Abstraction and Computation

Type Theory, Algebraic Structures, and Recursive Functions

Venanzio Capretta

ISBN 90-9015738-7

Typeset with LaTeX 2$_\varepsilon$
Printed by Print Partners Ipskamp, Enschede

The work in this thesis has been carried out under the auspices of the research school MRI (Mathematical Research Institute).

Abstraction and Computation

Type Theory, Algebraic Structures, and Recursive Functions

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 23 april 2002
des namiddags om 3.30 uur precies
door

Venanzio Capretta

geboren op 12 oktober 1969 te Valdobbiadene

Promotor:
Prof. dr. H.P. Barendregt
Copromotor:
Dr. H. J. Geuvers

Manuscriptcommissie:

Prof. dr. Peter Aczel	University of Manchester
Prof. dr. Peter Dybjer	Chalmers University of Technology
Prof. dr. Bart Jacobs	

Acknowledgements

Many people helped me in different ways during these four years. I want to thank the following persons for all their contributions.

Henk Barendregt for his guidance, helpful discussions and numerous *koan*s. His comments on various aspects of this thesis contributed to improve it greatly.

Herman Geuvers was always ready to listen to my questions and discuss issues in type theory and logic. He gave expert opinion an suggestions on different parts of the thesis.

Erik Barendsen helped me during the first part of my PhD program, guiding me in the first steps of my research.

To Ana Bove, Gilles Barthe, and Olivier Pons my thanks for their contributions that appear here.

Dick van Leijenhorst taught me complexity theory and helped me become a better teacher.

The friendly atmosphere and passionate discussions in the Foundations group in Nijmegen were a great incentive in my work. I would like to express all my gratitude to all of its members: Wil Dekkers, Silvia Ghilezan, Thierry Joly, Jan Willem Klop, Milad Niqui, Jasper Stein, Dan Synek, Freek Wiedijk, and Jan Zwanenburg.

Nicole Messink took care of most practical aspects of my research activity, bureaucratic matters, help in finding a house, and much more.

Wieb Bosma and Ton Levelt taught me computer algebra and algebraic geometry.

Silvio Valentini and Giovanni Sambin taught me mathematical logic and type theory.

Giovanni Curi shared with me the passion for mathematics and logic since our student days in Padova. He read my thesis and had many useful comments.

Marieke Huisman helped me with my Dutch and gave me good advise to improve this book.

Thanks to my neighbours Petra and Daphne for their friendship and their help in settling in a new country.

Finally, my greatest gratitude goes to my wife Terri. She supported me during these four years and lovingly shared the ups and downs of this adventure.

Contents

Chapter 1

Introduction

The most dramatic technological innovation of the last century is the introduction of computers in our daily lives. Computers are taking on more and more of the tasks that were once peculiar to humans. Almost every field of knowledge has been revolutionized by the advent of information technology. We can already glimpse the outlines of a future in which computing machines will be essential to almost every human activity.

One area in which computers failed to produce a radical renovation is mathematical reasoning. It is true that successes have been obtained in the mechanization of computations and Computer Algebra systems are today widely used. One of the most spectacular achievements of Computer Algebra was its use by Martinus J. G. Veltman and Gerardus 't Hooft to perform precise calculations in the non-abelian gauge theory of electro-weak interaction. This feat earned them the 1999 Nobel price for physics. But the most important activity of a mathematician, proving new theorems, has changed little with the introduction of computing devices and is still done in the old artisan way. This fact makes many mathematicians proud of their craft, it being a convincing proof that mathematics, far from being a dry subject dealing mostly with boring computations, is one of mankind's most creative and ingenuous products. These mathematicians are convinced that computers will never catch up, because they can get faster and more powerful, but not more creative. Their position is well expressed by Paul Halmos' statement:"Efficiency is meaningless. Understanding is what counts. So, is the computer important to mathematics? My answer is *no*. It is important, but not to mathematics."[Alb82] Similar proud declarations on the superiority of the human mind over the electronic chip were formerly uttered by many chess masters, some of whom didn't hesitate to offer high money prizes for any computer able to beat them. Their confidence that they would never have to pay has eventually been shattered.

But mathematics is much more complex than chess playing and the number of *moves* (proof steps) that are possible in a *game* (proof of a theorem) are unlimited. Today's computers cannot achieve the basic mathematical skill that a first year university student learns to master in the first weeks of her education.

The situation is even more dismal. Computer programs able to *check* a chess game have been around from the very beginning of the computer age. These are programs that do not play chess themselves, but simply verify that two human players are playing according to the rules and can tell when a check or checkmate occurs. However, no program capable of checking the correctness of a mathematical proof produced by a human exists. At present the proof of a theorem must be changed and extended with usually ignored details before it can be fed into a proof-checker, that is, a computer program that can verify its correctness. The amount of extra information that must be added is at times so large that the process becomes unfeasible. The present effort in the field of theorem proving tries to bridge the gap between informal mathematics and formalized mathematics. This thesis is a small contribution in that direction. Its main aim is the formalization of important parts of mathematics that have the characteristic of being of wide use in many proof methods and therefore are essential tools for theorem proving.

1.1 Constructive Mathematics

One question we must tackle before putting ourselves to work in the formalization of mathematics is what kind of mathematics we decide to formalize and especially what foundational theory we choose. In informal mathematics the foundational theory of reference is usually Zermelo-Fraenkel set theory (ZFC). This means that *in principle* all mathematics can be formalized in ZFC. Usually, a much weaker theory is sufficient. Peano Arithmetics (PA) is enough for much of Number Theory and each abstract theory has its own first order axiomatization. Analysis can be formalized in second order Peano Arithmetics (PA_2). The mathematician doesn't need to commit herself to one of these foundations as long as her proofs use standard techniques that are known to be formalizable in some system.

But when we use a computer to do mathematics we are forced to make the decision, since our program for proof-checking mathematical reasoning will have to be based on something. I will argue that the most sensible choice is not ZFC but some form of constructive type theory. This choice can be justified on philosophical but also on very practical grounds. Why do we want to do mathematics on a computer at all? The answer that the computer can check that our proofs are correct is not very convincing: peer reviewing works very well for that purpose, and if we try to insists on this point we will be selling a product that nobody needs. Of course, we could promise that computers will be able to prove things that humans can't, but we will not be able to keep that promise in the near future. We should instead turn our attention to what computers are really good at: computing! Nobody will question the utility of computer programs on the grounds that we can write those programs on a piece of paper faster and more easily, forgetting a lot of boring details. This would be a very silly position, since those programs written on paper cannot be executed. Similarly, the objection that formalizing a piece of mathematics in a computer

is useless because we can do it faster and more easily on a piece of paper could be disputed on similar grounds. But proofs of mathematical theorems cannot be *executed*, can they? Certainly classical proofs cannot, but constructive proofs can, in a sense. Constructive proofs give algorithms to compute all the objects of which they claim the existence and to decide all properties of which they claim decidability.

Some people find it strange to consider a proof a program, and even stranger that there can be different proofs of the same proposition that differ with respect to their *efficiency*. But we could convince them by starting our argument from the opposite end: instead of discussing the different ways to formalize mathematics and contending that the constructive approach is better, we discuss ways of extending existing programming languages with richer data types and new programming techniques. We don't even mention logic and mathematics, but just present a high level functional programming language and argue that it makes the job of the programmer easier by showing that some classical algorithms have very short and elegant formulations in it and that new interesting programs can be written. Eventually, we will point our interlocutor's attention to the fact that these new programs can be very naturally considered as proofs of theorems and that constructive mathematics is a subsystem of this programming language. The most important advantage is that programs written in this language are guaranteed to be correct by virtue of the their inherent logical content. We do this in the next section.

1.2 Extending functional programming

Now we imagine that we are talking to a computer programmer. We are trying to convince him to use a new programming language that we designed. This is a functional programming language similar to ML [MTHM97], Haskell [JHA+99], and Clean [dMJB+]. We assume that we already convinced him to use functional programming. So we have shown to him that our programming language has some basic data types as natural numbers $\mathbb{N}$, booleans $\mathbb{B}$ and types with a finite number n of elements $\mathbb{N}_n$. We also showed to him that he can define his own inductive types and that he can form new types from old ones with a number of type constructors, he can create the function type $A \to B$ from types A and B, the product type $A \times B$ and the sum type $A + B$, and the type of lists $\mathsf{List}(A)$. We explained that the main *instructions* of the language are not assignment and loops as in an imperative language, but pattern matching and recursion. Now we want to take him a step forward and show to him the most advanced features of the language. We begin with a practical example. We say to him: "How would you write a simple program that computes the sum of n-dimensional vectors of natural numbers?". At first he may think that the question is very easy and he will give a simple answer like:

$$\mathsf{VectorSum}\colon \mathbb{N}^n \times \mathbb{N}^n \to \mathbb{N}^n$$
$$\mathsf{VectorSum}(\langle a_1, \ldots, a_n\rangle, \langle b_1, \ldots, b_n\rangle) := \langle a_1 + b_1, \ldots, a_n + b_n\rangle.$$

We tell him that this answer is not good. This is not a program that we can run on our computer, because he used two elements that are not part of the programming language: he wrote $\mathbb{N}^n$ to indicate the n-fold repetition of the product of types, and he used dots '...'. We require that he makes explicit how he is going to implement these features. He may say that the meaning of those notations is clear and that when he is given the actual value of n, the program will become a real executable one, for example when $n = 3$ it will become:

$$\begin{array}{l}\mathsf{VectorSum}\colon (\mathbb{N}\times\mathbb{N}\times\mathbb{N})\times(\mathbb{N}\times\mathbb{N}\times\mathbb{N})\to\mathbb{N}\times\mathbb{N}\times\mathbb{N}\\ \mathsf{VectorSum}(\langle a_1,a_2,a_3\rangle,\langle b_1,b_2,b_3\rangle) := \langle a_1+b_1, a_2+b_2, a_3+b_3\rangle\end{array}$$

that is a valid program. We are not yet happy with this answer. We tell him that we want a program that takes n as an input, together with the two vectors to be summed. What he gave us is an infinite family of programs, one for each value of n. It is very likely that our computer programmer will come up with a typical solution that forgets about products of types and uses lists instead:

$$\begin{array}{l}\mathsf{VectorSum}\colon \mathbb{N}\times\mathsf{List}(\mathbb{N})\times\mathsf{List}(\mathbb{N})\to\mathsf{List}(\mathbb{N})\\ \mathsf{VectorSum}(0,l_1,l_2) := []\\ \mathsf{VectorSum}(n+1,l_1,l_2) := [\ \mathsf{head}(l_1)+\mathsf{head}(l_2) :\\ \qquad\qquad\qquad\qquad \mathsf{VectorSum}(n,\mathsf{tail}(l_1),\mathsf{tail}(l_2))\ ]\end{array}$$

After complimenting him for showing perfect understanding of the use of pattern matching and recursion, and for his cleverness in using lists instead of vectors, we tell him that we are not yet satisfied. We wanted the program to be well typed with respect to our original specification. This means that only the inputs in which the length of the vectors (here lists) is exactly that specified by the first input n are accepted, while in this program any lists are allowed. The spirit of functional programming is that the type structure guarantees that only sensible inputs can be given. He may propose to enlarge the program with an "error detecting" part that exits with a "wrong input" message when the lists are of the wrong length. But then we should change the type of the function, because it is not true anymore that it always gives an output in the type $\mathsf{List}(\mathbb{N})$. Our programmer will probably manage to modify his program so it does the desired job, but at the cost of loosing the elegance of functional programming by using tricks that are typical of imperative languages. Hoping that by now he is rather frustrated, we bestow our solution upon him:

$$\begin{array}{l}\mathsf{Vector}(0) := \mathbb{N}_1\\ \mathsf{Vector}(n+1) := \mathbb{N}\times\mathsf{Vector}(n)\end{array}$$

$$\begin{array}{l}\mathsf{VectorSum}(0,0_1,0_1) := 0_1\\ \mathsf{VectorSum}(n+1,\langle x_1,\vec{y}_1\rangle,\langle x_2,\vec{y}_2\rangle)) := \langle x_1+x_2, \mathsf{VectorSum}(n,\vec{y}_1,\vec{y}_2)\rangle\end{array}$$

He may now complain that we are cheating: we omitted writing the types of these two programs, and there is actually no good typing for them in the system so far defined. Specifically, the first program defines *types* by recursion, while we can only define *objects* belonging to a fixed type. The second program cannot be

typed because the type of the output depends on the input. We announce then that our system has more advanced types that allow us to give a type to the programs Vector and VectorSum. For Vector we need to see types themselves as objects belonging to a new type of types. We call this higher level type Type_1. Its elements are the basic types $\mathbb{N}$, $\mathbb{B}$ and $\mathbb{N}_n$, and all the types that can be obtained from these by finite applications of the type constructors $\times$, $+$ and List. Now we can give a good type to our first program:

$$\mathsf{Vector}\colon \mathbb{N} \to \mathsf{Type}_1.$$

To deal with VectorSum we must introduce dependent products. Vector is a dependent type indexed on the natural numbers. VectorSum is a function with three arguments. The first argument is a natural number, that is, an element $n\colon \mathbb{N}$. The second and third arguments are elements of the type $\mathsf{Vector}(n)$, and so is the result. If we fix the value n of the first argument, we obtain a function of the remaining two arguments of type $\mathsf{Vector}(n) \times \mathsf{Vector}(n) \to \mathsf{Vector}(n)$. We still need to abstract the first argument to obtain the desired function. We indicate the type of such functions with the standard mathematical notation $\Pi_{n:\mathbb{N}}\mathsf{Vector}(n) \times \mathsf{Vector}(n) \to \mathsf{Vector}(n)$ or, in our programming language, with $(n : \mathbb{N})\mathsf{Vector}(n) \times \mathsf{Vector}(n) \to \mathsf{Vector}(n)$. Now we can give a type to our second program:

$$\mathsf{VectorSum}\colon (n : \mathbb{N})\mathsf{Vector}(n) \times \mathsf{Vector}(n) \to \mathsf{Vector}(n).$$

Finally, here is the complete program that, we claim, is more elegant and has a better typing that the one our friend produced using lists:

$\mathsf{Vector}\colon \mathbb{N} \to \mathsf{Type}_1$
$\mathsf{Vector}(0) := \mathbb{N}_1$
$\mathsf{Vector}(n+1) := \mathbb{N} \times \mathsf{Vector}(n)$

$\mathsf{VectorSum}\colon (n : \mathbb{N})\mathsf{Vector}(n) \times \mathsf{Vector}(n) \to \mathsf{Vector}(n)$
$\mathsf{VectorSum}(0, 0_1, 0_1) := 0_1$
$\mathsf{VectorSum}(n+1, \langle x_1, \vec{y}_1\rangle, \langle x_2, \vec{y}_2\rangle) := \langle x_1 + x_2, \mathsf{VectorSum}(n, \vec{y}_1, \vec{y}_2)\rangle.$

The advantage of typed programming languages over untyped ones lies in the expressivity of specifications. An untyped program accepts any valid input and gives an output of which we do not know anything. On the other hand, a typed program specifies a set of admissible inputs and guarantees that the output will be of a certain kind. For example, a program of type $\mathbb{N} \to \mathbb{N}$ accepts only natural numbers as input and gives natural numbers as output. The compiler can check that the program satisfies the typing specification. Typing errors can then be caught at compile time. This type checking is similar to the dimensional checking for formulas in physics: Controlling that the two sides of an equation have the same physical dimensions does not verify whether the formula is correct, but it tests at least that it is meaningful. We can refine type specifications from generic ones to detailed ones. For example, if we are programming a sorting

algorithm on sequences of natural numbers, a first typing of the program may be

$$\mathsf{sort}\colon \mathsf{List}(\mathbb{N}) \to \mathsf{List}(\mathbb{N})$$

which just specifies that the program takes a list of natural numbers as input and computes a list of natural numbers as output. This guarantees that the program does not accept booleans as input and does not give pointers to strings as output, but it is by no means a complete specification of a sorting algorithm. We would desire, for example, that the output list has the same length as the input one. This is not ensured by the type. We may refine the specification by using vectors in place of lists:

$$\mathsf{sort}\colon (n\colon \mathbb{N})\mathsf{Vector}(n) \to \mathsf{Vector}(n).$$

Now we know for sure that the output has the same length as the input, but there is no guarantee that it is an ordered list. To obtain this property we need to further refine the type of the output. Let us say that the output vector must be in decreasing order. Then every component of the vector must be smaller or equal than the preceding one. If the vector is $\langle x_1, \ldots, x_n\rangle$, then x_{i+1} must be an element of $\mathbb{N}_{x_i+1}$ (remember that $\mathbb{N}_y = \{0_y, 1_y, \ldots, (y-1)_y\}$; for the sake of simplicity we identify i_y with i and consider $\mathbb{N}_y$ as a subtype of $\mathbb{N}$). For example, a pair of numbers such that the second is smaller or equal to the first can be typed by

$$\langle x, y\rangle\colon \mathbb{N} \times \mathbb{N}_{x+1}.$$

This notation is a bit sloppy, since the variable x is not bound in the type: The definition of the type should be independent from its elements. A more precise notation is $(x\colon \mathbb{N}) \times \mathbb{N}_{x+1}$. In type theory it is more common to indicate this type by $\Sigma x\colon \mathbb{N}.\mathbb{N}_{x+1}$, but we will stick to the '$\times$' notation in this section. The type of ordered vectors of length n can then be defined as

$$\mathsf{OrdVector}(n) := (x_1\colon \mathbb{N}) \times (x_2\colon \mathbb{N}_{x_1+1}) \times \cdots \times (x_n\colon \mathbb{N}_{x_{n-1}+1}).$$

This means that an element of $\mathsf{OrdVector}(n)$ is an n-tuple $\langle x_1, \ldots, x_n\rangle$ such that:

- $x_1\colon \mathbb{N}$;
- $x_2\colon \mathbb{N}_{x_1+1}$, that is, x_2 is a natural number smaller than or equal to x_1;
- in general, $x_{i+1}\colon \mathbb{N}_{x_i+1}$, that is, x_{i+1} is a natural number smaller than or equal to x_i.

Let us give an example. If we choose $n = 4$, then

$$\mathsf{OrdVector}(4) = (x_1\colon \mathbb{N}) \times (x_2\colon \mathbb{N}_{x_1+1}) \times (x_3\colon \mathbb{N}_{x_2+1}) \times (x_4\colon \mathbb{N}_{x_3+1}).$$

The tuple $\langle 5, 3, 3, 0\rangle$ is an element of this type because $5\colon \mathbb{N}$, $3\colon \mathbb{N}_{5+1} = \mathbb{N}_6 = \{0, 1, 2, 3, 4, 5\}$, $3\colon \mathbb{N}_{3+1} = \mathbb{N}_4 = \{0, 1, 2, 3\}$, and $0\colon \mathbb{N}_{3+1} = \mathbb{N}_4 = \{0, 1, 2, 3\}$. On the other hand, $\langle 5, 3, 4, 0\rangle$ is not an element of $\mathsf{OrdVector}(4)$, because 4 is not an element of $\mathbb{N}_{3+1} = \mathbb{N}_4 = \{0, 1, 2, 3\}$.

We define the family of types of ordered vectors similarly to how we defined the family Vector:

$$\begin{aligned}
&\mathsf{OrdVectorFrom}\colon \mathbb{N} \to \mathbb{N} \to \mathsf{Type}_1\\
&\mathsf{OrdVectorFrom}(0, x) := \mathbb{N}_1\\
&\mathsf{OrdVectorFrom}(n+1, x) := (y\colon \mathbb{N}_{x+1}) \times \mathsf{OrdVectorFrom}(n, y)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{OrdVector}\colon \mathbb{N} \to \mathsf{Type}_1\\
&\mathsf{OrdVector}(0) := \mathbb{N}_1\\
&\mathsf{OrdVector}(n+1) := (x\colon \mathbb{N}) \times \mathsf{OrdVectorFrom}(n, x).
\end{aligned}$$

The sorting algorithm can be typed as

$$\mathsf{sort}\colon (n\colon \mathbb{N})\mathsf{Vector}(n) \to \mathsf{OrdVector}(n).$$

This specification ensures that the algorithm takes a vector as input and gives an ordered vector of the same length as output. We managed to express in the typing much more that the simple condition that the program takes inputs of the right kind and produces outputs of right kind. We were able to specify part of the correctness requirement of the program in the typing.

