt{UnitSET}}, \llcorner\ldots\lrcorner> \\ \mathbb{N} & : & \texttt{Set} \\ & \equiv & <\texttt{nat}, =_{L}{}_{\texttt{nat}}, \llcorner\ldots\lrcorner>\end{array}$$

We define for any natural number $n$

$$\begin{array}{lcl}\underline{n} & : & \texttt{el}\,\mathbb{N} \\ & \equiv & (\texttt{S}^{\texttt{N}})^{n}(\texttt{zero})\end{array}$$

where $(\texttt{S}^{\texttt{N}})^{n}(\texttt{zero})$ stands for $n$ applications of `zero` with $\texttt{S}^{\texttt{N}}$.

3.1.7. Definition. Because we have PROP : SET we can also define the set of propositions as is done in Lindenbaum algebra's. Define $\Omega$ as the propositions modulo the if-and-only-if equivalence relation.

$$\begin{array}{lcl}\Omega & : & \texttt{Set} \\ & \equiv & <\texttt{PROP}, \texttt{iff}, \llcorner\ldots\lrcorner>\end{array}$$

By definition we have that $\texttt{el}\,\Omega =_{\beta\iota} \texttt{PROP}$ and for propositions $P, Q : \texttt{el}\,\Omega$

$$(P = Q) \iff (P \iff Q)\ .$$

Remark that for propositions $P, Q : \texttt{PROP}$, the term $P = Q$ is *not* typable. Namely, the argument mechanism of the type checker needs to find a setoid to which $P$ and $Q$ belong in order to find the underlying equality relation between $P$ and $Q$.

### 3.1.3 Partial equivalence relations

A third approach to model sets in type theory is to use *partial equivalence relations*. A relation is a partial equivalence relation (or a PER) if it is a symmetric and transitive relation.

Let $R$ be a PER over the type $T : \texttt{SET}$. The domain of $R$ is defined as precisely those $x : T$ for which $R\,x\,x$ holds. Then we have that $R$ is an equivalence relation on its domain.

[3]Instead of Leibniz equality we could use the inductively defined equality which is provably equivalent to Leibniz equality.

3.1.8. Definition. (i) Define the type PER of partial equivalence relations over a type $T$ : SET. Define PER-sets as a type $T$ together with a PER over $T$.

$$\begin{array}{lcl} \texttt{PER} & : & \texttt{SET}\to\texttt{SET} \\ & \equiv & \lambda T{:}\texttt{SET}\,\Sigma R{:}T\to T\to\texttt{PROP}.\,(\texttt{symmetric}\,R)\ \&\ (\texttt{transitive}\,R) \\ \texttt{Set}_p & : & \texttt{TYPE} \\ & \equiv & \Sigma T{:}\texttt{SET}.\,\texttt{PER}\,T \end{array}$$

(ii) Define terms to extract the various properties of a PER-set.

$$\begin{array}{lcl} \texttt{el}_p & : & \texttt{Set}_p\to\texttt{SET} \\ & \equiv & \lambda A{:}\texttt{Set}_p.\,\pi^3_1(A) \\ \_ =_p \_ & : & \Pi A{\mid}\texttt{Set}_p.\,(\texttt{el}_p\ A)\to(\texttt{el}_p\ A)\to\texttt{PROP} \\ & \equiv & \lambda A{\mid}\texttt{Set}_p.\,\pi^3_2(A) \end{array}$$

(iii) Define the domain of a PER-set.

$$\begin{array}{lcl} \texttt{dom}_p & : & \Pi A{:}\texttt{Set}_p.\,(\texttt{el}_p\ A)\to\texttt{PROP} \\ & \equiv & \lambda A{:}\texttt{Set}_p\,\lambda x{:}\texttt{el}_p\ A.\,x =_p x \end{array}$$

This way we have formalized the notion ‘element of’ by a term. So for a set $A$ : $\texttt{Set}_p$, we can express statements like ‘$x$ is an element of $A$ or it is not’.

$$\forall x : \texttt{el}_p\ A.(\texttt{dom}_p A\ x) \vee \neg(\texttt{dom}_p A\ x)$$

### 3.1.4 Product Sets and Vectors

One very basic mathematical notion we want to formalize is product sets. This can be done conveniently by either sigma types or an inductive definition. We choose to use sigma types because in the type checker we use, they provide a nicer denotation for pairing and projection and they can be type checked more efficiently.

3.1.9. Definition. Define the binary product type prod, an equivalence relation over prod, and the binary product set.

$$\begin{array}{lcl} \texttt{prod} & : & \texttt{SET}\to\texttt{SET}\to\texttt{SET} \\ & \equiv & \lambda S,T{:}\texttt{SET}.\,S\times T \\ \texttt{Eq}_{\texttt{prod}} & : & \Pi A,B{:}\texttt{Set}.\,(\texttt{prod}(\texttt{el}\,A)(\texttt{el}\,B))\to(\texttt{prod}(\texttt{el}\,A)(\texttt{el}\,B))\to\texttt{PROP} \\ & \equiv & \lambda A,B{:}\texttt{Set}\,\lambda p,q{:}\texttt{prod}(\texttt{el}\,A)(\texttt{el}\,B).\,(p.1 = q.1)\ \&\ (p.2 = q.2) \\ \texttt{Prod} & : & \texttt{Set}\to\texttt{Set}\to\texttt{Set} \\ & \equiv & \lambda A,B{:}\texttt{Set}.\,<\texttt{prod}(\texttt{el}\,A)(\texttt{el}\,B),\texttt{Eq}_{\texttt{prod}}\,A\,B,\llcorner\ldots\lrcorner> \end{array}$$

In the sequel, we will need to work with $n$-tuples for arbitrary natural numbers $n$. Therefore we need to define the type of $n$-tuples. Let $A_1,\ldots,A_n$ and $A$ be sets. We distinguish two cases, namely many-sorted $A_1\times\cdots\times A_n$ and single-sorted $A^n$ (also known as heterogeneous and homogeneous). The former case we call *$n$-ary products*, the latter *vectors*.

For the many-sorted case we need to define the type of sorts. This is done by a (finite) list of SETs, which we call SETS. Because SET lives in TYPE we need a list constructor of type TYPE→TYPE, which itself lives in BIGTYPE. We could avoid the use of BIGTYPE by defining SETS directly as an inductive type, but we prefer the current approach because it is more general.

3.1.10. Definition.

$$\begin{array}{lcl}
\mathtt{LIST} & : & \mathtt{TYPE}\to\mathtt{TYPE} \\
 & \equiv & \lambda T{:}\mathtt{TYPE}\,\mu X{:}\mathtt{TYPE}.\,(\mathtt{NIL}_T : X, \mathtt{CONS}_T : T\to X\to X) \\
\mathtt{SETS} & : & \mathtt{TYPE} \\
 & \equiv & \mathtt{LIST}\ \mathtt{SET}
\end{array}$$

For any type $A : \mathtt{SET}$ and $Bs : \mathtt{SETS}$, we will make use of the following abbreviations:

$$\begin{array}{rcl}
\mathtt{NIL}\,\mathtt{SET} & \equiv & \emptyset \\
\mathtt{CONS}\,\mathtt{SET}\,A\,Bs & \equiv & A\,\hat{}\,Bs
\end{array}$$

Now that we have the type of sorts, there are two equivalent ways to define $n$-ary products. Firstly, we can transform the list of sorts to an iterated application of the binary product. This is a definition by recursion on (the length of) the list SETS. Secondly, we can define $n$-ary products by an inductive type.

3.1.11. Definition. Define $n$-ary product in two ways. Firstly by iteration and secondly by an inductive type.

$$\begin{array}{lcl}
\mathtt{product} & : & \mathtt{SETS}\to\mathtt{SET} \\
 & \equiv & \varepsilon_{\mathtt{LIST}}\,\mathtt{SET}\ \emptyset \quad\quad \Longrightarrow\ \mathtt{UnitSET} \\
 & & \qquad\quad (T\,\hat{}\,Ts)\ \Longrightarrow\ \mathtt{prod}\,T\,(\mathtt{product}\ Ts) \\
\mathtt{product}_{\mathtt{ind}} & : & \mathtt{SETS}\to\mathtt{SET} \\
 & \equiv & \mu X{:}\mathtt{SETS}\to\mathtt{SET}.\,( \\
 & & \quad \mathtt{pnil} \ :\ X\,\emptyset, \\
 & & \quad \mathtt{pcons} \ :\ \Pi S{:}\mathtt{SET}\,\Pi l{:}\mathtt{SETS}.\,S\to(X\ l)\to X\ (S\,\hat{}\,l) \\
 & & )
\end{array}$$

An intuitive argument that product and $\mathtt{product}_{\mathtt{ind}}$ are equivalent is to look at the shape of their canonical inhabitants:

$$\begin{array}{lcl}
<s_1,\ldots,s_n,\mathtt{star}> & : & S_1\times\ldots\times S_n\times\mathtt{UnitSET} \\
 & & \quad =_\iota \mathtt{product}\,(S_1\,\hat{}\cdots\hat{}\,S_n\,\hat{}\,\emptyset) \\
\mathtt{pcons}\,s_1(\ldots(\mathtt{pcons}\,s_n\,\mathtt{pnil}))) & : & \mathtt{product}_{\mathtt{ind}}\,(S_1\,\hat{}\cdots\hat{}\,S_n\,\hat{}\,\emptyset)
\end{array}$$

Although the latter definition ($\mathtt{product}_{\mathtt{ind}}$) seems to be a bit more elegant, we choose to use the former one (product ). This is because proofs and definitions in which $n$-ary products occur are mostly done by induction on the set of sorts used.

Of course, vectors are just a special instance of $n$-ary products.

3.1.12. Definition. (i) Define a term which maps $A^n$ to $A\,\hat{}\cdots\hat{}\,A$ for any $A : \mathtt{SET}$, $n : \mathtt{nat}$.

$$\begin{array}{lcl}
\mathtt{ssorted} & : & \mathtt{SET}\to\mathtt{nat}\to\mathtt{SETS} \\
 & \equiv & \lambda T{:}\mathtt{SET}.\,\varepsilon_{\mathtt{nat}}\ \mathtt{zero} \quad \Longrightarrow\ \emptyset \\
 & & \qquad\qquad (\mathtt{S^N}\,n)\ \Longrightarrow\ T\,\hat{}(\mathtt{ssorted}\,T\,n)
\end{array}$$

(ii) Define the type of vectors in two ways. Firstly indirectly by using $n$-ary products and secondly by recursion on $n$.

$$\begin{array}{lcl}
\mathtt{vector_{prod}} & : & \mathtt{SET}{\rightarrow}\mathtt{nat}{\rightarrow}\mathtt{SET} \\
 & \equiv & \lambda T{:}\mathtt{SET}\,\lambda n{:}\mathtt{nat}.\,\mathtt{product}\,(\mathtt{ssorted}\,T\,n) \\[1ex]
\mathtt{vector} & : & \mathtt{SET}{\rightarrow}\mathtt{nat}{\rightarrow}\mathtt{SET} \\
 & \equiv & \lambda T{:}\mathtt{SET}.\,\varepsilon^2_{\mathtt{nat}}\ \begin{array}[t]{lcl} \mathtt{zero} & \Longrightarrow & \mathtt{UnitSET} \\ (\mathtt{S^N}\,\mathtt{zero}) & \Longrightarrow & T \\ (\mathtt{S^N}\,(\mathtt{S^N}\,n)) & \Longrightarrow & \mathtt{prod}\,T\,(\mathtt{vector}\,T\,(\mathtt{S^N}\,n)) \end{array}
\end{array}$$

Although the definition of vectors by means of $n$-ary products is a bit more abstract, we choose to use the second definition. The only reason is that it is a more low-level definition which involves half the number of $\iota$ reductions compared to the first definition. Also it has the very convenient property that $\mathtt{vector}\,T\,\underline{1} =_{\beta\iota} T$. In case we need to consider vectors as products, apply the next trivial lemma.

3.1.13. Lemma. $\forall T{:}\mathtt{SET}\,\forall n{:}\mathtt{nat}.\,(\mathtt{vector}\,T\,n){\rightarrow}(\mathtt{product}\,(\mathtt{ssorted}\,T\,n))$.

Proof. For a proof, see the LEGO library.

3.1.14. Definition. To complete the definitions for $n$-tuples, we extend them from types to setoids.

$$\begin{array}{lcl}
\mathtt{Sets} & : & \mathtt{TYPE} \\
 & \equiv & \mathtt{LIST\ Set} \\[1ex]
\mathtt{Product} & : & \mathtt{Sets}{\rightarrow}\mathtt{Set} \\
 & \equiv & \varepsilon_{\mathtt{LIST}}\,\mathtt{Set}\ \begin{array}[t]{lcl} (\mathtt{NIL\ Set}) & \Longrightarrow & \mathtt{UnitSet} \\ (\mathtt{CONS\ Set}\,A\,As) & \Longrightarrow & \mathtt{Prod}\,A\,(\mathtt{Product}\,As) \end{array}
\end{array}$$

Equality on $n$-ary products is defined using iterated conjunction of the equalities of the sorts.

3.1.15. Definition. (i) Define a term which maps $A^n$ to $A\widehat{\ }\cdots\widehat{\ }A$ for any setoid $A:\mathtt{Set}$ and natural number $n:\mathtt{nat}$.

$$\begin{array}{lcl}
\mathtt{SSorted} & : & \mathtt{Set}{\rightarrow}\mathtt{nat}{\rightarrow}\mathtt{Sets} \\
 & \equiv & \lambda A{:}\mathtt{Set}.\,\varepsilon_{\mathtt{nat}}\ \begin{array}[t]{lcl} \mathtt{zero} & \Longrightarrow & \mathtt{NIL\,Set} \\ (\mathtt{S^N}\,n) & \Longrightarrow & \mathtt{CONS}\,?\,A\,(\mathtt{SSorted}\,A\,n) \end{array}
\end{array}$$

(ii) Define the type of vectors by recursion on $n$.

$$\begin{array}{lcl}
\mathtt{Vector_{ind}} & : & \mathtt{Set}{\rightarrow}\mathtt{nat}{\rightarrow}\mathtt{Set} \\
 & \equiv & \lambda A{:}\mathtt{Set}.\,\varepsilon_{\mathtt{nat}}\ \begin{array}[t]{lcl} \mathtt{zero} & \Longrightarrow & \mathtt{UnitSet} \\ (\mathtt{S^N}\,n) & \Longrightarrow & \mathtt{Prod}\,A\,(\mathtt{Vector_{ind}}\,A\,n) \end{array}
\end{array}$$

(iii) Define the vector set.

$$\begin{array}{lcl}
\mathtt{Eq_{vector}} & : & \Pi A{:}\mathtt{Set}\,\Pi n{:}\mathtt{nat}.\,(\mathtt{vector}\,(\mathtt{el}\,A)\,n){\rightarrow}(\mathtt{vector}\,(\mathtt{el}\,A)\,n){\rightarrow}\mathtt{PROP} \\
 & \equiv & \lambda A{:}\mathtt{Set}.\,\varepsilon^2_{\mathtt{nat}}\ \begin{array}[t]{lcl} \mathtt{zero} & \Longrightarrow & =_{|}\mathtt{UnitSet} \\ (\mathtt{S^N}\,\mathtt{zero}) & \Longrightarrow & =_{|}A \\ (\mathtt{S^N}\,(\mathtt{S^N}\,n)) & \Longrightarrow & \\ \multicolumn{3}{l}{\quad\lambda v,w{:}\mathtt{vector}\,(\mathtt{el}\,A)\,(\mathtt{S^N}\,(\mathtt{S^N}\,n)).} \\ \multicolumn{3}{l}{\qquad(v.1 = w.1)\ \&\ (\mathtt{Eq_{vector}}\,?\,?\,v.2\,w.2)} \end{array} \\
\mathtt{Vector} & : & \mathtt{Set}{\rightarrow}\mathtt{nat}{\rightarrow}\mathtt{Set} \\
 & \equiv & \lambda A{:}\mathtt{Set}\,\lambda n{:}\mathtt{nat}.\,{<}\mathtt{vector}\,(\mathtt{el}\,A)\,n, \mathtt{Eq_{vector}}\,A\,n, \llcorner\ldots\lrcorner{>}
\end{array}$$

Although the definition $\texttt{Vector}_{\texttt{ind}}$ is much more straightforward then $\texttt{Vector}$, we still choose the latter one. Again this is just because of efficiency of reduction. $\texttt{Vector}$ gives us quick access to its components without going into recursion. So for each $n$, $\texttt{el}\,(\texttt{Vector}\,A\,\underline{n})$ converts in a single step to $\texttt{vector}\,(\texttt{el}\,A)\,\underline{n}$, where $\texttt{el}\,(\texttt{Vector}_{\texttt{ind}}\,A\,\underline{n})$ needs order $n$ steps.

### 3.1.5 Disjoint union

Disjoint sums and products are just dual structures, so all our definitions for products can trivially be converted to sums.

3.1.16. Definition. (i) Define the (binary) disjoint sum type inductively.

$$\begin{array}{lcl} \texttt{sum} & : & \texttt{SET}{\to}\texttt{SET}{\to}\texttt{SET} \\ & \equiv & \lambda S,T{:}\texttt{SET}\,\mu X{:}\texttt{SET}.\,(\texttt{inL}_{S,T} : S{\to}X, \texttt{inR}_{S,T} : T{\to}X) \end{array}$$

(ii) Define the setoid $\texttt{Sum}$ by adding an appropriate equivalence relation to $\texttt{sum}$.

$$\begin{array}{lcl} \texttt{Eq}_{\texttt{sum}} & : & \Pi A,B{:}\texttt{Set}.\,(\texttt{sum}\,A.\texttt{el}\;B.\texttt{el}\,){\to}(\texttt{sum}\,A.\texttt{el}\;B.\texttt{el}\,){\to}\texttt{PROP} \\ & \equiv & \lambda A,B{:}\texttt{Set}.\,\varepsilon^2_{\texttt{sum}}A.\texttt{el}\,B.\texttt{el} \\ & & \quad (\texttt{inL}\,A.\texttt{el}\,B.\texttt{el}\,x_1),(\texttt{inL}\,A.\texttt{el}\,B.\texttt{el}\,x_2) \;\Longrightarrow\; x_1 = x_2 \\ & & \quad (\texttt{inL}\,A.\texttt{el}\,B.\texttt{el}\,x_1),(\texttt{inR}\,A.\texttt{el}\,B.\texttt{el}\,y_2) \;\Longrightarrow\; \texttt{false} \\ & & \quad (\texttt{inR}\,A.\texttt{el}\,B.\texttt{el}\,y_1),(\texttt{inL}\,A.\texttt{el}\,B.\texttt{el}\,x_2) \;\Longrightarrow\; \texttt{false} \\ & & \quad (\texttt{inR}\,A.\texttt{el}\,B.\texttt{el}\,y_1),(\texttt{inR}\,A.\texttt{el}\,B.\texttt{el}\,y_2) \;\Longrightarrow\; y_1 = y_2 \\ \texttt{Sum} & : & \texttt{Set}{\to}\texttt{Set}{\to}\texttt{Set} \\ & \equiv & \lambda A,B{:}\texttt{Set}.\,{<}\texttt{sum}\,A.\texttt{el}\;B.\texttt{el}\,,\texttt{Eq}_{\texttt{sum}}\,A\,B,\llcorner\ldots\lrcorner{>} \end{array}$$

3.1.17. Definition. Define the $n$-ary disjoint type and set by recursion on $n$.

$$\begin{array}{lcl} \texttt{sums} & : & \texttt{SETS}{\to}\texttt{SET} \\ & \equiv & \varepsilon_{\texttt{LIST}}\,\texttt{SET}\;\; \emptyset \;\Longrightarrow\; \texttt{EmptySET} \\ & & \qquad\qquad (S\,\widehat{}\,Ss) \;\Longrightarrow\; \texttt{sum}\,S\,(\texttt{sums}\,Ss) \\ \texttt{Sums} & : & \texttt{Sets}{\to}\texttt{Set} \\ & \equiv & \varepsilon_{\texttt{LIST}}\,\texttt{Set}\;\;(\texttt{NIL}\,\texttt{Set}) \;\Longrightarrow\; \texttt{EmptySet} \\ & & \qquad\qquad (\texttt{CONS}\,\texttt{Set}\,A\,As) \;\Longrightarrow\; \texttt{Sum}\,S\,(\texttt{Sums}\,Ss) \end{array}$$

3.1.18. Definition. In Definition 3.4.6 we need a set of exactly two elements.

(i) Using the disjoint union it becomes straightforward to define a canonical set of precisely two elements.

$$\begin{array}{lcl} \texttt{TwoSet}_{sum} & : & \texttt{Set} \\ & \equiv & \texttt{Sum}\;\texttt{UnitSet}\,\texttt{UnitSet} \end{array}$$

(ii) We can also define a canonical set of two elements by an inductive definition.

$$\begin{array}{lcl} \texttt{TwoSET} & : & \texttt{SET} \\ & \equiv & \mu X{:}\texttt{SET}.\,(\texttt{unit}^2_1, \texttt{unit}^2_2 : X) \end{array}$$

(iii) Extend the definition to a setoid.

$$\begin{array}{lcllcl}
\mathtt{Eq}_{\mathtt{TwoSET}} & : & \multicolumn{4}{l}{\mathtt{TwoSET}\to\mathtt{TwoSET}\to\mathtt{PROP}} \\
 & \equiv & \varepsilon^2_{\mathtt{TwoSET}} & \mathtt{unit}^2_1, \mathtt{unit}^2_1 & \Longrightarrow & \mathtt{true} \\
 & & & \mathtt{unit}^2_1, \mathtt{unit}^2_2 & \Longrightarrow & \mathtt{false} \\
 & & & \mathtt{unit}^2_2, \mathtt{unit}^2_1 & \Longrightarrow & \mathtt{false} \\
 & & & \mathtt{unit}^2_2, \mathtt{unit}^2_2 & \Longrightarrow & \mathtt{true} \\
\mathtt{TwoSet} & : & \multicolumn{4}{l}{\mathtt{Set}} \\
 & \equiv & \multicolumn{4}{l}{<\mathtt{TwoSET}, \mathtt{Eq}_{\mathtt{TwoSET}}, \llcorner\ldots\lrcorner>}
\end{array}$$

The same scheme can be used to define sets of three or more elements. In those cases it is more convenient to define them as inductive types with three, four or more constructors as opposed to defining them by iteration of `Sum UnitSet`. This is because the elimination principle $\varepsilon_n$ of the set of $n$ elements gives us immediately all $n$ sub cases, whereas $\varepsilon_{\mathtt{sum}}$ gives us only two sub cases.

## 3.2 Functions

The next notion we define is functions on sets. We model these as type theoretic functions that respect the equality relation. However, this way some particular functions cannot be defined in our calculus. Therefore we present a second definition based on functions as graphs, and a third approach using case distinction.

### 3.2.1 Function space

Let the sets $A$ and $B$ be given. To construct a function from $A$ to $B$, we construct a $\lambda$-term $f$ of type $(\mathtt{el}\,A) \to (\mathtt{el}\,B)$. Also we have to supply a proof that this term $f$ preserves the equality from $A$ to $B$.

3.2.1. Definition. (i) Define the type of unary functions.

$$\begin{array}{lcl}
\mathtt{extensional} & : & \Pi A, B{\mid}\mathtt{Set}.\,((\mathtt{el}\,A)\to(\mathtt{el}\,B))\to\mathtt{PROP} \\
 & \equiv & \lambda A, B{\mid}\mathtt{Set}\,\lambda g{:}(\mathtt{el}\,A)\to(\mathtt{el}\,B) \\
 & & \quad \forall x, y{:}\mathtt{el}\,A.\,(x = y)\to(gx) = (gy) \\
\mathtt{Fun} & : & \mathtt{Set}\to\mathtt{Set}\to\mathtt{SET} \\
 & \equiv & \lambda A, B{:}\mathtt{Set}\,\Sigma f{:}(\mathtt{el}\,A)\to(\mathtt{el}\,B).\,\mathtt{extensional}\,f
\end{array}$$

(ii) Let $A, B{\mid}\mathtt{Set}$ be setoids and $f{:}\mathtt{Fun}\,A\,B$ a function from $A$ to $B$. Define a term `ap` which extracts the type theoretic function from $f$. Also define a term `exten` which gives a proof that $f$ preserves the equality of $A$ to $B$.

$$\begin{array}{lcl}
\mathtt{ap} & : & (\mathtt{el}\,A)\to(\mathtt{el}\,B) \\
 & \equiv & \pi^2_1(f) \\
\mathtt{exten} & : & \mathtt{extensional}\,(\mathtt{ap}\,f) \\
 & \equiv & \pi^2_2(f)
\end{array}$$

(iii) Extend the type of unary functions to a setoid.

$$\begin{array}{lcl}
\mathtt{eq}_{\mathtt{Fun}} & : & (\mathtt{Fun}\,AB)\to(\mathtt{Fun}\,AB)\to\mathtt{PROP} \\
 & \equiv & \lambda f, g{:}\mathtt{Fun}\,AB\,\forall x{:}\mathtt{el}\,A.\,(\mathtt{ap}\,f\,x) = (\mathtt{ap}\,g\,x) \\
\mathtt{Function} & : & \mathtt{Set}\to\mathtt{Set}\to\mathtt{Set} \\
 & \equiv & \lambda A, B{:}\mathtt{Set}.\,<\mathtt{Fun}\,AB, \mathtt{eq}_{\mathtt{Fun}}{\mid}A{\mid}B, \llcorner\ldots\lrcorner>
\end{array}$$

The term $\mathtt{ap}$ is used for function application. So $f(x)$ is formalized as $\mathtt{ap}\,f\,x$. This may be written as $f.\mathtt{ap}\,x$. Furthermore we extend pseudo-LEGO with the following abbreviation. Let $A, B$ be sets.

$$A{\Rightarrow}B \quad\equiv\quad \mathtt{Fun}\,A\,B$$

Besides unary functions, we also need binary functions a lot. It is trivial to extend the definitions above to the binary case.

3.2.2. Definition. We will only present the definition of binary functions. The definitions of $\mathtt{ap}^2$ and $\mathtt{exten}^2$ are merely the first and second projections.

$$\begin{array}{lll}\mathtt{Fun}^2 & : & \mathtt{Set}{\rightarrow}\mathtt{Set}{\rightarrow}\mathtt{Set}{\rightarrow}\mathtt{SET} \\ & \equiv & \lambda A,B,C{:}\mathtt{Set}\,\Sigma f{:}(\mathtt{el}\,A){\rightarrow}(\mathtt{el}\,B){\rightarrow}(\mathtt{el}\,C).\,\mathtt{extensional}^2 f\end{array}$$

For arbitrary structures we even need functions of arbitrary arity. We will define these in Section 3.4.2. We could have used arbitrary arity functions to instantiate them to the unary or binary case. Because functions are a rather basic notion which is used a lot in a proof checker, and because this instantiation would make type checking considerably slower, we choose to treat the unary and binary case separately.

### 3.2.2 Choice axioms

Some mathematicians have been reserved to make use of (variants of) the axiom of choice. Because of the contructive nature of the propositions-as-types paradigma, the choice principles are not *derivable* in our system.

3.2.3. Definition (Axiom of Choice). Let $A$ and $B$ be sets, and $S$ a subset of $A \times B$. If for each $a \in A$ there exists an element $b \in B$ such that $(a, b) \in S$, then there is a function $f \in A{\Rightarrow}B$ such that for each $a$ in $A$ we have $(a, f(a)) \in S$.

In constructive mathematics, existence is much more restrictive then the existence in classical mathematics. As (Bishop 1967, chapter 1, paragraph 3) writes "the only way to show that an object exists is to give a finite routine for finding it".

So suppose that, in order to get a choice function, we have proven for any $a$ the existence of $b$ such that $R(a, b)$ holds. Then we had to provide a finite routine $i$ for which $R(a, i(a))$ holds for every $a$. This routine need not preserve equality, so in general we do not get a choice *function*. Furthermore, this routine $i$ only exists on the meta theoretical level. So if we need the axiom of choice, we have to introduce it axiomatically.

There are many weaker forms for the axiom of choice. One of them, which we call the axiom of unique choice, is particularly useful if we need to construct a function object from its graph. As opposed to the axiom of choice, the underlying finite routine of the proof of $\exists! x.\phi(x)$ does preserve equality. So in constructive mathematics, the axiom of unique choice seems to be an acceptable principle to use.

3.2.4. Definition (Axiom of Unique Choice). Let $A$ and $B$ be sets. Every subset $S \subset A \times B$ for which

- for each $a \in A$ there exists an element $b \in B$ such that $(a, b) \in S$

– if $(a, b_1)$ and $(a, b_2)$ are elements of $S$, then $b_1 = b_2$

determines a function $f \in A \Rightarrow B$ such that for each $a$ in $A$ we have $(a, f(a)) \in S$.

When we formalize these axioms, in particular the existence property of a function $f$, we have to decide whether we take the strong or weak existential quantifier.

The next definition makes use of the notion of a *binary relation* over sets. Given $A, B : \mathtt{Set}$, inhabitants of $\mathtt{Rel}\,A\,B$ are equality preserving binary relations over the elements of $A \times B$. See Section 3.2.4 for the formal definition of $\mathtt{Rel}$.

3.2.5. Definition. (i) Define the axiom of choice over sets, binary relations and functions, and over types and operators.

$$\begin{array}{lcl} \mathtt{AC} & : & \mathtt{PROP} \\ & \equiv & \Pi A, B{:}\mathtt{Set}\,\forall R{:}\mathtt{Rel}\,A\,B.\,(\forall x{:}\mathtt{el}\,A\,\exists y{:}\mathtt{el}\,B.\,R.\mathtt{ap}\,x\,y) \to \\ & & \qquad \exists f{:}A \Rightarrow B\,\forall x{:}\mathtt{el}\,A.\,R.\mathtt{ap}\,x\,(f.\mathtt{ap}\,x) \\ \mathtt{ac} & : & \mathtt{PROP} \\ & \equiv & \Pi T, U{:}\mathtt{SET}\,\forall R{:}T \to U \to \mathtt{PROP}.\,(\forall x{:}T\,\exists y{:}U.\,R\,x\,y) \to \\ & & \qquad \exists f{:}T \to U\,\forall x{:}T.\,R\,x\,(f\,x) \end{array}$$

(ii) Define the axiom of unique choice over sets and functions using weak and strong existential quantification.

$$\begin{array}{lcl} \mathtt{AUC}_\exists & : & \mathtt{PROP} \\ & \equiv & \Pi A, B{:}\mathtt{Set}\,\forall R{:}\mathtt{Rel}\,A\,B.\,(\forall x{:}\mathtt{el}\,A\,\exists! y{:}\mathtt{el}\,B.\,R.\mathtt{ap}\,x\,y) \to \\ & & \qquad \exists f{:}A \Rightarrow B\,\forall x{:}\mathtt{el}\,A.\,R.\mathtt{ap}\,x\,(f.\mathtt{ap}\,x) \\ \mathtt{AUC}_\Sigma & : & \mathtt{TYPE} \\ & \equiv & \Pi A, B{:}\mathtt{Set}\,\forall R{:}\mathtt{Rel}\,A\,B.\,(\forall x{:}\mathtt{el}\,A\,\exists! y{:}\mathtt{el}\,B.\,R.\mathtt{ap}\,x\,y) \to \\ & & \qquad \Sigma f{:}A \Rightarrow B\,\forall x{:}\mathtt{el}\,A.\,R.\mathtt{ap}\,x\,(f.\mathtt{ap}\,x) \end{array}$$

The axiom of unique choice states that if we have a function as a graph, there exists a function in the type theoretic sense. The strong variant actually gives us this function as an object.

It is easy to prove that from $\mathtt{AC}$ follows $\mathtt{ac}$ and that from $\mathtt{ac}$ follows $\mathtt{AUC}_\exists$. Also $\mathtt{AUC}_\Sigma \to \mathtt{AUC}_\exists$ is provable, but the reverse implication is not. The following lemma shows that in our system the axiom of choice over sets implies classical logic.

3.2.6. Lemma. *The axiom of choice implies excluded middle.*

$$\mathtt{AC} \to \forall P{:}\mathtt{PROP}.\,P \vee \neg P$$

Proof. *We present an informal sketch of the proof. Suppose we have* $\mathtt{AC}$. *Let* $P : \mathtt{PROP}$ *be a proposition. First we define two predicates* $\phi$ *and* $\psi$ *over* $\Omega$*:*

$$\begin{array}{lcl} \phi(\alpha) & \equiv & \alpha \vee P \\ \psi(\alpha) & \equiv & \neg\alpha \vee P \ . \end{array}$$

*Next we construct a two-element set* $A$ *and a relation* $R$ *over* $A \times \Omega$ *as follows.*

$$\begin{array}{lcl} A & \equiv & \{\phi, \psi\} \\ R(\chi, \alpha) & \equiv & \chi(\alpha) \end{array}$$

*We show that $R$ is a graph. That is, we prove*

$$\forall\chi{:}\mathtt{el}\,A\,\exists\alpha{:}\mathtt{PROP}.\,R(\chi,\alpha)$$

*by case distinction on $\chi$ and substituting* `true` *respectively* `false` *for $\alpha$. Then we apply* `AC` *to obtain a predicate $f$ over $A$ for which*

$$\forall\chi{:}\mathtt{el}\,A.\,\chi(f(\chi))\ \ .$$

*In particular, $\phi(f(\phi))$ and $\psi(f(\psi))$ hold. We apply case distinction on $f(\phi)\vee P$ and $\neg f(\psi)\vee P$. For the case that $f(\phi)$ and $\neg f(\psi)$ hold, we assume $P$ is true. But then we have that $\psi=\phi$ and hence $f(\psi)=f(\phi)$ which is leads to a contradiction. So we give up the assumption and conclude $\neg P$. For the other three cases, we have $P$ immediately.*

The proof is based on the fact that the propositions form a set and that we have the comprehension axiom (see Section 3.3.3). This enables us to define the setoid $A$ as follows.

$$\begin{array}{lcl} =_P & : & (\mathtt{PROP}{\to}\mathtt{PROP}){\to}(\mathtt{PROP}{\to}\mathtt{PROP}){\to}\mathtt{PROP} \\ & \equiv & \lambda\phi,\psi{:}(\mathtt{el}\,\Omega){\to}(\mathtt{el}\,\Omega)\,\forall\alpha{:}\mathtt{PROP}.\,(\phi\alpha)=(\psi\alpha) \\ A & : & \mathtt{Set} \\ & \equiv & <\Sigma\chi{:}\mathtt{PROP}{\to}\mathtt{PROP}.\,\chi=_P\phi\vee\chi=_P\psi,\llcorner\ldots\lrcorner,\llcorner\ldots\lrcorner> \end{array}$$

For details, see the LEGO library.