In some systems (for example in PVS [OSRSC99]) it is possible to define a type like $\mathsf{OrdVector}(n)$ as a subtype of $\mathsf{Vector}(n)$ by saying that the elements of $\mathsf{OrdVector}(n)$ are those vectors that satisfy a suitable condition. We would then write $\mathsf{OrdVector}(n) = \{v\colon \mathsf{Vector}(n) \mid \phi(v)\}$, where ϕ is the suitable condition. This requires that ϕ is expressed in some logical language independent of the programming language in use. Here we advocate a different solution, in which the structure of the programming language itself allows the expression of logical properties.

The standard methodology in the verification of the correctness of software consists in constructing a program and then, separately, proving that it computes the desired result. With our expressive type system, we can include the correctness requirement directly in the type of the program.

There is still something missing: The elements of the output vector must be the same as the elements of the input vector. We do not pursue the complete specification here. We just state that it is possible to formalize it, and that eventually we would obtain a type for the sorting algorithm that looks like

$$\mathsf{sort}\colon (n\colon \mathbb{N})(v\colon \mathsf{Vector}(n))(w\colon \mathsf{OrdVector}(n)) \times \mathsf{Permutation}(v, w)$$

where $\mathsf{Permutation}(v, w)$ is the type of permutations that turn v into w. The type may be read as: sort is a function that takes a natural number n and a vector v of length n as input, and gives an ordered vector w of length n and a permutation that turn v into w as output. A program sort having this type is automatically correct, without the need of a proof.

We are content with having demonstrated the expressive power of type theory, and we hope to have convinced our computer programmer friend that it is a very interesting programming language.

We also obtained the goal of leading him into the realm of logic without him noticing it. In fact, giving a specification for the sorting algorithm that guarantees that the output is of the same length as the input and is ordered is equivalent of giving the simple specification (with lists) and then requiring that the programmer gives a proof that the algorithm satisfies those conditions. With type theory we integrate this logical information into the typing system. We will determine in more detail the relation between type theory and logic after giving a short overview of the different opinions about the philosophy of logic and mathematics.

1.3 Different philosophies of mathematics

Let us for a moment abandon our computer scientist, as he meditates on the expressivity of the new programming language, and let us turn to some philosophical issues. Since we are going to teach computers to do mathematics, it is essential that we know what mathematics is. This turns out to be a very thorny problem, from which most mathematicians turn away, deeming it a useless philosophical debate. As we already remarked, we do not have the luxury of avoiding the question, since from the answer we give will depend the nature of our computer implementation. Let us then summarize the main positions on the philosophy of mathematics. More exhaustive treatment can be found in [vH67], [Ewa96], or [MB93] (if you can read Italian).

1.3.1 Formalism

This is the position of David Hilbert and his pupils and followers (see, for example, [Hil05] and [Hil26]), and it is the most popular position among mathematicians because it is simple and avoids philosophical questions. It is in nature similar to the solution that Alexander the Great gave to the Gordian knot: he cut through it! According to formalism, logical formulas are just strings of symbols and logical principles just rules to manipulate those strings. Formulas have no meaning, or better, their meaning is not relevant to mathematics; only their formal properties are. The only part of mathematics to have meaningful content is finitary mathematics, that is, the analysis of concrete discrete objects and their decidable properties. To guarantee that the formal activity is not completely pointless, Hilbert postulated that a mathematical theory must have some model if it is consistent. In other words, he assumed that consistency, that is, the underivability of contradictory statements, coincides with satisfiability, that is, the existence of a mathematical universe in which all the derivable statements are true. The criterion for the acceptability of a theory is only its consistency:

> So in recent times we come upon statements like this: even if we could introduce a notion safely (that is, without generating contradictions) and if this were demonstrated, we would still not have established that we are justified in introducing the notion. Is this

> not precisely the same objection as the one formerly made against complex numbers, when it was said that one could not, to be sure, obtain a contradiction by means of them, but their introduction was nevertheless not justified, for, after all, imaginary magnitudes do not exist? No, if justifying a procedure means anything more than proving its consistency, it can only mean determining whether the procedure is successful in fulfilling its purpose.
>
> Hilbert [Hil26], p. 370

In this passage, Hilbert disagrees with those who reject the introduction of new notions in mathematics on philosophical grounds. He contends that a new concept is acceptable as long as its introduction is proved to be consistent. This claim was later proved by Kurt Gödel [Göd30]: It is enough to prove the consistency of a theory, to be sure that it is meaningful (that is, it has a model). Since finitary mathematics is the only solid ground to start from, the proof of consistency must be carried out by finitary means. Hilbert thought that this was possible, if we consider mathematical propositions as concrete discrete objects themselves and the application of logical rules as decidable properties over them. By considering logical systems as the object of mathematical investigation, Hilbert founded the field of proof theory. Unfortunately, this programme was shattered by Gödel's discovery [Göd31] that a proof of consistency for a theory always requires the use of a more powerful, and thus less reliable, theory. Infinitary mathematics cannot be proved consistent by finitary means.

One of the richest formal theories developed within this conception of mathematics is Zermelo-Fraenkel set theory ZFC, that also leans towards some of the ideas of logicism, another school of thought that we explain in the next section. It is usually considered to be the cornerstone of modern mathematics.

The formalist approach lends itself easily to implementation in a computer, because it already considers mathematical reasoning as a form of computation with discrete symbolic objects. Manipulating strings of symbols is the idiosyncrasy of computers. Implementations of ZFC are the computer system Mizar [Miz01] and the development of set theory in Isabelle by L. Paulson ([Pau94], pp. 203-233).

1.3.2 Logicism

Logicism tries to found the whole of mathematics on basic logical principles. It tries first of all to isolate the correct principles of reasoning, which have no specific content, but are just the ways to formulate concepts and propositions and to derive their interconnections. The main tenet of logicism is that mathematical objects are in fact logical constructions. For example, numbers are conceived as logical predicates satisfied by other predicates whose extensions are in mutual one-one correspondence. The purely logical nature of mathematics accounts for its abstraction. This philosophy of mathematics was initially proposed by Frege ([Fre79], [Fre84], [Fre03]).

Russell's paradox proved that Frege's system was inconsistent. But Russell himself and Whitehead managed to reformulate logicism in a way that avoids paradoxes, in a system called *the ramified theory of types*. Together they produced the monumental *Principia Mathematica* [WR27], that dominated mathematical logic during the twenties. They successfully managed to create a system in which, in principle, most of the mathematics known then could be formalized, but at the cost of blurring the logical simplicity of Frege's original ideas.

Some of the assumptions made in the *Principia* violate the fundamental dogma that no extralogical principles is to be allowed. In particular the *reduction principle*, essential for impredicative reasoning, was strongly criticized by predicativists like Henri Poincaré and by the intuitionists. The ramified theory of types divides the mathematical universe in an infinite hierarchy of types. Type 1 is the type of individuals, that is, mathematical objects that cannot be decomposed in components. The elements of Type 2 are classes of elements of Type 1, that is, classes of individuals. The elements of Type 3 are classes of elements of Type 2, that is, classes of classes of individuals. And so on. Every type is further divided into orders, according to the logical complexity of the predicate defining the classes. For example, the objects of Type 2 and Order 1 are those classes of individuals that can be defined only by predicates in which there are no quantifications, that is, all variables of Type 1 occur free; the objects of Type 2 and Order 2 are those classes of individuals that can be defined by predicates in which quantifications are done over variables of Type 1 and Order 1; the objects of Type 2 and Order 3 are those classes of individuals that can be defined by predicates in which quantifications are done over variables of Types 1 and Order 1 and of Type 1 and Order 2; and so on. Furthermore, there is the possibility of defining, for example, a class of Type 2 by quantifying over object of Type 3 and Order 1 and so on. Therefore, the orders do not form just a linear progression, but a ramified one. This gives the ramified structure of types and orders that form the base of Russell's system. If this was all, the theory would be perfectly predicative: to define an object of a certain type and order, we can use quantification only on simpler types and orders. However, this theory was not sufficient to formalize all mathematics. Russell introduced the *reduction principle*, stating that for every predicate of higher order there exist a predicate of first order with the same extension. There is no logical justification of such a principle, as Poincaré argued.

A simpler version of logicism, that avoids the reduction principle and still allows the reconstruction of all classical mathematics, was later presented by Frank Plumpton Ramsey [Ram31].

After Kurt Gödel proved that any logical system is inherently incomplete, the attempt to construct a single foundation for all mathematics was abandoned and the logicist approach to mathematics dwindled.

A logicist approach to the implementation of mathematics into a computer is feasible and sensible. It is natural to set up a system that represents the fundamental logical concepts and principles and use it as a sort of *logical programming language* to implement mathematics.

1.3.3 Intuitionism

> The question where mathematical exactness does exist, is answered differently by the two sides; the intuitionist says: in the human intellect, the formalist says: on paper.
>
> L. E. J. Brouwer [Bro13], p. 56

The philosophy of mathematics initiated by L. E. J. Brouwer [HF75] is based on the rejection of the Platonic notion of the existence of a plane of being in which mathematical objects reside. According to that conception, the activity of the mathematician consists in an attempt to probe this perfect metaphysical universe with her mind. In particular, it is assumed that mathematical objects have an ontological existence outside the minds of mathematicians and that there is an answer to every question about them.

Brouwer totally overturned this conception. He contended that mathematics is a human activity and its objects have meaning only inside the human mind. Numbers and functions are mental constructions and mathematical theorems express the capacity of the human mind to perform certain operations on these constructions.

This led to the rejection of two fundamental principles accepted in classical mathematics.

The principle of *the excluded middle*, that states that every mathematical proposition is either true or false, is based on the Platonic conviction that every question about mathematical objects must have an answer, even if we cannot find it. According to Brouwer a question has an answer when we find it; if we are unable to construct a proof or a confutation of a proposition, we cannot assume that that proposition must be either true or false.

The principle of *impredicativity*, that allows the construction of a new object of a certain class by referring to the whole class as a completed whole, was already questioned by Poincaré [Poi06, Poi16, Poi25] and Weyl [Wey18]. For a Platonist, mathematical objects and classes exist independently of the human mind; a definition is just a way to denote a preexisting entity. Therefore, both the defined object and the class to which it belongs exist independently of the definition and reference to the class is not problematic. For an intuitionist a definition creates the object it defines and we cannot treat a class as a whole until we have completely determined what its elements are. Therefore we cannot refer to a class while defining a new element of it.

Intuitionism, in Brouwer's form, doesn't lend itself to computer implementation, since Brouwer rejected the idea that mathematical reasoning is a formal activity. He even refused to use logical symbolism, preferring, whenever possible, to express himself in natural language. The human mind, he contended, is a *creative subject*, that can perform constructions that elude any formal process. He even contested that mathematics is based on logic, arguing that logic is just a part of mathematics and that mathematical activity is essentially nonlogical. Thus, intuitionism is radically opposed to both formalism and logicism.

Nevertheless, the intuitionist conception of mathematics gave rise to different streams of constructivism that have many connections with computer science (see [TvD88] and [Bee85]). A surprising fact is that the rejection of the principles of excluded middle and impredicativity turns out to have very important consequences for computability. It is also possible to reconcile intuitionism with its enemies and formulate a constructive foundation of mathematics that synthesize some of the core ideas of formalism, logicism and intuitionism.

1.3.4 Constructivism

The term *constructivism* is often used as a synonym of *intuitionism*. Here, I use it with a slightly different meaning: a refinement of intuitionistic mathematics based on the reinterpretation of Brouwer's *mental constructions* as *computer programs*.

When Brouwer proposed its new philosophy of mathematics, the modern computer had not appeared yet (though some of its ancestors had been around for centuries, for example the computing machines constructed by Babbage, Pascal, and Leibniz). But soon later the first cornerstones of theoretical computer science began to be laid by Turing [Tur36], Kleene [Kle36, Kle52] and Church [Chu32, Chu41]. The convergence of some of the basic ideas of intuitionism and computer science became soon evident. The starting point can be identified in the work of Arend Heyting [Hey56], Brouwer's student. As we saw, Brouwer rejected symbolic logic as a foundation of mathematics and consequently wasn't particularly interested in developing an intuitionist account of symbolic logic. Heyting provided the latter, providing an explanation of logical connectives and quantifiers in terms of intuitionistic mathematics. He contended that the meaning of logical formulas doesn't lie in absolute truths external to the human mind, but in the conditions under which we can achieve a proof of them. Consequently, the meaning of a formula is given by stating what its proofs are. The logical connectives and quantifiers are then explained in the following way:

- a proof of $A \wedge B$ is a pair consisting of a proof of A and a proof of B;
- a proof of $A \vee B$ is either a proof of A or a proof of B, together with the information on which of the two has been proved;
- a proof of $A \to B$ is a function that maps every proof of A to a proof of B;
- the formula $\bot$ has no proofs;
- a proof of $\forall x B[x]$ is a function that maps every object a in the domain of discourse to a proof of $B[a]$;
- a proof of $\exists x B[x]$ is a pair consisting of an object a and a proof of $B[a]$.

Consistently with the intuitionistic conviction that mathematical objects are pure mental contents not suitable to formal analysis, there is no further explanation of what is meant for the mathematical objects used, in particular of the notion of function.

Stephen C. Kleene provided a more precise notion by using recursive functions. He called the interpretation of logical connectives in recursion theory the *realizability interpretation* [Kle52]. In this interpretation every proposition is interpreted as a natural number, by using the encoding of pairs of natural numbers and of recursive functions as natural numbers (here, the interpretation of quantifiers assumes that the domain of discourse consists of the natural numbers; it can be easily adapted to domains that have an encoding into natural numbers):

- a proof of $A \wedge B$ is a natural number e that is the encoding of a pair $e = \#\langle a, b\rangle$, consisting of a proof a of A and a proof b of B;
- a proof of $A \vee B$ is a natural number e that is the encoding of a pair $e = \#\langle c, d\rangle$, where c is either 0 or 1 and, if $c = 0$, d is a proof of A, if $c = 1$, d is a proof of B;
- a proof of $A \to B$ is a natural number e such that for every proof a of A, $\{e\}(a)$ is a proof of B (here, $\{e\}$ indicates the recursive function with code e);
- the formula $\bot$ has no proofs;
- a proof of $\forall x B[x]$ is a natural number e such that for every natural number n, $\{e\}(n)$ is a proof of $B[n]$;
- a proof of $\exists x B[x]$ is a natural number e that is the encoding of a pair $e = \#\langle n, p\rangle$, consisting of a natural number n and a proof p of $B[n]$.

With this interpretation, proofs become themselves mathematical objects (natural numbers) about which we can reason. We have thus a framework in which to do metamathematics. The arithmetical character of proofs leads to interesting logical properties. The rules of intuitionistic logic are valid under this interpretation, but the law of excluded middle does not hold. Indeed Kleene formulated some axioms that are valid under this interpretation and are inconsistent with the excluded middle.

The realizability interpretation is very important for our aims. If proofs are represented as natural numbers, then they can be stored inside a computer and formal operations can be carried out on them. We must, though, stress a difficulty that still remains: in the definition of the interpretation of the implication and universal quantification, there remains a nonconstructive point. It is required, for example in the definition of a proof e of $A \to B$, that for every proof a of A, $\{e\}(a)$ is a proof of B. But this condition cannot in general be verified. If for a specific value a we compute the result of the application $\{e\}(a)$, we can check that it is indeed a proof of B. But A may have an infinite number of different proofs, and therefore we cannot verify by computation that we always get a proof of B for every proof of A. Besides, a code e represents a *partial* recursive function. It is necessary that the function $\{e\}$ terminates on all the proofs of A, but this is not in general decidable (not even for a given specific value a). Similar problems plague the definition of the universal quantifier.

Therefore, it is not in general decidable whether a code e is indeed a proof of a certain proposition. This is a serious drawback for the implementation on a computer, since the system would not be able to check whether a proof produced by the user or by a program is correct. There are two possible ways out of this difficulty. The first consists in enriching the definition of proofs to include also proofs of termination and correctness of the functions. But in this approach we run against a clear circularity in the definition of proof (in defining what a proof is we require a proof of a certain statement). The second solution is to restrict our notion of function to more manageable entities. This will restrict the logical strength of the interpretation (that was unlimited in the original formulation), but will give us a real computational model. The first restriction we can think of consists in using primitive recursive functions in place of general recursive ones. This solves the problem of termination (since primitive recursive functions are total), but still leaves open the problem of checking that the results of the computations are correct. Besides, we may be unsatisfied with the relatively low logical strength given by primitive recursive arithmetics.

If we consider the problem carefully, we realize that it is a *type checking* requirement: we want the output of our function to belong to a certain type (the type of proofs of B) and we are looking for a way to guarantee that this type specification is always met. In functional programming languages the rules are constructed in such a way that only well typed functions can be programmed. The very structure of the programming language guarantees that the programs always terminate and always give an output of the correct type. (We do not consider, here, functional programming languages with fixed point operators, or equivalent constructions, that allow the formulation of non-terminating programs.) This is all we need to modify the realizability interpretation so that proof checking becomes decidable. In this approach we do not interpret every proposition as a subset of natural numbers (codes representing proofs), but we associate to a proposition a type whose elements can be conceived as its proofs. After all we have said it becomes easy to see a structural analogy between logical operators and type constructors. This analogy has been made precise and is called the *Curry-Howard isomorphism* (see [dB70] and [How80]). It is at the base of modern constructive type theory. The first computer implementation of it was the system AUTOMATH (see [dB70] and [dB80]). Per Martin-Löf extended the isomorphism to include disjunction, existential quantification, and induction (see [ML98],[ML82] and [ML84]; some of the basic ideas were already present in [Sco70], who was inspired by de Bruijn's work on AUTOMATH).

1.4 Relation between mathematics and science

> The miracle of the appropriateness of the language of mathematics for the formulation of the laws of physics is a wonderful gift, which we neither understand nor deserve.
>
> Eugene P. Wigner [Wig67]

We have seen the philosophical basis on which different foundations of mathematics are built. But mathematics has also a practical justification. It is indispensable for science. Galileo wrote that "the book of nature is written in mathematical language", and every modern science has as its highest goal the formulation of precise mathematical laws for the phenomena it studies. A criterion by which to choose a foundation of mathematics could then be its usefulness in science. I claim that constructive mathematics is the best choice in this respect and the only one that has an epistemological justification.

Let me begin by outlining the basic tenets of modern philosophy of science (for an overview of the current state of philosophy of science, see, for example, [NS00]). A science is the human endeavor to find an explanation for a class of natural phenomena. Its starting point is the description of a finite number of such phenomena in a precise (preferably mathematical) language. The scientist tries to find an underlining order in these data. She formulates hypotheses on the causal relation between the phenomena. For these hypotheses to be scientifically meaningful, a series of requirements must be put on them. We can subsume these requirements under two general principles.

The first I call *the principle of exactness*. It states that the hypotheses must be clear and precise in what they state. Their meaning must be completely determined and the same for all human beings. If some elements are left ambiguous, the explicative power of the hypotheses must not depend on them. (Indeed all scientific hypotheses have a certain degree of abstractness in them, and abstractness means exactly ignoring those details that have no bearing on the explanation of the phenomena.) In short, the hypotheses must be precise and abstract, which is the most general possible definition of mathematics.

The second principle I call *the principle of predictability*. It states that from the hypotheses it must be possible to derive predictions about new phenomena in the class under study that have not yet been observed. Only then the scientist can proceed to a verification of the hypotheses by making new observations or constructing experiments and checking the results against the predictions of the theory.

The two principles together guarantee that the hypotheses are, according to Popper's terminology [Pop59, Pop65], *falsifiable*, that is, we have a procedure to test whether the hypotheses are wrong. If they are, then they are falsifiable, that is, we can find a phenomenon that is in disagreement with the prescription of the hypotheses.

Let us consider different conceptions of mathematics in the light of these two principles. The first principle is satisfied by all different schools of mathematics, since it coincides with a very general definition of mathematics. (One could have doubts about intuitionism, that relies on personal psychological experience, and Platonism, that stipulates the direct contact of the human mind with a perfect world of mathematical entities; but even these schools admit that mathematical knowledge must be unambiguously communicable.) But the second principle is much more restrictive. It rejects all theories that rely on undecidable principles. We could formulate it in a very precise mathematical way by saying that *a scientific theory is a recursive function* whose input is an

encoding of the description of the system under observation and whose output is the encoding of the outcome of the experimentally observed phenomenon. Of course, the theory may not be able to compute the outcome of the observation, that is, the recursive function may be undefined for some inputs.

We explain this point using Popper's definition of the empirical content of a scientific theory. We call *basic statements* those assertions that any observer, without disagreement, can verify or refute by direct observation of natural phenomena. Popper sees the theory as a generator of basic statements. If the theory T generates the basic statement S, we can directly check whether S is true or not in nature. If it is false, T has been falsified. We call *potential falsifier* any basic statement that, if confirmed by observation, would falsify the theory ("not S" in the example is a potential falsifier). By definition, the *empirical content* of the theory T is measured by the class of potential falsifiers of T. The intuitionistic and classical versions of T have exactly the same class of falsifiers, since the extra strength of classical logic can be used only to prove undecidable, that is non-basic, statements. Therefore the empirical content of a classical theory is equal to the empirical content of its intuitionistic version.