3.2.7. LEMMA. $\mathtt{AUC}_\Sigma$ *is conservative over* $\mathtt{AUC}_\exists$

PROOF (sketch). Suppose we have proved a statement $T$ by a proof $z:T$ which makes use of an application of $\mathtt{AC}_\Sigma$. So say for a relation $R$ and a proof $h$ we used in the proof $z$ the term

$$\mathtt{AC}_\Sigma\,R\,h\quad:\quad\Sigma f{:}A{\Rightarrow}B\,\forall a{:}\mathtt{el}\,A.\,R(a,f(s))\ \ .$$

Then we can prove $\forall f{:}A{\Rightarrow}B.\,[(\forall a{:}\mathtt{el}\,A.\,R(a,f(s)))\to T]$ without using $(\mathtt{AC}_\Sigma\ R\ h)$ anymore. By an application of $\mathtt{AC}_\exists$ we can get a new proof of $T$ without the strong axiom of choice.

The strong versions of the choice axioms as we formulated them have the remarkable properties that they make weak existential quantification strong.

3.2.8. LEMMA. *Suppose $A$¡*`Set`*, $\phi$:*`Pred` *$A$.*
(i) $\mathtt{AC}_\Sigma{\to}\ (\exists x{:}\mathtt{el}\,A.\,\phi(x))\to(\Sigma x{:}\mathtt{el}\,A.\,\phi(x))$
(ii) $\mathtt{AUC}_\Sigma{\to}\ (\exists! x{:}\mathtt{el}\,A.\,\phi(x))\to(\Sigma x{:}\mathtt{el}\,A.\,\phi(x)\ \&\ \mathtt{unique}(\phi,x))$

PROOF. For a proof, see the LEGO library.

So as soon as we assume $\mathtt{AC}_\Sigma$ or $\mathtt{AUC}_\Sigma$, the projection for the existential quantifiers $\exists$ respectively $\exists!$ become provable inside the system.

None the proofs presented in the case studies of the next chapter employed any choice principle at all. During the development of formalized mathematics, we encountered only two occasions where we felt the need for a choice principle. Firstly for a proof that the defined equality over categorical subsets is equivalent to the equality on predicates (see Section 3.3.1). Secondly, for the construction of an inverse function from a surjective injection. It is not hard to see that both statements are equivalent to the axiom of unique choice.

### 3.2.3 Graphs

Set theory uses graphs to represent functions. A graph is binary relation that is total and unique.

3.2.9. DEFINITION. (i) Define what it means for a relation to be *total* or *functional*.

$$\begin{array}{lcl} \texttt{total} & : & \Pi A, B{:}\texttt{Set}\,(\texttt{Rel}\,A\,B)\rightarrow\texttt{PROP} \\ & \equiv & \lambda A, B{:}\texttt{Set}\,\lambda R{:}\texttt{Rel}\,A\,B\,\forall x{:}\texttt{el}\,A\,\exists y{:}\texttt{el}\,B.\,R(x,y) \\ \texttt{functional} & : & \Pi A, B{:}\texttt{Set}\,(\texttt{Rel}\,A\,B)\rightarrow\texttt{PROP} \\ & \equiv & \lambda A, B{:}\texttt{Set}\,\lambda R{:}\texttt{Rel}\,A\,B \\ & & \quad\forall x{:}\texttt{el}\,A\,\forall y, y'{:}\texttt{el}\,B.\,R(x,y)\rightarrow R(x,y')\rightarrow y = y' \end{array}$$

(ii) Define functions in a set-theoretic way as total functional binary relations.

$$\begin{array}{lcl} \texttt{Fun}_{\texttt{Gr}} & : & \texttt{Set}\rightarrow\texttt{Set}\rightarrow\texttt{SET} \\ & \equiv & \lambda A, B{:}\texttt{Set}.\,\Sigma R{:}\texttt{Rel}\,A\,B.\,(\texttt{total}\,R)\ \&\ (\texttt{functional}\,R) \end{array}$$

Functions as graphs as have the advantage that the axiom of unique choice as formulated in the previous section becomes a tautology. So it will be no problem to define a function by cases on a decidable predicate.

3.2.10. LEMMA. *Let $A, B$ : $\texttt{Set}$ be a setoids, let $\phi$ : $(\texttt{el}\,A) \rightarrow$ Prop be a decidable predicate. Let $a, b$ : $\texttt{el}\,B$ be elements of $B$. Then there exists a function $F$ : $\texttt{Fun}_{\texttt{Gr}}\,A\,B$ such that*

$$\begin{array}{rcll} F(x) & = & a & \textit{if} \quad \phi(x) \\ & & b & \textit{else} \end{array}$$

PROOF. *Define $f$ as*

$$f \quad = \quad \lambda x, y{:}\texttt{el}\,A.\,(a = y\ \&\ (\phi x)) \vee (b = y\ \&\ \neg(\phi x))$$

*Then it is trivial to show that $f$ is a total functional relation for which the desired property holds.*

From a set-theoretic standpoint of view graphs are nice, for type theory these are highly inconvenient to use. Lambda calculus invites us to represent functions as by $\lambda$-terms. For example, to formalize '$\forall x.f(g(x)) = 2$' as graphs we have to write something like

$$\forall x \exists y.R_f(y, 2)\ \&\ R_g(x, y)$$

### 3.2.4 Predicates and relations

In this subsection we will define the notion of predicates. We will formalize them as functions into the set of propositions $\Omega$. Let us first present a definition of $n$-ary relations following (Mines et al. 1988, chapter 1, paragraph 2).

> An $n$-ary relation on a set $S$ is a property $P$ that is applicable to $n$-tuples of elements of $S$, and is extensional in the sense that if $x_i = y_i$, for $i = 1, \ldots, n$, then $P(x_1, \ldots, x_n)$ if and only if $P(y_1, \ldots, y_n)$.

The naive way to formalize predicates is by tuples of a type theoretic function $f$ into PROP, together with a proof that $f$ preserves equality. But since we have functions over sets and the setoid of propositions $\Omega$, it is easy to define predicates and relations as functions into $\Omega$. This has the advantage that all constructions and lemmas which are valid for functions also can be instantiated by predicates. In the following definitions we only present the case for predicates, while relations are a trivial extension.

3.2.11. Definition. (i) Define predicates as functions into the set of propositions.

$$\begin{array}{rcl} \mathtt{Pred} & : & \mathtt{Set}{\to}\mathtt{SET} \\ & \equiv & \lambda A{:}\mathtt{Set}.\,\mathtt{Fun}\,A\,\Omega \end{array}$$

(ii) Let $A : \mathtt{Set}$ be a setoid. Transform the type $\mathtt{Pred}\,A : \mathtt{SET}$ into a setoid.

$$\begin{array}{rcl} \mathtt{Predicate} & : & \mathtt{Set}{\to}\mathtt{Set} \\ & \equiv & \lambda A{:}\mathtt{Set}.\,\mathtt{Function}\,A\,\Omega \end{array}$$

Because Pred is defined in terms of functions, extensionality holds automatically for predicates also. This means that given a predicate $\phi : \mathtt{Pred}\,A$ for some setoid $A$, we have that for $x, y : \mathtt{el}\,A$,

$$(x = y) \quad \to \quad (\phi(x) \iff \phi(y)) \ .$$

This is ensured because the underlying equality of the set of propositions $\Omega$ is the if-and-only-if relation. For the definition of Predicate, pointwise equality of functions is used to compare subsets. This implies that for any two predicates $\phi$ and $\psi$ of type $\mathtt{Predicate}\,A$ we have

$$\phi = \psi \quad \equiv \quad \forall x{:}\mathtt{el}\,A.\,\phi(x) \iff \psi(x) \ .$$

3.2.12. Definition. (i) A predicate over a set $A$ is *decidable* if for every element of $A$ we know that $x$ is in the predicate or not. We distinguish predicates on the level of type theoretic maps into PROP and on the level of functions into $\Omega$.

$$\begin{array}{rcl} \mathtt{decidable\_pred} & : & \Pi T{\mid}\mathtt{SET}.\,(T{\to}\mathtt{PROP}) \to \mathtt{PROP} \\ & \equiv & \lambda T{\mid}\mathtt{SET}\,\lambda P{:}T{\to}\mathtt{PROP}\,\forall x{:}T.\,(P\,x) \vee \neg(P\,x) \\ \mathtt{decidable\_rel} & : & \Pi T{\mid}\mathtt{SET}.\,(T{\to}T{\to}\mathtt{PROP}) \to \mathtt{PROP} \\ & \equiv & \lambda T{\mid}\mathtt{SET}\,\lambda R{:}T{\to}T{\to}\mathtt{PROP}\,\forall x, y{:}T.\,(R\,x\,y) \vee \neg(R\,x\,y) \end{array}$$

(ii) Define decidability of predicates Pred and relations Rel.

$$\begin{array}{rcl} \mathtt{DecidablePred} & : & \Pi A{\mid}\mathtt{Set}.\,(\mathtt{Pred}\,A) \to \mathtt{PROP} \\ & \equiv & \lambda A{\mid}\mathtt{Set}\,\lambda P{:}\mathtt{Pred}\,A\,\mathtt{decidable\_pred}\,(\mathtt{ap}P) \\ \mathtt{DecidableRel} & : & \Pi A{\mid}\mathtt{Set}.\,(\mathtt{Rel}\,A\,A) \to \mathtt{PROP} \\ & \equiv & \lambda A{\mid}\mathtt{Set}\,\lambda R{:}\mathtt{Rel}\,A\,A\,\mathtt{decidable\_rel}\,(\mathtt{ap}^2R) \end{array}$$

(iii) We also can define what it is for a setoid to be *discrete*. A setoid is discrete if the underlying equality relation is decidable. Let $A : \mathtt{Set}$ be a setoid.

$$\begin{array}{rcl} \mathtt{Discrete} & : & \mathtt{PROP} \\ & \equiv & \mathtt{decidable\_rel}\,(= {\mid}A) \end{array}$$

## 3.3 Subsets

In this section we elaborate on subsets. Now that we have predicates, it seems natural to define subsets as predicates. We will look briefly to another approach, namely categorical subsets. Next we will show how to construct the power set, and show how we can transform subsets into sets. For decidable predicates, we will define a way to obtain the characteristic function by way of case distinction.

### 3.3.1 Subsets as predicates

Bishop defines subsets with use of an injection map. We quote (Bishop 1967, chapter 3, definition 1):

> A *subset* $(A, i)$ of a set $B$ consists of a set $A$ and a function $i : A \to B$, called the *inclusion map*, such that
>
> $$a_1 = a_2 \quad \text{if and only if} \quad i(a_1) = i(a_2)$$
>
> for all $a_1$ and $a_2$ in $A$.

This function $i$ injects every element of $A$ into the set $B$. So actually, $A$ is a subset of $B$ if $A$'s cardinality is smaller then the cardinality of $B$.

However, in type theory it seems to be more natural to formalize subsets as predicates. So given a set $A$, a subset $B$ is formed by indicating which elements of $A$ belong to $B$. The set-inclusion relation then is valid only for subsets over a common set.

3.3.1. Definition. (i) Define subsets as predicates and the power set by forming subsets into setoids.

$$\begin{array}{rcl} \texttt{Subset} & : & \texttt{Set}{\to}\texttt{SET} \\ & \equiv & \texttt{Pred} \end{array}$$

(ii) For readability, we define the 'is an element' relation for subsets. Let $A : \texttt{Set}$ be a setoid.

$$\begin{array}{rcl} \texttt{elem} & : & (\texttt{el}\,A){\to}(\texttt{Subset}\,A){\to}\texttt{PROP} \\ & \equiv & \lambda x{:}\texttt{el}\,A\,\lambda P{:}\texttt{Subset}\,A.\,\texttt{ap}\,P\,x \end{array}$$

3.3.2. Definition. A subset $S$ of a set $A$ is detachable if every element of $A$ belongs to $S$ or not.

$$\begin{array}{rcl} \texttt{detachable} & : & (\texttt{Subset}\,A) \to \texttt{PROP} \\ & \equiv & \texttt{DecidablePred}{\restriction}A \end{array}$$

In a classical setting, all subsets are detachable.

### 3.3.2 Power sets

Power set formation is considered as a very powerful operation. The way we have formalized predicates gives us the power sets immediately.

3.3.3. Definition.

$$\begin{array}{rcl} \texttt{Powerset} & : & \texttt{Set}{\to}\texttt{Set} \\ & \equiv & \lambda A{:}\texttt{Set}.\,\texttt{Predicate}\,A \end{array}$$

The crucial point is that in Definition 3.1.7, we allowed ourselves to form the set of propositions $\Omega$ by quotienting them by the *if-and-only-if* relation.

3.3.4. Definition. Let $A, B{:}\mathtt{Set}$ be a setoids. We define operators over subsets and the notions of image and pre-image of a subset.

(i) Let $S, T : \mathtt{el}\,(\mathtt{Powerset}\,A)$ be subsets of $A$. Define the empty subset, subset complement, union and intersection.

$$\begin{array}{lcl} \mathtt{void} & : & \mathtt{el}\,(\mathtt{Powerset}\,A) \\ & \equiv & <\lambda x{:}\mathtt{el}\,A.\,\mathtt{false}, \llcorner\ldots\lrcorner> \\ \mathtt{compl} & : & \mathtt{el}\,(\mathtt{Powerset}\,A) \\ & \equiv & <\lambda x{:}\mathtt{el}\,A.\,\neg(\mathtt{elem}\,x\,S), \llcorner\ldots\lrcorner> \\ \mathtt{union} & : & \mathtt{el}\,(\mathtt{Powerset}\,A) \\ & \equiv & <\lambda x{:}\mathtt{el}\,A.\,(\mathtt{elem}\,x\,S) \vee (\mathtt{elem}\,x\,T), \llcorner\ldots\lrcorner> \\ \mathtt{inter} & : & \mathtt{el}\,(\mathtt{Powerset}\,A) \\ & \equiv & <\lambda x{:}\mathtt{el}\,A.\,(\mathtt{elem}\,x\,S)\ \&\ (\mathtt{elem}\,x\,T), \llcorner\ldots\lrcorner> \end{array}$$

(ii) Let $f : \mathtt{Fun}\,A\,B$ be a function. Define the pre-image and image of $f$.

$$\begin{array}{lcl} \mathtt{PreImage} & : & (\mathtt{el}\,(\mathtt{Powerset}\,B)) \to (\mathtt{el}\,(\mathtt{Powerset}\,A)) \\ & \equiv & \lambda C{:}\mathtt{el}\,(\mathtt{Powerset}\,B).\,<\lambda a{:}\mathtt{el}\,A.\,\mathtt{elem}\,f(a)\,C, \llcorner\ldots\lrcorner> \\ \mathtt{Image} & : & (\mathtt{el}\,(\mathtt{Powerset}\,A)) \to (\mathtt{el}\,(\mathtt{Powerset}\,B)) \\ & \equiv & \lambda C{:}\mathtt{el}\,(\mathtt{Powerset}\,A). \\ & & \quad <\lambda b{:}\mathtt{el}\,B\,\exists a{:}\mathtt{el}\,A.\,(\mathtt{elem}\,a\,C)\ \&\ (f(a) = b), \llcorner\ldots\lrcorner> \end{array}$$

3.3.5. Lemma. *Given* $S, T : \mathtt{el}\,(\mathtt{Powerset}\,A)$ *be subsets of a setoid* $A : \mathtt{Set}$. *We have the following De Morgan laws.*

(i)

$$(\mathtt{inter}\,(\mathtt{compl}\,S)\,(\mathtt{compl}\,T)) \quad = \quad \mathtt{compl}\,(\mathtt{union}\,S\,T)$$

(ii) *If* $\mathtt{detachable}\,S$ *and* $\mathtt{detachable}\,T$ *hold, then*

$$(\mathtt{union}\,(\mathtt{compl}\,S)\,(\mathtt{compl}\,T)) \quad = \quad \mathtt{compl}\,(\mathtt{inter}\,S\,T)\ .$$

Proof. *See the LEGO library.*

### 3.3.3 Subsets into sets

The comprehension axiom plays an important role in set theory. The axiom states that

> for any property $\phi$ and set $A$ we can form the set $\{x \in A | \phi(a)\}$ of all elements of $A$ which satisfy property $\phi$ .

Let $A$ be a set and $S$ be a subset over $A$. We use sigma-types to define the type of all elements of $A$ that are member of $S$. This type is formed into a set by adding the equality relation of $A$.

$$\begin{array}{lcl} \mathtt{toSET} & : & \Pi T{:}\mathtt{SET}.\,(T{\to}\mathtt{PROP}){\to}\mathtt{SET} \\ & \equiv & \lambda T{:}\mathtt{SET}\,\lambda\phi{:}T{\to}\mathtt{PROP}.\,\Sigma x{:}T.\,\phi\,x \\ \mathtt{toSet} & : & \Pi A{:}\mathtt{Set}.\,(\mathtt{Subset}\,A){\to}\mathtt{Set} \\ & \equiv & \lambda A{:}\mathtt{Set}\,\lambda S{:}\mathtt{Subset}\,A.\,<T, \lambda x, y{:}T.\,x_1 = y_1, \llcorner\ldots\lrcorner> \\ & & \text{where } T : \mathtt{SET} \equiv \mathtt{toSET}\,(\mathtt{ap}\,S) \end{array}$$

The comprehension axiom has the danger that it may lead to a paradox. Namely, define $z \equiv \{y|y \notin y\}$, then we have $z \in z \iff z \notin z$ which is a contradiction. However, the Russell paradox will not arise because we demand that elements of $z$ belong to a set of which $z$ itself can not be a member. Also $\texttt{elem}\, z\, z$ is not typable.

### 3.3.4 Case distinction

Unless we have functions as graphs, it is not possible to define a function by cases in pure type systems like $\lambda C$. The reason for this is that we defined the logical or-connective by an impredicative definition. The or-elimination principle allows us only to eliminate into *propositions*. In general however, a function has a *type* as codomain. Of course, we could assume the strong variant of the axiom of unique choice to define a function by cases. But this extension of the context is not wanted nor necessary in most cases.

The inductively defined or-connective has a stronger elimination principle, namely elimination into types. Then we are able to define a function by case distinction. If we need case distinction we could substitute the impredicative definition of the or-connective by the inductive definition. This would mean redoing all our proofs and rebuilding all our libraries. But fortunately, there is a more elegant solution. As we will see, just the *definition* of the inductively defined or makes the impredicative defined or strong.

3.3.6. Definition. Recall the definition of the logical or-connective both impredicative and inductively from Section 2.1.

$$\begin{array}{lcl}
\vee & : & \texttt{PROP}\to\texttt{PROP}\to\texttt{PROP} \\
 & \equiv & \lambda P, Q{:}\texttt{PROP}\, \Pi X{:}\texttt{PROP}.\, (P\to X)\to(Q\to X)\to X \\
\texttt{inl} & : & \Pi P, Q{:}\texttt{PROP}.\, A\to(A\vee B) \\
 & \equiv & \llcorner\ldots\lrcorner \\
\texttt{inr} & : & \Pi P, Q{:}\texttt{PROP}.\, B\to(A\vee B) \\
 & \equiv & \llcorner\ldots\lrcorner \\
\texttt{Or} & : & \texttt{PROP}\to\texttt{PROP}\to\texttt{PROP} \\
 & \equiv & \lambda P, Q{:}\texttt{PROP}\, \mu X{:}\texttt{PROP}.\, (\texttt{Inl}_{P,Q} : P\to X, \texttt{Inr}_{P,Q} : Q\to X)
\end{array}$$

Then we have

$$\begin{array}{lcl}
\varepsilon_{\texttt{Or}}\, f\, g\, (\texttt{Inl}_{P,Q}\, p) & =_\iota & f\, p \\
\varepsilon_{\texttt{Or}}\, f\, g\, (\texttt{Inr}_{P,Q}\, q) & =_\iota & g\, q
\end{array}$$

where $P, Q : \texttt{PROP}$, $p : P$, $q : Q$, $\phi : (\texttt{Or}\, P\, Q)\to\texttt{SET}$, $f : \forall p{:}P.\, \phi\, (\texttt{Inl}_{P,Q}\, p)$, and $g : \forall q{:}Q.\, \phi\, (\texttt{Inr}_{P,Q}\, q)$.

Using $\texttt{Or}$ we can define a term $\texttt{select}$ for case distinction based on $P \vee Q$:

3.3.7. Definition. Let $P, Q : \texttt{PROP}$ be propositions.

$$\begin{array}{lcl}
\texttt{select} & : & (P\vee Q)\to\Pi T{:}\texttt{SET}.\, T\to T\to T \\
 & \equiv & \lambda z{:}P\vee Q\, \lambda T{:}\texttt{SET}\, \lambda a, b{:}T. \\
 & & \quad \varepsilon_{\texttt{Or}}\, (\lambda p{:}P.\, a)\, (\lambda q{:}Q.\, b)\, (z\, \texttt{Inl}_{P,Q}\, \texttt{Inr}_{P,Q})
\end{array}$$

3.3.8. Lemma. *Let $P, Q : \texttt{PROP}$ be propositions. Let $T : \texttt{SET}$ be a type and $a, b : T$.*

(i) *The or-connectives are equivalent.*

$$(P \vee Q) \leftrightarrow (\texttt{Or}\, P\, Q)$$

(ii) *Suppose $p : P$, and $q : Q$. Then*

$$\begin{aligned} \texttt{select}\,(\texttt{inl}_{P,Q}\, p)\, a\, b &=_{\beta\iota} a \\ \texttt{select}\,(\texttt{inr}_{P,Q}\, q)\, a\, b &=_{\beta\iota} b \ . \end{aligned}$$

(iii) *If $z : P \vee Q$, then*

$$\begin{aligned} \texttt{select}\, z\, a\, b &=_{\beta\iota} a \quad \textit{if for some } h{:}P \textit{ we have } z =_{\beta\iota} \texttt{inl}_{P,Q}\, h \\ \texttt{select}\, z\, a\, b &=_{\beta\iota} b \quad \textit{if for some } h{:}Q \textit{ we have } z =_{\beta\iota} \texttt{inr}_{P,Q}\, h \ . \end{aligned}$$

(iv) *If $z : P \vee (\neg P)$, then*

$$\begin{aligned} P &\rightarrow (\texttt{select}\ z\, a\, b) =_L a \\ \neg P &\rightarrow (\texttt{select}\ z\, a\, b) =_L b \ . \end{aligned}$$

Proof. The proof of (i) is trivial (see the LEGO library for details). To see that (ii) holds we have to realize that

$$\begin{aligned} \texttt{inl}_{P,Q}\, p\, \texttt{Inl}_{P,Q}\, \texttt{Inr}_{P,Q} &=_\beta \texttt{Inl}_{P,Q}\, p \\ \texttt{inr}_{P,Q}\, q\, \texttt{Inl}_{P,Q}\, \texttt{Inr}_{P,Q} &=_\beta \texttt{Inr}_{P,Q}\, q \ . \end{aligned}$$

Lemma (iii) follows directly from (ii). We only present a proof of the first part of (iv). Assume that $P$ holds. Expand the definition of `select` and apply the Or-elimination principle on

$$z\, \texttt{Inl}_{P,\neg P}\, \texttt{Inl}_{P,\neg P} \ : \ \texttt{Or}\ P\,(\neg P) \ .$$

Then we are left to prove $P \rightarrow (a =_L a)$ and $(\neg P) \rightarrow (b =_L a)$, which are both obvious because $P$ holds.

If the proof $z$ in Lemma 3.3.8 (iv) is constructive, $z$ will generally have a shape as stated in (iii). As a consequence we then get the stronger result that

$$\begin{aligned} P &\rightarrow (\texttt{select}\ z\, a\, b) =_{\beta\iota} a \\ \neg P &\rightarrow (\texttt{select}\ z\, a\, b) =_{\beta\iota} b \ . \end{aligned}$$

Case distinction via `select` is not always convenient. Sometimes we want to test whether two elements in a discrete set are equal or not.

3.3.9. Definition. Let $A{\mid}\texttt{Set}$ be a discrete setoid such that $A_{\texttt{discr}}$ is a prove for $\texttt{Discrete}\, A$. Let $x, y : \texttt{el}\, A$ be elements of $A$. Define

$$\begin{aligned} \texttt{if} &: \Pi T{\mid}\texttt{SET}.\, T \rightarrow T \rightarrow T \\ &\equiv \texttt{select}\,(A_{\texttt{discr}}\, x\, y) \ . \end{aligned}$$

Now we can write '$\texttt{if}\ A_{\texttt{discr}}\, x\, y\, a\, b$' which means 'if $x = y$ then $a$ otherwise $b$'.

The following extension of pseudo LEGO allows us to write case distinction for decidable predicates in a more readable way. Let $\phi : T \rightarrow \texttt{PROP}$ be a decidable predicate over a type $T : \texttt{SET}$ and $h_1, h_2$ be of type $T \rightarrow T$. Let $\phi_{\texttt{dec}} : \texttt{decidable\_pred}\, \phi$ be a proof of the decidability of $\phi$. Then

$$f \ \equiv \ \texttt{select}\,(\phi_{\texttt{dec}} x)(h_1\, x)(h_2\, x)$$

may be written as

$$f \quad \equiv \quad \lambda x{:}T.\ \begin{array}{ll} h_1\,x & \text{if } \phi\,x \\ h_2\,x & \text{otherwise} \end{array}$$

in pseudo LEGO.

3.3.10. Definition. Using case distinction it is easy to get the characteristic function of a decidable predicate.

(i) Let $T$:`SET` be a type, $\phi$:$T$→`PROP` and $\phi_{\texttt{dec}}$ : `decidable_pred` $\phi$ a proof that $\phi$ is decidable. Define the map

$$\begin{array}{lcl} \texttt{char} & : & T \to \texttt{nat} \\ & \equiv & \lambda x{:}T.\ \begin{array}{ll} \underline{0} & \text{if } \phi\,x \\ \underline{1} & \text{otherwise} \end{array} \end{array}$$

(ii) Because the map `char` preserves equality we can extend it to a function. Let $A$:`Set` be a setoid, $\phi$:`Pred` $A$ and $\phi_{\texttt{dec}}$ : `DecidablePred` $\phi$ a proof that $\phi$ is decidable. Define

$$\begin{array}{lcl} \texttt{Char} & : & \texttt{Fun}\,A\,\mathbb{N} \\ & \equiv & \texttt{<char}\,\phi_{\texttt{dec}}, \llcorner\ldots\lrcorner\texttt{>} \end{array}$$

3.3.11. Lemma. *Let $T$ : `SET` be a type, $\phi$ : $T$→`PROP` be a decidable predicate over $T$.*

$$\forall x{:}T.\,(\phi\,x) \iff (\texttt{char}\,\phi_{\texttt{dec}}\,x) = \underline{0}$$

Proof. *By Lemma 3.3.8 (iv) and some equational reasoning.*

For an application of `select` and `char`, the reader is referred to Section 4.2. There we define a prime generator using bounded minimalization.

## 3.4 Mathematical Structures

In this section we present a formalization of a general framework of syntactical descriptions and their realizations. For two reasons it is important to formalize mathematical structures as primitive notions. First, it makes it possible to define notions like homomorphism and substructures for arbitrary structures in a general way. So to obtain the type of homomorphisms over monoids, we just have to instantiate homomorphisms with the structure of monoids. Of course, all results for homomorphisms will immediately carry over to homomorphisms over monoids. Furthermore, we wish to be able to reason about structures in general. This will allow us to develop meta-theory inside our system. An example is equational reasoning which we present using a two-level approach in Chapter 5.

We will consider single-sorted structures only. The reason for this is threefold.

1. All our case studies do not need many-sorted structures. We don't work with metric spaces for example.

2. Most structures like 'function', 'terms', or 'homomorphism' are quite easier to define single-sorted. Also their application in formal proofs is simpler.

3. The construction of the single-sorted case as an instantiation of the many-sorted case is quite involved. It would make type-checking considerably less efficient.

Mathematical structures are constructed in two stages. First we define the syntactical notion of a signature. For the case of single-sorted algebras, the signature determines a countable set of function and predicate symbols. Also, every symbol is assigned an arity. Next the syntax is interpreted. This leads to the semantical notion of a structure. The interpretation consists of a carrier set and a function and predicate for each symbol of the signature.

We distinguish structures and models. Compared to structures, models satisfy additional axioms. We define the notion of axioms over *structures* instead of over signatures. This makes much more general formulations of axioms possible. A drawback might be that axioms do not have structure anymore about which we can reason. However, we feel that this is not out weighted by the advantage of freely structured axioms.

Recapitulating, structures are introduced by

1. a *signature* defining the alphabet,
2. a *carrier set* for the domain of the structure,
3. two *valuation maps* which interpret every symbol of the signature to a function or predicate over the carrier set.

Based upon structures we construct the notion of models by adding

4. a proposition over the interpreted symbols of the signature to formulate the *axioms* of a structure,
5. a *proof* of the axioms.

### 3.4.1 Syntax

Single-sorted signatures consist of two types representing the number of function and predicate symbols, together with two maps which assign an arity to each symbol. Constants are treated as functions with arity zero.

3.4.1. Definition. (i) We define the type of signatures inductively as follows.

$$\begin{array}{lcl} \mathtt{Signature} & : & \mathtt{TYPE} \\ & \equiv & \mu X{:}\mathtt{TYPE}.\,( \\ & & \quad \mathtt{Sig_{intro}} : \Pi F{:}\mathtt{SET}.\,(F{\to}\mathtt{nat}) \to \\ & & \qquad\qquad\quad \Pi P{:}\mathtt{SET}.\,(P{\to}\mathtt{nat}) \to X \\ & & ) \end{array}$$

(ii) Define projection functions to extract various properties of a signature.

$$\begin{array}{lcl} \mathtt{symbol_F} & : & \mathtt{Signature} \to \mathtt{SET} \\ & \equiv & \varepsilon_{\mathtt{Signature}}\,(\lambda F{:}\mathtt{SET}\,\lambda Ar_F{:}F{\to}\mathtt{nat}\,\lambda P{:}\mathtt{SET}\,\lambda Ar_P{:}P{\to}\mathtt{nat}.\,F) \\ \mathtt{symbol_P} & : & \mathtt{Signature} \to \mathtt{SET} \\ & \equiv & \varepsilon_{\mathtt{Signature}}\,(\lambda F{:}\mathtt{SET}\,\lambda Ar_F{:}F{\to}\mathtt{nat}\,\lambda P{:}\mathtt{SET}\,\lambda Ar_P{:}P{\to}\mathtt{nat}.\,P) \\ \mathtt{arity_F} & : & \Pi s{:}\mathtt{Signature}.\,(\mathtt{symbol_F}\,s) \to \mathtt{nat} \\ & \equiv & \varepsilon_{\mathtt{Signature}}\,(\lambda F{:}\mathtt{SET}\,\lambda Ar_F{:}F{\to}\mathtt{nat}\,\lambda P{:}\mathtt{SET}\,\lambda Ar_P{:}P{\to}\mathtt{nat}.\,Ar_F) \\ \mathtt{arity_P} & : & \Pi s{:}\mathtt{Signature}.\,(\mathtt{symbol_P}\,s) \to \mathtt{nat} \\ & \equiv & \varepsilon_{\mathtt{Signature}}\,(\lambda F{:}\mathtt{SET}\,\lambda Ar_F{:}F{\to}\mathtt{nat}\,\lambda P{:}\mathtt{SET}\,\lambda Ar_P{:}P{\to}\mathtt{nat}.\,Ar_P) \end{array}$$

3.4.2. Definition. Let $s$ : `Signature` be a signature. Define terms and formulas.

(i) First we define $n$-tuples of terms and terms.

$$\begin{array}{lll}
\mathtt{terms} & : & \mathtt{nat} \to \mathtt{SET} \\
 & \equiv & \mu X{:}\mathtt{nat} \to \mathtt{SET}.\,( \\
 & & \quad \begin{array}{lll}
 \mathtt{TFV} & : & \mathtt{nat} \to (X\underline{1}), \\
 \mathtt{TFC} & : & \Pi f{:}\mathtt{symbol}_{\mathtt{F}}\, s.\, (X(\mathtt{arity}_{\mathtt{F}}\, f)) \to (X\underline{1}), \\
 \mathtt{tnil} & : & X\underline{0}, \\
 \mathtt{tcons} & : & \Pi n{:}\mathtt{nat}.\, (X\underline{1}) \to (X n) \to (X(\mathtt{S}^{\mathtt{N}}\, n))
 \end{array} \\
 & & ) \\
\mathtt{term} & : & \mathtt{SET} \\
 & \equiv & \mathtt{terms}\,\underline{1}
\end{array}$$

Here `TFV` stands for *term formation variables* and `TFC` stands for *term formation constants* and functions. Variables are encoded by natural numbers.

(ii) Define (atomic) formulas inductively.

$$\begin{array}{lll}
\mathtt{aformula} & : & \mathtt{SET} \\
 & \equiv & \mu X{:}\mathtt{SET}.\,( \\
 & & \quad \begin{array}{lll}
 \mathtt{AFPred} & : & \Pi p{:}\mathtt{symbol}_{\mathtt{P}}\, s.\, (\mathtt{terms}\,(\mathtt{arity}_{\mathtt{P}}\, p)) \to X \\
 \mathtt{AFEq} & : & (\mathtt{terms}\,\underline{2}) \to X
 \end{array} \\
 & & ) \\
\mathtt{formula} & : & \mathtt{SET} \\
 & \equiv & \mu X{:}\mathtt{SET}.\,( \\
 & & \quad \begin{array}{lll}
 \mathtt{FFAtom} & : & \mathtt{aformula} \to X \\
 \mathtt{FFEx}, \mathtt{FFAll}, \mathtt{FFnot} & : & X \to X \\
 \mathtt{FFimp}, \mathtt{FFor}, \mathtt{FFand} & : & X \to X \to X
 \end{array} \\
 & & )
\end{array}$$

### 3.4.2 Semantics

We will define a structure over a signature $s$ as a set $A$ together with valuation functions for the function and predicate symbols of $s$. Given some axioms $ax$ over $s$, we define a model as a structure satisfying the axioms $ax$.

3.4.3. DEFINITION. (i) Define $n$-ary functions and predicates.

$$\begin{array}{lll}
\mathtt{Fun}^{\mathtt{N}} & : & \mathtt{Set} \to \mathbb{N} \to \mathtt{SET} \\
 & \equiv & \llcorner \ldots \lrcorner \\
\mathtt{Pred}^{\mathtt{N}} & : & \mathtt{Set} \to \mathbb{N} \to \mathtt{SET} \\
 & \equiv & \llcorner \ldots \lrcorner
\end{array}$$

(ii) Define structures and models.

$$\begin{array}{lll}
\mathtt{Structure} & : & \mathtt{Signature} \to \mathtt{TYPE} \\
 & \equiv & \lambda s{:}\mathtt{Signature}\, \Sigma A{:}\mathtt{Set}.\, (\Pi c{:}\mathtt{symbol}_{\mathtt{F}}\, s.\, \mathtt{Fun}^{\mathtt{N}}\, A\, (\mathtt{arity}_{\mathtt{F}}\, c)) \times \\
 & & \qquad\qquad\qquad\qquad\quad (\Pi p{:}\mathtt{symbol}_{\mathtt{P}}\, s.\, \mathtt{Pred}^{\mathtt{N}}\, A\, (\mathtt{arity}_{\mathtt{P}}\, p)) \\
\mathtt{Axioms} & : & \mathtt{Signature} \to \mathtt{TYPE} \\
 & \equiv & \lambda s{:}\mathtt{Signature}.\, (\mathtt{Structure}\, s) \to \mathtt{PROP} \\
\mathtt{Model} & : & \Pi s{:}\mathtt{Signature}.\, (\mathtt{Axioms}\, s) \to \mathtt{TYPE} \\
 & \equiv & \lambda s{:}\mathtt{Signature}\, \lambda ax{:}\mathtt{Axioms}\, s\, \Sigma str{:}\mathtt{Structure}\, s.\ ax\ str
\end{array}$$

Remark that in the definition of `Model` the lambda term *ax str* is a proposition.

The reader is referred to the LEGO library for the actual definition of $\texttt{Fun}^{\texttt{N}}$ and $\texttt{Pred}^{\texttt{N}}$. By definition we obtain the convenient properties that for any setoid $A : \texttt{Set}$

$$\begin{aligned}
\texttt{ap}\,(\texttt{Fun}^{\texttt{N}}\,A\,\underline{0}) &=_{\beta\iota} \texttt{el}\,A\\
\texttt{Fun}^{\texttt{N}}\,A\,\underline{1} &=_{\beta\iota} \texttt{Fun}\,A\,A\\
\texttt{Fun}^{\texttt{N}}\,A\,\underline{2} &=_{\beta\iota} \texttt{Fun}^2\,A\,A\,A\\
\texttt{Pred}^{\texttt{N}}\,A\,\underline{1} &=_{\beta\iota} \texttt{Pred}\,A\\
\texttt{Pred}^{\texttt{N}}\,A\,\underline{2} &=_{\beta\iota} \texttt{Rel}\,A\,A \ .
\end{aligned}$$

3.4.4. Definition. Define lambda terms to extract various properties of models and structures. Let $s{:}\texttt{Signature}$ be a signature.

(i) Let the proposition $ax{:}(\texttt{Axioms}\,s)$ formulate the axioms for a structure of the signature $s$. Let $M : (\texttt{Model}\,ax)$ be a model satisfying the axioms.

$$\begin{aligned}
\texttt{structure} &: \texttt{Structure}\,s\\
&\equiv \pi_1^2(M)\\
\texttt{axioms} &: ax\,\texttt{structure}\\
&\equiv \pi_2^2(M)
\end{aligned}$$

(ii) Let $ax{:}(\texttt{Axioms}\,s)$ be some axioms for a structure of the signature. Let $M$ : $(\texttt{Model}\,ax)$ be a model satisfying the axioms.

$$\begin{aligned}
\texttt{car} &: \texttt{Set}\\
&\equiv \pi_1^3(\texttt{structure}\,M)\\
\texttt{obj} &: \texttt{SET}\\
&\equiv \texttt{el}\,\texttt{car}\\
\texttt{int}_{\texttt{F}} &: \Pi c{:}\texttt{symbol}_{\texttt{F}}\texttt{s}\,.\,\texttt{Fun}^{\texttt{N}}\,\texttt{car}\ (\texttt{arity}_{\texttt{F}}\,c)\\
&\equiv \pi_2^3(\texttt{structure}\,M)\\
\texttt{int}_{\texttt{P}} &: \Pi p{:}\texttt{symbol}_{\texttt{P}}\,s.\,\texttt{Pred}^{\texttt{N}}\,\texttt{car}\ (\texttt{arity}_{\texttt{P}}\,p)\\
&\equiv \pi_3^3(\texttt{structure}\,M)
\end{aligned}$$

(iii) Define the type $\phi$ as

$$\begin{aligned}
\phi \equiv\ & \Pi A{:}\texttt{Set}.\\
& (\Pi c{:}\texttt{symbol}_{\texttt{F}}\,s.\,\texttt{Fun}^{\texttt{N}}\,A\,(\texttt{arity}_{\texttt{F}}\,c)) \rightarrow\\
& (\Pi p{:}\texttt{symbol}_{\texttt{P}}\,s.\,\texttt{Pred}^{\texttt{N}}\,A\,(\texttt{arity}_{\texttt{P}}\,p)) \rightarrow \texttt{PROP}\ .
\end{aligned}$$

For convenience, we define a lambda term to introduce the axioms of a structure as follows.

$$\begin{aligned}
\texttt{Axioms}_{\texttt{intro}} &: \phi \rightarrow \texttt{Axioms}\,s\\
&\equiv \lambda z{:}\phi\,\lambda str{:}\texttt{Structure}\,s.\,z\,\pi_2^3(str)\,\pi_3^3(str)
\end{aligned}$$

We need an interpretation function which, given some assignment, maps each term of the signature to an element of the carrier of a model.

3.4.5. Definition. Let $s{:}\texttt{Signature}$ be a signature, $ax{:}(\texttt{Axioms}\,s)$ some axioms for a structure of the signature. Let $M$ : $(\texttt{Model}\,ax)$ be a model satisfying the axioms.

(i) An assignment is a valuation of the variables into the carrier of $M$. We implement assignments by non-empty[4] lists. See the LEGO library for a definition of non-empty lists `neList`.

$$\begin{array}{lcl}\texttt{Assignment} & : & \texttt{Set}\\ & \equiv & \texttt{neList}\, M.\texttt{car}\end{array}$$

(ii) Define the interpretation of a term with respect to an assignment. First we deal with the case of n-tuples of terms. Let $\rho : \texttt{el}\,\texttt{Assignment}$ be an assignment.

$$\begin{array}{lcllll}\texttt{int}^{\texttt{n}}_{\rho} & : & \multicolumn{4}{l}{\Pi n{:}\texttt{nat}.\,(\texttt{terms}\,s\,n) \rightarrow (\texttt{product}\,M.\texttt{obj}\ n)}\\ & \equiv & \varepsilon_{\texttt{terms}}\ s & (\texttt{TFV}\,n) & \Longrightarrow & <\rho(n), \texttt{star}>\\ & & & (\texttt{TFC}\,f\,t) & \Longrightarrow & <(\texttt{int}_{\texttt{F}} f)(\texttt{int}^{\texttt{n}}_{\rho}\,t), \texttt{star}>\\ & & & \texttt{tnil} & \Longrightarrow & \texttt{star}\\ & & & \texttt{tcons}\,t\,l & \Longrightarrow & <\pi^2_1(\texttt{int}^{\texttt{n}}_{\rho}\,t), \texttt{int}^{\texttt{n}}_{\rho}\,l>\\ \texttt{int}_{\rho} & : & \multicolumn{4}{l}{(\texttt{term}\,s) \rightarrow (\texttt{obj}\,M)}\\ & \equiv & \multicolumn{4}{l}{\lambda t{:}\texttt{term}\,s.\,\pi^2_1(\texttt{int}^{\texttt{n}}_{\rho}\,t)}\end{array}$$

## Monoids

Defining a structure like the monoids falls into two parts. First, we define the signature. Second, we define axioms over the signature which should be satisfied to form a monoid.

3.4.6. Definition. We define the signature for monoids as two function symbols and no predicate symbols. One function symbol is used for a constant, another symbol for a binary function.

$$\begin{array}{lcl}\texttt{FnSymMN} & : & \texttt{SET}\\ & \equiv & \mu X{:}\texttt{SET}.\,(\texttt{identMN}, \texttt{opMN} : X)\\ \texttt{sigMN} & : & \texttt{Signature}\\ & \equiv & \texttt{Sig}_{\texttt{intro}}\ (\varepsilon_{\texttt{FnSymMN}}\ \texttt{identMN} \Longrightarrow \underline{0}\\ & & \qquad\qquad\qquad\qquad \texttt{opMN} \Longrightarrow \underline{2}\,)\\ & & \qquad\quad (\varepsilon_{\texttt{EmptySET}})\end{array}$$

3.4.7. Definition. (i) We need to define the notion of associativity and identity. Let $A{:}\texttt{SET}$ be a setoid, $f : (\texttt{Fun}^2\,A\,A\,A)$ a binary function over $A$ and $e : \texttt{el}\,A$ an element of $A$.

$$\begin{array}{lcl}\texttt{Associative} & : & \texttt{PROP}\\ & \equiv & \Pi x, y, z{:}\texttt{el}\,A.\, f(x, f(y, z)) = f(f(x, y), z)\\ \texttt{Identity} & : & \texttt{PROP}\\ & \equiv & (\Pi x{:}\texttt{el}\,A.\, f(e, x) = x)\ \&\ (\Pi x{:}\texttt{el}\,A.\, f(x, e) = x)\end{array}$$

[4]An assignment over $M$ is implemented by a non-empty list because we always need at least one element of the carrier of $M$. This element enables us to valuate variables outside the set of free variables of a term.

(ii) Now we are able to specify the axioms for a monoid.

$$\begin{array}{lll} \texttt{axioms}_{\texttt{MN}} & : & \texttt{Axioms sigMN} \\ & \equiv & \texttt{Axioms}_{\texttt{intro}}(\lambda A{:}\texttt{Set} \\ & & \quad \lambda F{:}(\Pi f{:}\texttt{symbol}_{\texttt{F}}\,\texttt{sigMN}.\,\texttt{Fun}^{\texttt{N}}\,A\,(\texttt{arity}_{\texttt{F}}\,f)) \\ & & \quad \lambda P{:}(\Pi p{:}\texttt{symbol}_{\texttt{P}}\,\texttt{sigMN}.\,\texttt{Pred}^{\texttt{N}}\,A\,(\texttt{arity}_{\texttt{P}}\,p)). \\ & & \quad (\texttt{Associative}\,(F\,\texttt{opMN})) \\ & & \quad\quad \& \\ & & \quad (\texttt{Identity}\,(F\,\texttt{opMN})(F\,\texttt{identMN}))) \end{array}$$

Although the definition of the axioms of monoid may look complex, the type checker will help us to build it. So for example using the `Intros` tactic, it will generate for us the types of the variables $F$ and $P$ automatically.

3.4.8. Definition. (i) Now we have specified the signature for a monoid structure and the axioms which should hold, the definition of a monoid is a simple task.

$$\begin{array}{lll} \texttt{Monoid} & : & \texttt{TYPE} \\ & \equiv & \texttt{Model axioms}_{\texttt{MN}} \end{array}$$

(ii) Also we are able to retrieve the multiplication and unit element of a monoid. Let $M$ : `Monoid` be a monoid.

$$\begin{array}{lll} 1_{\texttt{MN}} & : & \texttt{obj}\,M \\ & \equiv & \texttt{ap}\,(\texttt{int}_{\texttt{F}}\;M\,\texttt{identMN}) \\ \times_{\texttt{MN}} & : & \texttt{Fun}^2\,M.\texttt{car}\;M.\texttt{car}\;M.\texttt{car} \\ & \equiv & \texttt{int}_{\texttt{F}}\;M\,\texttt{opMN} \\ \texttt{assoc}_{\texttt{MN}} & : & \texttt{Associative}\,\times_{\texttt{MN}} \\ & \equiv & \texttt{fst}\;M.\texttt{axioms} \\ \texttt{ident}_{\texttt{MN}} & : & \texttt{Identity}\;\times_{\texttt{MN}}\,1_{\texttt{MN}} \\ & \equiv & \texttt{snd}\;M.\texttt{axioms} \end{array}$$

In our library we developed theory for monoids, groups, rings, up to fields. We introduced ordered field as follows.

3.4.9. Definition (Ordered field). Let $A$ be a set, 0 and 1 be two elements of $A$, `inv` and `recip` be unary functions over $A$ (for additive inverse and reciprocal), $+$ and $\times$ be binary functions over $A$, and `pos` be a predicate over $A$. $F = <A, 0, \texttt{inv}, +, 1, \texttt{recip}, \times, 1, \texttt{pos}>$ is an ordered field if it satisfies the field axioms, and moreover for every $x$, $y$ in $A$:

$$\begin{array}{l} \texttt{pos}(\texttt{inv}(x)) \vee \texttt{pos}(x) \\ \texttt{pos}(x) \to \texttt{pos}(\texttt{inv}(x)) \to x = 0 \\ \texttt{pos}(x) \to \texttt{pos}(y) \to \texttt{pos}(x + y) \\ \texttt{pos}(x) \to \texttt{pos}(y) \to \texttt{pos}(x \times y) \end{array}$$

Also we defined the notion of homomorphisms and subgroups over structures and showed some basic properties concerning them. The reader is referred to the LEGO library (Appendix A.3) for more details.

## 3.5 Towards Analysis

In this section we formalize a few number systems up to the field of complex numbers. We start with the natural numbers. They are defined by a construction. Next we build the integers from the naturals. Then using the integers, we construct the rationals. The real number system is formalized by an axiomatization. And finally the complex numbers are constructed out of the reals. Schematically:

$$\begin{tikzcd}
\texttt{nat} \arrow[d] & \mathbb{R} \arrow[d] \arrow[dr] & \\
\mathbb{N} \arrow[d] & \mathbb{C} \arrow[dr] & \mathbb{R}[X] \\
\mathbb{Z} \arrow[d] & & \mathbb{C}[X] \\
\mathbb{Q} & &
\end{tikzcd}$$

where `nat` stands for the inductively defined natural numbers. Polynomial rings will be introduced in Section 4.4.1.

### 3.5.1 Natural numbers

Let us first formulate a list of postulates which should be satisfied by any representation of the natural numbers.

3.5.1. Definition (The Peano postulates). Given a signature

$$<\texttt{N}, \texttt{0}, \texttt{S}, +, \times> \ .$$

The Peano postulates consist of the following list of sentences.

$$\begin{array}{lll}
\texttt{PP}_1 & : & \forall x \in \texttt{N}.(\texttt{S}\,x) \neq \texttt{0}, \\
\texttt{PP}_2 & : & \forall x, y \in \texttt{N}.((\texttt{S}\,x) = (\texttt{S}\,y)) \to (x = y), \\
\texttt{PP}_3 & : & \forall x \in \texttt{N}.(x + \texttt{0}) = x, \\
\texttt{PP}_4 & : & \forall x, y \in \texttt{N}.(x + (\texttt{S}\,y)) = (\texttt{S}\,(x + y)), \\
\texttt{PP}_5 & : & \forall x \in \texttt{N}.(x \times \texttt{0}) = \texttt{0}, \\
\texttt{PP}_6 & : & \forall x, y \in \texttt{N}.(x \times (\texttt{S}\,y)) = (x + (x \times y)), \\
\texttt{PP}_7 & : & \forall \phi \in \texttt{N} \to \texttt{PROP}.(\phi\,\texttt{0}) \to \\
 & & \qquad\qquad (\forall x \in \texttt{N}.(\phi\,x) \to \phi\,(\texttt{S}\,x)) \to \\
 & & \qquad\qquad (\forall x \in \texttt{N}.\phi\,x)
\end{array}$$

The formalization of natural numbers does not pose any real difficulties. Depending on the type system used, several approaches are possible. We will review these briefly. For all approaches we only define natural numbers as a *type*. We could transform them into a setoid by adding an appropriate equivalence relation like Leibniz equality.

#### Axiomatic

When we work in a second-order predicate logic, the following context may be used also as an axiomatization of the Peano postulates.

3.5.2. Definition. Define the context of Peano axioms.

$$\begin{array}{lcll}
\mathtt{PA}_{\mathrm{Ax}} & \equiv & \mathtt{nat}_A & : \mathtt{SET} \\
& & \mathtt{0} & : \mathtt{nat}_A, \\
& & \mathtt{S} & : \mathtt{nat}_A {\to} \mathtt{nat}_A, \\
& & \mathtt{PA}_1 & : \Pi x{:}\mathtt{nat}_A.\, \neg((\mathtt{S}\, x) = \mathtt{0}), \\
& & \mathtt{PA}_2 & : \Pi x, y{:}\mathtt{nat}_A.\, ((\mathtt{S}\, x) = (\mathtt{S}\, y)) {\to} (x = y), \\
& & \mathtt{PA}_7 & : \Pi \phi{:}\mathtt{nat}_A {\to} \mathtt{PROP}.\, (\phi\, \mathtt{0}) {\to} \\
& & & \qquad\qquad (\Pi x{:}\mathtt{nat}_A.\, (\phi\, x) {\to} \phi\, (\mathtt{S}\, x)) {\to} \\
& & & \qquad\qquad (\Pi x{:}\mathtt{nat}_A.\, \phi\, x)
\end{array}$$

where = stands for Leibniz equality.

By definition, Leibniz equality is compatible with $\mathtt{S}$. Because we work in second-order predicative logic, we are able to define primitive recursive functions like predecessor, addition or multiplication by some relation $R$. After we have shown that this $R$ is a graph, we feel safe to extend the context $\mathtt{PA}_{\mathrm{Ax}}$ with a fresh variable $\mathtt{f}_R$ together with a defining formula of $\mathtt{f}_R$ for $R$.

3.5.3. Definition. We introduce the addition as follows.

(i) Define the relation $\mathtt{plus}_{\mathtt{rel}}$ and prove it is a graph.

$$\begin{array}{lcl}
\mathtt{plus}_{\mathtt{rel}} & : & \mathtt{nat}_A {\to} \mathtt{nat}_A {\to} \mathtt{nat}_A {\to} \mathtt{PROP} \\
& \equiv & \lambda x, y, x{:}\mathtt{nat}_A \\
& & \forall P{:}\mathtt{nat}_A {\to} \mathtt{nat}_A {\to} \mathtt{nat}_A {\to} \mathtt{PROP}. \\
& & (\forall x{:}\mathtt{nat}_A.\, P\, x\, \mathtt{0}\, x) {\to} \\
& & (\forall x, y, z{:}\mathtt{nat}_A.\, (P\, x\, y\, z) {\to} P\, x\, (\mathtt{S}\, y) (\mathtt{S}\, z)) {\to} \\
& & (P\, x\, y\, z) \\
\mathtt{plus}_{\mathtt{graph}} & : & \forall x, y{:}\mathtt{nat}_A\, \exists! z{:}\mathtt{nat}_A.\, \mathtt{plus}_{\mathtt{rel}}\, x\, y\, z \\
& \equiv & \llcorner \ldots \lrcorner
\end{array}$$

(ii) Extend the context $\mathtt{PA}_{\mathrm{Ax}}$ with

$$\begin{array}{lcll}
\Gamma_{\mathtt{plus}} & \equiv & \mathtt{plus} & : \mathtt{nat}_A {\to} \mathtt{nat}_A {\to} \mathtt{nat}_A, \\
& & \mathtt{PA}_{3,4} & : \forall x, y, z{:}\mathtt{nat}_A.\, (\mathtt{plus}_{\mathtt{rel}}\, x\, y\, z) \leftrightarrow ((\mathtt{plus}\, x\, y) = z) \ .
\end{array}$$

It is easy to show that the addition thus defined is a commutative associative function for which $x + 0 = x$ and $x + (\mathtt{S}\, y) = \mathtt{S}\, (x + y)$ holds for any $x, y : \mathtt{nat}_A$. In a similar way multiplication can be introduced.

If we assume the strong axiom of unique choice $\mathtt{AUC}_\Sigma$ (see Section 3.2.2), there is no need to extend the context $\mathtt{PA}_{\mathrm{Ax}}$. This is because with $\mathtt{AUC}_\Sigma$ we can *define* $\mathtt{plus}$ such that $\mathtt{PA}_{3,4}$ holds.

## Encodings

Besides an axiomatization we can also realize the Peano postulates by an construction. Often it is desirable to keep the context of auxiliary axioms as small as possible. This is especially important when we wish to develop meta-theory like consistency. In $\lambda$-calculus, the Church-numeral encoding can be used to construct the natural numbers. The Peano postulates with respect to the Church numerals turn out to be derivable from a small auxiliary context.

3.5.4. Definition. (i) Define the polymorphic type of Church numerals.

$$\begin{array}{lcl}
\mathtt{nat}_C & : & \mathtt{SET} \\
 & \equiv & \Pi\alpha{:}\mathtt{PROP}.\, \alpha\to(\alpha\to\alpha)\to\alpha \\
\mathtt{0} & : & \mathtt{nat}_C \\
 & \equiv & \lambda\alpha{:}\mathtt{PROP}\, \lambda x{:}\alpha\, \lambda f{:}\alpha\to\alpha.\, x \\
\mathtt{S} & : & \mathtt{nat}_C\to\mathtt{nat}_C \\
 & \equiv & \lambda n{:}\mathtt{nat}_C\, \lambda x{:}\alpha\, \lambda f{:}\alpha\to\alpha.\, f\,(n\,\alpha\,x\,f)
\end{array}$$

(ii) Define a context $\mathtt{PA}_{\mathrm{Ch}}$ from which the Peano postulates are derivable.

$$\begin{array}{lcl}
\mathtt{ind} & : & \mathtt{nat}_C\to\mathtt{PROP} \\
 & \equiv & \lambda n{:}\mathtt{nat}_C\, \Pi\phi{:}\mathtt{nat}_C\to\mathtt{PROP}.\; (\phi\,\mathtt{0})\to \\
 & & \qquad (\Pi n{:}\mathtt{nat}_C.\, (\phi\, n)\to\phi\,(\mathtt{S}\, n))\to \\
 & & \qquad (\phi\, n) \\
\mathtt{PA}_{\mathrm{Ch}} & \equiv & \mathtt{PA}_0 : \neg((\mathtt{S}\,\mathtt{0}) = \mathtt{0}), \\
 & & \mathtt{PA}_7 : \Pi n{:}\mathtt{nat}_C.\, \mathtt{ind}\, n
\end{array}$$

The system needed in this case is second-order predicate logic. Also in this system we take Leibniz equality for equality over $\mathtt{nat}_C$.

3.5.5. Lemma. *From the context* $\mathtt{PA}_{\mathrm{Ch}}$*, the Peano postulates for the Church numerals are derivable.*

Proof. The postulate $\mathtt{PP}_7$ is satisfied by $\mathtt{PA}_7$. Define pairing $<,>$ and the first and second projection. Next we define the predecessor following Kleene as

$$\begin{array}{lcl}
\mathtt{P}^- & : & \mathtt{nat}_C\to\mathtt{nat}_C \\
 & \equiv & \lambda n{:}\mathtt{nat}_C.\, (n\,(\mathtt{nat}_C\times\mathtt{nat}_C)\,(\lambda z{:}\mathtt{nat}_C\times\mathtt{nat}_C.\, <z_2, \mathtt{S}\, z_1>)\, <\mathtt{0},\mathtt{0}>)_1 \ .
\end{array}$$

Then we show using the induction scheme $\mathtt{PA}_7$ that $\mathtt{P}^-\,(\mathtt{S}\,x) = x$ for all $x : \mathtt{nat}_C$. For technical reasons, we also need to show that $\mathtt{P}^-\,\mathtt{0} = \mathtt{0}$.

(i) Suppose $(\mathtt{S}\,x) = (\mathtt{S}\,y)$ for some $x, y : \mathtt{nat}_C$. Then we have $\mathtt{P}^-\,(\mathtt{S}\,x) = \mathtt{P}^-\,(\mathtt{S}\,y)$, and hence $x = y$. So we have $\mathtt{PP}_2$.

(ii) To prove $\mathtt{PP}_1$, we apply the induction axiom $\mathtt{PA}_3$. The base case is proven immediately by $\mathtt{PA}_0$. For the induction step, suppose $\mathtt{S}\,(\mathtt{S}\,x) = \mathtt{0}$ for some $x : \mathtt{nat}_C$. Then $\mathtt{P}^-\,(\mathtt{S}\,(\mathtt{S}\,x)) = \mathtt{P}^-\,\mathtt{0}$. So $\mathtt{S}\,x = \mathtt{0}$, and by the induction hypothesis we are done.

(iii) Define addition as follows.

$$\begin{array}{lcl}
\mathtt{plus} & : & \mathtt{nat}_C \to \mathtt{nat}_C \to \mathtt{nat}_C \\
 & \equiv & \lambda n, m{:}\mathtt{nat}_C\, \lambda\alpha{:}\mathtt{PROP}\, \lambda x{:}\alpha\, \lambda f{:}\alpha\to\alpha.\, m\,(n\,x\,f)\,f
\end{array}$$

Using $\mathtt{PA}_2$ and $\mathtt{PA}_7$ it is easy to prove $\mathtt{PA}_3$. $\mathtt{PA}_4$ holds directly by reflexivity of equality.

(iv) Define multiplication as follows.

$$\begin{array}{lcl}
\mathtt{times} & : & \mathtt{nat}_C \to \mathtt{nat}_C \to \mathtt{nat}_C \\
 & \equiv & \lambda n, m{:}\mathtt{nat}_C\, \lambda\alpha{:}\mathtt{PROP}\, \lambda x{:}\alpha\, \lambda f{:}\alpha\to\alpha.\, m\, x(\lambda x{:}\alpha.\, n\,x\,f)
\end{array}$$

$\mathtt{PA}_5$ and $\mathtt{PA}_6$ hold directly by reflexivity of equality.

Also other functions like exponentiation can be defined directly because Church numerals support iteration.

We could strip the context $\mathtt{PA}_{\mathrm{Ch}}$ to only $\mathtt{PA}_0$ if we relativize every quantification over natural numbers with the `ind` predicate. So for each formula $\phi(x)$ the translation of the expression $\forall x{:}\mathbb{N}.\,\phi(x)$ will be

$$\ulcorner\forall x{:}\mathbb{N}.\,\phi(x)\urcorner \quad = \quad \Pi x{:}\mathtt{nat}_C.\,(\mathtt{ind}\,x)\to\ulcorner\phi(x)\urcorner\ .$$

Using this mechanism, the assumption that the Church numeral `0` differs from the Church numeral `S 0` suffices to prove the Peano postulates. See (Ruys 1991) for details.

3.5.6. Remark. In most textbooks, Church numerals are defined as

$$\mathtt{nat}'_C \quad \equiv \quad \Pi\alpha{:}\mathtt{PROP}.\,(\alpha\to\alpha)\to\alpha\to\alpha\ .$$

Obviously, every closed inhabitant of $\mathtt{nat}_C$ has a corresponding term in $\mathtt{nat}'_C$. However, the term $\lambda\alpha{:}\mathtt{PROP}\,\lambda f{:}\alpha\to\alpha.\,f$, an $\eta$-reduct of the Church numeral `1`, lives in $\mathtt{nat}'_C$ but has no counterpart in $\mathtt{nat}_C$. Because $\mathtt{nat}_C$ has fewer closed inhabitants, we prefer the definition $\mathtt{nat}_C$ over $\mathtt{nat}'_C$.

**Inductively**

If we have inductive types, we can get rid of the context $\Gamma_{\mathtt{nat}}$ completely. Inductively defined natural numbers have the advantage that by definition they form the smallest type in which `0` lives and which is closed under `S`.

3.5.7. Definition. We recall `nat` from Definition 3.1.1 and use Leibniz equality to obtain the set of natural numbers.

$$\begin{array}{lcl}
\mathtt{nat} & : & \mathtt{SET} \\
 & \equiv & \mu\alpha{:}\mathtt{SET}.\,(\mathtt{zero} : \alpha, \mathtt{S}^{\mathtt{N}} : \alpha\to\alpha) \\
\mathbb{N} & : & \mathtt{Set} \\
 & \equiv & <\mathtt{nat}, =_L \llcorner\mathtt{nat}, \llcorner\ldots\lrcorner>
\end{array}$$

The system needed is second-order logic with inductive types. The reader is referred to Section 4.2 for more formalizations concerning natural numbers.

3.5.8. Example. As an example, we present the definition of addition and multiplication.

$$\begin{array}{lcllcl}
\_+\_ & : & \mathtt{nat}\to\mathtt{nat}\to\mathtt{nat} & & & \\
 & \equiv & \lambda x{:}\mathtt{nat}.\,\varepsilon_{\mathtt{nat}} & \mathtt{zero} & \Longrightarrow & x \\
 & & & \mathtt{S}^{\mathtt{N}}\,y & \Longrightarrow & \mathtt{S}^{\mathtt{N}}\,(x+y) \\
\_\times\_ & : & \mathtt{nat}\to\mathtt{nat}\to\mathtt{nat} & & & \\
 & \equiv & \lambda x{:}\mathtt{nat}.\,\varepsilon_{\mathtt{nat}} & \mathtt{zero} & \Longrightarrow & \mathtt{zero} \\
 & & & \mathtt{S}^{\mathtt{N}}\,y & \Longrightarrow & x+(x\times y)
\end{array}$$

**Binary**

Formalizations of the natural numbers based on Peano arithmetic are inadequate for real world applications. For example, in our case study 4.2 it is impossible to compute the seventh prime number using inductively defined naturals within reasonable time. The essence of the problem of Peano numbers is that it is a unary representation. Suppose that $f$ is a function which is defined by iteration on its argument. In most cases, the type checker needs order $n$ reduction steps to compute the value of $f(\mathtt{S}^{\,n}(\mathtt{0}))$. A possible solution is to use a binary representation. In (Huisman 1997) an efficient formalization of binary natural numbers is given.

## 3.5.2 Integers

From the natural numbers, integers can be defined as a quotient set over $\mathbb{N} \times \mathbb{N}$.

3.5.9. Definition. (i) An integer is a tuple of natural numbers. Two integers $x = {<}x_1, x_2{>}$ and $y = {<}y_1, y_2{>}$ are equal if and only if $x_1 - x_2 = y_1 - y_2$.

$$\begin{array}{lcl}
\texttt{int} & : & \texttt{SET} \\
 & \equiv & \texttt{prod}\,\texttt{nat}\,\texttt{nat} \\
\texttt{Eq}_{\texttt{int}} & : & \texttt{int}{\to}\texttt{int}{\to}\texttt{PROP} \\
 & \equiv & \lambda x, y{:}\texttt{int}.\, x_1 + y_2 = x_2 + y_1 \\
\mathbb{Z} & : & \texttt{Set} \\
 & \equiv & {<}\texttt{int}, \texttt{Eq}_{\texttt{int}}, \llcorner \ldots \lrcorner{>}
\end{array}$$

(ii) We define zero, the identity, negation, addition and multiplication as follows.

$$\begin{array}{lcl}
\texttt{0}_{\texttt{Z}} & : & \texttt{el}\,\mathbb{Z} \\
 & \equiv & {<}\underline{0}, \underline{0}{>} \\
\texttt{1}_{\texttt{Z}} & : & \texttt{el}\,\mathbb{Z} \\
 & \equiv & {<}\underline{1}, \underline{0}{>} \\
\texttt{neg}_{\texttt{Z}} & : & \mathbb{Z}{\Rightarrow}\mathbb{Z} \\
 & \equiv & {<}\lambda x : \texttt{el}\,\mathbb{Z}. {<}x_2, x_1{>}, \llcorner \ldots \lrcorner{>} \\
\texttt{add}_{\texttt{Z}} & : & (\mathbb{Z} \times \mathbb{Z}){\Rightarrow}\mathbb{Z} \\
 & \equiv & {<}\lambda x, y : \texttt{el}\,\mathbb{Z}. {<}x_1 + y_1, x_2 + y_2{>}, \llcorner \ldots \lrcorner{>} \\
\texttt{mult}_{\texttt{Z}} & : & (\mathbb{Z} \times \mathbb{Z}){\Rightarrow}\mathbb{Z} \\
 & \equiv & {<}\lambda x, y : \texttt{el}\,\mathbb{Z}. {<}x_1 \times y_1 + x_2 \times y_2, x_1 \times y_2 + x_2 \times y_1{>}, \llcorner \ldots \lrcorner{>}
\end{array}$$

3.5.10. Lemma. *The set $\mathbb{Z}$ forms a commutative ring with respect to the operators defined in Definition 3.5.9.*

Proof. Trivial.

3.5.11. Remark. We did not need the integers nor the rationals in our case studies. So we have not formalized the lemmas in this section and the next.

## 3.5.3 Rational numbers

Following a similar scheme, we can define the rationals.

3.5.12. Definition. A rational is a tuple of an integer and a natural number. Two rationals $x = {<}x_1, x_2{>}$ and $y = {<}y_1, y_2{>}$ are equal if and only if $x_1 \times (y_2 + 1) = y_1 \times (x_2 + 1)$.

$$\begin{array}{lcl}
\texttt{rat} & : & \texttt{SET} \\
 & \equiv & \texttt{prod}\,\texttt{int}\,\texttt{nat} \\
\texttt{S}^{\texttt{NZ}} & : & \texttt{nat} \to \texttt{int} \\
 & \equiv & \lambda x{:}\texttt{nat}. {<}\texttt{S}^{\texttt{N}}\, x, \underline{0}{>} \\
\texttt{Eq}_{\texttt{rat}} & : & \texttt{rat}{\to}\texttt{rat}{\to}\texttt{PROP} \\
 & \equiv & \lambda x, y{:}\texttt{rat}.\, x_1 \times (\texttt{S}^{\texttt{NZ}}\, y_2) = y_1 \times (\texttt{S}^{\texttt{NZ}}\, x_2) \\
\mathbb{Q} & : & \texttt{Set} \\
 & \equiv & {<}\texttt{rat}, \texttt{Eq}_{\texttt{rat}}, \llcorner \ldots \lrcorner{>}
\end{array}$$

After defining the constants zero and one in $\mathbb{Q}$, and the negation, the inverse function, and the addition and multiplication functions over $\mathbb{Q}$ in a proper way, we can show that $\mathbb{Q}$ forms a field.