However, the intuitionistic version can be much more efficient in the generation of potential falsifiers. Suppose that, for some parameters x and y of a physical system, $\Phi(x)$ is a property of the system before a phenomenon occurs, and $\Psi(y)$ is a property of the system after the phenomenon occurs. Both $\Phi(x)$ and $\Psi(y)$ are basic statements for every value of x and y. Then the relation $\Phi(x) \to \Psi(y)$ expresses the fact that if we make an experiment with an initial parameter of value x satisfying $\Phi(x)$, then the outcome of the experiment will have the value y for the final parameter and it will satisfy $\Psi(y)$. Now, assume that in our theory the theorem

$$\forall x.\exists y.\Phi(x) \to \Psi(y)$$

holds, establishing an empirical law about the physical system.

If the theorem was proved using constructive mathematics, we automatically obtain a computable function f such that

$$\forall x.\Phi(x) \to \Psi(f(x)).$$

All statements of the form

$$\Phi(x) \wedge \neg\Psi(f(x))$$

are potential falsifiers.

On the other hand, if the theorem was proved using classical mathematics, we cannot extract a computable function f from the proof. Therefore, we cannot generate the subclass of potential falsifiers. Every potential falsifier can still be produced in the classical theory (because every constructive proposition holds also classically), but they cannot be spawned uniformly. In other words, we have to perform an inefficient proof search to obtain the same potential falsifiers classically.

A possible critique to this position is that constructive mathematics is usually much more complicated than classical mathematics. Notions that are classically equivalent split into different concepts in a constructive setting. One could claim that only with classical mathematics we can formulate simple and elegant theories that really give us the feeling that we are understanding nature. Usually a dichotomy is made between theoretical and experimental science, and correspondingly between pure and applied mathematics. The job of the theoretician is to formulate elegant mathematical theories. The job of the applied mathematician and of the experimenter is to find ways to compute the results of the abstract theory and to test them. This dichotomy is expressed in the words of Albert Einstein: "The skeptic will say: 'It may well be true that this system of equations is reasonable from a logical standpoint. But this does not prove that it corresponds to nature.' You are right, dear skeptic. Experience alone can decide on truth. Yet we have achieved something if we have succeeded in formulating a meaningful and precise equation. The derivation, from the questions, of conclusions which can be confronted with experience will require painstaking efforts and probably new mathematical methods." (from *Scientific American*, April 1950, reprinted in the April 2000 issue, p. 6.) One of the strongest advocates of this *mathematical segregation* is Paul Halmos, who declared in an interview that pure mathematics is "beautiful and it is a work of art", while applied mathematics "solves problems about waterways, sloping beaches, airplane flights, atomic bombs, and refrigerators. But just the same, much too often it is bad, ugly, badly arranged, sloppy, untrue, undigested, unorganized, and unarchitected mathematics" [Alb82]. He even wrote a whole article entitled *Applied Mathematics Is Bad Mathematics* [Hal81].

(Don't draw the conclusion that constructive mathematics is ugly! This is just Halmos opinion. I argue that constructive mathematics is the beautiful way of doing applied mathematics.)

This attitude distinguishes between the proof of a theorem, a job for the pure mathematician, and the construction of algorithms that compute the contents of the theorem, a job for the applied mathematician. The latter activity is not so noble and artistic as the former. After all, *knowing* that something is true is the main aim of mathematics. Being able to compute the contents has only a practical interest. But a scientific theory can only be falsified to the extent to which we can derive measurable consequences from it. Therefore only the *dirty* algorithms of the applied mathematician can claim an epistemological status, while noncomputable pure mathematics, although elegant and artistic, does not satisfy the requirements for a good scientific theory.

Constructive mathematics is both elegant and abstract and has epistemological value.

Since one of the most important uses of computer algebra systems today consists in the computation of the results from scientific theories in order to compare them with the outcome of experiments and observation (as in the already mentioned work by Veltman and 't Hooft), we have another good reason to adopt the constructive standpoint.

1.5 Type theory for computer science and mathematics

The musings of the previous two sections lead to the choice of a framework for the development of computer mathematics that I adopted in my research project. Let me outline it in short. The next chapter will provide the formal definition.

We will use (constructive) type theory as a foundation for both computer science and mathematics. Type theory is a high level programming language. Its constituents are types and their elements. Its computational model consists in reduction of terms to canonical form. Type theory is richer than usual functional programming languages in that it contains dependent types and type universes. Its computational model is more restricted: only wellfounded recursion is allowed, while in other programming languages nonterminating functions can be defined. (But we see in Chapters 6 and 7 that there are ways of representing general recursion in type theory.) Logic is considered as a part of computer science, propositions being just types and their proofs elements of types. The activity of proving a theorem coincides with that of writing a program that satisfies a given typing specification. We can then formulate a new version of the realizability interpretation that associates a type $[\![A]\!]$ to every logical formula A. The elements of $[\![A]\!]$ are the proofs of A. In particular, we may refine the interpretation of $\forall x.P(x)$ as the type $\Pi_{x:\,D}[\![P(x)]\!]$ (or, with other notation, $(x\colon D)[\![P(x)]\!]$), where D is the domain-type of the variable x and $[\![P(x)]\!]$ is the type of the proofs of $P(x)$. Similarly, we can interpret $\exists x.P(x)$ as $\Sigma_{x:\,D}[\![P(x)]\!]$ (or $(x\colon D) \times [\![P(x)]\!]$ in the alternative notation).

There is not just one type theory, but infinitely many. Let us consider again the realizability interpretation: Proofs are interpreted as computable functions. It is not possible to use all computable total functions, because that is a undecidable class and proof-checking would also become undecidable. We may restrict the class of admissible computable functions in various ways. Accordingly we get various type theories with various logical strengths. The best general framework for the definition of type theories is Barendregt's formulation of *Pure Type Systems* [Bar92]. Not all Pure Type System result in a consistent logic. There are restrictions and extensions with sum types and inductive types. Two important restrictions are predicativism (Per Martin-Löf [ML82, ML84]) and categoricity (Bart Jacobs [Jac99]).

1.6 Contents

The title of this thesis refers to two fundamental elements that are essential in the formal development of mathematics in type theory.

Abstraction refers to the need of defining general structures and to keep the developments as general as possible, so to be able to reuse results. We tackle this issue by developing a general formulation of set theory in type theory by means of setoids, the development of Universal Algebra and equational reasoning.

Computation refers to the aim of maintaining a close link between formal proofs and algorithmic content. Type theory itself is an integrated foundation for logic and computation. We aim at developing tools that allow reasoning about general algorithms in type theory. To this purpose, we study the formulation of general recursion theory, and we present a case study in the formal verification of computer algebra, the Fast Fourier Transform.

The thesis consists in three parts.

Part I contains an introduction to type theory, specifically the Calculus of Constructions with inductive and coinductive types. In Chapter 2 we give the formal rules of the theory. In Chapter 3 we introduce coinductive types, first on an intuitive level, then by their formal rules.

Part II provides some tools that are necessary to begin formalizing mathematics in type theory. In Chapter 4 we face the problem of defining infinite families of inductive types; we offer several solutions to the problem. In Chapter 5, joint work with Gilles Barthe and Olivier Pons, we develop a theory of sets inside type theory. Sets are represented as pairs consisting of a type and a defined equality on it. Setoids come in different formulations in the literature. We give a systematic treatment and a mathematical analysis of the various possibilities. Chapters 6 and 7 tackle the same problem: How can we define general recursive functions in type theory. Chapter 6, joint work with Ana Bove, solves the problem by characterizing the domain of a recursive function by an inductive predicate. Chapter 7 solves the problem by using a coinductive type to represent infinite computations.

Part III describes the formal development of parts of mathematics in type theory, specifically formalizing them with the proof-assistant Coq[BBC+99]. In Chapter 8 we develop Universal Algebra and equational reasoning over algebraic structures. In Chapter 9 we present an implementation of the Fast Fourier Transform in Coq with the formal proof of its correctness.

Part I

Type Theory

Chapter 2

Rules of Type Theory

In this chapter, we introduce the fundamental concepts of type theory and we give the formal definition of the Calculus of Inductive Constructions, which is the version of type theory that we use in this thesis. The basic constituents of a theory of types are types and terms. Types are collections of elements, and terms are expressions indicating elements of the types. The notation $t\colon T$ asserts that the term t denotes an element of the type T. Different versions of type theory have different rules for the generation of types and the definition of their elements. In the next section, we explain the general form of these rules; in the rest of the chapter, we give the specific formal rules for the Calculus of Inductive Constructions.

2.1 General form of the rules

The version of type theory that we adopt is the Calculus of Inductive Constructions ([CH88],[Wer94]), but we never use its impredicative features, therefore everything could be done inside the intensional version of Constructive Type Theory ([ML82], [ML84], [NPS90]).

The theory consists of judgments and rules to derive them. In this respect, we take a different view from the standard notion of a theory as a class of derivable assertions. Our point of view is more intensional: We put more emphasis on how the assertions are derived. There could be two different sets of rules by which the same class of assertions can be derived. We still distinguish the two sets of rules as two different theories.

We assume that we have an infinite supply of variables and of identifiers for constants. A *judgment* is a sequent of the form

$$x_1\colon T_1, \ldots, x_n\colon T_n \vdash t\colon T,$$

where $x_1, \ldots, x_n$ are distinct variables; $T_1, \ldots, T_n$, and T are types; and t is a term. Intuitively it means that, if $x_1, \ldots, x_n$ represent objects of type $T_1, \ldots, T_n$, respectively, then t represents an object of type T. The structure of the term

t is a description of the procedure that constructs this object from the objects represented by the variables.

The theory consists of a collection of rules to derive judgments.

A *valid context* is a sequence $x_1 : T_1, \ldots, x_n : T_n$ such that there is a judgment having it as left side of the sequent, that is, there are a term t and a type T such that the judgment $x_1 : T_1, \ldots, x_n : T_n \vdash t : T$ is derivable. The rules are such that every type must be an object of a universe type, or sort. Then it can be shown that $x_1 : T_1, \ldots, x_n : T_n$ is a valid context if and only if the following judgments are derivable:

$$\begin{aligned} \vdash T_1 &: s_1 \\ x_1 : T_1 \vdash T_2 &: s_2 \\ &\vdots \\ x_1 : T_1, \ldots, x_{n-1} : T_{n-1} \vdash T_n &: s_n \end{aligned}$$

with $s_1, \ldots, s_n$ type universes and $x_1, \ldots, x_n$ distinct variables.

If $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ is a valid context, we say that $\Delta = y_1 : U_1, \ldots, y_n : U_m$ is a *valid extension* of Γ if Γ, Δ is also a valid context.

An *instantiation* of this context Δ is a series of values for its variables, possibly in an extension Γ' of Γ:

$$\begin{aligned} &\Gamma, \Gamma' \vdash u_1 : U_1 \\ &\Gamma, \Gamma' \vdash u_2 : U_2[y_1 := u_1] \\ &\qquad \vdots \\ &\Gamma, \Gamma' \vdash u_m : U_m[y_1 := u_1, \ldots, y_{m-1} := u_{m-1}]. \end{aligned}$$

Let us write $\Gamma, \Gamma' \vdash \bar{u} :: \Delta$ to indicate that $\bar{u}$ is an instantiation of Δ.

We often use the notation $t[x]$ to denote a term t in which a variable x may occur, but need not occur. So t and $t[x]$ denote the same term, but in the second expression we stress the dependence on x. After denoting t as $t[x]$, we may write $t[u]$, for a term u, to denote $t[x := u]$, that is, the result of the substitution of u for every free occurrence of x in t. The notation $[x]t$ indicates the function that to every u associates $t[u]$. It is not intended to be an element of a functional type, but a metafunction on the syntactic level. To be totally formal, we should then write $\lambda([x]t)$ to indicate the internal function. Anyway, we will usually identify the two where no confusion arises.

The general structure of the theory comprises an infinite hierarchy of predicative type universes denoted Type_0, Type_1, Type_2, ... The hierarchy is cumulative, that is, Type_i is contained in Type_{i+1} for every i. Under Type_0 there are two additional impredicative types Set and Prop.

Every type is given by four rules.

The first rule is the *formation rule*, which specifies the name of the type and the type universe in which it resides. For type constructors, for example products and sums, it specifies how the constructor is applied to other types to

obtain a new type. The formation rule can also be seen as an introduction rule for the universe to which the type belongs.

The second rule is the *introduction rule*, which specifies how the elements of the type are generated. There may be many introduction rules if there are many ways of constructing elements. If there is no introduction rule, then the type is empty. The general form of an introduction rule allows us to build a new element of the type by applying a *constructor* to a number of arguments. The arguments may come from other types or from the defined type itself, in which case we have an inductive type.

The third rule is the *elimination rule*, which specifies how the elements of the type can be used. The general form of the elimination rule states that we can do with an element of the type anything that could be done from the arguments from which it was constructed. Therefore, the elimination rule can be automatically obtained from the introduction rules. This is realized by an operator, called *destructor* or *selector*, that takes as arguments functions that produce a result of the desired type from the hypotheses of each of the introduction rules, and an element of the type. In some cases, for example, product types, a different but equivalent rule is used. Let us assume that the introduction rules of a type T are

$$\frac{a_{11} : H_{11}, \ldots, a_{1k_1} : H_{1k_1}}{\mathsf{c}_1(a_{11}, \ldots, a_{1k_1}) : \mathsf{T}},$$

$$\frac{a_{21} : H_{21}, \ldots, a_{2k_2} : H_{2k_2}}{\mathsf{c}_2(a_{21}, \ldots, a_{2k_2}) : \mathsf{T}},$$

$$\vdots$$

$$\frac{a_{n1} : H_{n1}, \ldots, a_{nk_n} : H_{nk_n}}{\mathsf{c}_n(a_{n1}, \ldots, a_{nk_n}) : \mathsf{T}}.$$

Here, $\mathsf{c}_1, \ldots, \mathsf{c}_n$ are the constructors of T.

We can formulate a corresponding elimination rule. First we give a simplified version, in which a function from the type T to some other type D is constructed:

$$\frac{\begin{array}{l} x_{11} : H_{11}, \ldots, x_{1k_1} : H_{1k_1} \vdash d_1[x_{11}, \ldots, x_{1k_1}] : D \\ x_{21} : H_{21}, \ldots, x_{2k_2} : H_{2k_2} \vdash d_2[x_{21}, \ldots, x_{2k_2}] : D \\ \vdots \\ x_{n1} : H_{n1}, \ldots, x_{nk_n} : H_{nk_n} \vdash d_n[x_{n1}, \ldots, x_{nk_n}] : D \end{array}}{t : \mathsf{T} \vdash \mathsf{T}\text{-}\mathsf{elim}([x_{11}, \ldots, x_{1k_1}]d_1, [x_{21}, \ldots, x_{2k_2}]d_2, \ldots, [x_{n1}, \ldots, x_{nk_n}]d_n, t) : D}.$$

The rule says that, if we can construct an element of D for each of the different canonical forms of elements of T specified by the introduction rule, then we can construct an element of D from every element of T.

The names of the constructors c_i and of the selector $\mathsf{T}\text{-}\mathsf{elim}$ do not have any independent meaning inside type theory: They become valid terms only when they are applied to arguments. If we want to define a constructor c_i as a

function, we need to take the abstraction of its full application

$$[a_{i1}\colon H_{i1}, \ldots, a_{ik_i}\colon H_{ik_i}]\mathsf{c}_i(a_{i1}, \ldots, a_{ik_i}).$$

However, for simplicity, we use the constructor name c_i to denote this function, when such notation does not lead to confusion. Similarly, we use the name $\mathsf{T\text{-}elim}$ to denote the function obtained from the abstraction of the full application of the selector.

The general elimination rule allows D to depend on the element t of T: assuming $t\colon T \vdash D[t]\colon s$ for some type universe s,

$$\frac{\begin{array}{l} x_{11} : H_{11}, \ldots, x_{1k_1} : H_{1k_1} \vdash d_1[x_{11}, \ldots, x_{1k_1}]\colon D[\mathsf{c}_1(x_{11}, \ldots, x_{1k_1})], \\ x_{21} : H_{21}, \ldots, x_{2k_2} : H_{2k_2} \vdash d_2[x_{21}, \ldots, x_{2k_2}]\colon D[\mathsf{c}_2(x_{21}, \ldots, x_{2k_1})], \\ \vdots \\ x_{n1} : H_{n1}, \ldots, x_{nk_n} : H_{nk_n} \vdash d_n[x_{n1}, \ldots, x_{nk_n}]\colon D[\mathsf{c}_n(x_{n1}, \ldots, x_{nk_n})] \end{array}}{t\colon \mathsf{T} \vdash \mathsf{T\text{-}elim}([x_{11}, \ldots, x_{1k_1}]d_1, [x_{21}, \ldots, x_{2k_2}]d_2, \ldots, [x_{n1}, \ldots, x_{nk_n}]d_n, t)\colon D[t]}.$$

Here, $\mathsf{T\text{-}elim}$ is the destructor of the type T. In some cases, it is necessary to restrict the type universe s of the elimination type D. This happens when impredicative universes are involved: In that case, the restriction on the type universe of the elimination type are necessary to avoid inconsistencies.

For inductive types, the elimination rule assumes also that the result of the elimination has already been computed for the recursive arguments, yielding the recursion and induction principles. Let us consider the simplified example with just two constructors:

$$\frac{a_{11}\colon H_{11} \qquad t_1\colon \mathsf{I}}{\mathsf{c}_1(a_{11}, t_1)\colon \mathsf{I}},$$

$$\frac{t_2\colon \mathsf{I} \qquad a_{22}\colon H_{22}}{\mathsf{c}_2(t_2, a_{22})\colon \mathsf{I}},$$

where we have that $H_{12} = \mathsf{I}$, $a_{12} = t_1$, $H_{21} = \mathsf{I}$, and $a_{21} = t_2$. We call t_1 and t_2 the *recursive arguments* of the constructors c_1 and c_2, respectively. The elimination rule assumes that a value of the target type has already been obtained for the recursive arguments:

$$\frac{\begin{array}{l} x_{11} : H_{11}, x\colon \mathsf{I}, h_x\colon D[x] \vdash d_1[x_{11}, x, h_x]\colon D[\mathsf{c}_1(x_{11}, x)], \\ y\colon \mathsf{I}, h_y\colon D[y], x_{22} : H_{22} \vdash d_2[y, h_y, x_{22}]\colon D[\mathsf{c}_2(y, x_{22})] \end{array}}{t\colon \mathsf{I} \vdash \mathsf{I\text{-}elim}([x_{11}, x, h_x]d_1, [y, h_y, x_{22}]d_2, t)\colon D[t]}.$$

The fourth rule is the *reduction rule*. It states that a term in which the elimination rule has been applied to a term obtained from an introduction rule can be simplified by applying directly the argument of the elimination rule to the arguments of the introduction rule. There is a reduction rule for each constructor. This rule constitutes the computational content of type theory. In our examples, the reduction rules for the type T are

$$\begin{array}{l} \mathsf{T\text{-}elim}([x_{11}, \ldots, x_{1k_1}]d_1, \ldots, [x_{n1}, \ldots, x_{nk_n}]d_n, \mathsf{c}_i(a_{i1}, \ldots, a_{ik_i})) \\ \rightsquigarrow d_i[a_{i1}, \ldots, a_{ik_i}] \end{array}$$

for $1 \leq i \leq n$, and the reduction rules for the inductive type I are

$$\begin{array}{l}\mathsf{I\text{-}elim}([x_{11}, x, h_x]d_1, [y, h_y, x_{22}]d_2, \mathsf{c}_1(a_{11}, t_1)) \\ \rightsquigarrow d_1[a_{11}, t_1, \mathsf{I\text{-}elim}([x_{11}, x, h_x]d_1, [y, h_y, x_{22}]d_2, t_1)]\end{array}$$

and

$$\begin{array}{l}\mathsf{I\text{-}elim}([x_{11}, x, h_x]d_1, [y, h_y, x_{22}]d_2, \mathsf{c}_2(t_2, a_{22})) \\ \rightsquigarrow d_2[t_2, \mathsf{I\text{-}elim}([x_{11}, x, h_x]d_1, [y, h_y, x_{22}]d_2, t_2), a_{22}].\end{array}$$

In the rules for I, the variables h_x and h_y are instantiated recursively by another application of $\mathsf{I\text{-}elim}$.