### 3.5.4 Real numbers

One approach to introduce the real number system is by a construction, for example by Cauchy sequences. This is worked out by (Jones 1991, Elbers 1993). Another approach is to introduce the reals axiomatically. Obviously this is much less work than a full construction. Because we will assume decidability of equality of real numbers anyway, the wish to be constructive is not so strong anymore.

One possible axiomatization of the real numbers is as an Archimedean ordered field which is Cauchy complete. See (Dieudonné 1960) for more details.

3.5.13. Definition. The real number system $\mathbb{R}$ is a set for which the following conditions hold.

(i) $\mathbb{R}$ is an *ordered field.*

(ii) $\mathbb{R}$ satisfies the *axiom of Archimedes*: for any pair $x, y$ of real numbers such that $0 < x, 0 \leq y$, there is an integer $n$ such that $y \leq n \times x$.

(iii) $\mathbb{R}$ satisfies the *axiom of nested intervals*: Given a sequence $([a_n, b_n])$ of closed intervals such that $a_n \leq a_{n+1}$ and $b_{n+1} \leq b_n$ for every $n \in \mathbb{N}$, the intersection of that sequence is not empty.

Another equivalent axiomatization of the real number system is by using Dedekind cuts. See (Huntington 1955) for more details on the Dedekind postulates.

3.5.14. Definition. The real number system $\mathbb{R}$ is set such that the following conditions hold.

(i) $\mathbb{R}$ is an *ordered field.*

(ii) $\mathbb{R}$ satisfies the *Dedekind postulate*: every non-empty bounded subset of $\mathbb{R}$ has a supremum.

We have chosen to use a weaker axiomatization, namely that of a real closed field. The reason for this decision was pragmatic. First, in our case studies, the axiomatization by a real closed field was strong enough to develop all theories. Furthermore, it would take quite some effort to prove the axioms of a real closed field from one of the two stronger axiomatizations. As a positive spin-off, all results for $\mathbb{R}$ in our work also hold for algebraic complex numbers.

On the other hand we strengthen our axiomatization by adding the assumption that the set of real numbers is *discrete.* In other words, we assume that the equality over reals is decidable. So we work in a classical setting. In fact, this is the only classical assumption we will make in our LEGO library.

3.5.15. Definition. The real number system $\mathbb{R}$ is defined as a real closed field. So $\mathbb{R}$ is

(i) a discrete ordered field,

(ii) such that every polynomial in $\mathbb{R}$ of odd degree has a root,

(iii) and which has a square root function over $\mathbb{R}$.

3.5.16. Example. Define the unary function *absolute value* over real numbers. We could do this using a definition by case distinction on the sign of the argument. Instead, we choose to use the square root because it more closely resembles the definition of the absolute value function in $\mathbb{C}$. Furthermore, we define a sign function over the reals.

$$\begin{array}{lll}
\mathtt{Abs_R} & : & \mathbb{R}{\Rightarrow}\mathbb{R} \\
 & \equiv & <\lambda x{:}\mathtt{el}\,\mathbb{R}.\,\mathtt{Sqrt_R}\,(\mathtt{times_R}\,(x,x)),\llcorner\ldots\lrcorner> \\
\mathtt{Sign_R} & : & \mathbb{R}{\Rightarrow}\mathbb{R} \\
 & \equiv & <\lambda x{:}\mathtt{el}\,\mathbb{R}.\,\mathtt{select}\,(\llcorner\ldots\lrcorner : x<0 \vee x \geq 0)\,(-1)\,1,\llcorner\ldots\lrcorner>
\end{array}$$

### 3.5.5 Complex numbers

From the reals we can construct the field of complex numbers. The approach using polar coordinates involves trigonometry. Because it would be quite a lot of effort to formalize trigonometry, we choose to define the complex numbers by ordered pairs of real numbers (Hamilton 1837).

3.5.17. Definition. (i) The complex numbers are defined as Cartesian coordinates of real numbers. A complex number consist of a real and an imaginary part. Two complex numbers are equal if their real and imaginary parts are equal as real numbers.

$$\begin{array}{lll}
\mathbb{C} & : & \mathtt{Set} \\
 & \equiv & \mathtt{Prod}\,\mathbb{R}\,\mathbb{R} \\
\mathtt{0_C} & : & \mathtt{el}\,\mathbb{C} \\
 & \equiv & <\mathtt{0_R},\mathtt{0_R}> \\
\mathtt{1_C} & : & \mathtt{el}\,\mathbb{C} \\
 & \equiv & <\mathtt{1_R},\mathtt{0_R}> \\
i & : & \mathtt{el}\,\mathbb{C} \\
 & \equiv & <\mathtt{0_R},\mathtt{1_R}>
\end{array}$$

(ii) We define negation, addition and multiplication over complex numbers as follows.

$$\begin{array}{lll}
\mathtt{Neg_C} & : & \mathbb{C}{\Rightarrow}\mathbb{C} \\
 & \equiv & <\lambda x{:}\mathtt{el}\,\mathbb{C}.\,<\mathtt{Neg_R}\,(x_1),\mathtt{Neg_R}\,(x_2)>,\llcorner\ldots\lrcorner> \\
\mathtt{Plus_C} & : & (\mathbb{C}\times\mathbb{C}){\Rightarrow}\mathbb{C} \\
 & \equiv & <\lambda x,y{:}\mathtt{el}\,\mathbb{C}.\,<\mathtt{Plus_R}\,(x_1,y_1),\mathtt{Plus_R}\,(x_2,y_2)>,\llcorner\ldots\lrcorner> \\
\mathtt{Minus_C} & : & (\mathbb{C}\times\mathbb{C}){\Rightarrow}\mathbb{C} \\
 & \equiv & <\lambda x,y{:}\mathtt{el}\,\mathbb{C}.\,\mathtt{Plus_C}\,(x,\mathtt{Neg_C}\,(y)),\llcorner\ldots\lrcorner> \\
\mathtt{Mult_C} & : & (\mathbb{C}\times\mathbb{C}){\Rightarrow}\mathbb{C} \\
 & \equiv & <\lambda x,y{:}\mathtt{el}\,\mathbb{C}.\,<\mathtt{Minus_R}\,(\mathtt{Mult_R}\,(x_1,x_2),\mathtt{Mult_R}\,(y_1,y_2)), \\
 & & \qquad\qquad \mathtt{Plus_R}\,(\mathtt{Mult_R}\,(x_1,y_2),\mathtt{Mult_R}\,(y_1,x_2))>,\llcorner\ldots\lrcorner> \\
\mathtt{Recip_C} & : & \mathbb{C}{\Rightarrow}\mathbb{C} \\
 & \equiv & <\lambda x{:}\mathtt{el}\,\mathbb{C}.\,<\mathtt{Div_R}\,(x,\mathtt{Add_R}\,(\mathtt{Square_R}\,(x_1),\mathtt{Square_R}\,(x_2))), \\
 & & \qquad\qquad \mathtt{Div_R}\,(\mathtt{Neg_R}\,(y), \\
 & & \qquad\qquad\qquad \mathtt{Add_R}\,(\mathtt{Square_R}\,(x_1),\mathtt{Square_R}\,(x_2)))>, \\
 & & \quad \llcorner\ldots\lrcorner>
\end{array}$$

3.5.18. Lemma. *The set $\mathbb{C}$ forms a commutative field of complex numbers with respect to the terms defined in Definition 3.5.17.*

PROOF. For a proof, see the LEGO library.

3.5.19. EXAMPLE. We can define for example the scalar product and absolute value for complex numbers.

$$\begin{array}{lcl}
\texttt{ScProdC} & : & (\mathbb{C} \times \mathbb{C}) \Rightarrow \mathbb{R} \\
 & \equiv & <\lambda x, y{:}\texttt{el}\,\mathbb{C}.\, \texttt{Plus}_{\texttt{R}}\,((\texttt{Times}_{\texttt{R}}\,(x.1, y.1)), (\texttt{Times}_{\texttt{R}}\,(x.2, y.2))), \\
 & & \llcorner\ldots\lrcorner> \\
\texttt{Abs}_{\texttt{C}} & : & \mathbb{C} \Rightarrow \mathbb{R} \\
 & \equiv & <\lambda z{:}\texttt{el}\,\mathbb{C}.\, \texttt{Sqrt}_{\texttt{R}}\,(\texttt{ScProd}_{\texttt{C}}\,(z, z)), \llcorner\ldots\lrcorner>
\end{array}$$

The reader is referred to the LEGO library for a development of the theory of complex numbers.

# Chapter 4

# Case Studies

Formalizing mathematics may be separated into three distinguished actions. Suppose we wish to prove a certain theorem. First we need *logic*: we need a framework in which we can reason. We need to know which rules we accept and which we do not. Next we need to *define* the mathematical objects we wish to reason about. Probably one already has a library of basic concepts like the ones defined in Chapter 3. But almost always one needs to extend it. And a third part of doing mathematics is to make *computations*.

In this chapter we present several case studies to make clear how these actions are performed in a proof-checker based on type theory. Note that here we study the *methodology* used in proofs. Hence none of the proofs presented here are mathematically original work. The first case study shows how one can use a type checker to reason. We will formalize a proof of a purely logical statement called the 'Drinkers Principle'. The next case study is a formalization of Euclid's theorem of the existence of infinitely many primes. Here we will show how one could automatically compute prime numbers from a constructive proof. The next formalization deals with a proof where a lot of equational reasoning is involved. The last case study is the most comprehensive one. It describes our attempt to give a fully formalized proof of the fundamental theorem of algebra. Here every aspect of formalization comes in, the biggest effort being the development of all notions and lemmas the theorem is built on.

## 4.1 Drinkers Principle

Smullyan's Drinkers Principle states that in every bar you can always point out someone with the following property: everyone is drunk whenever he is. This theorem is interesting because it is a purely logical sentence. Also the proof of this classically true statement is non-trivial. We will compare a proof in 'my best mathematical style' (Barendregt 1996) to a formal proof in natural deduction style, and to a proof in type theory.

4.1.1. Theorem. *Let* Cafe *be a non-empty set,* Drunk *a predicate over* Cafe.

$$\exists x[\mathrm{Drunk}(x) \to \forall y[\mathrm{Drunk}(y)]] \ .$$

Proof (Informal). By case distinction. If everyone is drunk, then the theorem is trivially true. Otherwise there will be somebody who is not drunk. This person makes the implication true.

We could not avoid to make use of an application of excluded middle. Excluded middle is logically equivalent to double negation, which is a rule only accepted in classical logic. When we formalize the proof, we will see that we actually need two applications of excluded middle. This appears in the next formal proof in natural deduction style.

PROOF (natural deduction). Suppose we have a signature consisting of a set $C$, a predicate $D$ and a constant $a$. The constant formalizes that $C$ is a non-empty set[1]. Define the predicate $\phi(x)$ as $D(x) \to \forall y[D(y)]$. Now we have to prove $\exists x[\phi(x)]$. Recall that $\neg A$ is defined as $A \to \bot$, and that from $\bot$ follows anything we wish ($\bot$-elimination).

$$\dfrac{\dfrac{\dfrac{\dfrac{D(a)^{5} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg D(z)^{8} \quad D(z)^{11}}{\bot}\,13}{\forall y[D(y)]}\,12}{\phi(z)}\,11}{\exists x[\phi(x)]}\,10 \qquad \neg\exists x[\phi(x)]^{2}}{\bot}\,9}{\neg\neg D(z)}\,8}{D(z)}\,7}{\forall y[D(y)]}\,6}{\phi(a)}\,5}{\exists x[\phi(x)]}\,4}{\bot}\,3}{\neg\neg\exists x[\phi(x)]}\,2}{\exists x[\phi(x)]}\,1$$

The emphasized steps 1 and 7 indicate an application of double negation.

In an interactive proof development tool like LEGO, a similar proof would be generated by use of tactics called *Refine* and *Intros*. The tactic *Refine* corresponds to modus ponens, and the tactic *Intros* to $\to$ introduction or to generalization. In contrary to mathematical practice, proofs are built by *backward* reasoning. That is why we numbered the natural-deduction proof starting from the last line and up.

PROOF (type checker). Let Dn $: \forall P{:}\texttt{PROP}.\, \neg\neg P \to P$ be the axiom of double negation. Let $C : \texttt{SET}$, assume we have an $a : C$ which makes $C$ non-empty, and let $D : C \to \texttt{PROP}$ be a predicate over $C$. Define a predicate $\phi$ as $\lambda x{:}C.\,(D\,x) \to \Pi y{:}C.\,D\,y$. Starting from our initial goal $\mathrm{Ex}\,\phi$, we execute the following LEGO tactics.

[1]Also we could formalize 'the set $A$ is non-empty' by assuming $\exists a \in A[\phi]$ for some arbitrary provable proposition $\phi$. This way, we do not have to extend the signature with a constant. In type theory both approaches are provably equivalent.

| # | Goal | Tactics | Rule |
|---|---|---|---|
| 1. | $\mathrm{Ex}\,\phi$ | Refine Dn ; | double negation |
| 2. | $\neg\neg\mathrm{Ex}\,\phi$ | Intros $H_2$; | $\rightarrow$ introduction |
| 3. | $\perp$ | Refine $H_2$; | modus ponens |
| 4. | $\mathrm{Ex}\,\phi$ | Refine ExIntro; Refine $a$; | $\exists$ introduction |
| 5. | $\phi\, a$ | Intros $H_5$; | $\rightarrow$ introduction |
| 6. | $\Pi\, y.D\, y$ | Intros z; | $\forall$ generalization |
| 7. | $D\, y$ | Refine Dn ; | double negation |
| 8. | $\neg\neg(D\, y)$ | Intros $H_8$; | $\rightarrow$ introduction |
| 9. | $\perp$ | Refine $H_2$; | modus ponens |
| 10. | $\mathrm{Ex}\,\phi$ | Refine ExIntro; Refine $z$; | $\exists$ introduction |
| 11. | $\phi\, y$ | Intros $H_{11}$; | $\rightarrow$ introduction |
| 12. | $\Pi\, y.D\, y$ | Refine ExFalso; | ex falso sequitur quodlibet |
| 13. | $\perp$ | Refine $H_8$; Refine $H_{11}$; | modus ponens |

See Section A.3.3 for an exact screen dump of the LEGO session above.

Of course, this is only a proof if we believe that the type checker only accepts tactics which leads to a proof of a true statement. The type checker generates a typed lambda-term as proof object. The type corresponds to the theorem, and the term corresponds to a proof. For a 'second opinion', any type checker could be used to verify that the term has indeed the indicated type.

Proof (Type theory).

$$\begin{array}{l}
\mathrm{Dn}\ (\mathrm{Ex}\,\phi)\\
\quad(\lambda H_2{:}\neg(\mathrm{Ex}\,\phi).\\
\quad\quad H_2\\
\quad\quad\quad(\mathrm{ExIntro}\,\phi\\
\quad\quad\quad\quad(\lambda H_5{:}D\,a.\\
\quad\quad\quad\quad\quad\lambda z{:}C.\\
\quad\quad\quad\quad\quad\quad\mathrm{Dn}\,(D\,z)\\
\quad\quad\quad\quad\quad\quad\quad(\lambda H_8{:}\neg(D\,z).\\
\quad\quad\quad\quad\quad\quad\quad\quad H_2\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad(\mathrm{ExIntro}\,\phi\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad(\lambda H_{11}{:}D\,z.\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\texttt{ExFalso}\,(\Pi y{:}C.\,D\,y)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad H_8\,H_{11})))))))\\
:\mathrm{Ex}\,\phi
\end{array}$$

The lambda term above is unreadable for human eyes. A type checker should type it by $\exists x[\phi(x)]$. So if we believe the correctness of the type checker, we will regard the term as a proof of the Drinkers Principle. Also the list of tactics for the type checker we presented is hard to grasp. The way an *interactive* type checker works is that for every step it will present us a list of assumptions and a subgoal to prove. This way, it is clear what to prove and which tactics to use. Also the type checker will do book-keeping like discharging local variables for us. Furthermore during a proof session, it will help us to focus on which subgoals still need to be proven.

If we look at the way an interactive type checker works, and how proofs are constructed in mathematical practice, one important difference comes up. The former forces us to reason backwards, the latter also allows forward reasoning. Backward reasoning needs a shift of attitude and may be counter intuitive for mathematicians. However, our experience showed that one gets used to backwards reasoning very easily.

## 4.2 There are infinitely many primes

Already Euclid showed that there exist infinitely many primes. We will formulate the theorem as 'for every natural number $x$, there exists a prime $y$ which is bigger than $x$'. We will give a constructive proof. By the nature of constructive proofs, our proof will consist of an algorithm which assigns to every natural number $x$ a $y$ together with a proof that $y$ is a prime bigger than $x$. We will be able to extract this algorithm and get a lambda term which really computes a bigger prime from an arbitrary starting point. By use of this function, we construct a prime generator which computes for every natural number $n$ the $n$-th prime number.

The aim of this case study is to see how *computation* is possible type theory. Also we will show how we can use proofs to construct functions.

First we define a predicate `is_prime` as

$$\texttt{is\_prime}\,(x) \quad \equiv \quad (x > 1)\;\&\;(\forall y{:}\texttt{nat}.\,(1 < y < x) \to (y \nmid x))$$

and give a constructive proof that this predicate is decidable. Applying the mechanism introduced in Section 3.3.4, we form the characteristic function $\mathtt{K}_{\mathtt{prime}}$ for which

$$\vdash \forall x{:}\texttt{nat}.\,(\mathtt{K}_{\mathtt{prime}}\,x) = \underline{0} \leftrightarrow \texttt{is\_prime}\,x \ \ ,$$

and such that for natural number $n$

$$(\mathtt{K}_{\mathtt{prime}}\,\underline{n}) =_{\beta\iota} \underline{0} \quad \text{iff} \quad n\,\text{is a prime number} \ \ .$$

Next we give a constructive proof that there are infinitely many primes, which we formalize by proving that for every natural number $x$ we can find a prime number $y$ which is bigger than $x$. Also we give an upper bound for this $y$, so we can apply bounded minimalization to define a function `prime` such that

$$\begin{aligned}
\texttt{prime}\,\underline{0} &\;=_{\beta\iota}\; \underline{2},\\
\texttt{prime}\,\underline{1} &\;=_{\beta\iota}\; \underline{3},\\
\texttt{prime}\,\underline{2} &\;=_{\beta\iota}\; \underline{5},\\
&\;\;\vdots
\end{aligned}$$

We will prove that `prime` generates, strictly increasing, all prime numbers.

4.2.1. Remark. If we had used classical logic, we would lose these computational equalities. So then the convertibility relations above would not hold. However, the 'logical' equations

$$\begin{aligned}
\texttt{prime}\,\underline{0} &\;=_{L}\; \underline{2},\\
\texttt{prime}\,\underline{1} &\;=_{L}\; \underline{3},\\
\texttt{prime}\,\underline{2} &\;=_{L}\; \underline{5},\\
&\;\;\vdots
\end{aligned}$$

would still be provable.

### 4.2.1 Formalization of Euclid's theorem

We will give a formalization of the theorem in more detail. We skip some basic definitions like the operations $<$, $|$, factorial. For a fully formalized proof with all the details, the reader is referred to the LEGO library.

4.2.2. Definition. Define a predicate 'is prime number', 'is not a prime number', and 'is a prime factor'.

$$\begin{array}{lcl}
\texttt{is\_prime} & : & \texttt{nat}\rightarrow\texttt{PROP} \\
 & \equiv & \lambda x{:}\texttt{nat}.\,(\underline{1} < x)\ \& \\
 & & \qquad (\forall y{:}\texttt{nat}.\,(\underline{1} < y)\rightarrow(y < x)\rightarrow(y \nmid x)) \\
\texttt{is\_dividable} & : & \texttt{nat}\rightarrow\texttt{PROP} \\
 & \equiv & \lambda x{:}\texttt{nat}.\,(\underline{1} < x)\rightarrow(\exists y{:}\texttt{nat}.\,(\underline{1} < y)\ \&\ (y < x)\ \&\ (y \mid x)) \\
\texttt{is\_prime\_factor} & & \\
 & : & \texttt{nat}\rightarrow\texttt{nat}\rightarrow\texttt{PROP} \\
 & \equiv & \lambda a, b{:}\texttt{nat}.\,(a \mid b)\ \&\ (\texttt{is\_prime}\,a)
\end{array}$$

Because we wish to be constructive, we need to be careful with the negation of propositions. So we formalized 'not is prime' as 'there exists a true divisor'. We can prove for example that 3 is a prime.

$$\begin{array}{lcl}
\texttt{ThreeIsPrime} & : & \texttt{is\_prime}\,\underline{3} \\
 & \equiv & \llcorner\ldots\lrcorner
\end{array}$$

For natural number $n$, the size of a proof of $\texttt{is\_prime}\,\underline{n}$ is linear in $n$. We will see how we can reduce it to a size independent of $n$.

4.2.3. Lemma. *The predicate* is_prime *is decidable, that is for any natural number* $x$ *we have* $(\texttt{is\_prime}\,x) \vee \neg(\texttt{is\_prime}\,x)$. *Formally,*

$$\begin{array}{lcl}
\texttt{is\_prime\_dec} & : & \texttt{decidable\_pred is\_prime} \\
 & \equiv & \llcorner\ldots\lrcorner\ .
\end{array}$$

Proof. *First we prove that*

$$\begin{array}{l}
\forall x, a{:}\texttt{nat}.\,(\forall y{:}\texttt{nat}.\,(\underline{1} < y)\rightarrow(y < a)\rightarrow(y \nmid x)) \vee \\
\qquad (\exists y{:}\texttt{nat}.\,(\underline{1} < y)\ \&\ (y < a)\ \&\ (y \mid x))\ .
\end{array}$$

*Then trivially*

$$\forall x{:}\texttt{nat}.\,(\texttt{is\_prime}\,x) \vee (\texttt{is\_dividable}\,x)$$

*holds too. Now let $x$ be a natural number. We distinguish two cases. Suppose* $\texttt{is\_prime}\,x$ *holds, then we are done. Suppose* $\texttt{is\_dividable}\,x$ *holds. We first prove that*

$$\forall x{:}\texttt{nat}.\,(\texttt{is\_prime}\,x)\rightarrow(\texttt{is\_dividable}\,x)\rightarrow\bot\ .$$

*Then we know* $\neg(\texttt{is\_prime}\,x)$ *and we are done.*

Note that from a classical point of view Lemma 4.2.3 is trivial. But if we would have used classical logic, the next definition would lose its nice properties.

4.2.4. Definition. Define the characteristic function $\texttt{K}_{\texttt{prime}}$ using the decidability of prime.

$$\begin{array}{lcl}
\texttt{K}_{\texttt{prime}} & : & \texttt{nat}\rightarrow\texttt{nat} \\
 & \equiv & \texttt{char is\_prime\_dec}
\end{array}$$

See Section 3.3.4 for the definition of `char`. Because the proof `is_prime_dec` is constructive, we have that for all natural numbers $n$

$$(\mathtt{K}_{\mathtt{prime}}\,\underline{n}) =_{\beta\iota} \underline{0} \quad \text{iff} \quad n \,\text{is a prime} \enspace .$$

4.2.5. EXAMPLE. Now it becomes very easy to check whether a natural number is a prime or not.

$$\begin{array}{lcl}
\mathtt{K\_prime\_intro} & : & \forall x{:}\mathtt{nat}.\,(\mathtt{is\_prime}\,x)\to((\mathtt{K}_{\mathtt{prime}}\,x) =_L \underline{0}) \\
 & \equiv & \llcorner\ldots\lrcorner \\
\mathtt{K\_prime\_elim} & : & \forall x{:}\mathtt{nat}.\,((\mathtt{K}_{\mathtt{prime}}\,x) =_L \underline{0})\to(\mathtt{is\_prime}\,x) \\
 & \equiv & \llcorner\ldots\lrcorner \\
\mathtt{FiveIsPrime} & : & \mathtt{is\_prime}\,\underline{5} \\
 & \equiv & \mathtt{K\_prime\_elim}\,(\mathtt{Refl}\,{}_L(\mathtt{K}_{\mathtt{prime}}\,\underline{5})) \\
\mathtt{NotSixIsPrime} & : & \neg(\mathtt{is\_prime}\,\underline{6}) \\
 & \equiv & \lambda H{:}\mathtt{is\_prime}\,\underline{6}.\,\mathtt{Succ\_not\_zero}\,\underline{0}\,(\mathtt{K\_prime\_intro}\,H)
\end{array}$$

where `Succ_not_zero` is a proof of

$$\forall x{:}\mathtt{nat}.\,\neg((\mathtt{S}^{\mathtt{N}}\,x) = \underline{0}) \enspace .$$

If we look at the definition of `FiveIsPrime`, we could be tempted to write the simpler term

$$\mathtt{K\_prime\_elim}\,(\mathtt{Refl}\,{}_L\underline{0}) \enspace .$$

However, this lambda term is not typable by the LEGO system because we make use of the argument synthesis mechanism in the definition of the term `K_prime_elim`. So the type checker has to be able to find an $x$ such that the next argument has type $((\mathtt{K}_{\mathtt{prime}}\,x) =_L \underline{0})$. But obviously, $(\mathtt{Refl}\,{}_L\underline{0})$ has type $(\underline{0} = \underline{0})$.

4.2.6. LEMMA. *Every natural number bigger then one has a prime factor.*

$$\begin{array}{lcl}
\mathtt{has\_prime\_factor} & : & \forall x{:}\mathtt{nat}.\,(\underline{1} < x)\to\exists y{:}\mathtt{nat}.\,\mathtt{is\_prime\_factor}\,y\,x \\
 & \equiv & \llcorner\ldots\lrcorner
\end{array}$$

PROOF. *By course of values induction.*

4.2.7. THEOREM. *Given a natural number $x$ there exists a natural number $y$ such that*

- *$y$ is a prime,*
- *$y$ is bigger than $x$, and*
- *for $y$ we have an upper bound depending only on $x$.*

*Formally:*

$$\begin{array}{l}
\mathtt{infinitely\_bounded\_primes\_exist} \\
\quad : \quad \forall x{:}\mathtt{nat}\,\exists y{:}\mathtt{nat}.\,(x < y)\ \&\ (y \le (\mathtt{S}^{\mathtt{N}}\,(\mathtt{fac}\,x)))\ \&\ (\mathtt{is\_prime}\,y) \\
\quad \equiv \quad \llcorner\ldots\lrcorner
\end{array}$$

PROOF. *Set $z = x! + \underline{1}$. By Lemma 4.2.6, $z$ has a prime factor $y$. Suppose $y \le x$. Then $y \mid x!$ holds. Because $y \mid z$, we must conclude $y = \underline{1}$, which by definition cannot be the case. So $x < y$.*

4.2.8. Corollary (Euclid). *Given a natural number $x$. We can always find a prime $y$ which is bigger than $x$.*

$$\begin{array}{rcl} \texttt{infinitely\_primes\_exist} & : & \forall x{:}\texttt{nat}\, \exists y{:}\texttt{nat}.\, (x < y)\ \&\ (\texttt{is\_prime}\, y) \\ & \equiv & \llcorner \ldots \lrcorner \end{array}$$

Proof. *Immediate from Theorem 4.2.7.*

### 4.2.2 Bounded minimalization

For decidable predicates $\phi(x)$ over the natural numbers, we wish to define bounded minimalization

$$\mu x < z.\, \phi(x)$$

such that it computes the smallest $x$ smaller than $z$ for which $\phi(x)$ holds. If there exists no such $x$, it should return $z$. Indeed, using case distinction we are able to define bounded minimalization.

4.2.9. Definition. Let $\phi$ be a predicate for which $\phi_{\texttt{dec}} : \texttt{decidable\_pred}\, \phi$ is a proof that $\phi$ is decidable. We define bounded minimalization as follows.

$$\begin{array}{rcllll} \texttt{mu}'_\phi & : & \texttt{nat}\to\texttt{nat}\to\texttt{nat} & & & \\ & \equiv & \varepsilon_{\texttt{nat}}\ \texttt{zero} & \Longrightarrow & \lambda z{:}\texttt{nat}.\, z & \\ & & (\texttt{S}^{\texttt{N}}\, x) & \Longrightarrow & \lambda z{:}\texttt{nat}.\, (\texttt{mu}'_\phi\, x\, x) & \text{if } \phi\, x \\ & & & & (\texttt{mu}'_\phi\, x\, z) & \text{otherwise} \\ \texttt{mu}_\phi & : & \texttt{nat}\to\texttt{nat} & & & \\ & \equiv & \lambda z{:}\texttt{nat}.\, \texttt{mu}'_\phi\, z\, z & & & \end{array}$$

Next we prove some basic properties of `mu`.

4.2.10. Lemma. *Let $\phi$ be a decidable predicate.*

(i) *Bounded minimalization is sound.*

$$\begin{array}{rcl} \texttt{mu\_phi} & : & \Pi z{:}\texttt{nat}.\, ((\texttt{mu}\, \phi_{\texttt{dec}}\, z) \neq z) \to \phi\, (\texttt{mu}\, \phi_{\texttt{dec}}\, z) \\ & \equiv & \llcorner \ldots \lrcorner \end{array}$$

(ii) *Bounded minimalization is bounded.*

$$\begin{array}{rcl} \texttt{mu\_UB} & : & \Pi z{:}\texttt{nat}.\, (\texttt{mu}\, \phi_{\texttt{dec}}\, z) \leq z \\ & \equiv & \llcorner \ldots \lrcorner \end{array}$$

(iii) *Bounded minimalization returns the smallest value, if it exists.*

$$\begin{array}{rcl} \texttt{mu\_OK} & : & \Pi x{:}\texttt{nat}.\, (\phi\, x) \to \Pi z{:}\texttt{nat}.\, (\texttt{mu}\, \phi_{\texttt{dec}}\, z) \leq x \\ & \equiv & \llcorner \ldots \lrcorner \end{array}$$

Proof. The reader is referred to the LEGO library.

### 4.2.3 Prime generator

Now we can use bounded minimalization to define a prime generator.

4.2.11. Definition. Define a relation is_new_prime which states that $y$ is a prime bigger than $x$.

$$\begin{array}{lcl}\texttt{is\_new\_prime} & : & \texttt{nat}\to\texttt{nat}\to\texttt{PROP} \\ & \equiv & \lambda x, p{:}\texttt{nat}.\,(\texttt{is\_prime}\,p)\ \&\ (x < p)\end{array}$$

4.2.12. Lemma. *Let $x$ be a natural number. The predicate* is_new_prime $x$ *is decidable.*

$$\begin{array}{lcl}\texttt{is\_new\_prime\_dec} & : & \texttt{decidable\_pred}\,(\texttt{is\_new\_prime}\,x) \\ & \equiv & \llcorner\ldots\lrcorner\end{array}$$

4.2.13. Definition. (i) Using bounded minimalization, define a function over $\mathbb{N}$ which returns a prime number bigger than its argument, called a new prime.

$$\begin{array}{lcl}\texttt{new\_prime} & : & \texttt{nat}\to\texttt{nat} \\ & \equiv & \lambda x{:}\texttt{nat}.\,\texttt{mu}\,(\texttt{is\_new\_prime\_dec}\,x)(\texttt{S}^{\texttt{N}}\,(x!))\end{array}$$

(ii) The prime generator is just an iteration of new_prime, starting from 2.

$$\begin{array}{lclcl}\texttt{prime} & : & \texttt{nat}\to\texttt{nat} & & \\ & \equiv & \varepsilon_{\texttt{nat}}\ \texttt{zero} & \Longrightarrow & \underline{2} \\ & & (\texttt{S}^{\texttt{N}}\,x) & \Longrightarrow & \texttt{new\_prime}\,(\texttt{prime}\,x)\end{array}$$

4.2.14. Lemma. (i) *The function* prime *only generates primes.*

$$\begin{array}{lcl}\texttt{prime\_is\_prime} & : & \forall i{:}\texttt{nat}.\,\texttt{is\_prime}\,(\texttt{prime}\,i) \\ & \equiv & \llcorner\ldots\lrcorner\end{array}$$

(ii) *The function* prime *is strictly increasing.*

$$\begin{array}{lcl}\texttt{prime\_grows} & : & \forall i{:}\texttt{nat}.\,(\texttt{prime}\,i) < (\texttt{prime}\,(\texttt{S}^{\texttt{N}}\,i)) \\ & \equiv & \llcorner\ldots\lrcorner\end{array}$$

(iii) *The function* prime *generates all primes.*

$$\begin{array}{lcl}\texttt{prime\_surj} & : & \forall x{:}\texttt{nat}.\,(\texttt{is\_prime}\,x)\to\exists i{:}\texttt{nat}.\,x =_L (\texttt{prime}\,i) \\ & \equiv & \llcorner\ldots\lrcorner\end{array}$$

Because the definition of the prime generator prime was a constructive definition, the type checker can compute prime numbers by normalizing. So the following convertibility relations hold.

$$\begin{array}{lcl}\texttt{prime}\,\underline{0} & =_{\beta\iota} & \underline{2}, \\ \texttt{prime}\,\underline{1} & =_{\beta\iota} & \underline{3}, \\ \texttt{prime}\,\underline{2} & =_{\beta\iota} & \underline{5}, \\ & \vdots & \end{array}$$

we can use this fact to quickly get a proof that

$$\begin{array}{lcl}\texttt{prime}_1 & : & (\texttt{prime}\,\underline{1}) =_L \underline{3} \\ & \equiv & \texttt{Refl}\,_L(\texttt{prime}\,\underline{1})\end{array}$$

Unfortunately, if the prime number we wish to compute grows, the verification time quickly gets out of hand[2].