Now that we explained the general shape of the rules, we start giving the specific rules for the Calculus of Inductive Constructions.

2.2 Structural Rules

Structural rules are rules concerning the general form of a judgment. They state that the elements of the sorts are types and therefore we are free to assume variables in them, and that contexts can be extended.

variable rule

$$\frac{\Gamma \vdash T : s}{\Gamma, x : T \vdash x : T}$$

weakening rule

$$\frac{\Gamma \vdash t : T}{\Gamma, \Delta \vdash t : T}$$

if Δ is a valid extension of Γ.

2.3 Type Universes

Type universes are given only by formation/introduction rules. It is possible to formulate an elimination rule in the form of a universe induction principle (see [Acz82]). However, such elimination rule would imply that the only types in the universes are the ones specified. Therefore, we would not be allowed to add new types later without changing the universe elimination rule. This would lead to a type theory that is not conservative with respect to its possible extensions.

Here are the introduction rules for universes:

$$\frac{}{\Gamma \vdash \mathsf{Set} : \mathsf{Type}_0} \quad \frac{}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_0} \quad \frac{}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}$$

for every valid context Γ and every i.

The fact that every universe is contained in the following ones is given by the *cumulativity rule*:

$$\frac{\Gamma \vdash A : \mathsf{Set}}{\Gamma \vdash A : \mathsf{Type}_i} \quad \frac{\Gamma \vdash A : \mathsf{Prop}}{\Gamma \vdash A : \mathsf{Type}_i} \quad \frac{\Gamma \vdash A : \mathsf{Type}_i}{\Gamma \vdash A : \mathsf{Type}_{i+1}}.$$

From now on s, with or without subscripts, will denote any of the type universes Set, Prop, or Type_i.

The universes Type_i for i a natural number form the *cumulative hierarchy* of predicative types. The impredicativity of the universes Set and Prop at the bottom is determined by the different restrictions on the universe of the types in the formation rules for product types.

We follow the type theory of Coq [BBC+99] in using two impredicative universes. Set is intended as the universe of data types and programs. Prop is intended as the universe of logic. The methodology implied consists in formalizing an algorithm in Set and then proving its correctness in Prop. While the elements of Set can be eliminated over Prop, that is, we can use computational content in the proofs, the elements of Prop cannot be eliminated over Set, that is, we cannot use a proof to develop a program.

In other versions of type theory, specifically in Lego, impredicativity is used only for the logical part. Then there is still an impredicative universe Prop, but no universe Set. Type_0 is used as the sort of data types and programs.

In this thesis we never use the impredicative features of Set and Prop. This means that our results are still valid if we just use Type_0 in place of both Set and Prop.

The presence of impredicativity forces us to restrict the scope of some elimination rules: We cannot eliminate a type belonging to an impredicative universe over a type belonging to a predicative one. Such unrestricted elimination would cause the system to be inconsistent [Coq86, Coq90, Coq94].

2.4 Dependent Product Types

Dependent product types are used to represent functions for which the type of the result depends on the argument. Let A be a type and B be a family of types depending on A, that is, for each element $a\colon A$, $B[a]$ is a type. Then the product type $(x\colon A)B$ is the type of the functions f such that, for every $a\colon A$, $(f\ a)\colon B[a]$. We already stated that, if $b[x]\colon B[x]$, we indicate by $[x]b$ the metafunction mapping every $a\colon A$ to $b[a]$. The constructor of the dependent type product is λ, so we have that $\lambda(A,[x]b)\colon (x\colon A)B$. As mentioned above, it is not necessary to distinguish the two, so we will in general identify $[x]b$ with $\lambda(A,[x]b)$. We write $[x\colon A]b$ when it is necessary to indicate the type of the argument variable.

formation

$$\frac{\Gamma \vdash A\colon s_1 \quad \Gamma, x\colon A \vdash B[x]\colon s_2}{\Gamma \vdash (x\colon A)B\colon s_3} \quad \langle s_1, s_2, s_3\rangle \in \mathcal{A}$$

introduction

$$\frac{\Gamma, x\colon A \vdash b[x]\colon B[x]}{\Gamma \vdash [x\colon A]b\colon (x : A)B}$$

elimination

$$\frac{\Gamma \vdash f\colon (x\colon A)B \quad \Gamma \vdash a : A}{\Gamma \vdash (f\ a)\colon B[a]}$$

reduction

$$([x\colon A]b\ a) \rightsquigarrow b[x := a]$$

The side condition $\langle s_1, s_2, s_3\rangle \in \mathcal{A}$ in the formation rule gives the universe restrictions on product types as in Pure Type Systems [Bar92]. Here

$$\begin{aligned}\mathcal{A} = \{\langle s_1, s_2, s_3\rangle \mid\ & s_2 = s_3 \in \{\mathsf{Set}, \mathsf{Prop}\} \text{ or} \\ & s_1 \in \{\mathsf{Set}, \mathsf{Prop}\}, s_2 = \mathsf{Type}_{i_2}, s_3 = \mathsf{Type}_{i_3} \text{ with } i_2 \leq i_3 \text{ or} \\ & s_1 = \mathsf{Type}_{i_1}, s_2 = \mathsf{Type}_{i_2}, s_3 = \mathsf{Type}_{i_3} \text{ with } i_1, i_2 \leq i_3\}.\end{aligned}$$

In general the universe of the product type must be higher or equal to the universes of the domain type A and of the dependent range type B. Only when the dependent type is in Set or in Prop, we are allowed to use any type as domain and have that the product is still in Set or Prop. This property is a formulation of *impredicativity*.

Notice that the elimination rule does not follow the general pattern illustrated in Section 2.1. According to the general form of elimination rule, we should state that we can do with an element of $(x\colon A)B$ all that could be done from the assumptions in the introduction rule. The assumption of the introduction rule is $\Gamma, x\colon A \vdash b[x] : B[x]$; that is, we can construct an element of $(x\colon A)B$ from a derivation of an element of B from the assumption of an element of A. If we try to follow the general form of the elimination rule, we should have something like: Let $y\colon (x\colon A)B \vdash T[y]\colon s$; given a generic derivation that from $x\colon A$ constructs an element of $B[x]$, if we can give a term of type $T[([x\colon A]b)]$; then we can construct a term of type $T[f]$ for every $f : (x\colon A)B$. The problem is that we cannot talk inside type theory about derivations from A to B.

One way to solve the problem is to distinguish meta-functional types [BV92]. Let us use the notation $\{x\colon A\}B$ for the type of meta-functions from A to B. That is, an element of $\{x\colon A\}B$ is not a function in the theory, but a map in the metatheory. Similarly we can use the notation $x \mapsto b$ for a metafunction, that is, the intuitive map that to every $a : A$ associates $b[x := a]$, but without saying that $x \mapsto b$ is a function in the theory.

Then we may give the rules for $(x\colon A)$ (the "real" function type in the theory) as

introduction

$$\frac{\Gamma \vdash m\colon \{x\colon A\}B}{\Gamma \vdash \lambda(m)\colon (x\colon A)B}$$

elimination

$$\frac{\Gamma, f\colon (x\colon A)B \vdash T[f]\colon s \quad \Gamma, y\colon \{x\colon A\}B \vdash e[y]\colon T[\lambda(y)]}{\Gamma, f\colon (x\colon A)B \vdash (\mathsf{fun\text{-}elim}\ [y]e\ f)\colon T[f]}$$

reduction

$$(\mathsf{fun\text{-}elim}\ [y]e\ \lambda(m)) \rightsquigarrow e[y := m]$$

If we do this, we must formalize the rules for the meta-types, and those are going to look like the rules for the function types that we have now. We prefer instead, for our purposes, to have the elimination rule in the form of the application of the function to an argument.

The type $(x\colon A)B$ can also be denoted by the expressions $(\Pi x\colon A.B)$, $\prod_{x\colon A} B$, and $\forall x\colon A.B$ in the case that the universe of B is Prop. This last convention will be justified when we talk about logic. If the variable x does not occur free in B, we write $A \to B$ for $(x\colon A)B$. When the type of the variable x is clear from the context, we write $[x]b$ in place of $[x : A]b$. The most common notation for $[x : A]b$ is the λ-notation $(\lambda x\colon A.b)$ or $(\lambda x.b)$.

Sometimes one or more arguments of a function type can be inferred from the type of the following arguments. In this case we adopt the convention of not explicitly writing these arguments, which we call *implicit arguments*. To mark that an argument is implicit we put the symbols $\lfloor\_\rfloor$ around its type in the type of the function. For example, if a function f is specified as having type $\lfloor(x\colon A)\rfloor B[x] \to C$, then its first argument does not need to be specified: If $a\colon A$ and $b\colon B[a]$, then we write $(f\ b)$ in place of $(f\ a\ b)$. Sometimes we need to make explicit an occurrence of an implicit argument. We do it by putting before the occurrence the symbol $n!$ where n is a numeral indicating that what follows is the nth implicit arguments. In the previous example we would then write $(f\ 1!a\ b)$. If we want to specify that all implicit arguments are explicitly written, then we put the symbol ! in front of the name of the function: $(!f\ a\ b)$.

Let Δ be a valid extension of the context Γ. That is, $\Delta = x_1\colon A_1, \ldots, x_n\colon A_n$ with

$$\begin{aligned} \Gamma &\vdash A_1\colon s_1 \\ \Gamma, x_1\colon A_1 &\vdash A_2\colon s_2 \\ &\vdots \\ \Gamma, x_1\colon A_1, \ldots, x_{n-1}\colon A_{n-1} &\vdash A_n\colon s_n. \end{aligned}$$

If $\Gamma, \Delta \vdash B\colon s_0$, we denote by $\Pi\Delta.B$ the product $(x_1\colon A_1)\cdots(x_n\colon A_n)B$. We have that

$$\Gamma \vdash \Pi\Delta.B\colon s$$

where s is the type universe with maximum index among $s_0, \ldots, s_n$, or $s = s_0 \in \{\mathsf{Set}, \mathsf{Prop}\}$. If B does not depend on any of the variables $x_1, \ldots, x_n$, we write $\Delta \to B$ for $\Pi\Delta.B$.

2.5 Nondependent Product Types

Nondependent product types are used to make the cartesian products of types. If A and B are types of the same sort, then $A \times B$ is the type of pairs whose first element comes from A and second element comes from B.

formation

$$\frac{\Gamma \vdash A\colon s \quad \Gamma \vdash B\colon s}{\Gamma \vdash A \times B\colon s}$$

introduction

$$\frac{\Gamma \vdash a\colon A \quad \Gamma \vdash b\colon B}{\Gamma \vdash \langle a, b\rangle\colon A \times B}$$

elimination

$$\frac{\Gamma, z\colon A \times B \vdash T[z]\colon s \quad \Gamma, x\colon A, y\colon B \vdash t[x,y]\colon T[\langle x, y\rangle]}{\Gamma, c\colon A \times B \vdash (\mathsf{prod\text{-}case}\ [x,y]t\ c)\colon T[c]}$$

reduction

$$(\mathsf{prod\text{-}case}\ [x,y]t\ \langle a, b\rangle) \rightsquigarrow t[a,b]$$

The elimination rule states that we can unpackage a pair into its components. We illustrates it with an example:

$$\frac{\Gamma \vdash C\colon s \quad \Gamma \vdash f\colon A \to B \to C}{\Gamma, z\colon A \times B \vdash (\mathsf{prod\text{-}case}\ f\ z)\colon C}$$

with the reduction

$$(\mathsf{prod\text{-}case}\ f\ \langle a, b\rangle) \rightsquigarrow (f\ a\ b).$$

In other words, $(\mathsf{prod\text{-}case}\ f)$ is the uncurried version of f: while $f\colon A \to B \to C$, $(\mathsf{prod\text{-}case}\ f)\colon A \times B \to C$.

If we generalize this example to a dependent elimination type, we obtain

$$\frac{\Gamma, z\colon A \times B \vdash C[z]\colon s \quad \Gamma \vdash f\colon (x\colon A)(y\colon B)C[\langle x, y\rangle]}{\Gamma, z\colon A \times B \vdash (\mathsf{prod\text{-}case}\ f\ z)\colon C[z]}$$

with exactly the same reduction

$$(\mathsf{prod\text{-}case}\ f\ \langle a, b\rangle) \rightsquigarrow (f\ a\ b).$$

Usually, the general elimination and reduction rules are replaced by the two projection rules and the corresponding reduction rules, that are equivalent to them:

$$\frac{\Gamma \vdash c\colon A \times B}{\Gamma \vdash (\pi_1\ c)\colon A}, \quad \frac{\Gamma \vdash c\colon A \times B}{\Gamma \vdash (\pi_2\ c)\colon B},$$

$$(\pi_1\ \langle a, b\rangle) \rightsquigarrow a, \quad (\pi_2\ \langle a, b\rangle) \rightsquigarrow b.$$

They can be obtained from the general elimination rule by taking $T[z] := A$ and $t[x,y] := x$, $T[z] := B$ and $t[x,y] := y$, respectively.

Nondependent products can be obtained as special cases of dependent product if we have a type $\mathbb{B}$ of boolean values, whose elements are true and false. In that case we may first construct the family of types $b\colon \mathbb{B} \vdash P[b]\colon s$ such that $P[\mathsf{true}] = A$ and $P[\mathsf{false}] = B$. Then $A \times B$ is isomorphic to $\Pi_{b\colon \mathbb{B}} P[b]$. However,

cartesian product is such a basic notion that we prefer to have a specific type constructor for it.

We use the notation $A \wedge B$ for $A \times B$ when the sort s is Prop.

We use the following simplified notation for an abstraction of an element of a product type:

$$\begin{array}{rcl}[z = \langle x, y\rangle]e & := & [z](\mathsf{prod\text{-}case}\ [x, y]e\ z) \\ & = & [z](e[x := (\pi_1\ z), y := (\pi_2\ z)]).\end{array}$$

Such notation is often indicated, in functional programming languages, by

$$\lambda z.\mathsf{let}\ z = \langle x, y\rangle\ \mathsf{in}\ e.$$

2.6 Nondependent Sum Types

The nondependent sum of two types A and B is a type containing copies of the elements of both types. The copy is introduced by a constructor specifying from which of the two types it comes from.

formation

$$\frac{\Gamma \vdash A\colon s \quad \Gamma \vdash B\colon s}{\Gamma \vdash A + B\colon s}$$

introduction

$$\frac{\Gamma \vdash a\colon A}{\Gamma \vdash (\mathsf{inl}\ a)\colon A + B} \qquad \frac{\Gamma \vdash b\colon B}{\Gamma \vdash (\mathsf{inr}\ b)\colon A + B}$$

elimination

$$\frac{\begin{array}{c}\Gamma, z\colon A + B \vdash T[z]\colon s' \\ \Gamma, x\colon A \vdash t_l[x]\colon T[(\mathsf{inl}\ x)] \\ \Gamma, y\colon B \vdash t_r[y]\colon T[(\mathsf{inr}\ y)]\end{array}}{\Gamma, c\colon A + B \vdash (\mathsf{case}\ [x]t_l\ [y]t_r\ c)\colon T[c]} \quad s' \in \mathcal{R}(s)$$

reduction

$$(\mathsf{case}\ [x]t_l\ [y]t_r\ (\mathsf{inl}\ a)) \rightsquigarrow t_l[a] \qquad (\mathsf{case}\ [x]t_l\ [y]t_r\ (\mathsf{inr}\ b)) \rightsquigarrow t_r[b]$$

The side condition $s' \in \mathcal{R}(s)$ in the elimination rule restricts the sorts of the elimination predicate. The admissible sorts are

$$\begin{array}{l}\mathcal{R}(\mathsf{Type}_i) = \{\mathsf{Type}_j \mid j \in \mathbb{N}\} \cup \{\mathsf{Set}, \mathsf{Prop}\}, \\ \mathcal{R}(\mathsf{Set}) = \{\mathsf{Set}, \mathsf{Prop}\}, \\ \mathcal{R}(\mathsf{Prop}) = \{\mathsf{Prop}\}.\end{array}$$

So sums of sets in the predicative cumulative hierarchy can be eliminated over any type, sums of elements of Set can be eliminated over Set and Prop, and sums of elements of Prop can be eliminated only over Prop.

The restrictions are necessary because the elimination of types of an impredicative sort over the predicative sorts leads to inconsistency [Coq86] and we do not allow the use of logical information, coming from Prop, in the construction of programs, which is done in Set.

We use the notation $A \vee B$ for $A + B$ when the sort s is Prop.

The expression (case $[x]t_l$ $[y]t_r$ c) can also be denoted by

$$\mathsf{Cases}\ c\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ x) \mapsto t_l \\ (\mathsf{inr}\ y) \mapsto t_r \end{cases}$$

that can be extended to multiple sums: For the sum of n types, $A_0 + A_1 + \cdots + A_{n-1}$, we use the notation $(\mathsf{in}_i\ a_i)$ for $(\mathsf{inr}^i\ (\mathsf{inl}\ a_i))$ if $i < n-1$, and $(\mathsf{inr}^n\ a_i)$ if $i = n-1$. For the elimination, if $c\colon A_0 + A_1 + \cdots + A_{n-1}$, we write

$$\mathsf{Cases}\ c\ \mathsf{of}\ \begin{cases} (\mathsf{in}_0\ x_0) & \mapsto t_0 \\ (\mathsf{in}_1\ x_1) & \mapsto t_1 \\ \qquad\vdots & \\ (\mathsf{in}_{n-1}\ x_{n-1}) & \mapsto t_{n-1} \end{cases}$$

for the term

$$\begin{array}{l} (\mathsf{case}\ [x_0]t_0 \\ \quad [y_1](\mathsf{case}\ [x_1]t_1 \\ \qquad\quad [y_2](\mathsf{case}\ \cdots\ [y_{n-2}](\mathsf{case}\ [x_{n-2}]t_{n-2}\ [x_{n-1}]t_{n-1}\ y_{n-2})\ \cdots\ ) \\ \qquad\quad y_1) \\ \quad c). \end{array}$$

Similar notation can be used for case analysis over any type with several constructors.

2.7 Dependent Sum Types

Dependent sum types, also called Σ-types, are a generalization of nondependent sum types in which the type of the second component of a pair may depend on the first component.

formation

$$\frac{\Gamma \vdash A\colon s_1 \quad \Gamma, x\colon A \vdash B[x]\colon s_2}{\Gamma \vdash (\Sigma x\colon A.B)\colon s_2}.$$

introduction

$$\frac{\Gamma \vdash a\colon A \quad \Gamma \vdash b\colon B[a]}{\Gamma \vdash \langle a, b\rangle\colon (\Sigma x\colon A.B)}$$

elimination

$$\frac{\Gamma, z\colon (\Sigma x\colon A.B) \vdash C[z]\colon s_3 \quad \Gamma, x\colon A, y\colon B \vdash c[x,y]\colon C[\langle x, y\rangle]}{\Gamma, p\colon (\Sigma x : A.B) \vdash (\mathsf{case}\ [x,y]c\ p)\colon C[p]} \quad s_3 \in \mathcal{R}(s_2)$$

reduction

$$(\mathsf{case}\ [x,y]c\ \langle a,b\rangle) \rightsquigarrow c[a,b]$$

We use the same pairing constructor $\langle \_, \_ \rangle$ and the same case selector case for nondependent product types and dependent sum types. This is justified because nondependent products can be obtained as special cases of dependent sums when the variable x does not occur free in B. In that case we can put $A \times B := (\Sigma x\colon A.B)$.

As for the nondependent product types, we may use projections in place of the destructor case. The projection rules are

$$\frac{\Gamma \vdash p\colon (\Sigma x\colon A.B)}{\Gamma \vdash (\pi_1\ p)\colon A}, \qquad \frac{\Gamma \vdash p\colon (\Sigma x\colon A.B)}{\Gamma \vdash (\pi_2\ p)\colon B[(\pi_1\ p)]},$$

$$(\pi_1\ \langle a,b\rangle) \rightsquigarrow a, \qquad (\pi_2\ \langle a,b\rangle) \rightsquigarrow b.$$

They can be obtained from the general elimination rule by taking $C[z] := A$ and $c[x,y] := x$, $C[z] := B[(\pi_1\ z)]$ and $c[x,y] := y$, respectively. Notice, however, that this is possible only if A is in the right type universe: $s_1 \in \mathcal{R}(s_2)$ ($s_2 \in \mathcal{R}(s_2)$ is always true).

We use the notation $\exists x\colon A.B[x]$ for $\Sigma x\colon A.B[x]$ when the sort s_2 is Prop.