[2]On our hardware, the normalization of prime $\underline{2}$ needed a heap space of more than 300 megabytes. It took a whole day to check. Not to mention how much resources it would take to compute prime $\underline{1999}$. This suggests that either the algorithm we used is not really efficient, or the type checker used is not implemented efficiently. Or both.

## 4.3 Binomial theorem

In this section will show how equational reasoning can be done in type theory. We take Newton's Binomial Theorem as a case study. Let us first formulate the theorem.

4.3.1. THEOREM. *Let $R$ be a commutative ring, $x$ and $y$ elements of $R$. Then for all $n$ in $\mathbb{N}$*

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^{n-i} y^i \ .$$

It seems to be mathematical practice to define the binomial function in terms of factorials.

$$\binom{n}{m} \quad \equiv \quad \frac{n!}{m!(n-m)!}$$

However proofs of properties of binomials get much more simple if we use a lower level definition[3]. So instead we present an equivalent definition of the binomial by nested recursion. This way we we avoid the need of factorials and division.

4.3.2. DEFINITION. We define the binomial coefficients directly by double recursion on its arguments.

$$\begin{array}{lllcl}
() & : & \multicolumn{3}{l}{\mathtt{nat}\to\mathtt{nat}\to\mathtt{nat}} \\
 & \equiv & \varepsilon^2_{\mathtt{nat}} \quad n, & \underline{0} & \Longrightarrow \underline{1} \\
 & & \quad \underline{0}, & \mathtt{S^N}\, m & \Longrightarrow \underline{0} \\
 & & \quad \mathtt{S^N}\, n, & \mathtt{S^N}\, m & \Longrightarrow \binom{n}{\mathtt{S^N}\, m} + \binom{n}{m}
\end{array}$$

Next we need a notion for summation. We always start from zero.

4.3.3. DEFINITION. Let $R$ be a ring. We define summation from zero by recursion.

$$\begin{array}{lll}
\sum_{i=0}^{(\_)}(\_) & : & (\mathtt{nat}\to(\mathtt{obj}\, R))\to\mathtt{nat}\to(\mathtt{obj}\, R) \\
 & \equiv & \lambda f{:}(\mathtt{nat}\to(\mathtt{obj}\, R)).\, \varepsilon_{\mathtt{nat}}\ \mathtt{zero} \quad \Longrightarrow \quad f\,\underline{0} \\
 & & \qquad\qquad\qquad\qquad (\mathtt{S^N}\, n) \quad \Longrightarrow \quad (\sum_{i=0}^{n} f) + f(\mathtt{S^N}\, n)
\end{array}$$

4.3.4. LEMMA. *Let $q$ be a natural number. Set $p \equiv \mathtt{S^N}\, q$ and $n \equiv \mathtt{S^N}\, p$. Then*

$$(x+y)^{\underline{0}} \quad = \quad \sum_{i=0}^{\underline{0}} (x^{\underline{0}-i} y^i) \binom{\underline{0}}{i} \quad , \tag{4.1}$$

$$(x+y)^{\underline{1}} \quad = \quad \sum_{i=0}^{\underline{1}} (x^{\underline{1}-i} y^i) \binom{\underline{1}}{i} \quad , \tag{4.2}$$

$$(x+y)^p x \quad = \quad \sum_{i=0}^{p} (x^{(\mathtt{S^N}\, p)-i} y^i) \binom{p}{i} \quad , \tag{4.3}$$

$$(x+y)^p y \quad = \quad \sum_{i=0}^{p} (x^{(\mathtt{S^N}\, p)-(\mathtt{S^N}\, i)} y^{\mathtt{S^N}\, i}) \binom{p}{i} \quad , \tag{4.4}$$

$$(x+y)^p x \quad = \quad (x^{n-\underline{0}} y^{\underline{0}}) \binom{n}{\underline{0}} + \sum_{i=0}^{p} (x^{n-(\mathtt{S^N}\, i)} y^{\mathtt{S^N}\, i}) \binom{p}{i} \quad . \tag{4.5}$$

[3]Here we offend the principle formulated in Section 3.0.1. But every rule should have exceptions.

PROOF. *By a lot of equational reasoning (see for details the LEGO library in Appendix A.3.)*

4.3.5. THEOREM. *Let $R$ be a commutative ring, $x$ and $y$ elements of $R$. Then for all $n$ in $\mathbb{N}$*

$$(x+y)^n \quad = \quad \sum_{i=0}^{n} x^{n-i} y^i \binom{n}{i} \quad .$$

PROOF. *By nested induction on $n$. The proof is straightforward, but involves a lot of rewriting of parentheses and applications of the ring axioms. For a fully formalized proof, the reader is referred to the LEGO library (Appendix A.3).*

## 4.4 Fundamental theorem of algebra

The formalization of a proof of the fundamental theorem of algebra forms the main part of our case studies. The reason we chose this particular theorem to formalize is twofold. First, although the theorem is easy to formulate and formalize, proofs of the theorem are highly non-trivial and often quite large. We need a huge database of definitions and supporting lemmas in order to be able to reach the level of mathematics at which we can prove the theorem. Although the name of the theorem suggests it belongs to algebra, there is also a great deal of analysis involved. Second, the theorem is interesting because during the last centuries many different proofs have been given by many great mathematicians. Some proofs are intuitionistic, some classical. Some proofs are highly abstract, using for example Galois Theory[4], and others are very syntactical of nature. Also some proofs make use of a lot of analysis, others are more based on algebra. Most proofs of the theorem, if not all, involve polynomials in the real and complex field. So we first develop the concept of polynomial rings.

### 4.4.1 Polynomial rings

In order to formalize polynomials and construct polynomial rings, let us cite (van der Waerden 1931).

4.4.1. DEFINITION. Let $R$ be a ring and let $G$ be an infinite cyclic group generated by an indeterminate $X$. Polynomials in $X$ over $R$ are elements of the set $R[X]$

$$f \quad \equiv \quad \sum_{i=0}^{n} a_i X^i \ ,$$

where $a_i \in R$. We call $a_i$ the coefficients of the polynomial $f$. The degree of a non-zero polynomial $f$ is $n$.

When we wish to formalize polynomials, we have to make a few implementation decisions. Let $R$ be a ring. Following the definition above, the most straightforward way to define a polynomial $f \in R[X]$ is as a list over $R$, such that the head of the list $f$ is non-zero. Then for example $f[X] = X^3 + \underline{4}X \in \mathbb{N}[X]$ would be denoted by

$$f \quad \equiv \quad \underline{1}\hat{\ }\underline{0}\hat{\ }\underline{4}\hat{\ }\underline{0}\hat{\ }\emptyset \ .$$

[4]For a large scale formalization of Galois Theory, see (Bailey 1998).

Two polynomials $f$ and $g$ are equal if and only if the lists $f$ and $g$ are element-wise equal. For reasons explained below, this implementation is not comfortable for more complex situations. Hence this definition is not suitable as a general representation of polynomials.

### Constructive definition

Suppose the carrier of the ring $R$ is not discrete. Then we cannot decide whether $x = y$ for arbitrary $x, y \in R$. So if we add two polynomials $f$ and $g$ of the same degree, we do not know in general if the resulting polynomial $f + g$ has the same degree, or a lower degree. To be constructive we have to admit trailing zeros in our representation of polynomials.

$$f \quad \equiv \quad \underline{0}\hat{\ }\underline{0}\hat{\ }\underline{1}\hat{\ }\underline{0}\hat{\ }\underline{4}\hat{\ }\underline{0}\hat{\ }\emptyset$$

could be used to denote $f[X] = X^3 + \underline{4}X$ too. Also we have to extend the equality relation over polynomials in order to quotient $f$ and $f'$. Unless the carrier of $R$ is a discrete set, a polynomial does not need to have a degree. We only have a notion of a maximum degree.

### Sparse polynomials

In some cases, the sequence of coefficients of a polynomial is sparse. For example, consider the polynomial $g[X] = \underline{1} + X^{183}$. Because most coefficients are zero, it would be rather inefficient to represent $g$ by a list of length 184, almost completely filled with zeros. So we prefer to represent polynomials by lists of tuples of type $R \times \mathbb{N}$. Every tuple $<a, i>$ stands for the monomial $aX^i$, and a list of tuples is used to represent a polynomial. That way we obtain a much more concise representation in the case of sparse polynomials. So

$$<\underline{1}, \underline{183}>\hat{\ }<\underline{1}, \underline{0}>\hat{\ }\emptyset$$

may be used to denote the polynomial $g$. Another advantage is that this representation allows us to define polynomials with non-constant indices. For example,

$$g \quad \equiv \quad \lambda n{:}\mathbb{N}.\, <\underline{1}, n>\hat{\ }<\underline{1}, \underline{0}>\hat{\ }\emptyset$$

represents $g_n[X] = \underline{1} + X^n$.

### Multivariate polynomials

In standard literature, multivariate polynomials, or polynomials in more indeterminates are usually defined by iteration of ring adjunction $R \to R[X]$:

$$R[X_1, \ldots, X_n] \quad \equiv \quad R[X_1, \ldots, X_{n-1}][X] \ .$$

Although it is a clear and simple definition, in practice it has two drawbacks. First, representatives for polynomials will quickly grow in complexity if the number of indeterminates increases. For example, to denote $h[X, Y] = 3XY^2 + X^3$ we need an expression like

$$h \equiv \quad <<\underline{1}, \underline{0}>\hat{\ }\emptyset, \underline{3}>\hat{\ }<<\underline{3}, \underline{2}>\hat{\ }\emptyset, \underline{1}>\hat{\ }\emptyset \ ,$$

which is quite unreadable. Another drawback of multivariate polynomials defined by iteration is that all theory concerning polynomial rings has to be developed twice: once for the case of the ring $R[X]$, another time for the case of the ring $R[X_1, \ldots, X_n]$. So we define the multivariate polynomials as a primitive notion, and consider the polynomials in one indeterminate as a special case by instantiation. For this we introduce the notion of index monoids. Index monoids will form the type of indices.

4.4.2. Definition. An index monoid is an abelian monoid such that

1. the cancellation law holds,
2. it is ordered, and
3. the carrier is a discrete set.

Examples of index monoids are $\mathbb{N}$ and $\mathbb{N}^n$. So our polynomial $h$ will be constructed by

$$h \quad \equiv \quad <\underline{3}, <\underline{1}, \underline{2}>>\hat{}<\underline{1}, <\underline{3}, \underline{0}>>\hat{}\emptyset \ .$$

**Relaxed polynomials**

Finally, we add one more degree of freedom to our representation of polynomials. We allow multiple coefficients of the same index. Also we remove the condition that the indices have to be ordered. An important advantage of such a liberal approach is that in most cases the definition of operators on polynomials is greatly simplified. For example, the addition operator merely becomes list concatenation. To be able to define equality over polynomials, we first define the coefficient operator. Let $f$ be a polynomial and $i$ an index. The coefficient $f_i$ is defined by traversing the list $f$ while adding all first projections of the pairs whenever the second projection equals $i$. Equality over two polynomials $f$ and $g$ is defined by

$$f = g \quad \equiv \quad \forall i[f_i = g_i] \ .$$

As pointed out in Section 3.0.1, there is a price to pay for our relaxed definition. Although now it is easy to define certain operators over polynomials, it is sometimes difficult to prove that these operators preserve equality over polynomials. The reason is that for operators over polynomials, we have lost reasoning by induction on the degree. We do have reasoning by induction on the length of the list representing the polynomial, but this induction principle is a bit weaker. But again, as pointed out in Section 3.0.1, we prefer relaxed polynomials in formalizing mathematics.

**Polynomial rings**

We summarize this section by giving the final formal definition of the set of polynomials we have chosen to work with. Note that because index monoids are discrete, we may define objects by case distinction on the equality of elements of an index monoid.

4.4.3. Definition. Let $I$ be an index monoid and let $R$ be a ring. Recall that

'$\texttt{if}\, x\, y\, a\, b$' should be read as 'if $x = y$ then $a$ otherwise $b$'.

$$\begin{array}{lcl}
\texttt{Monomial} & : & \texttt{Set} \\
 & \equiv & \texttt{Prod}\,(\texttt{car}\, R)(\texttt{car}\, I) \\
\texttt{poly} & : & \texttt{SET} \\
 & \equiv & \texttt{list}\,(\texttt{el}\, \texttt{Monomial}) \\
\texttt{coef} & : & \texttt{poly}{\rightarrow}(\texttt{obj}\, I){\rightarrow}(\texttt{obj}\, R) \\
 & \equiv & \varepsilon_{\texttt{list}}\ \begin{array}[t]{rcl} \emptyset & \Longrightarrow & \lambda i{:}\texttt{obj}\, I.\, 0_R \\ (t\hat{\ }f) & \Longrightarrow & \lambda i{:}\texttt{obj}\, I.\, \texttt{if}\ \begin{array}[t]{l} \pi_2^2(t)\ i \\ \pi_1^2(t) +_R (\texttt{coef}\, f\, i) \\ (\texttt{coef}\, f\, i) \end{array} \end{array} \\
=_{\texttt{poly}} & : & \texttt{poly}{\rightarrow}\texttt{poly}{\rightarrow}\texttt{PROP} \\
 & \equiv & \lambda f{:}\texttt{poly}\, \lambda f{:}\texttt{poly}\, \forall i{:}\texttt{obj}\, I.\, (\texttt{coef}\, f\, i) = (\texttt{coef}\, g\, i) \\
\texttt{Poly} & : & \texttt{Set} \\
 & \equiv & {<}\texttt{poly}, =_{\texttt{poly}}, \llcorner\ldots\lrcorner{>}
\end{array}$$

4.4.4. Definition. Let $I$ be an index monoid and let $R$ be a ring. Set $P \equiv \texttt{Poly} I\, R$.

$$\begin{array}{lcl}
0_{\texttt{Poly}} & : & \texttt{el}\, P \\
 & \equiv & \emptyset \\
1_{\texttt{Poly}} & : & \texttt{el}\, P \\
 & \equiv & {<}1_R, 0_I{>}\hat{\ }\emptyset \\
\texttt{plus}_{\texttt{Poly}} & : & (\texttt{el}\, P){\rightarrow}(\texttt{el}\, P){\rightarrow}(\texttt{el}\, P) \\
 & \equiv & \lambda f{:}\texttt{el}\, P.\, \varepsilon_{\texttt{list}}\ \begin{array}[t]{rcl} \emptyset & \Longrightarrow & f \\ (t\hat{\ }g) & \Longrightarrow & t\hat{\ }(\texttt{plus}_{\texttt{Poly}} f\, g) \end{array}
\end{array}$$

Allthough slightly more complicated, we can define the multiplication and inverse over polynomials in a similar fashion. Also we can show that $\texttt{Poly}$ forms a ring. This is done in the LEGO library.

4.4.5. Definition (polynomial application). Let $I$ be an index monoid, and $R$ be a commutative ring. Let $n$ be a natural number, set $\texttt{Rn} : \texttt{Set} \equiv \texttt{Vector}\,(\texttt{car}\, R)\, n$ and let

$$\texttt{Power} : \texttt{Fun}^2\, \texttt{Rn}\,(\texttt{car}\, I)\,(\texttt{car}\, R)$$

be an exponentiation function such that we have proofs for

$$\begin{array}{ll}
\forall x{:}\texttt{el}\, \texttt{Rn}.\, (\texttt{ap}^2\, \texttt{Power}\, x\, 0_I) = 1_R & \\
\forall x{:}\texttt{el}\, \texttt{Rn}\, \forall a, b{:}\texttt{obj}\, I. & (\texttt{ap}^2\, \texttt{Power}\, x\, (a +_I b)) = \\
 & ((\texttt{ap}^2\, \texttt{Power}\, x\, a) \times_I (\texttt{ap}^2\, \texttt{Power}\, x\, b))
\end{array}$$

Set $P \equiv \texttt{Poly}\, I\, R$. We define polynomial application as follows.

$$\begin{array}{lcl}
\texttt{apP} & : & (\texttt{el}\, P) \rightarrow (\texttt{el}\, \texttt{Rn}) \rightarrow (\texttt{obj}\, R) \\
 & \equiv & \lambda f{:}\texttt{el}\, P\, \lambda x{:}(\texttt{el}\, \texttt{Rn}).\, \llcorner\ldots\lrcorner \\
\texttt{ApP} & : & \texttt{Fun}^2\, P\, \texttt{Rn}\,(\texttt{car}\, R) \\
 & \equiv & {<}\texttt{apP}, \llcorner\ldots\lrcorner{>}
\end{array}$$

As we might expect, the $\llcorner\ldots\lrcorner$ in the definition of `apP` is easy to fill in. To compute the application of a polynomial $f$ and a value $x$, we essentially just sum $c\times x^i$ for all elements $<c,i>$ in the list $f$. The gap in the definition of `ApP` stands for a proof of the proposition that `apP` respects polynomial equality. This is not trivial to prove. The reader is referred to the LEGO library for the details.

Let $R$ be a ring. We introduce the pseudo LEGO notation 'R[X]' which stands for '$\texttt{el}\,(\texttt{Poly}\,\mathbb{N}\,R)$'. Furthermore for $f : \texttt{el}\,(R[X])$ and $x : \texttt{obj}\,R$, we abbreviate '$\texttt{apP}\,\mathbb{N}\,R\,\underline{1}\,(\texttt{Power}\,R)\,f\,x$' as '$f[x]$'.

## 4.4.2 k-th roots in $\mathbb{C}$

Before we start to work on $k$-th roots in $\mathbb{C}$ for arbitrary positive numbers $k$, we first need the cubic case ($k = 2$). The way we formalized the real number system gives us immediately the following result.

4.4.6. Lemma. *Every positive real number has a square root. That is, there exists a function $\sqrt{\_}$ over the real numbers, such that*

(i)

$$\texttt{SqrtR\_axiom1} \quad : \quad \forall x|\texttt{el}\,\mathbb{R}.\,(x \geq 0_R) \rightarrow (\sqrt{x}^2 = x)$$

(ii)

$$\texttt{SqrtR\_axiom2} \quad : \quad \forall x|\texttt{el}\,\mathbb{R}.\,(x \geq 0_R) \rightarrow (\sqrt{x} \geq 0_R)$$

Proof. Immediately by Definition 3.5.15.

We can construct the square root for complex numbers in terms of the square root for the real numbers.

4.4.7. Definition.

$$\begin{array}{lcl}
\texttt{sqrtC} & : & (\texttt{el}\,\mathbb{C}) \rightarrow (\texttt{el}\,\mathbb{C}) \\
 & \equiv & \lambda x{:}\texttt{el}\,\mathbb{C}.\ \sqrt{(W+a)/2} + i(\texttt{sign}\,b)\sqrt{(W-a)/2} \\
 & & \qquad \text{where } a \equiv \texttt{re}\,x, b \equiv \texttt{im}\,x, W \equiv \sqrt{a^2+b^2} \\
\texttt{SqrtC} & : & \mathbb{C} \Rightarrow \mathbb{C} \\
 & \equiv & <\texttt{sqrtC}, \llcorner\ldots\lrcorner>
\end{array}$$

We extend pseudo LEGO such that we may replace '$\texttt{ap}\,\texttt{SqrtC}\,x$' by the more readable '$\sqrt{x}$'. Let $x : \texttt{el}\,\mathbb{C}$ be a complex number. The equation $y^2 = x$ actually has two solutions, namely $\sqrt{x}$ and $-\sqrt{x}$. So we have the following key lemma.

4.4.8. Lemma.

$$\forall x{:}\texttt{el}\,\mathbb{C}.\,\sqrt{x}^2 = (-\sqrt{x})^2 = x$$

Proof. *For a proof in full detail, the reader is referred to the LEGO library (Appendix A.3).*

For the definition of functions like $\texttt{cp} : (\texttt{el}\,\mathbb{R}){\rightarrow}(\texttt{el}\,\mathbb{C})$, $\texttt{re} : (\texttt{el}\,\mathbb{C}){\rightarrow}(\texttt{el}\,\mathbb{R})$, and $\texttt{im} : (\texttt{el}\,\mathbb{C}){\rightarrow}(\texttt{el}\,\mathbb{R})$, the reader is referred to the LEGO library. Also for the more complex definition of the degree ($\delta$) of a polynomial in a discrete ring, we refer to the same library.

In order to be able to prove the existence of the $k$-th roots in $\mathbb{C}$ we need the following continuity result.

4.4.9. Lemma. *Every polynomial over the real numbers of odd degree has a root.*

$$\forall P{:}\mathtt{el}\,\mathbb{R}[X].\,(\mathtt{odd}\,(\delta\,P)) \to \exists y{:}\mathtt{el}\,\mathbb{R}.\,P[y] = 0_R$$

Proof. Immediatly by Definition 3.5.15.

4.4.10. Corollary.

$$\forall x{:}\mathtt{el}\,\mathbb{R}\,\forall n{:}\mathtt{el}\,\mathbb{N}.\,(\mathtt{odd}\,n) \to \exists y{:}\mathtt{el}\,\mathbb{R}.\,y^n = x$$

Proof. *Given $x$ and odd $n$, apply Lemma 4.4.9 with $P \equiv X^n - x$.*

4.4.11. Theorem.

$$\forall n{:}\mathtt{el}\,\mathbb{N}\,\forall x{:}\mathtt{el}\,\mathbb{C}.\,(n \neq \underline{0}) \to \exists y{:}\mathtt{el}\,\mathbb{C}.\,(y^n = x)$$

Proof. Apply course of values induction on $n$. We distinguish two cases.

Suppose $n$ is even. Say $n = 2m$. Because $n \neq \underline{0}$ we have that $m \neq \underline{0}$ and $m < n$. By the induction hypothesis for $m$ and $\sqrt{x}$, we obtain a $y : \mathtt{el}\,\mathbb{C}$ such that $y^m = \sqrt{x}$. So $y^n = (\sqrt{x})^2 = x$, and we are done.

For the second case, suppose $n$ is odd. Then we have a $z : \mathtt{el}\,\mathbb{C}$ such that $|z| = 1_R$ and $(\mathtt{cp}\,|x|)z = x$. By Corollary 4.4.10 we have $y_0 : \mathtt{el}\,\mathbb{R}$ such that $y_0^n = |x|$. Again we distinguish two cases.

Suppose $x \in \mathbb{R}$. By Corollary 4.4.10 we have a $y_1 : \mathtt{el}\,\mathbb{R}$ such that $y_1^n = \mathtt{re}\,z$. Set $y \equiv \mathtt{cp}\,y_1$, then $y^n = \mathtt{cp}\,y_1^n = \mathtt{cp}\,(\mathtt{re}\,z) = z$, and we are done.

Suppose $x \notin \mathbb{R}$. Set $d \equiv \sqrt{z}$. Then $d \notin \mathbb{R}$. Define

$$\begin{array}{lcl} P & : & \mathtt{el}\,\mathbb{C}[X] & \equiv & i(\overline{d}(X+i)^n - d(X-i)^n) \\ Q & : & \mathtt{el}\,\mathbb{R}[X] & \equiv & \mathtt{re}\,P\ . \end{array}$$

We know that $P = \overline{P}$, so $P = \mathtt{cp}\,(\mathtt{re}\,P) = \mathtt{cp}\,Q$. Because $\delta\,P = n$ we have that $\mathtt{odd}\,(\delta\,Q)$. Apply Lemma 4.4.9 to obtain a $y_1 : \mathtt{el}\,\mathbb{R}$ such that $Q[y_1] = 0_R$. Set $y_2 : \mathtt{el}\,\mathbb{C} \equiv \mathtt{cp}\,y_1$, then $y_2 \neq i$, and then

$$P[y_2] = (\mathtt{cp}\,Q)[\mathtt{cp}\,y_1] = \mathtt{cp}\,(Q[y_1]) = \mathtt{cp}\,0_R = 0_C\ .$$

Take $y_3 : \mathtt{el}\,\mathbb{C} \equiv (y_2 + i)/(y_2 - i)$. Then we know that $y_3^n = d/\overline{d}$, so we have $y_3^n = z$. Take $y : \mathtt{el}\,\mathbb{C} \equiv (\mathtt{cp}\,y_0)y_3$. Then $y^n = (\mathtt{cp}\,y_0^n)y_3^n = (\mathtt{cp}\,|x|)z = x$, and again, we are done.

The proof is based on the proof found in (Ebbinghaus, Hermes, Hirzebruch et al. 1990, Chapter 3, Section 3). The fully formalized proof contains a lot of equational reasoning[5].

4.4.12. Remark. Usually one makes use of the complex exponential function to prove the preceding theorem. This leads to a considerable simpler proof. However, the conversion of the representation of complex numbers from ordered pairs to polar coordinates is rather nontrivial to establish. We did not wish to develop the necessary theories like trigonometry from scratch.

[5]The proof found in our LEGO libraray consists of more than 400 lines of LEGO code.

### 4.4.3 Fundamental theorem of algebra

So far, all our effort in formalizing mathematics was done with as ultimate goal the formalization of the fundamental theorem of algebra. Proofs for this theorem caught the attention of mathematicians for several centuries. Only in 1799, Gauss gave a full proof. To prove the theorem, one needs a large base of analysis concerning the real and complex number systems. Also a lot of algebraic concepts are involved. So it is a good measure for the usefulness of current proof-development tools, and type checkers in particular. Let us first formulate the theorem.

4.4.13. Theorem (The fundamental theorem of algebra). *Every non-constant complex polynomial has one or more zero's in the field* $\mathbb{C}$.

A field $K$ is said to be *algebraically closed* if every polynomial $f \in K[X]\backslash K$ has a zero in $K$. So the theorem is equivalent to the statement that the field $\mathbb{C}$ of complex numbers is algebraically closed.

#### Historical notes

To place the fundamental theorem in a context, we make a few historical remarks taken from (Ebbinghaus et al. 1990, Chapter 4, Section 1).

The Flemisch mathematician Albert Girard was the first to assert that there are always $n$ solutions. He did not give a proof, only a few examples of polynomials for which his thesis holds.

4.4.14. Thesis (Girard, 1692). *For every polynomial* $f \in \mathbb{R}[X]$ *of degree* $n$ *there exists a field* $K$*, an extension of* $\mathbb{R}$*, such that* $f$ *has exactly* $n$ *zeros (not necessarily distinct) in* $K$*. The field* $K$ *may perhaps be a proper overfield of* $\mathbb{C}$.

In a letter to Goldbach, Euler asserted in 1742 the following thesis.

4.4.15. Thesis (Fundamental theorem of algebra for real polynomials). *Every polynomial* $f \in \mathbb{R}[X]$ *of the* $n$*-th degree has precisely* $n$ *zeros in the extension field* $\mathbb{C}$.

In 1749 Euler gave a sketch of a proof of this thesis. The first serious attempt to prove the fundamental theorem of algebra was done three years earlier by Jean le Rond d'Alembert in 1746. His idea was to try to minimize the absolute value of the polynomial $f$ by an appropiate choice of its argument. But it was not until 1799 that Gauss gave a rigorous proof of the theorem. So far questions were raised like what form the roots would have. Gauss's proof did not calculate a root, but it proved its existence. Later, Gauss gave three other proofs of the theorem, the last one in 1849.

In (Huntington 1955, Chapter IV, Appendix II) a very syntactical proof is given based on geometry that does not make use of trigonometry, nor of the method of separating a complex quantity into its real and pure imaginary parts.

#### Our results

To formalize the Fundamental Theorem, we choose the proof found in (Ebbinghaus et al. 1990, Chapter 3, Section 4). This proof is attractive for our purposes for two reasons. First, it makes a clear distinction between the analytical part and the algebraic part. Second, the proof does not make use of large theories which should be proven first. In fact the proof is rather syntactical and seemed to be straightforward to formalize.

We spend considerable effort formalizing all kinds of concepts and proofs which can be found in the LEGO library (Appendix A.3), and in the other chapters in this thesis. Eventually, reached as far as a complete formal proof of the existence of $k$-th complex roots (Theorem 4.4.11). So we did not formalize the Fundamental Theorem in type theory. The reason for this is basicly that with current technology it is not possible to formalize non-trivial mathematical bodies within reasonable time. In the conclusion (Chapter 6), we will elaborate on the limitations of type checkers. In the next chapter, we will treat one limitation in particular, namely equation reasoning.

# Chapter 5

# A Two-Level Approach

This chapter is based on the paper (Barthe, Ruys and Barendregt 1996).

We present a simple and effective methodology for equational reasoning in proof checkers. The method is based on a two-level approach distinguishing between syntax and semantics of mathematical theories. The method is very general and can be carried out in any type system with inductive and congruence types. The potential of our two-level approach is illustrated by some examples developed in LEGO.

## 5.1 Introduction

The main actions in writing mathematics consist of defining, reasoning and computing (symbolically; this is also called 'equational reasoning'). Whereas defining and reasoning are reasonably well captured by an interactive proof-developer, the formalization of computations has caused problems. This chapter studies the possibilities of a partial automation of equational reasoning, which is from the authors' experience, one of the most recurrent source of problems in formalizing mathematics using a proof-developer (Barthe 1995a). We describe several methods using elementary techniques from universal algebra which provide an efficient tool to solve problems of an equational nature in any type theory with inductive types and term rewriting (inductive types are required for a formalization of universal algebra, in particular for the formalization of the type of terms of a signature).

Our main goal is to solve equational problems of the form $a =_{\mathcal{A}} b$, where $\mathcal{A}$ is a model of a given equational theory $\mathcal{S} = (\Sigma, E)$, $a$ and $b$ are (expressions for) elements of $\mathcal{A}$, and $=_{\mathcal{A}}$ is the equality relation of the carrier of $\mathcal{A}$. To do so, we use two naming principles:

*for satisfiability:* we recast the problem $a =_{\mathcal{A}} b$ in a syntactic form $[\![\ulcorner a \urcorner]\!]_\alpha^{\mathcal{A}} =_{\mathcal{A}} [\![\ulcorner b \urcorner]\!]_\alpha^{\mathcal{A}}$ where $\alpha$ is an assignment and $\ulcorner a \urcorner$ and $\ulcorner b \urcorner$ are two $\Sigma$-terms such that

$$[\![\ulcorner a \urcorner]\!]_\alpha^{\mathcal{A}} = a \quad \text{and} \quad [\![\ulcorner b \urcorner]\!]_\alpha^{\mathcal{A}} = b$$

where $[\![\_]\!]_\alpha^{\mathcal{A}}$ denotes the $\alpha$-interpretation of $\Sigma$-terms into the model $\mathcal{A}$. (Note that such terms always exist and one can even find optimal terms). By the soundness theorem, the latter problem follows from $\mathcal{S} \vdash \ulcorner a \urcorner \doteq \ulcorner b \urcorner$ (we use this informal notation to state that $(\ulcorner a \urcorner, \ulcorner b \urcorner)$ is a theorem of $\mathcal{S}$). If $\mathcal{S}$ is equivalent to a canonical term rewriting system $\mathcal{R}$, then the last problem can be solved automatically by taking the $\mathcal{R}$-normal forms of $\ulcorner a \urcorner$ and $\ulcorner b \urcorner$ and check whether they are equal. We internalize the whole informal process using *congruence types* (Barthe and Geuvers 1996); the rewrite system is grafted to the type theory in such a way that the conversion rule itself is changed and checking whether $[\ulcorner a \urcorner] = [\ulcorner b \urcorner]$ (the equality here is Leibniz equality) boils down to a reflexivity test, which can be done by the proof checker.

*for extensionality:* often we need a proof object for statements of the form

$$s =_{\mathcal{A}} t \quad \Rightarrow \quad \phi(s) =_{\mathcal{A}} \phi(t) \tag{5.1}$$

where $s$, $t$ and $\phi(x)$ be (expressions for) elements of $\mathcal{A}$. If this is done in the way taught in books on logic (applying several times the axioms of equational logic) a proof object for this fact becomes rather large: quadratic in the size of the expression '$\phi$'. However, using the naming principle one can solve (5.1) by proving the meta-result

$$s =_{\mathcal{A}} t \quad \Rightarrow \quad [\![\ulcorner \phi \urcorner]\!]^{\mathcal{A}}_{\alpha\,(x:=s)} =_{\mathcal{A}} [\![\ulcorner \phi \urcorner]\!]^{\mathcal{A}}_{\alpha\,(x:=t)}$$

for all $\ulcorner \phi \urcorner$. This result has a proof of fixed size.

In this chapter, we shall give a detailed presentation of these methods (and some minor variants) and demonstrate with non-trivial examples that they provide a suitable tool for a partial automation of equational reasoning in proof-checking. The distinctive features of our approach are:

– it applies to type systems where equality is treated axiomatically (intensional frameworks) and with proof-objects; the only requirement is the presence of (first-order) inductive types and so-called congruence types;

– the size of the implementation of the proof-checker is kept fairly small; the whole process can be carried out within the proof-checker;

– the proof-checker is built upon formal systems whose meta-theory is easy to understand.

The chapter is organized as follows: in Section 5.2, we introduce the relevant mathematical background for the subsequent parts of the chapter. In Section 5.3, we specify the nature of equational reasoning and delimit the range of equational problems whose resolution can be automated. In Section 5.4, we discuss the possible approaches to the automation of equational reasoning and present our own solution in terms of congruence types. In Section 5.5, we present a preliminary implementation of the two-level approach in LEGO. Large parts of the chapter are of expository nature; they have been included because (i) the material we present has never been presented elsewhere with a view to use it for our specific purpose (ii) the main contribution of this chapter is to specify the problem and devise a methodology to solve it (but the methodology does not use any new technique).

The work presented in this chapter bears some similarities with the work of the NuPrl team on reflection (Constable 1993, Howe 1988), although the specific use of naming principles to automate equational reasoning seems to be new.

## 5.2 Mathematical Background

In this section, we review some standard material on equational logic and term rewriting. During the last few years, there has been an explosion in the number of variants of equational logic: many-sorted, order-sorted, conditional... We shall only be concerned with the simplest formalism, unsorted equational logic. For convenience, we separate the presentation in two parts; the first part is concerned with syntax, equational deduction and term rewriting. The second part is devoted to semantics. See (Cohn 1981, Klop 1992) for a longer introduction to the notions involved.

### 5.2.1 Equational Logic and Term Rewriting

The basic notions of universal algebra are those of signature and equational theory. As the notions are standard, we give them without any further comment.

5.2.1. Definition. (i) A *signature* is a pair $\Sigma = (F_\Sigma, \mathsf{Ar})$ where $F_\Sigma$ is a set of function symbols and $\mathsf{Ar} : F_\Sigma \to \mathbb{N}$ is the arity map.

(ii) Let $\Sigma$ be a signature. Let $V$ be a fixed, countably infinite set of variables. The set $T_\Sigma$ of $\Sigma$-*terms* is defined as follows:

– if $x \in V$, then $x \in T_\Sigma$,

– if $f \in F_\Sigma$ and $t_1, \ldots, t_{\mathsf{Ar}f} \in T_\Sigma$, then $f(t_1, \ldots, t_{\mathsf{Ar}f}) \in T_\Sigma$.

(iii) A map $\theta : T_\Sigma \to T_\Sigma$ is a $\Sigma$-*substitution* if for every $f \in F_\Sigma$ and $\Sigma$-terms $t_1, \ldots, t_{\mathsf{Ar}f}$ we have $\theta(f(t_1, \ldots, t_{\mathsf{Ar}f})) = f(\theta t_1, \ldots, \theta t_{\mathsf{Ar}f})$.

(iv) The relation $\leq$ is defined by $t, t' \in T_\Sigma$, $t \leq t'$ if there exists $\theta$ such that $\theta t = t'$. The pre-order induced by $\leq$ is denoted by $T_\Sigma^{\leq}$.

(v) The set $\mathsf{var}(s)$ of variables of a term $s$ is defined inductively as follows:

– if $x \in V$, then $\mathsf{var}(x) = \{x\}$,

– $\mathsf{var}(f(t_1, \ldots, t_{\mathsf{Ar}f})) = \bigcup_{1 \leq i \leq n} \mathsf{var}(t_i)$.

(vi) if $s$ and $t$ are $\Sigma$-terms and $u$ is an occurrence of $s$, $s[u \leftarrow t]$ is the term obtained by replacing the sub term of $s$ at $u$ by $t$.

Note that every (partial) map $\theta : V \to T_\Sigma$ yields a $\Sigma$-*substitution* in an obvious way. We shall sometimes refer to such maps as partial substitutions. The standard terminology can be carried over to partial substitutions, so we will also talk about partial renamings.

**Equational Logic.** A $\Sigma$-*equation* is a pair of $\Sigma$-terms $(s, t)$, usually written as $s \doteq t$.

5.2.2. Definition. An *equational theory* is a pair $\mathcal{S} = (\Sigma, E)$ where $\Sigma$ is a signature and $E$ is a set of $\Sigma$-equations.

The rules for equational deduction are given in the following table:

| **Rules for equational deduction** | |
|---|---|
| $\dfrac{}{s \doteq s}$ | Reflexivity |
| $\dfrac{s \doteq t}{t \doteq s}$ | Symmetry |
| $\dfrac{s \doteq t \quad t \doteq u}{s \doteq u}$ | Transitivity |
| $\dfrac{s_1 \doteq t_1 \ldots s_n \doteq t_n}{f(s_1, \ldots, s_n) \doteq f(t_1, \ldots t_n)}$ | Compatibility |
| $\dfrac{s \doteq t}{\theta s \doteq \theta t}$ | Instantiation |

where $\theta$ is a substitution.

5.2.3. Definition. Let $\mathcal{S} = (\Sigma, E)$ be an equational theory. A $\Sigma$-equation $s \doteq t$ is a *theorem* of $\mathcal{S}$ (written $\mathcal{S} \vdash s \doteq t$) if it is deducible from $E$ using the rules for equational deduction.

**Term Rewriting.** Let $\Sigma$ be a signature.

5.2.4. Definition. A $\Sigma$-*rewrite rule* is a pair of $\Sigma$-terms $(s, t)$, usually written $s \to t$, such that $s$ is a non-variable term and $\mathsf{var}(t) \subseteq \mathsf{var}(s)$. A $\Sigma$-*rewrite system* is a set of rewrite rules.

As usual, we talk about rewrite rules and rewrite systems when there is no risk of confusion. Note that every $\Sigma$-rewrite system $\mathcal{R}$ induces an equational theory $(\Sigma, \mathcal{R})$, simply by seeing rewrite rules as equations. By *abus de notation*, we shall denote this equational theory by $\mathcal{R}$.

Let $\mathcal{R}$ be a rewrite system and $s$ and $t$ be two $\Sigma$-terms. We say that $s$ *one step* $\mathcal{R}$*-rewrites* to $t$ (notation $s \to_{\mathcal{R}} t$) if there exist an occurrence $u$ of $s$, a rewrite rule $(l, r)$ in $\mathcal{R}$ and a $\Sigma$-substitution $\theta$ satisfying $s/u = \theta l$ and $t = s[u \leftarrow \theta r]$.

We let $\twoheadrightarrow_{\mathcal{R}}$ and $\leftrightarrow_{\mathcal{R}}$ be respectively the reflexive transitive and the reflexive, symmetric and transitive closure of $\to_{\mathcal{R}}$. Finally, $s \downarrow_{\mathcal{R}} t$ if there exists $u$ such that $s \twoheadrightarrow_{\mathcal{R}} u$ and $t \twoheadrightarrow_{\mathcal{R}} u$. Note that $\downarrow_{\mathcal{R}} \subseteq \leftrightarrow_{\mathcal{R}}$.

5.2.5. Definition. A rewrite system $\mathcal{R}$ is *confluent* if $\downarrow_{\mathcal{R}} = \leftrightarrow_{\mathcal{R}}$ and *terminating* if there is no infinite reduction sequence $t \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$. A rewrite system is *canonical* if it is both confluent and terminating.

5.2.6. Proposition. *Let $\mathcal{R}$ be a confluent rewrite system.*

$$(s \downarrow_{\mathcal{R}} t) \quad \Leftrightarrow \quad (s \leftrightarrow_{\mathcal{R}} t) \quad \Leftrightarrow \quad \mathcal{R} \vdash s \doteq t \ .$$

5.2.7. Remark. Algebraic structures are usually described equationally rather than as term rewriting systems. However, some of them can be turned into term rewriting systems using the Knuth-Bendix completion procedures (Klop 1992).

### 5.2.2 The Semantics of Equational Logic and the Completeness Theorem

Equational theories are syntactical descriptions of mathematical objects. The objects satisfying these descriptions are the mathematical structures themselves. In this section, we define a semantics for equational theories. As we are interested in using universal algebra to solve the problem of equational reasoning in type theory, our semantics is ultra-loose, i.e. the equality relation between terms is interpreted as an arbitrary equivalence relation rather than as the underlying equality of the model.

5.2.8. Definition. An $\Sigma$-*algebra* $\mathcal{A}$ for a signature $\Sigma$ consists of a set $A$, an equivalence relation $=_{\mathcal{A}}$ on $A$ and for each function symbol $f$ of arity $n$, a function $f^{\mathcal{A}} : A^n \to A$ such that for every $(a_1, \ldots, a_n), (a'_1, \ldots, a'_n) \in A^n$,

$$a_1 =_{\mathcal{A}} a'_1, \ldots, a_n =_{\mathcal{A}} a'_n \quad \Rightarrow \quad f^{\mathcal{A}}(a_1, \ldots, a_n) =_{\mathcal{A}} f^{\mathcal{A}}(a'_1, \ldots, a'_n) \ .$$

For implementation purposes, we use a slightly modified definition of assignment and satisfiability. Of course, the resulting semantics is equivalent to the standard one.

5.2.9. Definition. An $\mathcal{A}$-*assignment* is a partial map $\alpha : V \rightharpoonup A$ with a non-empty, finite domain.

Any $\mathcal{A}$-assignment can be extended inductively to a partial function $[\![\_]\!]^{\mathcal{A}}_{\alpha}$ on the set of $\Sigma$-terms:

$$\begin{array}{rcll} [\![x]\!]^{\mathcal{A}}_{\alpha} & \simeq & \alpha x & \text{if } x \in \mathsf{dom}\ \alpha \\ [\![f(t_1, \ldots, t_n)]\!]^{\mathcal{A}}_{\alpha} & \simeq & f^{\mathcal{A}}([\![t_1]\!]^{\mathcal{A}}_{\alpha}, \ldots, [\![t_n]\!]^{\mathcal{A}}_{\alpha}) \ . & \end{array}$$

5.2.10. Definition. Let $\mathcal{A}$ be a $\Sigma$-algebra. Two $\mathcal{A}$-assignments $\alpha$ and $\beta$ are *compatible* if $\mathsf{dom}\ \alpha = \mathsf{dom}\ \beta$ and $\alpha x =_{\mathcal{A}} \beta x$ for all $x \in \mathsf{dom}\ \alpha$.

The following lemma shows that compatible assignments model the same equations.

5.2.11. Lemma (Compatibility lemma). *Let $\mathcal{A}$ be a $\Sigma$-algebra. Let $\alpha$ and $\beta$ be two compatible $\mathcal{A}$-assignments. Let $t$ be a $\Sigma$-term such that* $\mathsf{var}(t) \subseteq \mathsf{dom}\ \alpha$. *Then* $[\![t]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\beta}$.

We write $\mathcal{A} \models s \doteq t$ if for all $\mathcal{A}$-assignments $\alpha$ such that $\mathsf{var}(s) \cup \mathsf{var}(t) \subseteq \mathsf{dom}\ \alpha$,

$$[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha} \ .$$

5.2.12. Definition. Let $\mathcal{S} = (\Sigma, E)$ be an equational theory. A $\Sigma$-algebra $\mathcal{A}$ is a $\mathcal{S}$-model if $\mathcal{A} \models s \doteq t$ for all the equations $s \doteq t$ in $E$.

We say that $\mathcal{S} = (\Sigma, E)$ *semantically entails* a $\Sigma$-equation $s \doteq t$ (notation $\mathcal{S} \models s \doteq t$) if $\mathcal{A} \models s \doteq t$ for every $\mathcal{S}$-model $\mathcal{A}$. The fundamental theorem of equational logic establishes the compatibility between syntax and semantics.

5.2.13. Theorem (Soundness/Completeness). *For every $\Sigma$-equation $s \doteq t$,*

$$\mathcal{S} \vdash s \doteq t \quad \Leftrightarrow \quad \mathcal{S} \models s \doteq t \ .$$

The completeness result is proved by constructing the term model $T_{\mathcal{S}}$ as the quotient of $T_\Sigma$ by the provability relation $\sim_{\mathcal{S}}$. The crucial fact that we shall exploit later is that for every term $s$ and $t$,

$$\mathcal{S} \vdash s \doteq t \quad \Leftrightarrow \quad [s] = [t]$$

where $[\_] : T_\Sigma \to T_{\mathcal{S}}$ is the canonical map assigning to every term its equivalence class under the provability relation.

## 5.3 The Naming Principles

In this section, we define a methodology to solve equational problems in type theory. Our methodology is very flexible and can be carried out in any type system with inductive types. In particular, it can be carried out in the underlying type systems of LEGO (Luo and Pollack 1992), Coq (Dowek et al. 1993), Alf (Magnusson and Nordström 1994) and NuPrl (Constable et al. 1986).

### 5.3.1 Specifying the Problem to be Solved

Our first task is to fix the boundaries of the problem to be solved. In its most general form, equational reasoning is concerned with determining whether two elements $s$ and $t$ of a set $V$ of values are related by an equality relation $R$. Naturally, the problem is far too general to have an automated solution. Yet there is a well-understood branch of mathematical logic, namely equational logic, which is concerned with equational theories, i.e. first-order languages with a single (binary) predicate symbol $=$. Equational logic provides the *right level of generality* to tackle the problem of equational reasoning for several reasons:

1. the problem is general enough: a wide collection of mathematical theories can be presented equationally, for example the theories of monoids, groups and rings;
2. one might expect a useful and automated solution to the problem: in some cases, it is possible to provide an algorithm to test whether an equation of a given theory $\mathcal{S}$ is a theorem of this theory;
3. this work can provide a theoretical foundation to integrate computer algebra systems and proof checkers: computer algebra systems, with their impressive power, are mostly concerned with equational theories.

This justifies the following choice for the form of an equational problem.

THE PROBLEM. *Let $\mathcal{S}$ be an equational theory. Let $\mathcal{A}$ be a model of $\mathcal{S}$. Let $a$ and $b$ be expressions for elements of $\mathcal{A}$. Does $a =_{\mathcal{A}} b$?*

Note that the problem makes sense within a type system with inductive types as one can formalize all basic notions of universal algebra in such a system. Here are a few examples of equational problems.

5.3.1. EXAMPLES. – Let $\mathbb{Z}_n$ be the ring of integers modulo $n$, where $n \geq 3$. Does $2(n-1) = 0$?

– Let $\boldsymbol{D}_8$ be the dihedral group with eight elements. Let $\sigma, \tau \in \boldsymbol{D}_8$. Does $\tau\sigma = \sigma^3\tau$? Here the problem is quantified over all elements of $\boldsymbol{D}_8$.

– Let $(M, =_M, \circ_M, e_M)$ be a monoid. Let $x, y \in M$. Does $(x \circ_M e) \circ_M y =_M x \circ_M y$? Here the problem is quantified over all $x, y \in M$ and monoids $M$.

To solve the problem, we will first relate it to equational logic and then use equational logic to solve the problem automatically.

For the remainder of this section, we work with the formalization of universal algebra in the type system. In particular, an equational theory is an inhabitant of the type of equational theories, and a model of a theory is an inhabitant of the type of models of this theory. To alleviate the presentation, we will still use the ordinary language of universal algebra.

In the sequel, we let $\mathcal{S} = (\Sigma, E)$ be a fixed equational theory and $\mathcal{A}$ be a model of $\mathcal{S}$.

### 5.3.2 Equational Logic, Local Equational Logic and Equational Reasoning

Equational logic is *global* in the sense that it is used to determine whether a $\mathcal{S}$-equation $s \doteq t$ is true in all models of $\mathcal{S}$, i.e. whether $\mathcal{S} \models s \doteq t$. In contrast, equational reasoning is *local*, in the sense that one is also interested whether a given equality holds in a specific model, i.e. $a =_\mathcal{A} b$ for some specific $a$ and $b$ in a specific model $\mathcal{A}$ of $\mathcal{S}$. An intermediate formal system is *local equational logic*, a variant of equational logic whose deductive system allows to infer whether $\mathcal{A} \models s \doteq t$ for a specific model $\mathcal{A}$ of $\mathcal{S}$. One could even go one step further and develop a formal system to infer whether $[\![s]\!]^\mathcal{A}_\alpha =_\mathcal{A} [\![t]\!]^\mathcal{A}_\alpha$ in a specific model $\mathcal{A}$ and for a specific assignment $\alpha$. This last problem, which we call the *local satisfiability problem* is in fact a special instance of equational reasoning. If we analyze the logical formulations of local satisfiability and semantical entailment, we see that the latter represents a uniform notion of the former[1]. One concludes that *the goal of equational logic is to know whether a uniform collection of equational problems is satisfied.*

Local satisfiability is a very common form of equational problem. However, not all equational problems arising in the formalization of mathematics are concerned with local satisfiability. An equally important instance of equational problem is the *extensionality problem*: given a $\mathcal{S}$-term $t$, a model $\mathcal{A}$ of $\mathcal{S}$ and two interpretations $\alpha, \beta$ in $\mathcal{A}$, does $[\![t]\!]^\mathcal{A}_\alpha =_\mathcal{A} [\![t]\!]^\mathcal{A}_\beta$? In fact, those two problems (local satisfiability and extensionality) form the core of equational reasoning.

### 5.3.3 The Naming Principles

As outlined in the previous subsection, there is a divergence between equational logic as a formal system and equational reasoning as it occurs in mathematics. We have

*a goal:* an equational problem, i.e. an equality $a =_\mathcal{A} b$;

*some tools:* equational logic, which can be used to solve a local satisfiability problem, and the compatibility lemma, which can be used to solve an extensionality problem.

[1]By the soundness/completeness theorem, $\mathcal{S} \models s \doteq t$ is equivalent to the collection of local satisfiability problems $([\![s]\!]^\mathcal{A}_\alpha =_\mathcal{A} [\![t]\!]^\mathcal{A}_\alpha)_{(\mathcal{A} \in \mathcal{M}, \alpha \in \mathcal{V}(\mathcal{A})}$ where $\mathcal{M}$ is the collection of $\mathcal{S}$-models and for $\mathcal{A} \in \mathcal{M}$, $\mathcal{V}(\mathcal{A})$ is the set of $\mathcal{A}$-assignments.

The difficulty in applying the tools to solve the goal is that equational problems are essentially of a semantical nature while equational logic is designed to solve syntactical problems. In order to apply equational logic to equational reasoning, one must perform a preliminary manipulation on equational problems, so that they present themselves in a form which is amenable to be solved by equational logic. What is needed here is a *naming principle* which transforms a semantical equational problem into a local satisfiability problem or an extensionality problem. For the clarity of the discussion, we will therefore distinguish between the *naming principle for satisfiability* (for short NPS) and the *naming principle for extensionality* (for short NPE). One fundamental feature of these naming principles is that they do not require any extension of the type system; indeed, the naming principles are a special instance of conversion rules. We introduce these principles below.

**The Naming Principle for Satisfiability.**

The aim of the naming principle for satisfiability is to recast a local equation $a =_{\mathcal{A}} b$ into an equation of the form $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$, where

– $s$ and $t$ are terms of the theory $T$,

– $\alpha$ is an assignment,

– $[\![s]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$,

– $[\![t]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow b$.

Of course, the equation to be solved has not changed; what has changed is the way to look at it. The equation in its second form makes it clear that the problem to be solved is an instance of a uniform collection of equational problems, as defined in the previous section. The advantage of this switch of perspective is that the equation in its second form is more amenable to be solved by standard syntactic tools. Indeed, $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$ is an immediate consequence of $\mathcal{S} \vdash s \doteq t$. This yields a semi-complete[2] method to prove $a =_{\mathcal{A}} b$:

1. apply the NPS; this reduces the equational problem to one of the form $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$;
2. apply any method available to prove $\mathcal{S} \vdash s \doteq t$.

Of course, the efficiency of the method depends on the choice of $s$ and $t$[3]. Fortunately, there is always an optimal application of the NPS.

5.3.2. Definition. Let $\mathcal{A}$ be a model of $\mathcal{S}$. Let $a$ be an element of $\mathcal{A}$. The *pre-order of codes* of $a$ is the sub-pre-order of $T^{\leq}_{\Sigma}$ whose elements are the terms $t$ for which there exists an assignment $\alpha$ such that $[\![t]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$.

For every element $a$ of $\mathcal{A}$, the pre-order of codes of $a$ has a top element (unique up to renaming), called the *optimal code* of $a$. We write $\ulcorner a \urcorner$ for the optimal code of $a$.

Similarly, we can define a *code for an equational problem* $a =_{\mathcal{A}} b$ to be an equation $s \doteq t$ such that for some assignment $\alpha$, $[\![s]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$ and $[\![t]\!]^{\mathcal{A}}_{\beta} \twoheadrightarrow b$. Every

[2]The method can fail even if the equational problem is true.

[3]Indeed, some uses of the NPS can be less than judicious. Every equational problem $a =_{\mathcal{A}} b$ can be reduced by the NPS to $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$ where $s$ and $t$ are distinct variables and $\alpha$ is any assignment satisfying $\alpha s = a$ and $\alpha t = b$. In order to solve the problem according to the proposed method, we must now solve $\mathcal{S} \vdash s \doteq t$. This only holds if the theory is inconsistent!

equational problem $a =_{\mathcal{A}} b$ has an *optimal code* $\ulcorner a \urcorner \doteq \ulcorner b \urcorner$ (one can verify that $\ulcorner a \urcorner$ and $\ulcorner b \urcorner$ are optimal codes for $a$ and $b$ respectively) with the two properties:

– $\ulcorner a \urcorner \doteq \ulcorner b \urcorner$ is a code for $a =_{\mathcal{A}} b$;

– $\mathcal{S} \vdash \ulcorner a \urcorner \doteq \ulcorner b \urcorner$ if and only if $\mathcal{S} \vdash s \doteq t$ for some code $s \doteq t$ of $a =_{\mathcal{A}} b$.

The conclusion is that one can define an algorithm which performs the optimal choice for the NPS. In the sequel, it is understood that the NPS is always applied for such an optimal choice.

**The Naming Principle for Extensionality.**

The aim of the naming principle for extensionality is to recast a local equation $a =_{\mathcal{A}} b$ into an equation $[\![t]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\beta}$, where

– $t$ is a term of the theory $T$,

– $\alpha$ and $\beta$ are assignments,

– $[\![t]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$,

– $[\![t]\!]^{\mathcal{A}}_{\beta} \twoheadrightarrow b$.

In the second form, the equation can be immediately deduced from $\alpha x =_{\mathcal{A}} \beta x$ for all $x \in \mathsf{var}(t)$. As for the NPS, the method is only semi-complete. Yet it is a very important tool for formal proof development. Indeed, the standard representation of sets in most type systems uses the so-called setoids; consequently all the reasoning takes place with book equalities and extensionality matters do come up very often. As for the NPS, the NPE can be applied optimally. Indeed, one can find for every equational problem $a =_{\mathcal{A}} b$ a term $t$ (the *optimal code* for NPE) such that

– there exist two assignments $\alpha$ and $\beta$ such that $[\![t]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$ and $[\![t]\!]^{\mathcal{A}}_{\beta} \twoheadrightarrow b$;

– for every term $t'$ and assignments $\delta$ and $\gamma$ such that $[\![t']\!]^{\mathcal{A}}_{\delta} \twoheadrightarrow a$ and $[\![t']\!]^{\mathcal{A}}_{\gamma} \twoheadrightarrow b$, there exists a substitution $\theta$ such that $\theta t' \twoheadrightarrow t$.

Note that it is possible to extend the naming principle for extensionality to formulae.

**Combining Both Principles.**

In the previous subsections, we have considered two different naming principles which can be used to solve equational problems. However, the method that we have described disregards the possibility of using assumptions present in the context. In fact, the NPS is too weak to be useful in this more general case. For example, if one has to prove in a monoid $M$ that

$$(a \circ b) \circ c =_{\mathcal{A}} a' \circ (b \circ c) \tag{5.2}$$

for some elements $a$, $a'$, $b$ and $c$ of $M$ such that $a =_{\mathcal{A}} a'$, the NPS reduces the problem to

$$[\![(x \cdot y) \cdot z]\!]^{\mathcal{A}}_{\gamma} =_{\mathcal{A}} [\![x' \cdot (y \cdot z)]\!]^{\mathcal{A}}_{\gamma} \tag{5.3}$$

for a suitable assignment $\gamma$. Moreover, one cannot invoke the NPE principle to reduce equation (5.3) further. However, one can combine the NPS and the NPE to obtain a powerful naming principle (NPSE) which can be used to solve equational problems in a context. This new principle takes as input an equational problem $a =_{\mathcal{A}} b$ and returns as output an equation $[\![s]\!]^{\mathcal{A}}_{\alpha} = [\![t]\!]^{\mathcal{A}}_{\beta}$ where $s$ and $t$ are two terms $s$ and $t$ and $\alpha$ and $\beta$ are two assignments such that

- $[\![s]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$,
- $[\![t]\!]^{\mathcal{A}}_{\beta} \twoheadrightarrow b$,
- $\mathsf{dom}\ \alpha = \mathsf{dom}\ \beta$ and $\alpha x = \beta x$ for every $x \in \mathsf{dom}\ \alpha$.

As for the NPS, the equation follows from $\mathcal{S} \vdash s \doteq t$. With this new principle, equation (5.2) can be reduced to $[\![(x \cdot y) \cdot z]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![x \cdot (y \cdot z)]\!]^{\mathcal{A}}_{\beta}$ and $\alpha x = \beta x$ for suitable $\alpha$ and $\beta$. This shows that the NPSE is stronger than the combination of the NPS and the NPE. However, it is difficult to find an optimal use of the NPSE for obvious reasons. Fortunately, one can recover the power of the NPSE from the NPS by grafting a simple procedure on top of the NPE. The procedure, called *collapsing procedure* (or CP for short),

- takes as input a problem of the form $[\![s]\!]^{\mathcal{A}}_{\alpha} = [\![t]\!]^{\mathcal{A}}_{\alpha}$ and two variables $x$ and $y$ in the domain of $\alpha$,
- returns as output the problems $[\![s[y/x]]\!]^{\mathcal{A}}_{\alpha} = [\![t[y/x]]\!]^{\mathcal{A}}_{\alpha}$ and $\alpha x = \alpha y$.

The benefits of the CP are similar to those of the NPSE. For example, the CP can be called to reduce equation (5.3) into the two problems

$$\begin{aligned} [\![(x \cdot y) \cdot z]\!]^{\mathcal{A}}_{\gamma} &=_{\mathcal{A}} [\![x \cdot (y \cdot z)]\!]^{\mathcal{A}}_{\gamma} \\ \gamma x &=_{\mathcal{A}} \gamma x' \ . \end{aligned}$$

The CP provides an easy means to make use of the optimal naming of an equational problem via the NPS. Unfortunately, there does not seem to be any obvious counterpart for making use of the optimal naming of an equational problem via the NPE[4].

## 5.4 Congruence Types

### 5.4.1 How to Automate Equational Reasoning?

As mentioned earlier, the naming principles do not solve equational problems. A naming principle is a special kind of conversion rule which recasts an equational problem into a specific form. Here these specific forms are local satisfiability and extensionality problems. The point is the naming principles make apparent terms of an equational theory. In the special case where we look at a local satisfiability problem, the equational problem will become of the form $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$. By the soundness/completeness theorem, the equality is a consequence of $\mathcal{S} \vdash s \doteq t$. Reducing an equational problem to a problem of the form $\mathcal{S} \vdash s \doteq t$ is useful because we dispose of techniques to determine whether an equation is in the deductive closure of an equational theory:

*using computer algebra systems.* Current computer algebra systems are excellent at equational reasoning. They have various clever algorithms to compute all kinds of equations at a symbolic (syntactical) level. We could use such a system to compute $s \doteq t$ and, if this succeeds, we let our proof checker assume the statement as an axiom. This is what we call the *external believing* way.

[4]Consider a monoid $H$ and three elements $a, b, c$ of $H$ such that $a \circ b =_H a' \circ b'$. The optimal use of the NPE on

$$(a \circ b) \circ c =_H (a' \circ b') \circ c$$

will yield the two subproblems $a =_H a'$ and $b =_H b'$. Indeed, the NPE will be applied with $(x \circ y) \circ z$ as code whereas it would have been better to take $x \circ y$ as code.

*using term rewriting.* Another technique to check $\mathcal{S} \vdash s \doteq t$ is of course term rewriting: if $\mathcal{S}$ can be completed into a confluent and terminating term rewriting system $\mathcal{R}$, we can look at the normal form of $s$ and $t$ with respect to the completion of $\mathcal{S}$. For such theories, equational reasoning can be partially automated by using the naming principle and importing in some way term rewriting into the type theory as done for example in (Breazu-Tannen 1988). We call this method the *internal believing* way, because the problem is solved without any outside help. This is the method proposed in this chapter.

*the autarkic way.* We might want to define a map `nf` which assigns to every term its normal form in $\mathcal{R}$ and to show that for every term $t$ and assignment $\alpha$, we have $[\![t]\!]^{\mathcal{A}}_{\alpha} =_{\beta\iota} [\![\texttt{nf}\ t]\!]^{\mathcal{A}}_{\alpha}$. In order to check $s \doteq t$, we just have to verify $(\texttt{nf}\ s) = (\texttt{nf}\ t)$, where $=$ denotes Leibniz equality. This comes down to a reflexivity test. This method is called the *autarkic* way because it does not involve any change to the type theory or the proof-checker. It must be said that this method seems currently too inefficient to be used in practice.

Most proposals in the literature opt for the external believing approach (Ballarins, Homann and Calmet 1995, Harrison and Théry 1993, Jackson 1994). Indeed, the external believing way has an obvious advantage: hybrid systems offer a shortcut to integrate term rewriting in proof checking. However, the approach has two disadvantages:

- proof checkers are based on well-understood languages whose logical and computational status are well understood. It is not always the case for computer algebra systems.
- proof checkers generate from scripts proof-objects; if the computer algebra system is used as an congruence, then all calculations performed by the computer algebra system have to be taken as axioms by the proof checker. Such a process threatens the reliability of the hybrid system[5].

One can remedy to these two problems by using the computer algebra system not as an congruence but as a guide, as done in (Harrison and Théry 1993). In this case, the answer of the computer algebra system is used to solve an equation. We call this method the *skeptic* way because the proof-checker does not trust the computer algebra system. This technique is superior over the external believing one in that it eliminates the holes in the proof-terms. Moreover, the problem of the reliability of the computer algebra system is circumvented. However the skeptic way seems infeasible in a proof-checker such as LEGO because of the absence of tactics.

### 5.4.2 The Internal Believing Approach via Congruence Types

In this section, we introduce congruence types. The formalism, which is based on algebraic, inductive and quotient types, is well-suited for the introduction of canonical term rewriting systems. We refer the reader to (Barthe and Geuvers 1996) for a general scheme for congruence types and focus on a specific example of congruence type used to solve equational problems for groups. It consists of two types:

[5]Sometimes the user has to make sure that the necessary side conditions are satisfied. For example, several computer algebra systems will state that $(\sqrt{x})^2$ equals $x$, without bothering about the condition that $x \geq 0$.

– an inductive type $\underline{G}$ corresponding to the set of terms of the signature of groups,
– the quotient $G$ of $\underline{G}$ by the deductive closure of the theory of groups; $G$ is defined as an algebraic type, i.e. equality between inhabitants of $G$ is forced by the rewrite rules.

Both types are related by a map $[\_] : \underline{G} \to G$ which assigns to every term its equivalence class under the provability relation. There is an axiom to reflect the universal property of quotients as it is used in the completeness theorem: an equation $s \doteq t$ holds in every group if $[s] = [t]$. If we work in ECC (Luo 1994), the rules are:

$$\frac{}{\vdash \underline{G} : \Box_0} \qquad \frac{}{\vdash \underline{e} : \underline{G}} \qquad \frac{}{\vdash \underline{i} : \underline{G} \to \underline{G}} \qquad \frac{}{\vdash \underline{o} : \underline{G} \to \underline{G} \to \underline{G}}$$

$$\frac{}{\vdash G : \Box_0} \qquad \frac{}{\vdash e : G} \qquad \frac{}{\vdash i : G \to G} \qquad \frac{}{\vdash o : G \to G \to G}$$

$$\frac{}{\vdash \underline{a} : \mathbb{N} \to \underline{G}} \qquad \frac{}{\vdash a : \mathbb{N} \to G} \qquad \frac{\Gamma \vdash b : \underline{G}}{\Gamma \vdash [b] : G} \qquad \frac{\Gamma \vdash p : [a] = [b]}{\Gamma \vdash \texttt{noconf}\ p : a =_{\underline{G}} b}$$

$$\frac{\Gamma \vdash C : \Box_0 \quad \begin{array}{c}\Gamma \vdash f_e : C \\ \Gamma \vdash f_a : \mathbb{N} \to C\end{array} \quad \begin{array}{c}\Gamma \vdash f_i : \underline{G} \to C \to C \\ \Gamma \vdash f_o : \underline{G} \to \underline{G} \to C \to C \to C\end{array}}{\epsilon^C\ [f_a, f_e, f_i, f_o] : \underline{G} \to C}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash B : s}{\Gamma \vdash a : B} \qquad \text{if } A \to_{\beta\chi\iota\rho} B \text{ or } B \to_{\beta\chi\iota\rho} A$$

where $=_{\underline{G}}$ is the (impredicatively defined) deductive closure of the theory of groups, $[\_]$ is a new constructor and $\mathbb{N}$ are the inductively defined natural numbers. The computational content of the system is given by $\beta$-reduction and the following reduction relations:

– $\iota$-reduction; let $\vec{f} = (f_a, f_e, f_i, f_o)$. The rules are

$$\begin{array}{rcl}
\epsilon^C\ [\vec{f}]\ (a\ i) & \to_\iota & f_a\ i \\
\epsilon^C\ [\vec{f}]\ e & \to_\iota & f_e \\
\epsilon^C\ [\vec{f}]\ (i\ x) & \to_\iota & f_i\ x\ (\epsilon^C\ [\vec{f}]\ x) \\
\epsilon^C\ [\vec{f}]\ (o\ x\ y) & \to_\iota & f_o\ x\ y\ (\epsilon^C\ [\vec{f}]\ x)\ (\epsilon^C\ [\vec{f}]\ y)
\end{array}$$

– $\rho$-reduction; the rules correspond to the Knuth-Bendix completion of the axioms of groups:

$$\begin{array}{rclrcl}
o\ e\ x & \to_\rho & x & i\ e & \to_\rho & e \\
o\ x\ e & \to_\rho & x & o\ (o\ x\ (i\ y))\ y & \to_\rho & x \\
o\ x\ (o\ y\ z) & \to_\rho & o\ (o\ x\ y)\ z & o\ (o\ x\ y)\ (i\ y) & \to_\rho & x \\
o\ (i\ x)\ x & \to_\rho & e & i\ (i\ x) & \to_\rho & x \\
o\ x\ (i\ x) & \to_\rho & e & i\ (o\ x\ y) & \to_\rho & o\ (i\ y)\ (i\ x)
\end{array}$$

– $\chi$-reduction; for every $x, y : \underline{G}$,

$$\begin{array}{rcl}
[\underline{o}\ x\ y] & \to_\chi & o\ [x]\ [y] \\
[\underline{i}\ x] & \to_\chi & i\ [x] \\
[\underline{e}] & \to_\chi & e
\end{array}$$

## 5.5 Formalization in LEGO

Type theory based proof checkers such as Alf, Coq and LEGO are expressive enough for the two-level approach described above to be developed within the system itself. We present an implementation of the two-level approach in LEGO. The reason to choose LEGO is that it allows for the user to input its own rewrite rules, thus offering the possibility to implement congruence types.

### 5.5.1 Formalization of Equational Logic

Formalizing equational logic in LEGO is relatively easy. There are no major difficulties in developing the whole theory along the lines of Section 5.2.