As with nondependent products, we use the simplified notation for an abstraction of an element of a Σ-type:

$$\begin{array}{rcl} [z = \langle x,y\rangle]e & := & [z](\mathsf{case}\ [x,y]e\ z) \\ & = & [z](e[x := (\pi_1\ z), y := (\pi_2\ z)]). \end{array}$$

2.8 The Unit type

The type Unit is a type containing a single element $\bullet$.

formation

$$\vdash \mathsf{Unit}\colon s$$

introduction

$$\vdash \bullet\colon \mathsf{Unit}$$

elimination

$$\frac{\Gamma, z\colon \mathsf{Unit} \vdash C[z]\colon s' \quad \Gamma \vdash c\colon C[\bullet]}{\Gamma, u\colon \mathsf{Unit} \vdash (\mathsf{Unit\text{-}case}\ c\ u)\colon C[u]}$$

reduction

$$(\mathsf{Unit\text{-}case}\ c\ \bullet) \rightsquigarrow c$$

Notice that we could strengthen the reduction rule to

$$(\mathsf{Unit\text{-}case}\ c\ u) \rightsquigarrow c$$

for every $u\colon \mathsf{Unit}$, since any element of Unit must be equal to $\bullet$, but we keep it as it is to preserve the usual interpretation of the elimination rule.

The version of the unit type in Prop is usually denoted $\top$.

2.9 The Empty type

The type Empty is a type containing no element.

formation

$$\vdash \mathsf{Empty}\colon s$$

introduction There is no introduction rule.

elimination

$$\frac{\Gamma, z\colon \mathsf{Empty} \vdash C[z]\colon s'}{\Gamma, e\colon \mathsf{Empty} \vdash (\mathsf{Empty\text{-}case}\ e)\colon C[e]}$$

reduction There is no reduction rule.

The version of the empty type in Prop is usually denoted $\bot$.

2.10 Record Types

We generalize further the notion of Σ-types by defining record types with an arbitrary number of elements, each of which has a type that may depend on the previous ones. Assume that that Γ is a valid context and that also Γ, Δ is a valid context. The variables in Δ are the parameters of the record type, which is actually a family of record type indexed on these parameters. We define a type of record with n fields $f_1, \ldots, f_n$ of types $A_1, \ldots, A_n$, respectively, with A_i possibly dependent of f_j for $j < i$:

$$\begin{gathered}\Gamma, \Delta \vdash A_1\colon s_1 \\ \Gamma, \Delta, x_1\colon A_1 \vdash A_2[x_1]\colon s_2 \\ \vdots \\ \Gamma, \Delta, x_1\colon A_1, \ldots, x_{n-1}\colon A_{n-1} \vdash A_n[x_1, \ldots, x_{n-1}]\colon s_n\end{gathered}$$

Then we can declare the record type by the command

$$\mathsf{Record}\ R\colon s\ [\Delta]\ :=\ c\ \begin{cases} f_1 & : A_1 \\ f_2 & : A_2[f_1] \\ \vdots & \\ f_n & : A_n[f_1, \ldots, f_{n-1}] \end{cases}$$

which introduces the family of types R with fields $f_1, \ldots, f_n$ and constructor c.

We we speak of declarations in the system we mean the possibility of the user of defining new types and functions, in this case R, c, and $f_1, \ldots, f_n$. This means that after the declaration the system will contain those constants and the corresponding rules.

formation

$$\Gamma, \Delta \vdash (R\ \Delta)\colon s$$

introduction

$$\frac{\Gamma, \Delta \vdash a_1 \colon A_1 \quad \Gamma, \Delta \vdash a_2 \colon A_2[a_1] \quad \dots \quad \Gamma, \Delta \vdash a_n \colon A_n[a_1, \dots, a_{n-1}]}{\Gamma, \Delta \vdash (c\ a_1\ a_2\ \cdots\ a_n) \colon (R\ \Delta)}$$

elimination

$$\frac{\begin{array}{l}\Gamma, \Delta, z \colon (R\ \Delta) \vdash C[z] \colon s' \\ \Gamma, \Delta, \quad x_1 \colon A_1, \\ \qquad\quad x_2 \colon A_2[x_1], \\ \qquad\quad \vdots \\ \qquad\quad x_n \colon A_n[x_1, \dots, x_{n-1}] \\ \qquad\quad \vdash d[x_1, \dots, x_n] \colon C[(c\ x_1\ \cdots\ x_n)]\end{array}}{\Gamma, \Delta, r \colon (R\ \Delta) \vdash (R{-}\mathsf{case}\ [x_1, \dots, x_n] d\ r) \colon C[r]} \quad s' \in \mathcal{R}(s)$$

reduction

$$(R{-}\mathsf{case}\ [x_1, \dots, x_n] d\ (c\ a_1\ a_2\ \cdots\ a_n)) \rightsquigarrow d[a_1, \dots, a_n]$$

The general elimination rule is not commonly used. Commonly the field selector functions are used. These extract the components of the records. However, this extraction is possible only if the sort of the field type is in $\mathcal{R}(s)$.

$$\begin{array}{ll}\Gamma, \Delta, r \colon (R\ \Delta) \vdash (f_1\ r) \colon A_1 & \text{if } s_1 \in \mathcal{R}(s) \\ \Gamma, \Delta, r \colon (R\ \Delta) \vdash (f_2\ r) \colon A_2[(f_1\ r)] & \text{if } s_2 \in \mathcal{R}(s) \\ \vdots & \\ \Gamma, \Delta, r \colon (R\ \Delta) \vdash (f_n\ r) \colon A_n[(f_1\ r), \dots, (f_{n-1}\ r)] & \text{if } s_n \in \mathcal{R}(s)\end{array}$$

with the extra obvious restriction that if a field selector f_i is not defined and x_i occurs in $A_j[x_1, \dots, x_{j-1}]$ then also x_j is undefined. For the defined field selectors we have the reduction rule

$$(f_i\ (c\ a_1\ a_2\ \cdots\ a_n)) \rightsquigarrow a_i.$$

The field selectors can be derived from the elimination rules, but are usually treated as primitive. On the other hand, the elimination rule cannot be derived from the field selectors if some of them are undefined.

We sometimes use the more common notation $\langle f_1 = a_1, \dots, f_n = a_n \rangle$ for $(c\ a_1\ a_2\ \cdots\ a_n)$, and the *field selector notation* $r \cdot f_i$ for $(f_i\ r)$.

The notion of declaration may seem mathematically questionable: The rules change before and after the declaration. We could treat record types in a more mathematical way by using the notations $\mathsf{Record}(A_1, \dots, A_n)$ for the record type R, $\mathsf{constr}(A_1, \dots, A_n)$ for the constructor c, and $\mathsf{field}_i(A_1, \dots, A_n)$ for the field selector f_i. Then we could just say that the record type, its constructor, and its field selectors just exist, without the need for the user to declare anything. We prefer the version with declarations because it gives a smoother and more intuitive notation.

2.11 Inductive Types

Inductive types are types whose elements are constructed recursively, using previously defined ones. An inductive type is declared by a command of the form

$$\begin{array}{l}\mathsf{Inductive}\ I\ [x_1 : P_1; \ldots; x_h : P_h] : (y_1 : Q_1; \ldots; y_m : Q_m) s_I := \\ \quad c_1 : (z_{11} : R_{11}) \cdots (z_{1k_1} : R_{1k_1})(I\ t_{11}\ \cdots\ t_{1m}) \\ \qquad \vdots \\ \quad c_n : (z_{n1} : R_{n1}) \cdots (z_{nk_n} : R_{nk_n})(I\ t_{n1}\ \cdots\ t_{nm}) \\ \mathsf{end.}\end{array}$$

We will indicate with Δ the environment $x_1 : P_1, \ldots, x_h : P_h$ and we call x_1, $\ldots$, x_h *general parameters* of I. We will indicate with Θ the environment $y_1 : Q_1, \ldots, y_m : Q_m$ and we call y_1, $\ldots$, y_m *recursive parameters* of I. We will indicate with Ξ_i the environment $z_{i1} : R_{i1}, \cdots, z_{ik_i} : R_{ik_i}$ and we call z_{i1}, $\ldots$, z_{ik_i} *arguments of the i-th constructor* of I. We call c_i *the i-th constructor* of I. Before giving the formal definition, we look at some examples that clarify the role of each element of the definition.

Natural Numbers

The type of natural numbers is defined by

$$\begin{array}{l}\mathsf{Inductive}\ \mathbb{N}\ :\ \mathsf{Set} := \\ \quad \mathsf{0} : \mathbb{N} \\ \quad \mathsf{S} : \mathbb{N} \to \mathbb{N} \\ \mathsf{end.}\end{array}$$

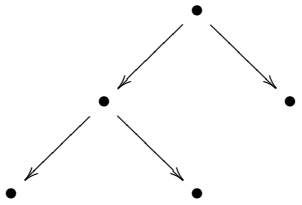
The definition of natural numbers has no general or recursive parameters. It has two constructors $\mathsf{0}$ and S. The constructor $\mathsf{0}$ has no argument, while S has one argument of type $\mathbb{N}$. This is a recursive argument, since its type is the inductive type that we are defining.

Binary Trees

Binary trees are trees constructed from leaves by nodes having two branches. Their type is defined as the inductive types

$$\begin{array}{l}\mathsf{Inductive}\ \mathbb{T}_2\ :\ \mathsf{Set} := \\ \quad \mathsf{leaf} : \mathbb{T}_2 \\ \quad \mathsf{node} : \mathbb{T}_2 \to \mathbb{T}_2 \to \mathbb{T}_2 \\ \mathsf{end.}\end{array}$$

For example, the binary tree depicted as



is represented by the term

$$(\mathsf{node}\ (\mathsf{node}\ \mathsf{leaf}\ \mathsf{leaf})\ \mathsf{leaf}) \colon \mathbb{T}_2.$$

Trees

Let us assume that we have a family of types $\mathbb{N}\_ \colon \mathbb{N} \to s$, such that $\mathbb{N}_n$ is a finite type with n elements:

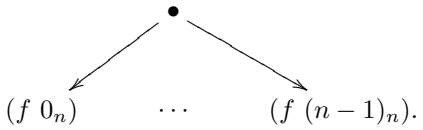
$$\mathbb{N}_n = \{0_n, 1_n, \ldots, (n-1)_n\}.$$

Then we can define the type of trees of finite branching degree by

$$\begin{array}{l}\mathsf{Inductive\ Tree}\ :\ s :=\\ \quad \mathsf{tree} \colon (n \colon \mathbb{N})(\mathbb{N}_n \to \mathsf{Tree}) \to \mathsf{Tree}\\ \mathsf{end}.\end{array}$$

Notice that, in the type of the second argument of the constructor tree, $\mathbb{N}_n \to \mathsf{Tree}$, the defined type Tree occurs only as the result type of a product type or, in other words, occurs at the right of the arrow. This is a requirement for every inductive type. We will formalize it precisely later.

The term $(\mathsf{tree}\ n\ f)$ is intuitively understood as a tree whose root node has n branches; the subtree attached to the i-th branch is $(f\ i)$:



Ordinals of the second number class

The type of countable ordinals is defined by

$$\begin{array}{l}\mathsf{Inductive\ Ord}\ :\ s :=\\ \quad \mathsf{0} \colon \mathsf{Ord}\\ \quad \mathsf{S} \colon \mathsf{Ord} \to \mathsf{Ord}\\ \quad \mathsf{lim} \colon (\mathbb{N} \to \mathsf{Ord}) \to \mathsf{Ord}\\ \mathsf{end}.\end{array}$$

This definition comes from [CP90]. It is similar to the definition of $\mathbb{N}$, but with an extra constructor lim, that takes an argument of type $\mathbb{N} \to \mathsf{Ord}$. The constructors 0 and S allow the construction of finite ordinals, that is, natural numbers. The constructor lim allows the construction of the limit ordinal of a countable family of ordinals. For example, the first infinite ordinal ω can be obtained by first defining the sequence of all finite ordinals

$$\begin{array}{l}\mathsf{finord}\colon \mathbb{N} \to \mathsf{Ord}\\ \mathsf{finord}(0) := 0\\ \mathsf{finord}((\mathsf{S}\ n)) := (\mathsf{S}\ \mathsf{finord}(n)\end{array}$$

(where 0 and S on the left side refer to the constructors of $\mathbb{N}$, and to the right side to the constructors of Ord) and then defining the first infinite ordinal as their limit

$$\omega := (\mathsf{lim}\ \mathsf{finord}).$$

Aczel's model for set theory

We can generalize further the example of the second number class by allowing any type to appear in place of $\mathbb{N}$ in the type $\mathbb{N} \to \mathsf{Ord}$ of the argument of the third constructor. In this case the inductive type must belong to an higher universe than the universe of the types that we admit in this argument type.

$$\begin{array}{l}\mathsf{Inductive\ V}\ :\ \mathsf{Type}_{i+1} :=\\ \quad \mathsf{sup}\colon (A\colon \mathsf{Type}_i)(A \to \mathsf{V}) \to \mathsf{V}\\ \mathsf{end}\end{array}$$

The intuitive idea behind this definition is that the term $(\mathsf{sup}\ A\ f)$ represents the set $\{(f\ a) \mid a\colon A\}$. There is no need for a base case in the definition, since the empty set can be defined as $\emptyset := (\mathsf{sup}\ \ [e](\mathsf{Empty\text{-}case}\ e))$. This definition was given by Peter Aczel, who studied its relation with constructive axiomatic set theory in [Acz78, Acz82, Acz86].

Lists

Given any type A, we define the type of lists of elements of A by

$$\begin{array}{l}\mathsf{Inductive\ List}\ [A\colon s]\colon s :=\\ \quad \mathsf{nil}\colon (\mathsf{List}\ A)\\ \quad \mathsf{cons}\colon A \to (\mathsf{List}\ A) \to (\mathsf{List}\ A)\\ \mathsf{end.}\end{array}$$

We have one general parameter A. A general parameter is a value that is fixed during the inductive definition. That is, we choose the value of A and then we give constructors for $(\mathsf{List}\ A)$ that can have recursive arguments only

from (List A) and not from a different instantiation (List A') of the inductive definition.

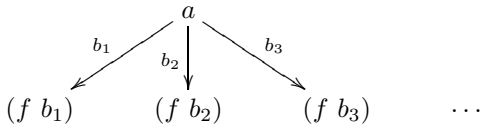
We use the notation $[a : l]$ for (cons a l), if $a\colon A$ and $l\colon$ (List A), and $[a_1, a_2, \ldots, a_n]$ for (cons a_1 (cons a_2 $\ldots$ (cons a_n nil))), if $a_1, \ldots, a_n\colon A$.

Well-orderings

We can generalize the construction of trees to that of a type of trees that has restrictions on the kind of nodes and on the branching degree of the nodes. If $A\colon s$ is a type, understood as the type of node labels; and $B\colon A \to s$ is a family of types, such that $(B\ a)$ is understood as the type of branch labels of a node of kind a, we define the corresponding type of trees by

$$\begin{array}{l}\mathsf{Inductive}\ \mathbb{W}\ [A\colon s; B\colon A \to s]\colon s :=\\ \quad \mathsf{wtree}\colon (a\colon A)((B\ a) \to (\mathbb{W}\ A\ B)) \to (\mathbb{W}\ A\ B)\\ \mathsf{end}.\end{array}$$

If $a\colon A$ and $f\colon (B\ a) \to (\mathbb{W}\ A\ B)$ is a family of trees indexed on $(B\ a)$, and we assume that $b_1, b_2, b_3, \ldots$ are the elements of $(B\ a)$; then the tree represented by the term (wtree a f) is



Vectors

If we want to have a type of lists of fixed length, that is, vectors, we must modify the definition of List to obtain

$$\begin{array}{l}\mathsf{Inductive\ Vector}\ [A\colon s]\colon \mathbb{N} \to s :=\\ \quad \mathsf{vnil}\colon (\mathsf{Vector}\ 0)\\ \quad \mathsf{vcons}\colon (n\colon \mathbb{N})A \to (\mathsf{Vector}\ A\ n) \to (\mathsf{Vector}\ A\ (\mathsf{S}\ n))\\ \mathsf{end}.\end{array}$$

Now we are not defining a single inductive type, but a family of types indexed on the natural numbers. Besides the general parameter A, we have a recursive parameter of type $\mathbb{N}$. That means that one element of the family (Vector A n), has the parameter $A\colon s$, that does not change in the course of the inductive definition, and the recursive parameter n, that may change during the inductive definition. In fact, the constructor vcons takes an argument of type (Vector A n) and produces an element of type (Vector A (S n)), thus changing the value of the recursive parameter.

We use the notation $\langle a : l\rangle$ for (vcons n a v), if $a\colon A$ and $v\colon$ (Vector A n), and $\langle a_1, a_2, \ldots, a_n\rangle$ for (vcons $n-1$ a_1 (vcons $n-2$ a_2 $\ldots$ (vcons 0 a_n vnil))), if $a_1, \ldots, a_n\colon A$.

General Trees

The definition of well-orderings has been further generalized by Kent Petersson and Dan Synek in [PS89] to the notion of general tree types (see also Chapter 16 of [NPS90]). We define a whole family of types of trees, indexed on a type A. For each index $a\colon A$ we have a type of labels for the nodes of the trees belonging to the corresponding tree type, $(B\ a)$. For every node label $b\colon (B\ a)$, we must give a type $(C\ a\ b)$ of labels for the branches stemming from a node labeled by b. Finally, for every branch with label $c\colon (C\ a\ b)$, we must specify to which member of the $\mathbb{T}$ family the subtree attached to the branch belongs; we do this by giving its index $(d\ a\ b\ c)\colon A$.

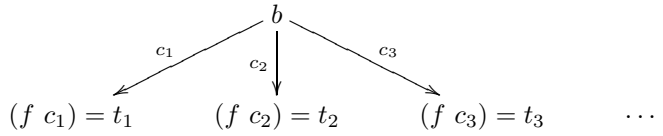
$$\begin{array}{l}\mathsf{Inductive}\ \mathbb{T}\ \left[\begin{array}{l}A\colon s;\\ B\colon A\to s;\\ C\colon (x\colon A)(B\ x)\to s;\\ d\colon (x\colon A; y\colon (B\ x))(C\ x\ y)\to A\end{array}\right]\colon A\to s :=\\ \quad \mathsf{gtree}\colon\ (x\colon A; y\colon (B\ x))\\ \qquad\qquad ((z\colon (C\ x\ y))(\mathbb{T}\ A\ B\ C\ d\ (d\ x\ y\ z)))\to(\mathbb{T}\ A\ B\ C\ d\ x)\\ \mathsf{end}\end{array}$$

Here A, B, C, and d are general parameters, while $x\colon A$ is a recursive parameter.

Let $a_1, a_2, a_3\colon A$ be indexes for types of trees; let also $t_1\colon (\mathbb{T}\ A\ B\ C\ d\ a_1)$, $t_2\colon (\mathbb{T}\ A\ B\ C\ d\ a_2)$, and $t_3\colon (\mathbb{T}\ A\ B\ C\ d\ a_3)$ be trees in the corresponding types. Let $a\colon A$. We are going to construct a tree of type $(\mathbb{T}\ A\ B\ C\ d\ a)$ that has t_1, t_2, and t_3 as subtrees. To this end, let us assume that $b\colon (B\ a)$ and $(C\ a\ b) = \{c_1, c_2, c_3, \ldots\}$, such that $(d\ a\ b\ c_1) = a_1$, $(d\ a\ b\ c_2) = a_2$, $(d\ a\ b\ c_3) = a_3$. First of all we construct the function that gives the subtrees:

$$\begin{array}{l} f\colon (z\colon (C\ a\ b))(\mathbb{T}\ A\ B\ C\ d\ (d\ a\ b\ z))\\ f(c_1) := t_1\\ f(c_2) := t_2\\ f(c_3) := t_3.\end{array}$$

Then the term $(\mathsf{gtree}\ a\ b\ f)\colon (\mathbb{T}\ A\ B\ C\ d\ a)$ represents the tree



The relationship between $\mathbb{W}$ and $\mathbb{T}$ types are similar to that between List and Vector: while in $\mathbb{W}$ all trees belong to the same type, in $\mathbb{T}$ there is a family of types and the constructor takes trees from some members of the family and gives a tree in another member.

Definition of inductive type

We observed, talking about the examples, that the type that we are defining can appear in the types of the constructor arguments only as the result type or,

in other words, only to the right of arrows. This condition is formalized by the notion of strictly positive type.

Definition 2.11.1 *Let Γ be a valid context and $\Theta = y_1 \colon Q_1, \ldots, y_m \colon Q_m$ be a valid extension of Γ. Let I be a family of types indexed on the context Θ, $\Gamma \vdash I \colon \Theta \to s_I$. We want to define what it means for I to occur strictly positively in a type R. This happens when R is of the form specified as follows. Let $\Upsilon = v_1 \colon A_1, \ldots, v_l \colon A_l$ be a valid extension of Γ in which Θ can be instantiated: $\Gamma, \Upsilon \vdash \bar{q} :: \Theta$. We recall that this means that*

$$\begin{aligned}
&\Gamma, \Upsilon \vdash q_1 \colon Q_1 \\
&\Gamma, \Upsilon \vdash q_2 \colon Q_2[y_1 := q_1] \\
&\qquad \vdots \\
&\Gamma, \Upsilon \vdash q_m \colon Q_m[y_1 := q_1, \ldots, y_{m-1} := q_{m-1}].
\end{aligned}$$

Then we say that I is strictly positive *in the type $\Pi\Upsilon.(I\ \bar{q})$. Let us write, for short, R for $\Pi\Upsilon.(I\ \bar{q})$.*

Given a type operator on the family I,

$$\Gamma, \Theta, w \colon (I\ \bar{y}) \vdash T[\bar{y}, w] \colon s,$$

we can extend it to the strictly positive type R by the definition

$$\Gamma, f \colon R \vdash T_{[R,I]}[f] \colon s \qquad \textit{with} \qquad T_{[R,I]}[f] := \Pi\Upsilon.T[\bar{q}, (f\ \bar{v})]$$

or, more explicitly,

$$T_{[R,I]}[f] := (v_1 \colon A_1; \ldots; v_l \colon A_l)(T\ q_1\ \cdots\ q_m\ (f\ v_1\ \cdots\ v_k)).$$

Similarly, if we have a term

$$\Gamma, \Theta, w \colon (I\ \bar{y}) \vdash z[\bar{y}, w] \colon T[\bar{y}, w],$$

we can extend it to

$$\Gamma, f \colon R \vdash z_{[R,I]}[f] \colon T_{[R,I]}[f],$$

defined as

$$z_{[R,I]}[f] := [v_1 \colon A_1; \ldots; v_l \colon A_l] z[\bar{q}, (f\ \bar{v})].$$

An inductive type must occur strictly positively in the types of the arguments of its constructors.

Definition 2.11.2 *Let Γ be a valid context. The definition of an inductive type in the context Γ requires the following components.*

A valid extension Δ of the context Γ: $\Delta = x_1 \colon P_1, \ldots, x_h \colon P_h$, whose components we call the general parameters *of the inductive definition.*

A valid extension Θ of the context Γ, Δ: $\Theta = y_1 \colon Q_1, \ldots, y_m \colon Q_m$, whose components we call the recursive parameters *of the inductive definition.*

A sequence of valid extensions Ξ_i *of the context* $\Gamma, \Delta, I\colon \Theta \to s_I$*:* $\Xi_i = u_{i1}\colon R_{i1}, \ldots, u_{ik_i}\colon R_{k_i}$ *for* $i = 1, \ldots n$*.*

An instantiation $t_{i1}, \ldots, tim$ *of* Θ *for each* Ξ_i*:* $\Gamma, \Delta, I\colon \Theta \to s_I \vdash \bar{t_i} :: \Theta$*, such that* I *does not occur free in any of the* $t_{i,j}$*.*

Given these components, we can define an inductive family of types I *by the declaration*

$$\begin{array}{l}\mathsf{Inductive}\ I\ [\Delta]\colon \Theta \to s_I :=\\ \quad c_1\colon (z_1\colon R_{11}) \cdots (z_{k_1}\colon R_{1k_1})(I\ t_{11}\ \cdots\ t_{1m})\\ \qquad \vdots\\ \quad c_n\colon (z_1\colon R_{n1}) \cdots (z_{k_n}\colon R_{nk_n})(I\ t_{n1}\ \cdots\ t_{nm})\\ \mathsf{end}.\end{array}$$

This declaration introduces the new family of types I *with the rules*

formation

$$\frac{\Gamma \vdash \bar{p} :: \Delta \quad \Gamma \vdash \bar{q} :: \Theta[\bar{x} := \bar{p}]}{\Gamma \vdash (I\ \bar{p}\ \bar{q})\colon s_I}$$

introduction

$$\frac{\Gamma \vdash \bar{p} :: \Delta \quad \Gamma \vdash \bar{r_i} :: \Xi_i[\bar{x} := \bar{p}]}{\Gamma \vdash (c_i\ \bar{p}\ \bar{r_i})\colon (I\ \bar{p}\ t_{i1}[\bar{x} := \bar{p}, \bar{u_i} := \bar{r_i}]\ \cdots\ t_{ik_i}[\bar{x} := \bar{p}, \bar{u_i} := \bar{r_i}])}\ 1 \le i \le n$$

elimination

$$\frac{\begin{array}{l}\Gamma \vdash \bar{p} :: \Delta\\ \Gamma, \Theta[\bar{x} := \bar{p}], v\colon (I\ \bar{p}\ \bar{y}) \vdash T[\bar{y}, v]\colon s_T\\ \Gamma,\ u_{11}\colon R_{11}, v_{11} : T_{[R_{11},I]}[u_{11}], \ldots,\\ \quad u_{1k_1}\colon R_{1k_1}, v_{1k_1} : T_{[R_{1k_1},I]}[u_{1k_1}] \vdash a_1\colon (T\ \bar{p}\ \bar{t_1})\\ \vdots\\ \Gamma,\ u_{i1}\colon R_{i1}, v_{i1} : T_{[R_{i1},I]}[u_{i1}], \ldots,\\ \quad u_{ik_i}\colon R_{ik_i}, v_{ik_i} : T_{[R_{ik_i},I]}[u_{ik_i}] \vdash a_i\colon (T\ \bar{p}\ \bar{t_i})\end{array}}{\Gamma, \Theta[\bar{x} := \bar{p}], w\colon (I\ \bar{p}\ \bar{y}) \vdash (\mu\text{-}\mathsf{ind}\ \bar{p}\ \bar{a}\ \bar{y}\ w)\colon T[\bar{y}, w]} \quad s_T \in \mathcal{R}(s_I)$$

where we mean that the variable v_{ij} *is present only if* I *occurs (in a strictly positive position) in the argument type* R_{ij}*. In that case* v_{ij} *is the induction hypothesis, or the recursive call if we are defining a recursive function.*

reduction

$$\begin{array}{l}(\mu\text{-}\mathsf{ind}\ \bar{p}\ \bar{a}\ \bar{t_i}[\bar{x} := \bar{p}, \bar{u_i} := \bar{r_i}]\ (c_i\ \bar{p}\ \bar{r_i})) \rightsquigarrow\\ \quad a_i[\ u_{i1} := r_{i1}, v_{i1} := (\mu\text{-}\mathsf{ind}\ \bar{p}\ \bar{a}\ w)_{[R_{i1},I]}[u_{i1}], \ldots,\\ \qquad u_{ik_i} := r_{ik_i}, v_{ik_i} := (\mu\text{-}\mathsf{ind}\ \bar{p}\ \bar{a}\ w)_{[R_{ik_i},I]}[u_{ik_i}]\ ]\end{array}$$

Inductive types as fixed points of strictly positive operators

A simplification of the notion of inductive types can be obtained if we consider only nondependent types.

If the types of the constructors do not use dependent products, that is, they are in the form $R_{i1} \to \cdots \to R_{ik_i} \to I$, and the constructor argument types R_{ij} similarly do not use dependent products, we can use the alternative formulation of inductive types as fixed points of strictly positive type operators (see, for example, [Dyb97]). It is less intuitive but simpler for theoretical purposes, so we adopt it. Suppose we have such an inductive definition

$$\begin{array}{l}\mathsf{Inductive}\ I : s_I := \\ \quad c_1 \colon R_{11} \to \cdots \to R_{1k_1} \to I \\ \qquad \vdots \\ \quad c_n \colon R_{n1} \to \cdots \to R_{nk_n} \to I \\ \mathsf{end}.\end{array}$$

We can associate to it the type operator

$$X \colon s \vdash F[X] := ((R_{11} \times \cdots \times R_{1k_1}) + \cdots + (R_{n1} \times \cdots \times R_{nk_n}))[I := X].$$

The inductive type I is the least fixed point of the operator F.

The restriction on the form of the argument types becomes now a condition that the type operator must satisfy: strict positivity. An operator $X \colon s \vdash F[X] \colon s$ is strictly positive if X occurs in $F[X]$ only to the right of arrows.

Definition 2.11.3 *A* strictly positive type operator *is an operator $X \colon s \vdash F[X] \colon s$ obtained by a finite application of the following constructions:*

- *If X does not occur free in $F[X]$, that is, if $F[X] = K$ for some constant type K, then F is strictly positive;*
- *The operator $X \colon s \vdash X$ is strictly positive;*
- *If $X \colon s \vdash F_1[X] \colon s$ and $X \colon s \vdash F_2[X] \colon s$ are strictly positive operators, then $X \colon s \vdash F_1[X] \times F_2[X] \colon s$ is a strictly positive operator;*
- *If $X \colon s \vdash F_1[X] \colon s$ and $X \colon s \vdash F_2[X] \colon s$ are strictly positive operators, then $X \colon s \vdash F_1[X] + F_2[X] \colon s$ is a strictly positive operator;*
- *If $X \colon s \vdash F[X] \colon s$ is a strictly positive operator and K is a constant type (not depending on X), then $X \colon s \vdash K \to F[X] \colon s$ is a strictly positive operator.*

Every strictly positive operator $X \colon s \vdash F[X] \colon s$ has a functorial extension: if $X, Y \colon s$ and $f \colon X \to Y$, then there exists $\Phi_F[f] \colon F[X] \to F[Y]$ preserving identity and composition (see [CP90] and [PR99]). To simplify notation and comply with categorical conventions, we write $F[f]$ for $\Phi_F[f]$. This existence of a functorial extension is sufficient to formulate the rules for inductive types

(Matthes [Mat99] gives an extension of *system F* in which this is the only condition required for inductive types). In Chapter 4.2, we consider extensions of the positivity condition that still have the functorial property. The rules for inductive types are then the same as in the following definition, with the corresponding property replacing *strictly positive*.

Definition 2.11.4 *Let* $X\colon s \vdash F[X]\colon s$ *be a strictly positive operator. The* inductive type $\mu_X(F)$ *is defined by the following rules (where we write* I *for* $\mu_X(F)$*):*

formation

$$I\colon s$$

introduction

$$\frac{y\colon F[I]}{(\mu\text{-intro } y)\colon I}$$

elimination

$$\frac{x\colon I \vdash (P\ x)\colon s' \quad z\colon F[(\Sigma\ I\ P)] \vdash u[z]\colon (P\ (\mu\text{-intro } (F[\pi_1]\ z)))}{x\colon I \vdash (\mu\text{-ind } [z]u\ x)\colon (P\ x)} \quad s' \in \mathcal{R}(s)$$

reduction

$$(\mu\text{-ind } [z]u\ (\mu\text{-intro } y)) \rightsquigarrow u[(F[[x]\langle x, (\mu\text{-ind } [z]u\ x)\rangle]\ y)]$$

As a special case of the elimination rule when the elimination predicate P is a constant type T, we get the recursion rule, and, when the recursion term u does not depend on the induction variable, the iteration rule:

$$\frac{B\colon s' \quad z\colon F[I \times B] \vdash u[z]\colon B}{x\colon I \vdash (\mu\text{-rec } [z]u\ x)\colon B} \qquad \frac{B\colon s' \quad y\colon F[B] \vdash t[y]\colon B}{x\colon I \vdash (\mu\text{-it } [y]t\ x)\colon B}$$

$$(\mu\text{-rec } [z]u\ (\mu\text{-intro } y)) \rightsquigarrow u[(F[[x]\langle x, (\mu\text{-rec } [z]u\ x)\rangle]\ y)] \qquad (\mu\text{-it } [y]t\ (\mu\text{-intro } y)) \rightsquigarrow t[(F[(\mu\text{-it } [y]t)]\ y)]$$

by defining $(\mu\text{-rec } [z]u\ x) := (\mu\text{-ind } [z]u\ x)$, with $(P\ x) := T$, and $(\mu\text{-it } [y]t\ x) := (\mu\text{-rec } [z]t[y := (F[\pi_2]\ z)]\ x)$.

It is a well known fact that the recursion and iteration principles are equivalent, while the full induction principle is a proper extension of them (see, for example, [Geu92] or [PR99]).

The types of natural numbers, binary trees and lists over a type A can be defined as $\mathbb{N} := \mu_X(\mathbb{N}_1 + X)$, $\mathbb{T}_2 := \mu_X(\mathbb{N}_1 + X \times X)$ and $\mathsf{List}(A) := \mathbb{N}_1 + A \times X$, respectively. Their constructors can be defined in terms of the single constructor μ-intro:

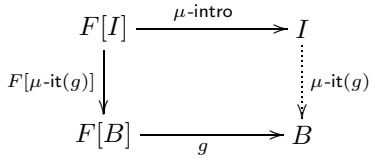
$$\begin{array}{rclrcl} 0 & := & (\mu\text{-intro } (\text{inl } 0_1)) & \mathsf{S} & := & [n](\mu\text{-intro } (\text{inr } n)) \\ \mathsf{leaf} & := & (\mu\text{-intro } (\text{inl } 0_1)) & \mathsf{node} & := & [x_1, x_2](\mu\text{-intro } (\text{inr } \langle x_1, x_2\rangle)) \\ \mathsf{nil} & := & (\mu\text{-intro } (\text{inl } 0_1)) & \mathsf{cons} & := & [a, l](\mu\text{-intro } (\text{inr } \langle a, l\rangle)) \end{array}$$

The rules for inductive types can be synthetically and intuitively expressed in categorical terms by saying that I is an initial F-algebra.

Definition 2.11.5 *Let $\mathcal{C}$ be a category and $F\colon \mathcal{C} \to \mathcal{C}$ an endofunctor on it. An* F-algebra *is a pair $\langle A, f\rangle$, with A an object of $\mathcal{C}$ and $f\colon F[A] \to A$ a morphism in $\mathcal{C}$.*

In particular, every type universe can be considered as a category, thus we consider endofunctors on the category (sort) s: An operator $X\colon s \vdash F[X]\colon s$, with an extension to functions, $X, Y\colon s; f\colon X \to Y \vdash F[f]\colon F[X] \to F[Y]$.

Definition 2.11.6 *An* initial F-algebra *is the initial object in the category of F-algebras, that is, an F-algebra $\langle I, \mu\text{-intro}\rangle$ such that, for every other F-algebra $\langle B, g\rangle$, there exists a unique morphism $\mu\text{-it}(g)\colon I \to B$ that makes the following diagram commute:*

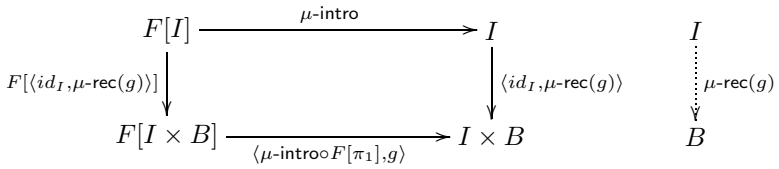


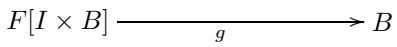
So $\langle I, \mu\text{-intro}\rangle$ is an initial F-algebra if and only if

$$\forall B\colon s.\forall g\colon F[B] \to B.\exists!\mu\text{-it}(g)\colon I \to B.\mu\text{-it}(g) \circ \mu\text{-intro} = g \circ F[\mu\text{-it}(g))$$

This categorical characterization corresponds to the iteration principle for inductive types.

We can also characterize the recursion principle by the categorical diagram



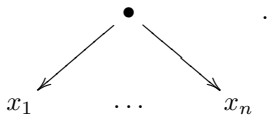


corresponding to the property

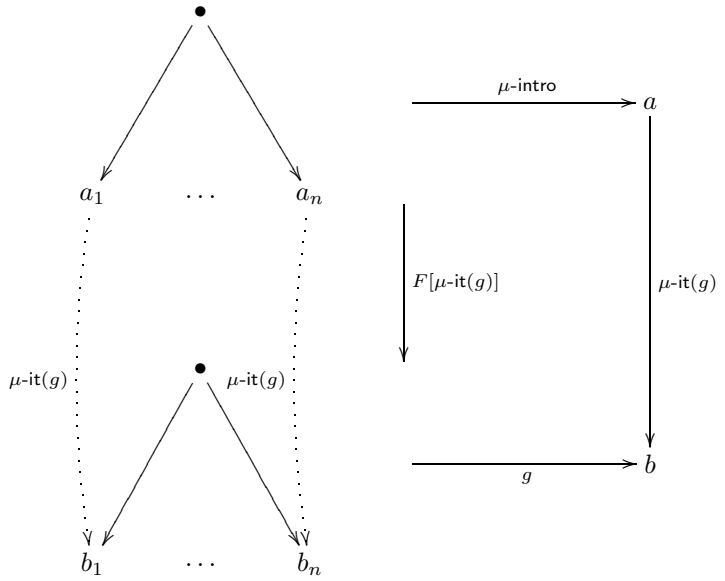
$$\begin{aligned}&\forall B\colon s.\forall g\colon F[I \times B] \to B.\exists!\mu\text{-rec}(g)\colon I \to B.\\ &\quad \mu\text{-rec}(g) \circ \mu\text{-intro} = g \circ F[\langle id_B, \mu\text{-rec}(g)\rangle].\end{aligned}$$

As the principles of iteration and recursion are equivalent, so are the two categorical characterizations.

We give a graphical representation of the elements of an inductive type. Let us represent, for a given type X, the elements of $F[X]$ as trees having elements of X as their leaves:

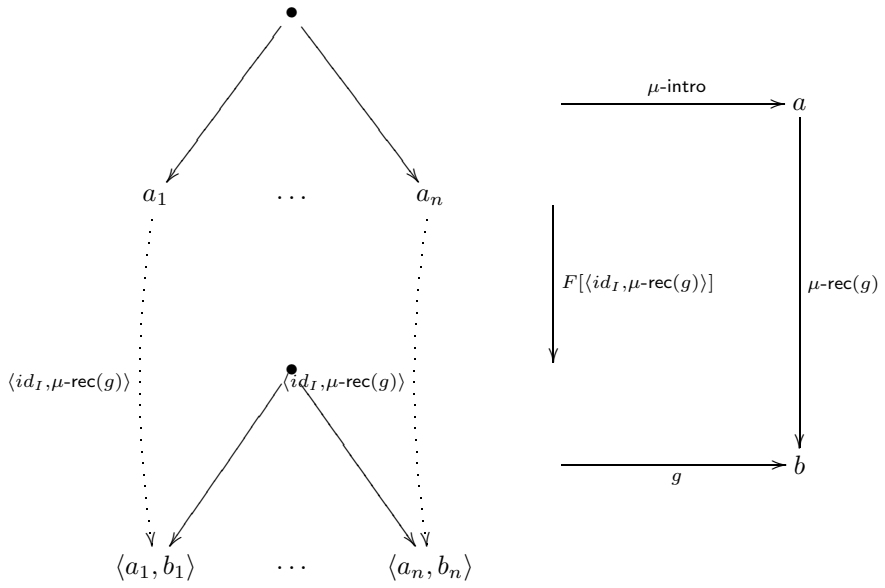


Then the rule of iteration can be depicted as follows:



Every element $a\colon I$ must be of the form (μ-intro y) for some $y\colon F[I]$. Recursively applying μ-it(g) to the leaves, that is, applying $F[\mu\text{-it}(g)]$ to y, we obtain an element of $F[B]$, to which we apply g to obtain the desired result b of B.

A similar depiction illustrates the rule of recursion. Here we keep the original leaves in the recursive call.



2.12 The representation of logic

The sort Prop is intended to be used as the universe of logical propositions. According to the Curry-Howard isomorphism we can interpret every proposition as a type whose elements are its proofs (see [How80], [ML82], and [SU98]). We get the following table of correspondence between logical operators and type constructors:

$\top$	Unit
$\bot$	Empty
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$A \Rightarrow B$	$A \to B$
$\neg A$	$A \to \mathsf{Empty}$
$\forall x\colon A.P(x)$	$\Pi x\colon A.(P\ x)$
$\exists x\colon A.P(x)$	$\Sigma x\colon A.(P\ x)$

It is easily verified that the laws of intuitionist first order logic are satisfied by this interpretation. For a detailed study of the relationship between logic and type systems see [Geu93, SU98].

Besides first order logic, we need primitive predicates and relations on the types. The most important of all is the equality relation. For every type A, we want to have a relation $(=)\colon A \to A \to \mathsf{Prop}$, such that $a_1 = a_2$ is inhabited if a_1 and a_2 denote the same object.

There are two standard ways do define equality types.