Equations are defined as pairs of terms and equational theories as signatures together with a predicate over the type of equations. One can even formalize the deductive closure of a set of equations by formalizing first the notion of simultaneous substitution. It is equally easy to define the semantics of equational logic. The definitions of algebra, assignment, satisfaction and model are immediate adaptations of the definitions introduced in Section 5.2. See Section 3.4 for a detailed presentation of our implementation of universal algebra in LEGO.

### 5.5.2 Formalization of the Naming Principles

LEGO does not offer support for the naming and extensionality principles[6]. Yet they are special instances of conversion rules, so they can be performed manually using the `Equiv` command. We present three examples, one using the NPE, a second using the CP and the third one using the NPS. These examples are meant to give an idea of the method used. To understand them fully, the reader should read first Section 5.6. In each case, the proofs turn out to be remarkably short. Note that in our implementation we did not use (nor need) specifications of equational theories.

First, we give an example where the NPE is used to solve an equational problem. Here `G` is an algebra for the signature of groups, `obj G` is an element of its carrier, `times` is the multiplication on `G` and `inv` is the inverse on `G`. `TIMES` and `INV` are function symbols of the signature of groups. `int` is the interpretation function which, given an assignment `rho`, assigns a symbol of the signature to an element of `G` whose set of variables is contained in the domain of `rho`. Note that `[x:A]b` stands for $\lambda x{:}A.b$, `{x:A}B` for $\Pi x{:}A.B$, `<x:A>B` for $\Sigma x{:}A.B$, `Set` stands for the type of setoids, `Eq` for the equality of a `Set`, `el` for the elements of a `Set`, `obj` for the elements of a model and `Q` is Leibniz equality.

```
Lego> Goal {x,y,z:obj G} (Eq x y) ->
                         (Eq (times (inv x) z) (times (inv y) z));
Goal
  ?0 : {x,y,z:obj G}(Eq x y)->Eq (times (inv x) z) (times (inv y) z)
Lego> intros;
intros (4)
  x : obj G
  y : obj G
  z : obj G
  H : Eq x y
```

[6]An extension of the LEGO system is proposed in (Barthe and Elbers 1996) to solve this problem.

```
  ?1 : Eq (times (inv x) z) (times (inv y) z)
Lego> rho == necons x (necons y (base z));
defn  rho = necons x (necons y (base z))
      rho : nelist (obj G)
Lego> t   == TIMES (INV (VAR ZeroN)) (VAR TwoN);
defn  t = TIMES (INV (VAR ZeroN)) (VAR TwoN)
      t : termGr
Lego> u   == TIMES (INV (VAR  OneN)) (VAR TwoN);
defn  u = TIMES (INV (VAR OneN)) (VAR TwoN)
      u : termGr
Lego> Equiv Eq (int G rho t) (int G rho u);
Equiv
  ?2 : Eq (int G rho t) (int G rho u)
Lego> Refine SubstitutionLemma G ZeroN;
Refine by  SubstitutionLemma G ZeroN
  ?9 : Eq (int G rho (TFV sig ZeroN)) (int G rho (VAR OneN))
Lego> Refine H;
Refine by  H
Discharge..  rho H z y x
*** QED ***
```

Note that the NPE yields the goal ?2. The SubstitutionLemma is used to obtain ?9 is a specific instance of the compatibility lemma. The next example uses the CP procedure. Here we are working in a context in which times_assoc states that times is associative. The CP procedure is called by the term CP.

```
Lego> Goal {a,b,b',c:obj G} (Eq b b') ->
           Eq (times a (times b c)) (times (times a b') c);
Goal
  ?0 : {a,b,b',c:obj G} (Eq b b') ->
                          (Eq (times a (times b c)) (times (times a b') c)
Lego> intros;
intros (5)
  a : obj G
  b : obj G
  b' : obj G
  c : obj G
  H : Eq b b'
  ?1 :Eq (times a (times b c)) (times (times a b') c)
Lego> rho == necons a (necons b (necons b' (base c)));
defn  rho = necons a (necons b (necons b' (base c)))
      rho : nelist (obj G)
Lego> t   == TIMES (VAR ZeroN) (TIMES (VAR OneN) (VAR ThreeN));
defn  t = TIMES (VAR ZeroN) (TIMES (VAR OneN) (VAR ThreeN))
      t : termGr
Lego> u   == TIMES (TIMES (VAR ZeroN) (VAR TwoN)) (VAR ThreeN);
defn  u = TIMES (TIMES (VAR ZeroN) (VAR TwoN)) (VAR ThreeN)
      u : termGr
Lego> Equiv Eq (int G rho t) (int G rho u);
Equiv
  ?1 : Eq (int G rho u) (int G rho u)
Lego> Refine CP G OneN (VAR TwoN);
Refine by  CP G OneN (VAR TwoN)
  ?9 : Eq (int G rho (TFV sig OneN)) (int G rho (VAR TwoN))
  ?10 : Eq (int G rho (Subst t OneN (VAR TwoN)))
           (int G rho (Subst u OneN (VAR TwoN)))
Lego> Refine H;
Refine by  H
?10 : ...
Lego> Refine times_assoc;
Refine by  times_assoc
Discharge..  rho H c b' b a
*** QED ***
```

The final example uses the NPS. Congruence types are used to give a short proof of an equality on groups. In the sequel, Q_refl is a proof of the reflexivity of Leibniz equality, comm and conj respectively denote the commutator and the conjugate of two elements. For comparison, we have included a traditional proof of this fact in Section 5.6.

```
Goal {x,y,z:obj G} Eq (conj (comm x y) z) (comm (conj x z) (conj y z));
  intros;
  rho == necons x (necons y (base z));
  t == CONJ (COMM (VAR ZeroN) (VAR OneN)) (VAR TwoN);
  u == COMM (CONJ (VAR ZeroN) (VAR TwoN)) (CONJ (VAR OneN) (VAR TwoN));
  Equiv Eq (int G rho t) (int G rho u);
  Refine Soundness;
  Refine Q_refl;
Save comm_conj;
```

## 5.6 Examples

This section contains examples of equational problems solved using our approach. To keep the presentation simple, we introduce the group axioms without using an equational theory. Note that because of the two-level approach, the number of LEGO commands of the proof comm_conj is very small (in essence only four). This in contrast to the traditional proof comm_conj_hand. Because of a lot of applications of the transitivity of equality and the group axioms, the proof explodes up to a few pages of LEGO commands.

```
Module Examples Import syntax semantics;

(* Define the signature and the terms of a Group. *)

[sigGr  : Signature                = ...]
[termGr : SET                      = term sigGr]
[VAR    : nat -> termGr            = TFV sigGr]
[ONE    : termGr                   = ...]
[INV    : termGr -> termGr         = ...]
[TIMES  : termGr-> termGr -> termGr = ...];
[DIV    : termGr-> termGr -> termGr = ...];

(* Let G be a group, satisfying the group axioms. *)

[G : Algebra sigGr];
```

```
[One   : el (car G)                   = ...]
[Inv   : Fun (car G) (car G)          = ...]
[Times : Fun2 (car G) (car G) (car G)  = ...]
[one   : obj G                        = ...]
[inv   : (obj G) -> (obj G)           = ...]
[times : (obj G) -> (obj G) -> (obj G) = ...];

[One_ident   : Identity Times One   ]
[Inv_invers  : Inverse Times One Inv]
[Times_assoc : Associative Times    ];

(* Show y = z  ->  z ((x/y) y) = z ((x/z) z) *)

Goal {x,y,z:obj G} (Eq y z) -> Eq (times (times y (times x (inv y))) z)
                                  (times (times z (times x (inv z))) z);
  intros;
  rho == necons x (necons y (base z));
  t == TIMES (TIMES (VAR OneN) (DIV (VAR ZeroN) (VAR OneN))) (VAR TwoN);
  u == TIMES (TIMES (VAR TwoN) (DIV (VAR ZeroN) (VAR TwoN))) (VAR TwoN);
  Equiv Eq (int G rho t) (int G rho u);
  Refine SubstitutionLemma G OneN;
  Refine H;
Save Example_1;

(* Show b = b'  ->  a (b c) = (a b') c *)

Goal {a,b,c,d:obj G} (Eq b d) -> Eq (times a (times b c))
                                    (times (times a d) c);
  intros;
  rho == necons a (necons b (necons c (base d)));
  t   == TIMES (VAR ZeroN) (TIMES (VAR OneN) (VAR TwoN));
  u   == TIMES (TIMES (VAR ZeroN) (VAR ThreeN)) (VAR TwoN);
  Equiv Eq (int G rho t) (int G rho u);
  Refine CP G OneN (VAR TwoN);
  Refine H;
  Refine Times_assoc;
Save Example_2;

(* ====================================================================
   Use Oracle Types to implement term rewriting.                     *)

[FreeGroup : SET];

[varFg   : nat -> FreeGroup];
[oneFg   : FreeGroup];
[invFg   : FreeGroup -> FreeGroup];
[timesFg : FreeGroup -> FreeGroup -> FreeGroup];

(* Define the Knuth-Bendix completion of the group equations. *)

[ [x,y,z : FreeGroup]

   timesFg oneFg x                ==> x
|| timesFg x oneFg                ==> x
|| timesFg (invFg x) x            ==> oneFg
|| timesFg x (invFg x)            ==> oneFg
|| invFg oneFg                    ==> oneFg
|| timesFg (timesFg x (invFg z)) z ==> x
|| timesFg (timesFg x y) (invFg y) ==> x
|| timesFg x (timesFg y z)        ==> timesFg (timesFg x y) z
|| invFg (invFg z)                ==> z
|| invFg (timesFg z y)            ==> timesFg (invFg y) (invFg z)
];

[class : termGr -> FreeGroup = ...];

[Soundness : {s,t:termGr} {rho:el (Assignment G)}
             (Q (class s) (class t)) -> Eq (int G rho s) (int G rho t)];

(* The conjugate of a commutator equals the commutator of the conjugate.
   Define the commutator [x,y] == (x y)/(y x)
   and the conjugate      x*y  == y (x/y)                              *)

[comm [x,y : obj G]   : obj G  = times (times x y) (inv (times y x))]
[COMM [x,y : termGr] : termGr = DIV (TIMES x y) (TIMES y x)]
[conj [x,y : obj G]   : obj G  = times y (times x (inv y))]
[CONJ [x,y : termGr] : termGr = TIMES y (TIMES x (INV y))];

(* Show [x,y]*z = [x*z,y*z] using the two-level approach. *)

Goal {x,y,z:obj G} Eq (conj (comm x y) z) (comm (conj x z) (conj y z));
  intros;
  rho == necons x (necons y (base z));
  t   == CONJ (COMM (VAR OneN) (VAR OneN)) (VAR TwoN);
  u   == COMM (CONJ (VAR OneN) (VAR TwoN)) (CONJ (VAR OneN) (VAR TwoN));
  Equiv Eq (int G rho t) (int G rho u);
  Refine Soundness;
  Refine Q_refl;
Save comm_conj;

(* Proof the last lemma again on the traditional way.
   First show x^{-1}^{-1} = x.                                         *)

Goal Involutive Inv;
  Intros x;
  Refine Eq_trans (times one (inv (inv x)));
    Refine Eq_sym; Refine fst One_ident;
  Refine Eq_trans (times (times x (inv x)) (inv (inv x)));
    Refine exten2 Times ??.Eq_refl;Refine Eq_sym; Refine snd Inv_invers;
  Refine Eq_trans (times x (times (inv x) (inv (inv x))));
    Refine Eq_sym; Refine Times_assoc;
  Refine Eq_trans (times x one);
    Refine exten2 Times ?.Eq_refl; Refine snd Inv_invers;
  Refine snd One_ident;
Save Inv_invol;
```

```
(* Show (x y)^{-1} = y^{-1} x^{-1} *)

Goal {x,y:obj G} Eq (inv (times x y)) (times (inv y) (inv x));
  intros;
  Refine Eq_sym;
  Refine Eq_trans (times (times (inv y) (inv x)) one);
    Refine Eq_sym; Refine snd One_ident;
  Refine Eq_trans (times (times (inv y) (inv x))
                         (times (times x y) (inv (times x y))));
    Refine exten2 Times ?.Eq_refl; Refine Eq_sym; Refine snd Inv_invers;
  Refine Eq_trans (times (times (times (inv y) (inv x)) (times x y))
                         (inv (times x y)));
    Refine Times_assoc;
  Refine Eq_trans (times one (inv (times x y)));
    Refine +1 fst One_ident;
  Refine exten2 ? ? ?.Eq_refl;
  Refine Eq_trans (times (inv y) (times (inv x) (times x y)));
    Refine Eq_sym; Refine Times_assoc;
  Refine Eq_trans (times (inv y) y);
    Refine +1 fst Inv_invers;
  Refine exten2 Times ?.Eq_refl;
  Refine Eq_trans (times (times (inv x) x) y);
    Refine Times_assoc;
  Refine Eq_trans (times one y);
    Refine exten2 Times ? ?.Eq_refl; Refine fst Inv_invers;
  Refine fst One_ident;
Save Times_Inv;

(* Show (x*z)(y*z) = (x y)*z *)

Goal {x,y,z:obj G}Eq (times (conj x z) (conj y z)) (conj (times x y) z);
  intros;
  Refine Eq_trans (times z (times (times x (inv z)) (conj y z)));
    Refine Eq_sym; Refine Times_assoc;
  Refine exten2 Times ?.Eq_refl;
  Refine Eq_trans (times x (times (inv z) (conj y z)));
    Refine Eq_sym; Refine Times_assoc;
  Refine Eq_trans (times x (times y (inv z)));
    Refine +1 Times_assoc;
  Refine exten2 Times ?.Eq_refl;
  Refine Eq_trans (times (times (inv z) z) (times y (inv z)));
    Refine Times_assoc;
  Refine Eq_trans (times one (times y (inv z)));
    Refine +1 fst One_ident;
  Refine exten2 Times ? ?.Eq_refl;
  Refine fst Inv_invers;
Save Times_Conj;

(* Show (x*y)^{-1} = x^{-1}*y *)

Goal {x,y:obj G} Eq (inv (conj x y)) (conj (inv x) y);
  intros;
  Refine Eq_trans (times (inv (times x (inv y))) (inv y));
```

```
    Refine Times_Inv;
  Refine Eq_trans (times (times y (inv x)) (inv y));
    Refine +1 Eq_sym; Refine +1 Times_assoc;
  Refine exten2 Times ? ?.Eq_refl;
  Refine Eq_trans (times (inv (inv y)) (inv x));
    Refine Times_Inv;
  Refine exten2 Times ? ?.Eq_refl;
  Refine Inv_invol;
Save Inv_conj;

(* And finally, show [x,y]*z = [x*z,y*z] *)

Goal {x,y,z:obj G} Eq (conj (comm x y) z) (comm (conj x z) (conj y z));
  intros;
  Refine Eq_sym;
  Refine Eq_trans (times (conj (times x y)z) (conj (inv(times x y)) z));
    Refine +1 Times_Conj;
  Refine exten2 Times; Refine Times_Conj;
  Refine Eq_trans (inv (conj (times x y) z));
    Refine exten Inv; Refine Times_Conj;
  Refine Inv_conj;
Save comm_conj_hand;
```

# Chapter 6

# Conclusions

We spent quite some effort in formalizing all kinds of mathematical definitions and lemma's in type theory. What did we learn from the exercise? Perhaps most surprising is that it took a lot more time to formalize proofs than we initially thought it would take. In the beginning of the project, most effort was spent in building from scratch a library full of very trivial lemma's. When we arrived at a higher level where we tried to prove more involved lemma's, formalizing was still hard. Namely, compared to the original mathematical texts, we had to add a lot of details and side conditions not mentioned in the original proof. For example, half a page of mathematical text was blown up to ten pages of LEGO code. We feel that the current state of technology of proof checkers is not suitable to fully formalize non-trivial parts of mathematics. Currently, proof development systems are little used by mathematicians. This is not because of unwillingness to use computer systems, nor because of lack of persistence on their side. The reason is that the effort to fully formalize a proof is out of proportion to the insight we gain.

On the other hand we do feel that when proof checkers get smarter, and when good libraries of formalized mathematics are built, in time these systems are promising and can be extremely useful.

## 6.1 Recommendations

Before we present some recommendations to be implemented in proof development systems, let us first point out two fundamental obstacles one has to overcome when starting a large formalization.

### Start from scratch

Every library of formalized mathematics is based on certain choices which are not always the most optimal in all circumstances. Especially the decision which logical system is used, is often an area of ongoing discussion. As a consequence, often one finds it necessary to develop from scratch all theory needed to prove or to formalize a particular theorem. As the author has experienced, this is a highly elaborate process. So we have no alternative then to reuse existing libraries of formalized mathematics, and to extend them to the level needed.

**Level of detail**

Although at first sight it may seem surprising, proofs found in mathematical texts are almost never integral proofs. Proofs are merely a list of arguments which should convince the reader that the formulated lemma is true indeed. The author gives a sketch of how to prove the theorem. All steps in the proof which are trivial for the intended reader, are left out. So often, all side conditions are discarded and it is for the diligent reader to verify them. For this reason, the level of detail of proofs is never the same level needed by proof development systems. When a proof is fed into such a system we have to add the low-level proofs ourselves. However, proof checkers should be able to generate proofs automatically to fill in all the gaps as much as feasible. Especially in the area of equational reasoning, we think a lot is to be gained. See Section 6.2 for more on this topic.

### 6.1.1 Requests for implementators

Let us formulate a few areas in which present proof development systems could be enhanced. Some of our findings are only valid for type checkers, or even just a particular type checker like LEGO, other observations have a more global value.

**Expressive power**

Most proof development systems have too little expressive power. As a result, even simple mathematical statements become unreadable when formalized. For example, consider the following lemma.

6.1.1. Lemma. *Let $G$ be an abelian group. Then*

$$\forall x, y \in G\ [y + 0 + x = x + y] \ .$$

In LEGO, we formalize this like

$$\begin{array}{lcl} \texttt{lemma611} & : & \texttt{Prop} \\ & \equiv & \{G : \texttt{AbGroup}\}\{x, y : \texttt{car}\, G\} \\ & & \texttt{Eq}\ \ (\texttt{ap}\,(\texttt{PlusAG}\, G)\, y\, (\texttt{ap}\,(\texttt{PlusAG}\, G)\, (\texttt{UnitAG}\, G)\, x)) \\ & & \quad\ (\texttt{ap}\,(\texttt{PlusAG}\, G)\, x\, y) \ \ . \end{array}$$

Obviously, the expression `lemma611` is hard to read. We mention four items for improvement.

*Annotations.* Similar to programming languages, the expression `lemma611` consists of a chain of ASCII characters. For example, instead of $\forall x \in A[\phi(x)]$, we have to write $\{x : A\}\texttt{phi}(x)$. Since we do not have annotations like sub- or superscript, fonts or typefaces like in type-setting languages, expressions are flattened. Annotations would add structure to the expression for easy parsing.

*Infix notation.* The lack of infix operators[1]. If we take a look at the part '$x + y$' of Lemma 6.1.1, we wish to formalize it not as '($\texttt{ap PlusAG}\ G$) $x$ $y$', but as:

$$x\ (\texttt{ap PlusAG}\ G)\ y \ \ .$$

[1]In LEGO, infix notation can be imitated up to a certain degree, namely by the dot-application. In LEGO, '$x$.`plus` $y$' stands for '`plus` $x\, y$'. See the LEGO manual (Luo and Pollack 1992) for details.

*Overloading.* The lack of overloading prevents us to write

$$x\ (\texttt{ap}\ \texttt{Plus}\ G)\ y\ \ .$$

So for example, each time we introduce a new additional structure, we have to define a new term for addition and assign it a unique name. Finding unique names which are easy to recall is a hard task.

*Context dependent parsing.* Argument synthesis, see (Luo and Pollack 1992), is in some cases extremely useful. For example, when we apply the polymorphic operator of function composition, argument synthesis allows us to leave out the types of the applicants. The mechanism is able to derive these types from the context in which they appear. So instead of writing

$$\texttt{comp}\ T\ U\ V\ f\ g\ \ ,$$

for types $T$, $U$ and $V$, and for functions $f : T \to U$ and $g : U \to V$, we may just write

$$\texttt{comp}\ f\ g\ \ .$$

Another example where argument synthesis is very convenient, is the polymorphic defined book equality (see Definition 3.1.5). However in some other cases, this mechanism is insufficient. For example, let us take a look again at our formalization of '$x + y$' in LEGO:

$$x\ (\texttt{ap}\ \texttt{Plus}\ G)\ y$$

First, $G$ should be derivable from the type of $x$ and $y$, which is $\texttt{car}\,G$. Second, if we examine the definition of `Plus`, we will see it consists of two parts, namely the product of some function $p$ together with a proof that $p$ preserves equality. Then it should be clear to our proof checker that we need the former part of `Plus`. If the type checker could derive this for us, we would be able to simplify the running expression to merely

$$x\ \texttt{Plus}\ y\ \ ,$$

which very closely resembles the original '$x + y$'.

To summarize, we would like that our type checker has a lot more expressive power in the sense of annotations, infix notation and overloading, together with some kind of sophisticated mechanism of smart context dependent parsing.

### Efficiency

Compared to other proof checkers, LEGO seems to be rather inefficient in type checking. The system consumes both a lot of processing time, as a lot of system memory. We do have to realize that LEGO is in fact just a proto-type, and was never designed for actually doing really large proofs. However with respect to efficiency, we can point out a few areas in which LEGO, and most other proof checkers, could be improved.

*Arithmetic.* Inductively defined natural numbers are essentially represented by lists. So it is a unary representation which is totally unsuitable for computations involving large numbers. For example, even a simple operation like addition already has complexity $\Theta(n)$. We wish to have types like the natural numbers, the integers and the rationals as first class citizens in our system. Only then computations may become acceptably fast. For other solutions see (Huisman 1997). Remark that one could argue that in formalizing mathematics, we never deal with canonical natural numbers bigger than perhaps ten.

*Annotated lambda terms.* LEGO has a mechanism to file the current proof state of all verified proofs to disk. All named proofs and their accompanying types are saved. The next time the user wishes to restore a LEGO session, the type checker does not have to reconstruct all terms again by executing every tactical. Loading the saved lambda terms, and checking their types is much quicker. Surprisingly enough, in our experience, the opposite is true. Reconstructing via tacticals is faster then verifying types. The reason for this is that in the original interactive session, now-and-then we helped the proof checker by giving hints how convert a type to an intermediate term. We sometimes do this to speed up the type checking of our interactive session. Besides lambda terms and types, also these intermediate steps should be saved to disk as annotations.

*Reduction.* When running large scale formalisations, fast proof checking is important. The implementation of term rewriting and $\beta$-, $\iota$- and $\delta$-reduction must be very efficient. Users of proof checkers are almost insatiable with respect to speed.

**Environment**

Another area in which most proof development systems are weak is the proof-development environment. Already a lot of research is done in user interfaces. We mention the Centaur project. Among other aspects, part of this project is to produce textual proofs out of lambda terms (Bertot, Kahn and Théry 1994). When we focus ourselves on LEGO, we mention two items.

*Presenting proofs.* We need tools for presenting proofs in various formats. For example, a pretty printer for LaTeX is desirable. Sometimes we wish to automatically strip all proofs leaving the plain definitions, lemmas and comments. The idea of merging code and documentation is already used in the system (C)WEB (Knuth 1983, Knuth 1992). Besides LaTeX, another useful output format is hypertext markup language (HTML). In Appendix A.4.3 we present a so-called CGI-program that allows us to present LEGO-files via a web server via the internet. LEGO source is presented in such a way that definitions and lemmas become hyperlinks. To unfold a lemma or definition, the user has to follow the link.

*Modules.* LEGO, like some other proof development systems, lacks a well developed module mechanism. When working on large projects where large formalizations of mathematics are done by more people, we need a module mechanism. At least two elements are needed. First, given a source file, it should be clear which definitions and lemmas are public and exported, and which are local. The implementation should be hidden, or at least clearly separated from the export part. Second, we need to be able to indicate dependencies between source files in order to properly make theories and load all depending files first[2]. Examples of practical module mechanisms can be

[2]Recent versions of LEGO do support this via the `Module` keyword and `Make` command.

found in programming languages like Clean, Modula or Ada. Also we refer to (Courant 1997, Courant 1998), who studied modules with respect to type theory.

**Logic**

The final two aspects of possible improvement of proof development systems we wish to bring up concerns logic.

*Forward reasoning.* As pointed out in Section 4.1, type checkers mainly construct proofs by backwards reasoning. Because this is counter-intuitive to mathematical practice, we wish to have a mechanism for forward reasoning. This could be achieved by allowing nested lemmas. So while proving a lemma, we wish to state and prove a local (sub)lemma, which in turn may be used in the proof of the 'parent' lemma. A sublemma then is in fact just a definition local to a lemma, interactively build by tacticals.

*Proof theoretical strength.* Most proof development systems are based on a predefined, fixed set of logical environments. For example, Mizar (Trybulec and Blair 1985, Rudnicki 1992) verifies proofs based on a variant of Zermelo-Fraenkel set theory, and LEGO supports the Calculus of Constructions, with the option of inductive types or sigma types. Although not every mathematician may be concerned about the logical system he uses to prove a theorem, in some occasions it is important to know. For example, do we need, or allow, second order logic? Or higher order logic? Do we want to build purely constructive proofs, and if not, at which steps do we really need classical logic? So we wish to be able to tune the logic used in our proof development system. With respect to type theory, a good mechanism for this seems to be Pure Type Systems (PTSs, Barendregt 1992). The type checker Constructor (Helmink 1993) does support a large subset of PTSs. However, Constructor is not supported anymore.

## 6.2 Lean proof checking

The proof of the binomial theorem in Section 4.3 shows in extreme how equational reasoning can get out of hand. For this reason, we have developed a simple, flexible and rather efficient method to solve equational problems in type theory. The main ingredients of our method are a two-level formalization of universal algebra based on congruence types. The approach chosen is also intimately related to the design of hybrid systems and can be seen as an attempt to lay the foundations for a theoretical understanding of the interaction between proof checkers and computer algebra systems. In the future, it seems worthwhile to try to extend the framework to equational theories which do not yield a confluent terminating term rewriting system. A longer term goal related to this research is the understanding of computer algebra algorithms. A full understanding of their nature as term rewriting systems is necessary to see whether a type system with (a reasonable variant of) congruence types can provide a theoretical framework in which the integration of proof checkers and computer algebra systems can be justified.

## 6.3 Related work

Beside Automath (de Bruijn 1970), one of the first projects in which large bodies of mathematical texts are verified in a thorough way, is the Mizar project (Trybulec and Blair 1985, Rudnicki 1992). In Mizar a so called *mathematical vernacular* is used to represent mathematical texts. Mizar is not based on type theory, but on a variant of Zermelo-Fraenkel set theory. Hence it does not produce proof objects as first class citizens. During the last decade, a large number of verified mathematical texts where added to the Mizar Mathematical Library (MML). More ambitious is the QED project(Anonymous 1994), which has as goal to build a repository that represents *all* important mathematical knowledge.

Paul Jackson worked on formalizations of abstract algebra in his PhD thesis (Jackson 1995). The proof assistant used is the NuPrl system. Also Jackson introduced a notion of reflection which is related to our two-level approach.

Zhaohui Luo introduced the notion of *coercive subtyping* in (Luo 1997) in type theory. The idea is to use subtyping as a mechanism for notational abbreviations. For example, if `Group` is a subtype of `Monoid` via a coercion function, then a group of type `Group` can be regarded as an object of type `Monoid`. These ideas are worked out furthermore in (Jones, Luo and Soloviev 1998).

In the context of our thesis, we also must refer to Anthony Bailey's work on formalizations of abstract algebra (Bailey 1998). Bailey formalized the fundamental theorem of Galois Theory in the type theory using the LEGO system. For this, he designed a variant of LEGO to implement the synthesizing of implicit coercions. Following Bailey, coercions are implemented in the proof system Coq by (Saïbi 1997). Furthermore, Bailey also investigated *literate formalizations*. Literate formalization stems from Donald Knuth's concept of literate programming. The idea is to write source files of mathematics which can both be mechanically checked, as well as easily be pretty printed for human reading.

## 6.4 Results

We end this chapter by repeating some points presented in the previous chapters 3 and 4.

1. As expressed in the introduction of Chapter 3, we prefer to keep definitions as simple as possible. So while formalizing a mathematical notion, we are eager to give a definition as clearly as possible, so that it can be easily validated that the definition models our intention. The price to pay is that some proofs may get more involved. But after all, for verification we have the proper tools.
2. Introducing sets as the product type of a type $T$, a binary relation $R$ over $T$, and a proof that $R$ is an equivalence relation, appears to be a suitable definition for formalizing (constructive) sets (see Section 3.1.2). Following this concept, we defined functions over a type $T$ as a tuple of an arrow $f$, together with a proof that $f$ preserves equality (see Section 3.2.1).
3. When we allow the formation of the set of propositions $\Omega$ as a setoid using if-and-only-if as equivalence relation (see Definition 3.1.7), this implies that:
   (a) Predicates (and subsets) can be defined as functions into $\Omega$ (Section 3.2.4).
   (b) The power set of a set is a set again (Section 3.3.2).

4. When we define the or-connective by inductively, the impredicatively defined or-connective inherits the strong elimination rule. As a result, we can define the case distinction over discrete sets and even the characteristic function of a decidable predicate (see Section 3.3.4). For an application, see Section 4.2.3 where we define a prime generator using bounded minimalization. Because all proofs are constructive, the prime generator actually computes primes by normalization. Also in some cases, proof objects become much shorter (often merely reflexivity).
5. Mathematical structures are defined in a generic way using signatures, a carrier set and valuation functions (Definition 3.4.3), all packed together by sigma types.
6. We formalized a non-constructive proof of the existence of $k$-th roots in the complex field in Section 4.4.2. For this we introduced in Definition 3.5.15 the real number system $\mathbb{R}$ as a discrete ordered field such that every polynomial in $\mathbb{R}$ of odd degree has a root, and which has a square root function over $\mathbb{R}$. Polynomial rings are defined in Section 4.4.1 as sparse, multi-variate, relaxed polynomials.
7. We developed a library of formal mathematics. Over 18,000 lines of code were written which consist of roughly 600 definitions and 1200 theorems. It takes a few hours for LEGO to load and check all the code on a Sun SparcStation 20 with TMS390Z55 processors and 384MB of internal memory.
8. We wrote a front-end for LEGO source and proof-objects files to view these pretty-printed via a web-server on the world-wide-web[3].

[3]See Appendix A.4.3 and `http://www.cs.kun.nl/fnds/lego/markr.shtml`.

# Appendix A

# The LEGO System

LEGO is a type checker written by Randy Pollack in the functional language NJ-SML (New Jersey Standard Meta Language). The source is freely available from the web site `http://www.dcs.ed.ac.uk/home/lego/`. There you will also find documentation, libraries of formalized mathematics, tools, and other useful information.

## A.1 Quick introduction to LEGO

For an introduction to LEGO the reader is referred to the manual (Luo and Pollack 1992). In this section we just give some high-lights.

We initialize LEGO by `Init XCC;`, so we work in the Extended Calculus of Constructions *ECC* (Luo 1990) with inductive types. Actually we don't need the full power of *ECC*, but it is the weakest system LEGO offers which has more then two sorts and inductive types.

### A.1.1 Argument Synthesis

LEGO has a concept of meta variables, called existential variables, for lambda terms. The type checker will try to resolve existential variables automatically. If it fails, we should supply the correct terms ourselves. An existential variable is denoted by a question mark. So, suppose we have defined function composition as

$$\begin{array}{rcl}\texttt{Comp1} & : & \Pi A,B,C{:}\texttt{SET}.\,(A\to B)\to(B\to C)\to(A\to C)\\ & \equiv & \lambda A,B,C{:}\texttt{SET}\,\lambda f{:}A\to B\,\lambda g{:}B\to C\,\lambda x{:}A.\,g\ (f\ x)\end{array}$$

and suppose $A,B,C:\texttt{SET}$, $f:A\to B$ and $g:B\to C$, then we may write

$$\texttt{Comp1}\ ?\ ?\ ?\ \texttt{f g}$$

and LEGO will resolve the existential variables to $A$, $B$, and $C$ respectively. Furthermore, in LEGO we can annotate abstractions to indicate that they will be applied to implicit existential variables. So define

$$\begin{array}{rcl}\texttt{Comp2} & : & \Pi A,B,C{\mid}\texttt{SET}.\,(A\to B)\to(B\to C)\to(A\to C)\\ & \equiv & \lambda A,B,C{\mid}\texttt{SET}\,\lambda f{:}A\to B\,\lambda g{:}B\to C\,\lambda x{:}A.\,g\ (f\ x)\end{array}$$

and suppose once again that $A,B,C:\texttt{SET}$, $f:A\to B$ and $g:B\to C$, then we may write

$$\texttt{Comp2}\,f\ g\ .$$

| Description | LEGO Code | Pseudo LEGO |
|---|---|---|
| sorts | `Prop:Type(0):Type(1)` :Type(2) | `PROP:SET:TYPE` `:BIGTYPE` |
| variables | `x, y, z, A, B ...` | $x, y, z, A, B, \ldots$ |
| definitions | `ap, Eq ...` | `ap`, `Eq`, ... |
| meta-variables | `?` | ? |
| $\lambda$-abstraction | `[x:A]B` | $\lambda x{:}A.\,B$ |
| | `[_:A]B`   `A\B` | $\lambda\_{:}A.\,B$ |
| | `[x\|A]B` | $\lambda x{\mid}A.\,B$ |
| application | `ap f x` | `ap` $f\ x$ |
| | `f.ap x` | $f$.`ap` $x$ |
| | `Eq\|A` | `Eq`$\mid A$ |
| $\Pi$-abstraction | `{x:A}B` | $\Pi x{:}A.\,B$   $\forall x{:}A.\,B$ |
| | `{_:A}B`   `A -> B` | $A{\rightarrow}B$ |
| | `{x\|A}B` | $\Pi x{\mid}A.\,B$ |
| $\Sigma$-abstraction | `<x:A>B` | $\Sigma x{:}A.\,B$ |
| | `<_:A>B`   `A#B` | $A \times B$ |
| projection | `z.1`   `z.2` | $\pi_1(z)$   $\pi_2(z)$ |
| pairing | `<x,y>` | $<x, y>$ |

Table A.1: Informal definition of LEGO terms and of pseudo LEGO.

### A.1.2 Pseudo LEGO

Informally, LEGO terms are build following the lines of Table A.1. This table also contains the first part of an informal definition of a *pseudo LEGO* we use throughout this thesis.

In Table A.2, we extend the pseudo LEGO by adding some syntactical sugar. This improves the readability of LEGO source considerably. For example, LEGO has only projection of two-tuples as primitives, although it allows us to write $<x_1, \ldots, x_n>$. The scheme for projections in Table A.2 gives us projections of $n$-tuples for arbitrary fixed $n$ based on $\pi_1$ and $\pi_2$.

| Description | LEGO Code | Pseudo LEGO |
|---|---|---|
| sets | `Eq x y` | $x = y$ |
| | `x:el A` | $x \in A$ |
| functions | `Fun A B` | $A{\Rightarrow}B$ |
| | `BFun A B C` | $(A \times B){\Rightarrow}C$ |
| | `Function A B` | $A{\Longrightarrow}B$ |
| predicates | `R.ap2 x y` | $<x, y> \in R$   $(x, y) \in R$ |
| | `S:Pred A` | $S \subset A$ |
| interpretation | `int A` | $[\![A]\!]$ |
| projection | `t.`$\ulcorner\pi_i^n\urcorner$ | $\pi_i^n(t)$ |
| where | $\ulcorner\pi_1^i\urcorner \equiv 1$ | |
| | $\ulcorner\pi_2^2\urcorner \equiv 2$ | |
| | $\ulcorner\pi_{j+1}^{i+2}\urcorner \equiv 2.\ulcorner\pi_j^{i+1}\urcorner$ | |

Table A.2: Extension of pseudo LEGO.

### A.1.3 Recursive definitions

We introduced a definition of the inductively defined natural numbers in Section 1.3.1 as follows.

$$\mathbb{N} \quad \equiv \quad \mu X : *.(0 : X, S^+ : X \to X)$$

Using the elimination principle $\varepsilon_{\mathbb{N}}$, we defined the addition by recursion on the secord argument:

$$\texttt{add} \quad \equiv \quad \lambda x{:}\mathbb{N}.\, \varepsilon_{\mathbb{N}}\, (\lambda y{:}\mathbb{N}.\, \mathbb{N})\, 0\, (\lambda y{:}\mathbb{N}\, \lambda h{:}\mathbb{N}.\, S^+ h) \ .$$

Because this definition is hard to read, we extent pseudo LEGO to allow a different notation for inductive definitions.

$$\begin{array}{lllll} \texttt{add} & \equiv & \lambda x{:}\mathbb{N}.\, \varepsilon_{\mathbb{N}} & 0 & \Longrightarrow & x \\ & & & (S^+ y) & \Longrightarrow & S^+(\texttt{add}\, x\, y) \end{array}$$

So we leave out the term $(\lambda y{:}\mathbb{N}.\, \mathbb{N})$, we drop the abstractions $\lambda y{:}\mathbb{N}\, \lambda h{:}\mathbb{N}$ and substitute the name of the definiendum $(\texttt{add}\, x)$ for the induction hypothesis $h$.

## A.2 Naming conventions

Another topic concerns the 'type of' relation which assigns a type to a term in typed $\lambda$-calculus. What might look trivial, but still needs some attention, is how one should read '$A : B$' for types $A$ and $B$. Several opportunities come up.

1. $A$ is of type $B$, $A$ lives in $B$.
2. $A$ proves $B$, $B$ holds, '$B$'.
3. $A$ is a member of $B$, $A$ is an element of $B$.
4. $A$ is a $B$, let $A$ be a $B$.

The first option is used when we don't want to give an interpretation to the typing relation. The second is used for propositions, when $B$ : `PROP`. The third is used when we want to view types as sets and terms as elements. However, as Table A.3 shows, option three is not as convenient as the fourth. So although 3 is a more literal interpretation, we have chosen to use 4 for readability. The whole point of this discussion is that `Set`, `el` , `Subset` , …now should be given singular names despite that they stand for collections.

By definition, `Subset` $A$ and `el` (`Powerset` $A$) are $\beta\iota$-convertible. Often the first is used because it is shorter. The second has an advantage when working with equality of subsets.

## A.3 The LEGO library

The LEGO source files have a size of about 580,000 tokens. If printed out, it will takeup about 18,000 lines and 300 pages. All definitions and proofs from chapters 3 and 4 and more are collected as a large LEGO library which is publicly available (see below).

| mathematics | type theory | interpretation |
|---|---|---|
| $A$ is a set | $A$ : `Set` | $A$ **is a member of** the class of sets |
| | | $A$ **is a** set |
| $x \in A$ | $x$ : `el` $A$ | $x$ **is a member of** the elements of $A$ |
| | | $x$ **is an** element of $A$ |
| $S \subset A$ | $S$ : `Subset` $A$ | $S$ **is an element** of the subsets of $A$ |
| | | $S$ **is a** subset of $A$ |
| $S \in \wp(A)$ | $S$ : `el` (`Powerset` $A$) | $S$ **is a member of** the elements of the powerset of $A$ |
| | | $S$ **is an** element of the powerset of $A$ |
| $f \in A \Rightarrow B$ | $f$ : `Fun` $A\,B$ | $f$ **is a** function |

Table A.3: How to read the 'type of' relation.

### A.3.1 Technical background

From the original LEGO library we only use the file `lib_logic.l`. For reasons of speed we were forced to make occasionally use of the `Freeze` command. `Freeze` invalidates `Make`, so instead all files should be `Loaded`. Surprisingly, we have experienced that `Load` is quicker than `Make` anyway. You may want to increase the limit on your heap space (60 Mb should do.) Before you start LEGO you can change the limit by typing `ulimit -d` 60000 for bash, or `limit data 60000` for csh. It takes a few hours to load and check all the files on a Sun Sparc Station 20 with TMS390Z55 processors and 384MB of internal memory. The files consist of more then 18,000 lines of LEGO code.

### A.3.2 Availability

The LEGO library we developed is obtainable via the web[1]. All files are shown by the pretty-printer Lego2html. Furthermore, one can use anonymous ftp to get the library as single compressed zip file.

```
ftp ftp.cs.kun.nl
Name: anonymous
Password: enter your e-mail address
cd /pub/CSI/CompMath.Found/lego
bin
get RuysLegoLib.zip
```

The size of this file is approximately 135 Kb, uncompressed about 600 Kb.

### A.3.3 Demo Session

As an example of an interactive LEGO session we include a complete screen dump. It is the proof of the Drinkers Principle (see Section 4.1). All text after the LEGO prompt `Lego>` is typed by the user. The rest is system output. Note that we have left out a small unimportant portion of text and replaced it by three consecutive dots.

[1]`http://www.cs.kun.nl/fnds/lego/markr.shtml`

```
Standard ML with LEGO
Generated  Thu Jul  4 16:58:07 MET DST 1996
using Standard ML of New Jersey, Version 0.93, February 15, 1993
Sun4 with inversion and Then tactical
use command 'Help' for info on new commands.
'Qrepl' configured
Extended CC: Initial State!
strong predicative Sigma-types
Lego> Init XCC; Logic;
'Qrepl' configured
Extended CC: Initial State!
strong predicative Sigma-types
...
Lego> [ExFalso = [P:Prop][H:absurd] H P];
defn  ExFalso = [P:Prop][H:absurd]H P
      ExFalso : {P:Prop}absurd->P
Lego> [Dn : {P:Prop} ~~P->P] [C : Type] [a : C] [D : C->Prop];
decl  Dn : {P:Prop}(not (not P))->P
decl  C : Type
decl  a : C
decl  D : C->Prop
Lego> [phi = [x:C] (D x) -> {y:C} D y];
defn  phi = [x:C](D x)->{y:C}D y
      phi : C->Prop
Lego> Goal Ex phi;
Goal
  ?0 : Ex phi
Lego> Refine Dn;
Refine by  Dn
  ?2 : not (not (Ex phi))
Lego> Intros H2;
Intros (1) H2
  H2 : not (Ex phi)
  ?3 : absurd
Lego> Refine H2;
Refine by  H2
  ?4 : Ex phi
Lego> Refine ExIntro; Refine a;
Refine by  ExIntro
  ?7 : C
  ?8 : phi ?7
Refine by  a
  ?8 : phi a
Lego> Intros H5;
Intros (1) H5
  H5 : D a
  ?9 : {y:C}D y
Lego> Intros z;
Intros (1) z
```

```
  z : C
  ?10 : D z
Lego> Refine Dn;
Refine by  Dn
  ?12 : not (not (D z))
Lego> Intros H8;
Intros (1) H8
  H8 : not (D z)
  ?13 : absurd
Lego> Refine H2;
Refine by  H2
  ?14 : Ex phi
Lego> Refine ExIntro; Refine z;
Refine by  ExIntro
  ?17 : C
  ?18 : phi ?17
Refine by  z
  ?18 : phi z
Lego> Intros H11;
Intros (1) H11
  H11 : D z
  ?19 : {y:C}D y
Lego> Prf;
  H2 : not (Ex phi)
  H5 : D a
  z : C
  H8 : not (D z)
  H11 : D z
  ?19 : {y:C}D y
Lego> Refine ExFalso;
Refine by  ExFalso
  ?21 : absurd
Lego> Refine H8; Refine H11;
Refine by  H8
  ?22 : D z
Refine by  H11
Discharge..  H11
Discharge..  H8
Discharge..  z
Discharge..  H5
Discharge..  H2
*** QED ***
Lego> Save DrinkersPrinciple;
"DrinkersPrinciple"  saved as global, unfrozen
```

## A.4 Tools

Essentially, LEGO is just a plain command-line program, without an interface. A convenient way to start the LEGO system is in a shell inside an editor like emacs. In that way you can browse through the output and edit your input. Thomas Schreiber wrote a very useful LEGO-interface for emacs in elisp for editing LEGO proof scripts.

We wrote a few other programs which are quite helpful. These tools can be downloaded via the web[2].

### A.4.1 legogrep

Legogrep is a Perl script which recursively searches through LEGO modules for definitions, proved lemmas, et cetera. It takes into account the commands `Make`, `Load`, `Reload`, `Include`, `Module`, and is aware of the `LEGOPATH` environment variable. Regular expressions are allowed. An example is to search all occurrences of `cancel.*plus`.

```
omega> legogrep cancel.*plus test_all
lib_nat_plus_thms: Save cancel_plus
lib_nat_plus_thms: Save cancel_plus
lib_nat_times_thms: Refine cancel_plus (suc a)
lib_nat_Le: Refine cancel_plus
lib_int_nat_lemmas: Refine cancel_plus
lib_int_basics: Refine cancel_plus n1
lib_nat_Le: Refine cancel_plus
```

### A.4.2 legostat

Legostat is also a Perl script for generating statistical information of a LEGO session. It is mostly used to measure the speed (or slowness) of checking large LEGO files. The output will look like

```
omega> echo "Load lib_nat" | lego | legostat
parameters.l      time=  0.02 sec  0:00:00 hms    gc=  0.00 sec
lib_logic.l       time=  2.06 sec  0:00:02 hms    gc=  0.10 sec
lib_ML_eq.l       time=  0.64 sec  0:00:00 hms    gc=  0.01 sec
lib_start_up.l    time=  0.38 sec  0:00:00 hms    gc=  0.01 sec
lib_nat.l         time=  1.72 sec  0:00:01 hms    gc=  0.06 sec
                  time=  4.82 sec  0:00:04 hms    gc=  0.18 sec
```

The column labeled `time` stands for the cumulative execution time. The `gc` column is the amount of time spend at (major) garbage collections.

### A.4.3 lego2html

Lego2html is a CGI program written in Perl. It generates HTML pages out of lego source files. It is intended to present a lego file or a library of lego files in nice and more readable way on the web. Proofs are replaced by hyperlinks. Following this link unfolds the proof. A next link goes even one level deeper to a new page which

[2]`http://www.dcs.ed.ac.uk/home/lego/html/tools.html`

shows the actual lambda term of the proof objects. Also a search form is added at the bottom of each page. It uses the legogrep tool to search for a definitions or lemmas in the lego files presented. Our LEGO library can be best viewed using this tool. The hyperlink is:
`http://www.cs.kun.nl/fnds/lego/markr.shtml`

# Bibliography

Abramsky, S., D.M. Gabbay and T.S.E. Maibaum (eds.) (1992). *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press.

ACM (1997). *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France.

Anonymous (1994). The QED manifesto, *in:* Alan Bundy (ed.), *12th International Conference on Automated Deduction*, LNAI 814, Springer-Verlag, Nancy, France, pp. 238–251.

Appel, K. and W. Haken (1976). Every planar graph is four colourable, *Bulletin of the American Mathematical Society.*

Aspers, W.A.M., J.W. de Bakker, P.J.W. ten Hagen, M. Hazewinkel, P.J. van der Houwen and H.M. Nieland (eds.) (1996). *Images of SMC Research 1996*, Stichting Mathematisch Centrum.

Bailey, A. (1993). *Representing algebra in LEGO*, Master's thesis, University of Edinburgh.

Bailey, Anthony (1998). *The Machine-Checked Literate Formalisation of Algebra in Type Theory*, Dissertation, University of Manchester.

Ballarins, C., K. Homann and J. Calmet (1995). Theorems and algorithms: an interface between Maple and Isabelle, *Proceedings of ISSAC '95*.

Barendregt, H.P. (1984). *The lambda calculus: its syntax and semantics*, Studies in Logic and the Foundation of Mathematics, North Holland.

Barendregt, H.P. (1992). Lambda calculi with types, *in:* Abramsky, Gabbay and Maibaum (1992), pp. 117–309.

Barendregt, H.P. (1996). The quest for correctness, *in:* Aspers, de Bakker, ten Hagen, Hazewinkel, van der Houwen and Nieland (1996), pp. 39–58.

Barendregt, H.P. and T. Nipkow (eds.) (1994). *Types for Proofs and Programs (International workshop TYPES '93, selected papers)*, Nijmegen, The Netherlands, Lecture Notes in Computer Science 806, Nijmegen, The Netherlands, Springer-Verlag, Berlin.

Barthe, G. (1995a). Formalising mathematics in type theory: fundamentals and case studies, *Technical Report CSI-9508*, Computing Science Institute, University of Nijmegen.

Barthe, G. (1995b). Towards a mathematical vernacular, *Technical Report CSI-N9502*, Computing Science Institute, University of Nijmegen.

Barthe, G. and H.J. Elbers (1996). Towards lean proof checking, *in:* J. Calmet and C. Limongelli (eds.), *Proceedings of DISCO '96*, Lecture Notes in Computer Science 1128, Springer-Verlag, pp. 61–??

Barthe, G. and J.H. Geuvers (1996). Congruence types, *in:* H. Kleine Buening (ed.), *Proceedings of CSL '95*, Lecture Notes in Computer Science 1092, Springer-Verlag, pp. 36–51.

Barthe, G., M.P.J. Ruys and H.P. Barendregt (1996). A two-level approach towards lean proof-checking, *in:* Berardi and Coppo (1996), pp. 16–35.

Beeson, M.J. (1985). *Foundations of Constructive Mathematics*, Springer-Verlag, Berlin.

van Benthem Jutting, L.S. (1977). *Checking Landau's "Grundlagen" in the AUTOMATH system*, Dissertation, Eindhoven University of Technology. Mathematical Centre Tracts nr. 83, Math. Centre Amsterdam 1979.

van Benthem Jutting, L.S. (1994). Description of AUT-68, *in:* Nederpelt, Geuvers and de Vrijer (1994), pp. 251–273.

Berardi, S. (1989). *Type dependence and Constructive Mathematics*, Dissertation, Mathematical Institute, University of Turin, Italy.

Berardi, Stefano and Mario Coppo (eds.) (1996). *Types for Proofs and Programs (International workshop TYPES '95, selected papers)*, Torino, Italy, Lecture Notes in Computer Science 1158, Torino, Italy, Springer-Verlag, Berlin.

Bertot, Yves, Gilles Kahn and Laurent Théry (1994). Proof by pointing, *in:* M. Hagiya and J.C. Mitchell (eds.), *Proceedings of the International Symposium on Theoretical Aspects of Computer Softward*, Springer-Verlag LNCS 789, Sendai, Japan, pp. 141–160.

Bishop, Errett (1967). *Foundations of Constructive Analysis*, McGraw-Hill Book Company, New York.

Boole, George (1854). *An Investigation of the Laws of Thought*, Macmillan, London.

Boyer, R.S. and J.S. Moore (1988). *A Computational Logic Handbook*, Academic Press, San Diego.

Breazu-Tannen, V. (1988). Combining algebra and higher-order types, *Proceedings of LICS '88*, IEEE Computer Society Press, pp. 82–90.

Brouwer, L.E.J. (1907). *Over de grondslagen der wiskunde*, Dissertation, Amsterdam. In Dutch.

Bruijn, N.G. de (1970). The mathematical language AUTOMATH, its usage, and some of its extensions, *in:* Laudet, Lacombe and Schuetzenberger (1970), pp. 29–61.

Char, B.W., K.O. Geddes, G.H. Gonnet, B. Leong, M.B. Monagan and S.M. Watt (1992). *First leaves– a tutorial introduction to Maple V*, Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc.

Choi, J., J.J. Dongarra, R. Pozo and D.W. Walker (1992). ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers, *in:* H.J. Siegel (ed.), *The Fourth Symposium on the Frontiers of Massively Parallel Computation: Frontiers '92 / October 19–21, 1992, McLean, Virginia*, IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, pp. 120–127. IEEE catalog number 92CH3185-6.

Church, A. (1932,1933). A set of postulates for the foundation of logic, *Anals of mathematic*, Vol. 33, 34, pp. 346–366, 839–864.

Church, A. (1940). A formulation of the simple theory of types, *J. Symbolic Logic* **5**, pp. 56–68.

Cohn, Paul Moritz (1981). *Universal algebra*, Mathematics and its Applications 6, D. Reidel.

Constable, R.L. (1993). Metalevel programming in constructive type theory, *in:* F.L. Bauer, W. Brauer and H. Schwichtenberg (eds.), *Logic and Algebra of Specification*, Springer-Verlag, Berlin.

Constable, R.L. et al. (1986). *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall, Englewood Cliffs, New Jersey.

Coquand, Th. (1985). *Une théorie des constructions*, Dissertation, UniversitéParis VII, Paris, France.

Coquand, Th. and G. Huet (1985). Constructions: A higher order proof system for mechanising mathematics, in *EUROCAL 85* (*EUROCAL 85* 1985).

Coquand, Th. and G. Huet (1988). The calculus of constructions, *Information and Computation*.

Courant, Judicaël (1997). An applicative module calculus, *Theory and Practice of Software Development 97*, Lecture Notes in Computer Science, Springer-Verlag, Lille, France.

Courant, Judicaël (1998). *Un calcul de modules pour les systmes de types purs*, Thse de Doctorat, Ecole Normale Suprieure de Lyon.

Curry, H.B. (1934). Functionality in combinatory logic, *Proc. Nat. Acad. Science USA*, Vol. 20, pp. 584–590.

van Dalen, Dirk (1978). *Filosofische grondslagen van de wiskunde*, Van Gorcum, Assen/Amsterdam. In Dutch.

Dieudonné, J. (1960). *Foundations of Modern Analysis*, Academic Press, New York and London.

Dowek, G. et al. (1993). The Coq proof assistant user's guide, *Technical report*, INRIA.

Ebbinghaus, H.-D., H. Hermes, F. Hirzebruch et al. (1990). *Numbers*, Graduate texts in mathematics, Springer-Verlag, Berlin.

Elbers, H.J. (1993). *Computer aided construction of real numbers*, Master's thesis, Computing Science Institute, University of Nijmegen.

Elbers, H.J. (1998). *Connecting Informal and Formal Mathematics*, Dissertation, Eindhoven University of Technology, Eindhoven, The Netherlands.

*EUROCAL 85* (1985). LNCS 203, Springer.

Frege, Friederich Ludwig Gottlob (1879). *Begriffschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*, Nebert, Halle.

Girard, J.-Y., Y. Lafont and P. Taylor (1989). *Proofs and types*, Tracts in Theoretical Computer Science 7, Cambridge University Press.

Goossens, K.G.W. (1992). *Embedding Hardware Description Languages in Proof Systems*, Dissertation, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Gordon, M.J.C. (1991). Introduction to the HOL system, *in:* M. Archer, J.J. Joyce, K.N. Levitt and P.J. Windley (eds.), *International Workshop on Higher Order Logic Theorem Proving and its Applications*, IEEE Computer Society, ACM SIGDA, IEEE Computer Society Press, Davis, California, pp. 2–3.

Hamilton, W.R. (1837). Theory of conjugate functions, or algebraic couples with a preliminary and elementary essay on algebra as the science of pure time, *Trans. Royal Irish Academy* **17**, pp. 293–422.

Harrison, J. and L. Théry (1993). Extending the HOL theorem prover with a computer algebra system to reason about the reals, *Proceedings of HOL '93*, Lecture Notes in Computer Science.

Harrison, John (1996). Formalized mathematics, *Technical Report TUCS-TR-36*, Turku Centre for Computer Science, Finland.

Helmink, L. (1993). *Tools for Proofs and Programs*, Dissertation, University of Amsterdam.

Heyting, A. (1956). *Intuitionism, an Introduction*, Studies in Logic, North-Holland, Amsterdam.

Hilbert, D. and P. Bernays (1934, 1939). *Grundlagen der Mathematik*, Springer-Verlag, Berlin.

Hindley, J.R. and J.P. Seldin (eds.) (1980). *To H.B. Curry: essays on combinatory logic, lambda calculus and formalism*, Academic Press.

Howard, W.A. (1980). The formulae-as-types notion of construction, *in:* Hindley and Seldin (1980), pp. 479–490.

Howe, D. (1988). *Automating reasoning in an implementation of constructive type theory*, Dissertation, Cornell University.

Huisman, Marieke (1997). Binary addition in LEGO, *Technical Report CSI-R9714*, Computing Science Institute, University of Nijmegen.

Huntington, Edward V. (1955). Fundamental propositions of algebra, *in:* J.W.A. Young (ed.), *Monographs on topics of Modern Mathematics*, Dover Publications, chapter IV, pp. 201–207. First edition in 1911.

Jackson, Paul B. (1994). Exploring abstract algebra in constructive type theory, *Proceedings of CADE-12*, LNAI 814.

Jackson, Paul B. (1995). *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*, Dissertation, Cornell University.

Johnson, Eugene (1992). Working with `linalg`, *Maple Technical Newsletter* **0**(7), pp. 25–30.

Jones, A., Z. Luo and S. Soloviev (1998). Some algorithmic and proof-theoretical aspects of coercive subtyping, *Lecture Notes in Computer Science* **1512**, pp. 173–??

Jones, Claire (1991). Completing the rationals and metric spaces in LEGO, *in:* Gerard Huet, Gordon Plotkin and Claire Jones (eds.), *Proceedings of the second workshop on Logical Frameworks*, Edinburgh.

Jones, Cliff B. (1990). *Systematic Software Development Using VDM*, second edition, Prentice-Hall International, Englewood Cliffs, New Jersey. ISBN 0-13-880733-7.

Klop, J.W. (1992). Term rewriting systems, *in:* Abramsky et al. (1992).

Knuth, Donald E. (1983). The WEB system of structured documentation, *Technical Report CS-TR-83-980*, Stanford University, Department of Computer Science.

Knuth, Donald E. (1992). *Literate Programming*, CSLI Lecture Notes Number 27, Stanford University Center for the Study of Language and Information, Stanford, CA, USA.

Laudet, M., D. Lacombe and M. Schuetzenberger (eds.) (1970). *Symposium on Automatic Demonstration*, IRIA, Versailles, Lecture Notes in Math. 125, IRIA, Versailles, Springer-Verlag, Berlin.

Luo, Z. (1997). Coercive subtyping in type theory, *Proceedings of CSL '96*, LNCS 1258, Utrecht.

Luo, Zhaohui (1990). *An Extended Calculus of Constructions*, Dissertation, University of Edinburgh. Also as Report CST-65-90/ECS-LFCS-90-118.

Luo, Zhaohui (1994). *Computation and Reasoning: a type theory for computer science*, Monographs on Computer Science 11, Clarendon Press, Oxford.

Luo, Zhaohui and Robert Pollack (1992). LEGO proof development system: User's manual, *LFCS Report ECS-LFCS-92-211*, Department of Computer Science, University of Edinburgh.

Magnusson, L. and B. Nordström (1994). The Alf proof editor and its proof engine, *in:* Barendregt and Nipkow (1994).

Martin-Löf, P. (1972). An intuitionistic theory of types, Manuscript.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*, Bibliopolis.

McCune, W. (1990). OTTER 2.0, *in:* M. Stickel (ed.), *Proceedings of the 10th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, Vol. 449*, Springer-Verlag, New York, pp. 663–664. Extended abstract.

Mines, Ray, Fred Richman and Wim Ruitenburg (1988). *A course in constructive algebra*, Universitext, Springer-Verlag, Berlin.

Nederpelt, R.P., J.H. Geuvers and R.C. de Vrijer (eds.) (1994). *Selected Papers on AUTOMATH*, Studies in Logic 133, North-Holland, Amsterdam.

Nipkow, T. and L.C. Paulson (1992). Isabelle-91, *in:* Deepak Kapur (ed.), *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, LNAI 607, Springer, Saratoga Springs, NY, pp. 673–676.

Nordström, B., K. Peterson and J.M. Smith (1990). *Programming in Martin-Löf's Type Theory : an introduction*, Oxford Science Publications.

Olthof, E.H.T. (1996). *The vibrational-rotational-tunneling dynamics of Van der Waals and hydrogen bonded complexes*, Dissertation, University of Nijmegen.

Owre, S., J.M. Rushby and N. Shankar (1992). PVS: A prototype verification system, *in:* Deepak Kapur (ed.), *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence 607, Springer-Verlag, Saratoga, NY, pp. 748–752.

Peterson, Ivars (1995). *Fatal Defect: chasing killer computer bugs*, Random House, Inc., New York.

Pollack, Robert (1994). *The Theory of LEGO*, Dissertation, University of Edinburgh.

Pollack, Robert (1996). How to believe a machine-checked proof, *Twenty Five Years of Constructive Type Theory, Proceedings of the Venice Meeting*.

Rudnicki, P. (1992). An overview of the mizar project, *Proceedings of the Workshop on Types for Proofs and Programs*, Chalmers University of Technology, Båstad, Sweden.

Ruess, H., N. Shankar and M.K. Srivas (1996). Modular verification of SRT division, *Lecture Notes in Computer Science* **1102**, pp. 123–??

Ruitenburg, Wim (1991). Constructing roots of polynomials over the complex numbers, *in:* A.M. Cohen (ed.), *Computational aspects of Lie group representations and related topics*, Centre for Mathematics and Computer Science (CWI), CWI Tract 84, Amsterdam, pp. 107–128.

Rushby, John M. and Freidrich von Henke (1993). Formal verification of algorithms for critical systems, *IEEE Transactions on Software Engineering Software Engineering Notes, 16(5) Dec 1991, pp 1-15. Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems, New Orleans, LA, Dec 4-6, 1991*. Published as IEEE Transactions on Software Engineering Software Engineering Notes, 16(5) Dec 1991, pp 1-15. Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems, New Orleans, LA, Dec 4-6, 1991, volume 19, number 1.

Ruys, M.P.J. (1991). *$\lambda P\omega$ is not conservative over $\lambda P2$*, Master's thesis, Computing Science Institute, University of Nijmegen.

Saïbi, Amokrane (1997). Typing algorithm in type theory with inheritance, *in:* ACM (1997), pp. 292–301.

Samuel, Pierre (1967). *Théorie Algébrique des Nombres*, Collection Méthodes, Hermann Paris.

Scott, Dana (1970). Constructive validity, *in:* Laudet et al. (1970), pp. 237–275.

Sellink, Alex (1996). *Computer-Aided Verification of Protocols*, Dissertation, Universiteit Utrecht.

Shankar, N. (1986). *Proof-Checking Metamathematics*, Dissertation, Computer Science Department, University of Texas at Austin.

Shankar, N. (1994). *Metamathematics, Machines, and Gödel's Proof*, Cambridge Tracts in Theoretical Computer Science 38, Cambridge University Press.

Tarski, Alfred (1953). *Inleiding tot de logica*, Noord-Hollandse Uitgevers Maatschappij, Amsterdam. In Dutch.

Terlouw, J. (1989). Een nadere bewijstheoretische analyse van GSTT's, *Technical report*, Dept. of Computer Science, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands. In dutch.

Trybulec, Andrzej and Howard Blair (1985). Computer assisted reasoning with MIZAR, *in:* Aravind Joshi (ed.), *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, Los Angeles, CA, pp. 26–28.

van der Waerden, B.L. (1931). *Moderne Algebra*, Frederick Ungar Publishing, New York.

Whitehead, Alfred North and Bertrand Russell (1910–1913). *Principia Mathematica*, Cambridge University Press, Cambridge.

Wolfram, Stephen (1991). *Mathematica–A System for Doing Mathematics by Computer*, second edition, Addison-Wesley, Reading, MA, USA.

Wupper, Hanno and Hans Meijer (1997). A taxanomy for computing science, *Technical Report CSI-R9713*, Computing Science Institute, University of Nijmegen.

# Samenvatting

Het uitgangspunt van gemechaniseerde bewijsverificatie is het gebruik van computersystemen om een hoge graad van zekerheid te verkrijgen dat een bepaald bewijs een gegeven stelling inderdaad bewijst. Tot op heden worden bewijsverificatiesystemen door wiskundigen nog maar sporadisch gebruikt. Door middel van het bestuderen van deze systemen is meer inzicht verkregen wat de obstakels zijn en hoe de bruikbaarheid van bewijsverificatiesystemen verhoogd kan worden.

Dit proefschrift behandelt een speciale klasse van bewijsverificatiesystemen, namelijk zij die op type-theorie gebaseerd zijn. De type-theorie leent zich onder andere goed voor bewijsverificatie omdat er een sterke analogie is tussen enerzijds typen en haar inwoners en anderzijds, stellingen met hun bewijzen. Reductie van lamdatermen correspondeert dan met normalisatie van bewijzen. Omdat inwoners van typen *geconstrueerd* worden door zogeheten lamda-termen, past de constructieve logica het beste bij type-theorie. In deze logica wordt de aandacht van waarheid en onwaarheid verschoven naar *bewijsbaarheid.* In hoofdstuk 2 is kort ingegaan op een aantal benaderingen van logica, te weten de formalistische, de logicistische en de intuitionistische wijze. Ook is hier gekeken hoe logica geformaliseerd kan worden in type-theorie.

Het beoefenen van wiskunde kan verdeeld worden in drie handelingen: het bewijzen, het definiëren en het rekenen. Voordat wiskunde bedreven kan worden, moeten eerst alle concepten waarvan gebruik gemaakt wordt, eenduidig *gedefiniëerd* worden. Het is gebleken dat dit proces bij mechanische verificatie uiterst moeizaam is, omdat begonnen moet worden vanaf de meest elementaire wiskunde. Hoofdstuk 3 richtte zich volledig op het formaliseren van basale wiskundige begrippen als 'verzameling', 'monoide' en bijvoorbeeld 'de complexe getallen'.

Wat betreft het *rekenen* is in hoofdstuk 4 de stelling van Euclides geformaliseerd die zegt dat er oneindig veel priemgetallen zijn: gegeven een willekeurig natuurlijk getal is er altijd een groter getal te vinden dat ook priem is. Omdat het bewijs constructief gegeven is, kon uit het existentiebewijs een 'getuige' gevonden worden: gegeven een getal rekent het systeem voor ons het eerst volgende priemgetal uit. Dit wordt puur verkregen door normalisatie. In hetzelfde hoofdstuk is tevens nog een aanzet gemaakt tot de complete formalisatie van de hoofdstelling van de algebra. Dit bleek echter te hoog gegrepen met de voorhanden zijnde middelen. Wel zijn we gekomen tot de formalisatie van een klassiek bewijs van het bestaan van wortels in het complexe vlak van willekeurige graad. Hiervoor is onder andere een korte studie gedaan naar mogelijke definities van veeltermen.

Om aan te geven hoe omslachtig het redeneren met gelijkheden is, is als voorbeeld een bewijs geformaliseerd van de binomiumstelling. In hoofdstuk 5 is een methode geïntroduceerd om het redeneren met gelijkheden in bepaalde gevallen te

kunnen automatiseren. Basisidee is om een scheiding te maken tussen syntax en semantiek en om de rekenregels van groepen via herschrijfregels aan het bewijsverificatiesysteem toe te voegen.

# Curriculum Vitae

Op 24 maart 1966 ben ik geboren in Rochester, N.Y., de Verenigde Staten van Amerika. Een paar maanden later ben ik per boot naar Nederland gebracht. In Tilburg bezocht ik vanaf 1978 het Theresia Lyceum. Mijn middelbare-schooltijd werd in 1984 afgesloten met het behalen van het diploma VWO, met als zevende vak Latijn.

Aansluitend ben ik Informatica gaan studeren aan de Katholieke Universiteit Nijmegen, om in 1991 af te studeren met als hoofdvak Grondslagen van de Informatica. Vervolgens ben ik in 1992 aangesteld als AIO (assistent in opleiding) door dezelfde universiteit. Deze aanstelling werd gefinancieerd door het TLI (Taal Logica en Informatica) netwerk. Ik was werkzaam bij de afdeling Grondslagen van de vakgroep Informatica onder leiding van prof.dr. H.P. Barendregt.

Op 10 mei 1997 ben ik getrouwd met Manon van Lieshout en op 3 juni het jaar erna werd onze dochter Ilse geboren. Begin 1997 heb ik het internet-bedrijf Paracas Software opgericht. In 1998 ben ik in dienst getreden bij het automatiseringsbedrijf Logica.

| | |
|---|---|
| *Adres:* | Opaalhof 21 |
| | 3402 ZJ IJsselstein |
| | the Netherlands |
| *E-mail:* | markr@cs.kun.nl |
| | mark@paracas.nl |
| | ruysm@logica.com |