The first is called *Leibniz equality* and is based on the principle of *identity of the indistinguishables*, formulate by Gottfried Wilhelm Leibniz in the book *Monadology*. According to this principle two entities that do not have any properties allowing to distinguish them should be seen as a single entity. In our language we may formulate it as: a_1 and a_2 are equal if every predicate satisfied by a_1 is also satisfied by a_2. It can be formalized by an impredicative relation:

$$(a_1 = a_2) := \forall P\colon A \to \mathsf{Prop}.(P\ a_1) \to (P\ a_2).$$

We can define equality in general as the term

$$\begin{array}{ll}(=) & : \Pi_{X\colon s} X \to X \to \mathsf{Prop} \\ & := [X\colon s][x_1, x_2\colon X]\forall P\colon X \to \mathsf{Prop}.(P\ x_1) \to (P\ x_2).\end{array}$$

Alternatively, we can define equality as the relation that every element has only with itself. This is achieved by the relation with the following rules.

formation

$$\frac{\Gamma \vdash A\colon s \quad \Gamma \vdash a_1\colon A \quad \Gamma \vdash a_2\colon A}{\Gamma \vdash a_1 = a_2\colon \mathsf{Prop}}$$

introduction

$$\frac{\Gamma \vdash a\colon A}{\Gamma \vdash (\mathsf{refl}\ a)\colon (a = a)}$$

elimination

$$\frac{\Gamma, x\colon A, y\colon A, h\colon (x = y) \vdash T[x, y, h]\colon s \quad \Gamma, x\colon a \vdash t[x]\colon T[x, x, (\mathsf{refl}\ x)]}{\Gamma, x\colon A, y\colon A, h\colon (x = y) \vdash (\mathsf{eq\text{-}elim}\ [x]t\ x\ y\ z)\colon T[x, y, z]}$$

reduction

$$(\mathsf{eq\text{-}elim}\ [x]t\ a\ a\ (\mathsf{refl}\ a)) \rightsquigarrow t[a]$$

Notice that Leibniz principle can be derived from the elimination rule: Let $P\colon A \to \mathsf{Prop}$ and put $T[x, t, h] := (P\ x) \to (P\ y)$. Then the premise of the elimination rule is automatically satisfied:

$$\Gamma, x\colon A \vdash [p\colon (P\ x)]p\colon (P\ x) \to (P\ x).$$

So we obtain

$$\Gamma, x\colon A, y\colon A, h\colon (x = y) \vdash (\mathsf{eq\text{-}elim}\ [x][p]p\ x\ y\ z)\colon (P\ x) \to (P\ y).$$

This second definition of equality is the one that we adopt in this thesis.

Chapter 3

Coinductive Types

> By these remarks I wanted to show only that the definitive clarification of the *nature of the infinite* has become necessary, not merely for the special interests of the individual sciences, but rather for the *honor of the human understanding* itself.
>
> Hilbert [Hil26], p. 371

Coinductive types are the categorical dual of inductive types. Their intuitive explanation is that they are types of possibly infinite elements. For example, they are used to represent streams, that is, infinite lists, and infinite trees.

They are the type-theoretic version of *non-well-founded sets*, sets in which there may be infinite chains in the membership relation. Peter Aczel introduced and studied the axiomatic theory of non-well-founded sets in [Acz88]. There are actually several earlier works on the notion of non-well-founded set, discussed in Appendix A of [Acz88]; specifically, Aczel traces the notion back to Miramanoff and gives Forti and Honsell's [FH83] the merit of formulating the anti-foundation axiom for the first time. A full introduction of the theory and its application can be found in [BM96].

The expression of coinductive types in the polymorphic λ-calculus has been studied by Wraith [Wra89], Geuvers [Geu92], and Paulson [Pau97]. Coquand [Coq93] formalized coinductive types in type theory using the notion of guarded recursion. Giménez [Gim94] proved that definitions by guarded recursion are equivalent to corecursion. Coinductive types have been formalized in the proof assistants Isabelle [Pau94] and Coq (see [LPM93], [Gim94], [Gim95], and [Gim98]).

We begin with a categorical presentation of coalgebras, to convey an intuitive understanding of them. (Coinductive methods in computer science are the subject of annual conference. For an overview of current research see the proceedings [JMRR01].) Then we give a formal definition of coinductive types.

3.1 Coalgebras

From the categorical point of view, coalgebras are the dual notion of algebras.

Definition 3.1.1 *Let $\mathcal{C}$ be a category and $F\colon \mathcal{C} \to \mathcal{C}$ an endofunctor on it. An F-*coalgebra *is a pair $\langle B, g\rangle$, with B an object of $\mathcal{C}$ and $g\colon B \to F[B]$ a morphism in $\mathcal{C}$.*

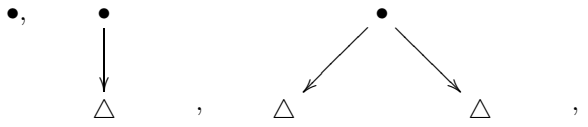
As an example we take the functor $F[X] := X^0 + X^1 + X^2$, where $X^0 := \mathsf{Unit}$, $X^1 := X$, and $X^2 := X \times X$. Let us compare the notions of F-algebra and F-coalgebra. According to Definition 2.11.5, an F-algebra is a pair $\langle A, f\rangle$ such that $f\colon F[A] \to A$. In the category of sets, this means that we have a set A and a function f from $F[A] = A^0 + A^1 + A^2$ to A. Such a function is equivalent to three components $f_0\colon A$, $f_1\colon A \to A$, and $f_2\colon A^2 \to A$. In fact we have that, given f, we can define $f_0 := (f\ (\mathsf{in}_0\ \bullet))$, $f_1 := \lambda y\colon A.(f\ (\mathsf{in}_1\ y))$, and $f_2 := \lambda z\colon A^2.(f\ (\mathsf{in}_2\ z))$, where in_i is the injection function from a set to the i-th component of a sum (disjoint union) of sets. Vice versa, if f_0, f_1, and f_2 are given, we can define

$$f := \lambda x\colon A^0 + A^1 + A^2.\mathsf{Cases}\ x\ \mathsf{of}\ \begin{cases} (\mathsf{in}_0\ x) \mapsto f_0 \\ (\mathsf{in}_1\ y) \mapsto (f_1\ y) \\ (\mathsf{in}_2\ z) \mapsto (f_2\ z). \end{cases}$$

We obtain the usual notion of an algebra as a set with some operations on it. The operations are functions that take some arguments from the algebra (and possibly from other sets) and produce a new element of the algebra. So an algebra is a set with methods to compute elements of the set. There are no methods to extract information from elements of the algebra: There are no functions from the algebra to other sets.

Coalgebras take the inverse approach. We have functions to extract information from the elements of the coalgebra, but not to generate new elements. An F-coalgebra is a pair $\langle B, g\rangle$ with $g\colon B \to F[B]$. The function g can be seen as a method extracting information from an element of B. If $b\colon B$, we can check $(g\ b)$ to see in which of the forms $(\mathsf{in}_0\ x)$, $(\mathsf{in}_1\ y)$, and $(\mathsf{in}_2\ z)$ it is. In computer science, coalgebras are useful to represent abstract data types. Abstract data types are types whose implementation is hidden from the user/programmer. Information about an element of an abstract data type can be obtained only through *methods*, that is, functions from the data type to other types (possibly the data type itself).

To give some intuitive understanding of the meaning of the elements of our example coalgebras, let us say that we want to define a data type whose elements can be thought of as trees of branching degree at most two. So we want to represent objects of the three forms



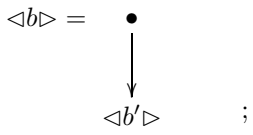
where the $\triangle$s represent other trees of the same general form.

An F-coalgebra $\langle B, g\rangle$ can be understood as a representation of a set of trees. We are not allowed to look into the representation, but we have a function g

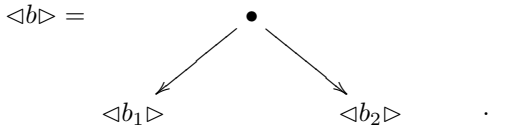
that gives us information on the structure of the tree. We use the notation $\lhd b \rhd_{\langle B,g\rangle}$ to denote the tree associated with the element b of the F-coalgebra $\langle B, g\rangle$. If the coalgebra is clear from the context, we omit it and write simply $\lhd b \rhd$. If $b\colon B$, we can compute $(g\ b)$ and deduce the shape of the tree by its value: if $(g\ b) = (\mathsf{in}_0\ \bullet)$, then the tree represented by b is simply a leaf,

$$\lhd b \rhd = \bullet;$$

if $(g\ b) = (\mathsf{in}_1\ b')$, then the tree has just one branch, at the end of which the subtree represented by b' is attached,



if $(g\ b) = (\mathsf{in}_2\ \langle b_1, b_2\rangle)$, then the tree has two branches, at the end of which the subtrees represented by b_1 and b_2 are attached,



In the last two cases, g not only gives the shape of the tree, but also the objects representing the subtrees, b' and b_1, b_2, respectively. The function g can be applied to them in turn to refine the information.

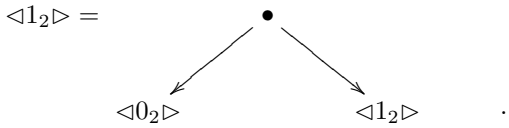
Let us give some concrete examples. First, let us choose the simple case of the type with two elements $\mathbb{N}_2 = \{0_2, 1_2\}$ and the function

$$\begin{aligned}&\mathsf{bg}\colon \mathbb{N}_2 \to \mathbb{N}_2^0 + \mathbb{N}_2^1 + \mathbb{N}_2^2\\ &\mathsf{bg}(0_2) := (\mathsf{in}_0\ \bullet)\\ &\mathsf{bg}(1_2) := (\mathsf{in}_2\ \langle 0_2, 1_2\rangle).\end{aligned}$$

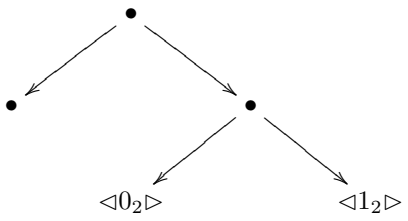
(bg stands for "a g on the booleans", since we can consider $\mathbb{N}_2$ as the type of booleans.) The pair $\langle \mathbb{N}_2, \mathsf{bg}\rangle$ is an F-coalgebra with two elements 0_2 and 1_2. We use the method bg to probe the structure of the associated trees. The tree associated with 0_2 is, since $\mathsf{bg}(0_2) = (\mathsf{in}_0\ \bullet)$, a simple leaf,

$$\lhd 0_2 \rhd = \bullet.$$

The tree associated with 1_2 is, since $\mathsf{bg}(1_2) = (\mathsf{in}_2\ \langle 0_2, 1_2\rangle)$, the binary tree with two subtrees represented by 0_2 and 1_2,

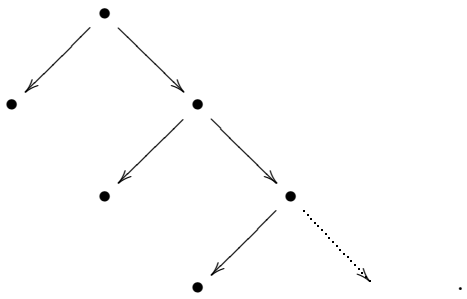


Now we can refine further our knowledge of the structure of the tree associated with 1_2 by probing the structures of the trees associated with 0_2 and 1_2. Since we already know the shape of the trees associated with them, we can immediately conclude that

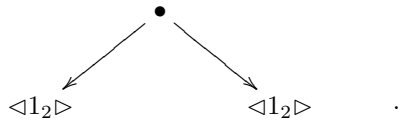
$\lhd 1_2 \rhd =$ 

.

It is then clear that a complete unfolding of the structure of the tree associated with 1_2 would give the infinite *comb*

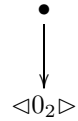
$\lhd 1_2 \rhd =$ 
.

This example shows that, already with very simple coalgebras, we obtain representations of infinite objects. Coalgebras are not in general closed under tree-constructing operations. For example, we may want to construct a binary tree having two copies of the tree associated with 1_2 as subtrees:



.

There is no element in the F-coalgebra $\langle \mathbb{N}_2, \mathsf{bg} \rangle$ representing this tree. Similarly, there is no element representing the unary tree



.

We call a coalgebra *closed under tree formation* if it contains every tree constructed by one step from trees already in it. This means that, first, it must contain an element representing a leaf; second, for every element x in it, it must

contain some other element y such that $(g\ y) = (\mathsf{in}_1\ x')$ for some x' such that $\lhd x' \rhd = \lhd x \rhd$; third, for every two elements x_1 and x_2 in it, it must contain some other element z such that $(g\ z) = (\mathsf{in}_2\ \langle x'_1, x'_2 \rangle)$ for some x'_1 and x'_2 such that $\lhd x'_1 \rhd = \lhd x_1 \rhd$ and $\lhd x'_2 \rhd = \lhd x_2 \rhd$.

Coalgebras closed under tree formation are also algebras, they are fixed points of the functor F. The least and largest fixed points of F are the inductive and coinductive types associated with F, respectively.

Let us look now at a more complex example. This time we choose the natural numbers as type of the elements of the coalgebra. The structure function is defined as

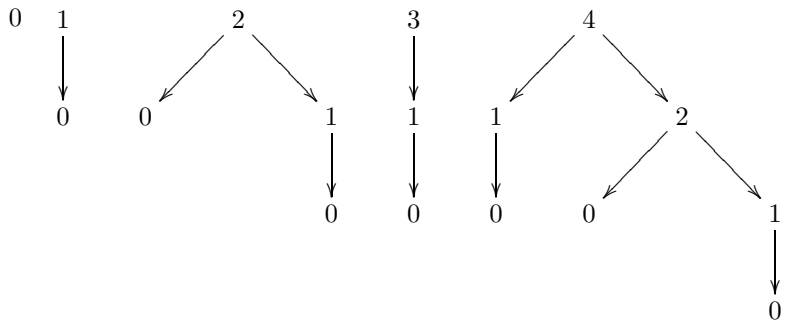
$$\begin{array}{l} \mathsf{ng}\colon \mathbb{N} \to \mathbb{N}^0 + \mathbb{N}^1 + \mathbb{N}^2 \\ \mathsf{ng}(0) := (\mathsf{in}_0\ \bullet) \\ \mathsf{ng}((\mathsf{S}\ n)) := \mathsf{Cases}\ \mathsf{ng}(n)\ \mathsf{of} \left\{ \begin{array}{ll} (\mathsf{in}_0\ x) & \mapsto (\mathsf{in}_1\ 0) \\ (\mathsf{in}_1\ m) & \mapsto (\mathsf{in}_2\ \langle m, (\mathsf{S}\ m) \rangle) \\ (\mathsf{in}_2\ \langle m_1, m_2 \rangle) & \mapsto (\mathsf{in}_1\ m_1 + m_2). \end{array} \right. \end{array}$$

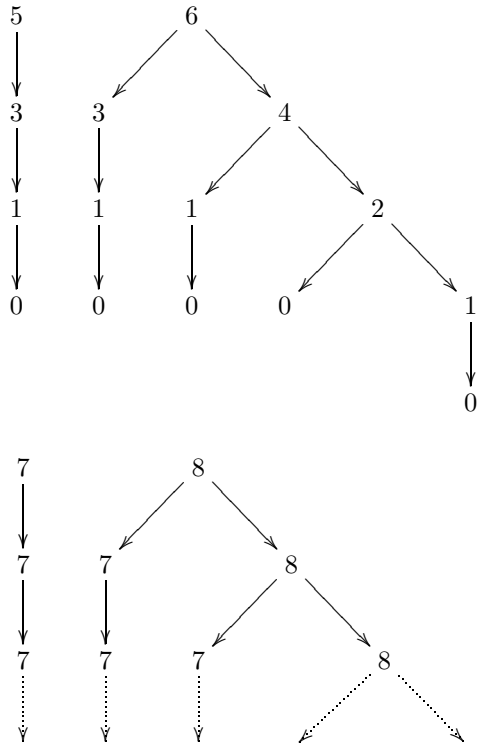
(ng stands for "a g defined on the natural numbers".)

Let us compute the first few values of ng:

$$\begin{array}{lll} \mathsf{ng}(0) = (\mathsf{in}_0\ \bullet) & \mathsf{ng}(6) = (\mathsf{in}_2\ \langle 3, 4 \rangle) & \mathsf{ng}(12) = (\mathsf{in}_2\ \langle 31, 32 \rangle) \\ \mathsf{ng}(1) = (\mathsf{in}_1\ 0) & \mathsf{ng}(7) = (\mathsf{in}_1\ 7) & \mathsf{ng}(13) = (\mathsf{in}_1\ 63) \\ \mathsf{ng}(2) = (\mathsf{in}_2\ \langle 0, 1 \rangle) & \mathsf{ng}(8) = (\mathsf{in}_2\ \langle 7, 8 \rangle) & \mathsf{ng}(14) = (\mathsf{in}_2\ \langle 63, 64 \rangle) \\ \mathsf{ng}(3) = (\mathsf{in}_1\ 1) & \mathsf{ng}(9) = (\mathsf{in}_1\ 15) & \mathsf{ng}(15) = (\mathsf{in}_1\ 127) \\ \mathsf{ng}(4) = (\mathsf{in}_2\ \langle 1, 2 \rangle) & \mathsf{ng}(10) = (\mathsf{in}_2\ \langle 15, 16 \rangle) & \mathsf{ng}(16) = (\mathsf{in}_2\ \langle 127, 128 \rangle) \\ \mathsf{ng}(5) = (\mathsf{in}_1\ 3) & \mathsf{ng}(11) = (\mathsf{in}_1\ 31) & \mathsf{ng}(17) = (\mathsf{in}_1\ 255). \end{array}$$

We can now associate a tree to every element of the coalgebra $\langle \mathbb{N}, \mathsf{ng} \rangle$. For clarity, in the following pictures we write the natural number corresponding to a tree in the main node.



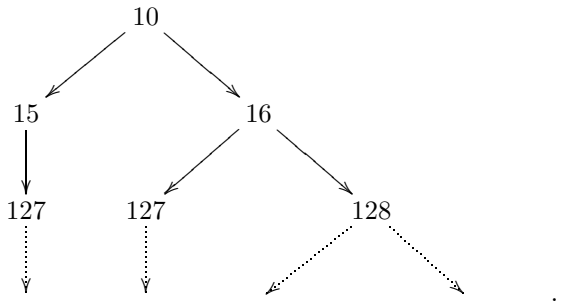


The first infinite tree is associated with the number 7; it is a unary tree having itself as subtree. The tree associated with 8 is a binary tree having the tree of 7 as left subtree and itself as right subtree. These two trees are *regular*, that is, they have a finite number of subtrees. Regular trees are called *hereditarily finite trees* in [Acz88], pg. 7. Specifically, the tree of 7 has only itself as subtree and the tree of 8 has the tree of 7 and itself as subtree. There may exist trees with an infinite number of distinct subtrees. We call such trees *irregular*. A candidate for an irregular tree is the tree associated with 9:

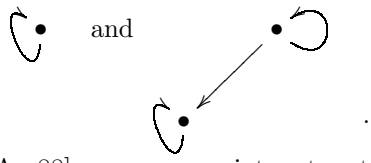
.

This tree has subtrees given by always increasing numbers. However, the tree itself is just an infinite unary tree, that is, the same tree associated with the

value 7. A better candidate could be 10:



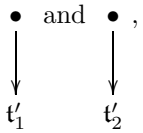
We will prove that this tree is also regular; it is actually equal to the tree associated with 8. We can represent trees that have themselves as subtrees by letting the branches point back at the root. In this way the trees associated with 7 and 8 are



In general, following [Acz88], we can associate a tree to every *accessible pointed graph*, that is, to every directed graph with a distinct node, the *point*, in which every other node can be reached by a path from the point.

Every element of the F-coalgebra $\langle \mathbb{N}, \mathsf{ng} \rangle$, except the first seven, is associated to one of those two trees. The proof of this fact illustrates the principle of *proof by coinduction*. Let us give an intuitive explanation of the principle. Let b_1 and b_2 be two elements of an F-coalgebra $\langle B, g \rangle$, representing the trees $\mathfrak{t}_1$ and $\mathfrak{t}_2$, respectively; that is, $\lhd b_1 \rhd = \mathfrak{t}_1$ and $\lhd b_2 \rhd = \mathfrak{t}_2$.

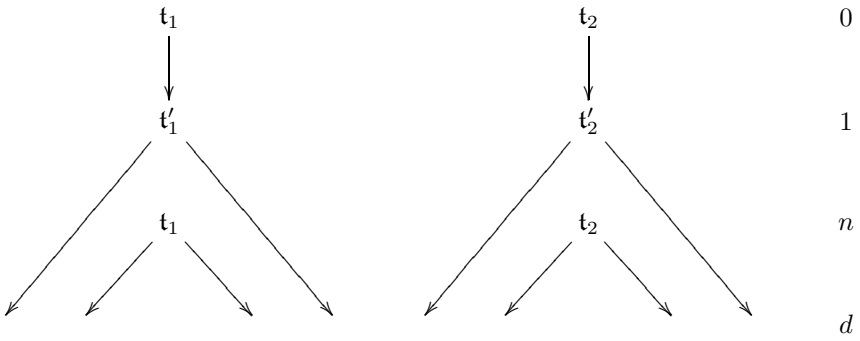
We want to prove that $\mathfrak{t}_1$ and $\mathfrak{t}_2$ are equal. To do this we look at their immediate structure, that is, we check whether they are simple leafs, trees with one branch, or trees with two branches. We can do this by looking at $g(b_1)$ and $g(b_2)$, which are elements of $F[B] = B^0 + B^1 + B^2$. They must be both of the same shape. Let us say, for example, that $g(b_1) = \mathsf{in}_2(b'_1)$ and $g(b_2) = \mathsf{in}_2(b'_2)$. This means that that the corresponding trees are



with $\lhd b'_1 \rhd = \mathfrak{t}'_1$ and $\lhd b'_2 \rhd = \mathfrak{t}'_2$. Now, we need to prove that $\mathfrak{t}'_1$ and $\mathfrak{t}'_2$ are equal. Copies of the original trees $\mathfrak{t}_1$ and $\mathfrak{t}_2$ may occur in $\mathfrak{t}'_1$ and $\mathfrak{t}'_2$, since infinite trees may have themselves as subtrees. In proving that $\mathfrak{t}'_1$ and $\mathfrak{t}'_2$ are equal, we may assume that these occurrences are equal. This assumption is called *coinductive hypothesis*. It may seem circular to assume what we have to prove, but we are

only allowed to use this assumption to prove the equality of subtrees of the original trees, not of the trees themselves.

We can justify the principle of proof by coinduction by showing its validity in a more standard way: We can prove that $\mathfrak{t}_1$ and $\mathfrak{t}_2$ are equal by showing that they are the same at every *level*, that is, that their structure is the same to any depth. Let us indicate by $\mathfrak{t}|d$ the *pruning* of a tree $\mathfrak{t}$ at depth d, that is, the finite tree that coincides with $\mathfrak{t}$ up to depth d and doesn't have any branches of depth larger than d. We can prove the equality of $\mathfrak{t}_1$ and $\mathfrak{t}_2$ by proving $\forall d\colon \mathbb{N}.\mathfrak{t}_1|d = \mathfrak{t}_2|d$. We prove this statement by induction on d. The base case consists in proving that the immediate structure of the two trees is the same. In the inductive step, assume that $\mathfrak{t}_1$ and $\mathfrak{t}_2$ occur as proper subtrees of themselves, with their roots at depth n.



Then, to prove that $\mathfrak{t}_1|d = \mathfrak{t}_2|d$ we can assume, by inductive hypothesis, that $\mathfrak{t}_1|(d-n) = \mathfrak{t}_2|(d-n)$. In other words, this proof by induction actually shows that the coinductive hypothesis is used only up to a smaller depth than the present one.

Therefore we have shown that every use of the principle of proof by coinduction for these trees can be reduced to simple course of value induction on the depth of trees.

The coinduction principle can be used in a more general way to prove simultaneously a property of all trees associated with elements of a coalgebra $\langle B, g\rangle$. Suppose that we have to prove the goal

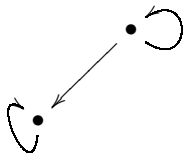
$$\forall b\colon B.b \approx f(b).$$

for some function $f\colon B \to B$, where the relation $\approx$ is the equality of the associated trees, that is, $b_1 \approx b_2$ is the same as $\lhd b_1 \rhd = \lhd b_2 \rhd$. This equality is called *bisimulation* and we define it formally in the next section. We first prove that the immediate structure of every $b\colon B$ is the same as that of $f(b)$. Then, to prove that their subtrees are the same, we can assume the property to prove for all subtrees of b: If the tree of b' occurs as a proper subtree of the tree of b, then we can assume $b' \approx f(b')$ for this occurrence in proving that the subtrees of of the tree of b are equal to the subtrees of the tree of $f(b)$.

This principle can be similarly justified by induction on the depth of the trees.

Proposition 3.1.2 *In the F-coalgebra $\langle \mathbb{N}, \mathsf{ng} \rangle$, for all elements $n\colon \mathbb{N}$ such that n is odd and $n \geq 7$, $\lhd n \rhd$ is equal to*

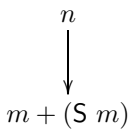
while for all elements $n\colon \mathbb{N}$ such that n is even and $n \geq 8$, $\lhd n \rhd$ is equal to



Proof This is a first example of a proof by coinduction.

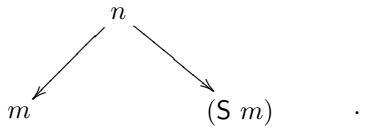
We start with the base case: We prove that the equality holds for the base-level structure of the trees. We use the following property of the function ng: For every $n\colon \mathbb{N}$ such that $n \geq 7$ we have that, if n is odd, then $\mathsf{ng}(n) = (\mathsf{in}_1\ m)$ with m also odd and $m \geq 7$; if n is even, then $\mathsf{ng}(n) = (\mathsf{in}_2\ \langle m, (\mathsf{S}\ m)\rangle$ with m odd and $m \geq 7$. This can easily be proved by induction on n.

Now we prove the statement for odd values. It is true for the value 7. For a value n larger that 7, thus $n \geq 9$, the result of $\mathsf{ng}(n)$ depends on $\mathsf{ng}(n-1)$. Since $n-1$ is even and $n-1 \geq 8$, $\mathsf{g}(n-1) = (\mathsf{in}_2\ \langle m, (\mathsf{S}\ m)\rangle)$ for some $m\colon \mathbb{N}$ with $m \geq 7$. Then, $\mathsf{ng}(n) = (\mathsf{in}_1\ m + (\mathsf{S}\ m))$. So $\lhd n \rhd$ is



and the value $m + (\mathsf{S}\ m)$ is an odd number larger that 7. We can assume coinductively that the statement holds for $m + (\mathsf{S}\ m)$. Then it must also hold for n.

Finally, we prove the statement for even values. Let $n\colon \mathbb{N}$ be even and larger or equal to 8. The value of $\mathsf{ng}(n)$ depends on the value of $\mathsf{ng}(n-1)$. Since $n-1$ is odd, $\mathsf{ng}(n-1) = (\mathsf{in}_1\ m)$, with $m \geq 7$ odd. Then $\mathsf{ng}(n) = (\mathsf{in}_2\ \langle m, (\mathsf{S}\ m)\rangle$, so $\lhd n \rhd$ is



Since m is odd and larger or equal to 7, the left branch is of the desired form, and since $(\mathsf{S}\ m)$ is even and larger that 8 we can assume that the statement holds for it by coinductive hypothesis. So the statement holds also for n. □

The fundamental step is the use of the *coinduction principle*: In proving that the statement holds for a tree, we assume that it already holds for all its subtrees (this is the assumption that we called *coinductive hypothesis*). Be advised: This is not a proof by well-founded induction, since the trees may be infinite.

We therefore have that the F-coalgebra $\langle \mathbb{N}, \mathsf{ng} \rangle$ can represent only a finite number of trees. The problem arises of how expressive a coalgebra can be: What subsets of the set of finite and infinite trees of branching degree at most two can be represented by coalgebras? Let us first give an intuitive answer: All subsets closed under the subtree relation. Let us call $\mathfrak{Tree}$ the set of all trees. We call every subset $\mathfrak{F} \subseteq \mathfrak{Tree}$ closed under the subtree relation a *forest*.

Every F-coalgebra $\langle B, g \rangle$ represents the elements of a forest. The function g is the structural function that gives the subtrees of a given tree. Vice versa, for every forest $\mathfrak{F}$, we can find an F-coalgebra representing it. Just take the coalgebra $\langle \mathfrak{F}, \mathfrak{sub} \rangle$, where $\mathfrak{sub}$ is the function giving the subtrees of a tree.

This was an intuitive mathematical explanation. In type theory, the second part of the result is not immediate, because a subset of a type is not itself a type. It can be realized by coinductive types, which we introduce in the next section. However, in this case, we can show that a type of all trees can be constructed without the help of coinductive definitions.

We have the following classification of trees: finite trees; infinite regular trees, that is, infinite trees with a finite number of subtrees; infinite irregular trees. The examples we gave so far contain only finite and infinite regular trees. Let us give an example of a coalgebra representing infinite irregular trees.

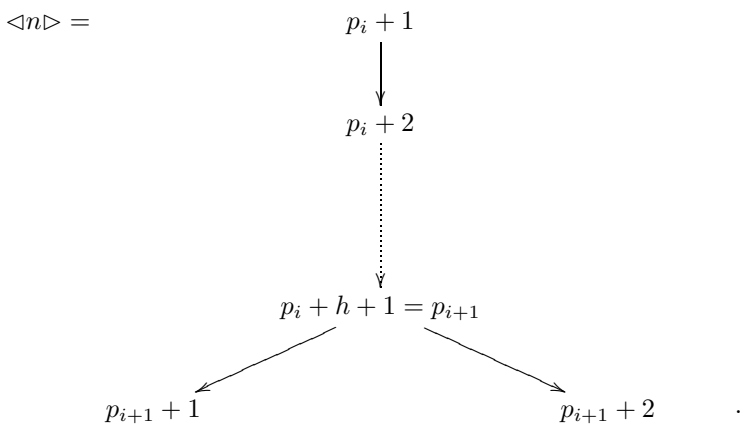
The type of the elements of the coalgebra is again $\mathbb{N}$. The structural function is

$$\mathsf{nig} \colon \mathbb{N} \to \mathbb{N}^0 + \mathbb{N}^1 + \mathbb{N}^2$$
$$\mathsf{nig}(n) := \begin{cases} (\mathsf{in}_1\ n+1) & \text{if } n \text{ is not prime} \\ (\mathsf{in}_2\ n+1\ n+2) & \text{if } n \text{ is prime.} \end{cases}$$

(nig stands for "a g on the natural numbers that produces irregular trees".)

Proposition 3.1.3 *The F-coalgebra $\langle \mathbb{N}, \mathsf{nig} \rangle$ represents an infinite number of trees. Every tree represented in the coalgebra is infinite and irregular.*

Proof To prove that there are an infinite number of trees, consider two consecutive prime numbers p_i and p_{i+1}. The tree associated with $n = p_i + 1$ begins with $h = p_{i+1} - p_i - 1$ single branches:



The gap between two consecutive prime numbers grows without bound. This is a known fact in number theory. For completeness, let us give a short proof. (See, for example, [AZ99], page 7.) We want to show that, for a given natural number n, there are two consecutive prime numbers p_i and p_{i+1} such that $p_{i+1} - p_i > n$. Let $N = (n+1)!$. Then $N + k$ is divisible by k for every $k \leq n+1$, so the numbers $N+2, \ldots, N+n+1$ are not prime. If p_i is the largest prime number that is smaller that $N+2$, we have that the next prime number p_{i+1} must be larger than $N + n + 1$, and therefore the gap between the two is larger than n.

As a consequence, we have that there are trees that begin with larger and larger initial segments of single branches. Therefore there is an infinite number of different trees.

Every natural number n has an associated tree in which the trees of all numbers m larger that n occur. So all but the first n trees occur as subtrees of the tree of n. Therefore the tree of n has an infinite number of subtrees, that is, it is irregular. □

The last question we want to answer is whether there exists a coalgebra that represents all trees. Of course, on an intuitive level, the answer is trivially yes, because we can choose the set of all trees as carrier of the algebra, and the subtree function as structural function. But we want to see if we can answer that question in type theory. In the next section, we introduce coinductive types, exactly to represent collections of infinite objects like $\mathfrak{Tree}$. For now, let us see if we can find a solution using ordinary types. First of all, we remark that it is impossible to find such a solution in the form $\langle \mathbb{N}, g \rangle$, for any structural function g, because the set of all trees is uncountable. We need to use an uncountable carrier type.

We start with a more modest goal: to find a coalgebra where we can represent a single tree. We can do it always using a coalgebra whose elements are natural numbers.

Proposition 3.1.4 *Given a tree* $\mathfrak{t}$: $\mathfrak{Tree}$*, there exists an* F*-coalgebra* $\langle \mathbb{N}, g_{\mathfrak{t}} \rangle$ *in which* 0 *represents* $\mathfrak{t}$*.*

Proof Given $\mathfrak{t}$, we construct a correspondence between the nodes of $\mathfrak{t}$ and $\mathbb{N}$. First of all we look at the main node of $\mathfrak{t}$. We associate it to the number 0. If it is a leaf, we put $g_{\mathfrak{t}}(0) = (\mathsf{in}_0\ \bullet)$, and we can give arbitrary values to $g_{\mathfrak{t}}(n)$ for $n > 0$. If the main node of $\mathfrak{t}$ has a single branch, then we put $g_{\mathfrak{t}}(0) = (\mathsf{in}_1\ 1)$ and we associate 1 to the direct subtree of $\mathfrak{t}$. If the main node of $\mathfrak{t}$ has two branches, then we put $g_{\mathfrak{t}}(0) = (\mathsf{in}_2\ \langle 1, 2 \rangle)$ and we associate 1 to the main node of the left subtree and 2 to the main node of the right subtree.

We can then proceed similarly for the other nodes: At every stage, we choose the least number n that has been assigned to a node of the tree but for which the value of $g_{\mathfrak{t}}(n)$ has not yet been defined. If the node associated with n is a leaf, we put $g_{\mathfrak{t}}(n) = (\mathsf{in}_0\ \bullet)$. If the node associated with n has a single branch, we put $g_{\mathfrak{t}}(n) = (\mathsf{in}_1\ m)$ where m is the least number that has not yet been associated to a node of $\mathfrak{t}$, and we associate m to the main node of the direct subtree of n.

If the node associated with n has two branches, we put $g_{\mathfrak{t}}(n) = (\mathsf{in}_2\ \langle m_1, m_2\rangle)$ where m_1 and m_2 are the least two numbers that have not yet been associated to a node of $\mathfrak{t}$, we associate m_1 to the main node of the left subtree of the tree of n and m_2 to the main node of the right subtree.

In the case the process terminates, which happens if and only if $\mathfrak{t}$ is finite, we will associate an arbitrary value to the remaining numbers.

It is clear that $\lhd 0 \rhd = \mathfrak{t}$. □

We have proved that every tree can be represented by 0 by choosing an appropriate element of $\mathbb{N} \to \mathbb{N}^0 + \mathbb{N}^1 + \mathbb{N}^2$. In general, every pair $\langle g, n\rangle$, where $g\colon \mathbb{N} \to \mathbb{N}^0 + \mathbb{N}^1 + \mathbb{N}^2$ and $n\colon \mathbb{N}$, represents a tree, and every tree can be represented by such a pair (choosing always $n = 0$). Let us put $\mathbb{T} := (\mathbb{N} \to \mathbb{N}^0 + \mathbb{N}^1 + \mathbb{N}^2) \times \mathbb{N}$ and define a structural function on it:

$$\mathsf{tg}\colon \mathbb{T} \to \mathbb{T}^0 + \mathbb{T}^1 + \mathbb{T}^2$$

$$\mathsf{tg}(\langle g, n\rangle) := \mathsf{Cases}\ g(n)\ \mathsf{of} \begin{cases} (\mathsf{in}_0\ x) & \mapsto (\mathsf{in}_0\ x) \\ (\mathsf{in}_1\ m) & \mapsto (\mathsf{in}_1\ \langle g, m\rangle) \\ (\mathsf{in}_2\ \langle m_1, m_2\rangle) & \mapsto (\mathsf{in}_2\ \langle\langle g, m_1\rangle, \langle g, m_2\rangle\rangle) \end{cases}$$

Theorem 3.1.5 *In the F-coalgebra $\langle \mathbb{T}, \mathsf{tg}\rangle$ every element of $\mathfrak{Tree}$ can be represented.*

Proof Given $\mathfrak{t}\colon \mathfrak{Tree}$, by the previous proposition there exists a function $g_{\mathfrak{t}}$ such that 0 represents $\mathfrak{t}$ in the F-coalgebra $\langle \mathbb{N}, g_{\mathfrak{t}}\rangle$. Then $\langle g, 0\rangle$ represents $\mathfrak{t}$ in $\langle \mathbb{T}, \mathsf{tg}\rangle$. □

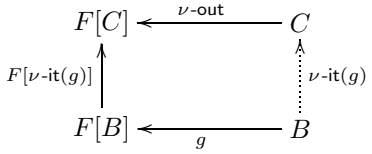
We will see in the next section that $\langle \mathbb{T}, \mathsf{tg}\rangle$ is a final F-coalgebra, and, therefore, is isomorphic to the coinductive type associated with F.

3.2 Coinductive Types

As for inductive types, the rules for coinductive types can be synthetically and intuitively expressed in categorical terms.

Let F be an endofunctor on the category s, that is, $X\colon s \vdash F[X]\colon s$ and there is an extension of F to functions, $X, Y\colon s; f\colon X \to Y \vdash \Phi_F[f]\colon F[X] \to F[Y]$. For the sake of readability we usually write $F[f]$ for $\Phi_F[f]$.

Definition 3.2.1 *A* final F-coalgebra *is the final object in the category of F-coalgebras, that is, an F-coalgebra $\langle C, \nu\text{-}\mathsf{out}\rangle$ such that, for every other F-coalgebra $\langle B, g\rangle$, there exists a unique morphism $\nu\text{-}\mathsf{it}(g)\colon B \to C$ that makes the following diagram commute:*

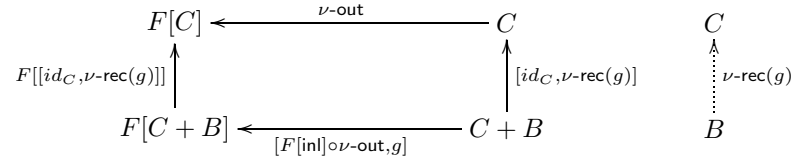


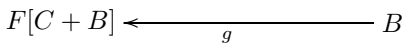
So $\langle C, \nu\text{-}\mathsf{out}\rangle$ *is a final* F*-coalgebra if and only if*

$$\forall B\colon s.\forall g\colon B \to F[B].\exists!\nu\text{-}\mathsf{it}(g)\colon B \to C.\nu\text{-}\mathsf{out} \circ \nu\text{-}\mathsf{it}(g) = F[\nu\text{-}\mathsf{it}(g)] \circ g$$

This categorical characterization gives the coiteration principle, which is the dual of the iteration principle for initial F-algebras.

We can also characterize the corecursion principle, the dual of the recursion principle for initial F-algebras, by the categorical diagram





corresponding to the property

$$\begin{aligned}&\forall B\colon s.\forall g\colon B \to F[C+B].\exists!\nu\text{-}\mathsf{rec}(g)\colon B \to C.\\ &\quad \nu\text{-}\mathsf{out} \circ [id_C, \nu\text{-}\mathsf{rec}(g)] = F[[id_C, \nu\text{-}\mathsf{rec}(g)]] \circ [F[\mathsf{inl}] \circ \nu\text{-}\mathsf{out}, g]\end{aligned}$$

This equality breaks down to two equalities when precomposed with the two injections inl and inr, the first of the two being an identity.

$$\begin{aligned}&\nu\text{-}\mathsf{out} \circ [id_C, \nu\text{-}\mathsf{rec}(g)] \circ \mathsf{inl}\colon C \to F[C]\\ &= \nu\text{-}\mathsf{out}\\ &F[[id_C, \nu\text{-}\mathsf{rec}(g)]] \circ [F[\mathsf{inl}] \circ \nu\text{-}\mathsf{out}, g] \circ \mathsf{inl}\colon C \to F[C]\\ &= F[[id_C, \nu\text{-}\mathsf{rec}(g)]] \circ F[\mathsf{inl}] \circ \nu\text{-}\mathsf{out}\\ &= F[[id_C, \nu\text{-}\mathsf{rec}(g)] \circ \mathsf{inl}] \circ \nu\text{-}\mathsf{out}\\ &= F[id_C] \circ \nu\text{-}\mathsf{out} = \nu\text{-}\mathsf{out}\\ &\nu\text{-}\mathsf{out} \circ [id_C, \nu\text{-}\mathsf{rec}(g)] \circ \mathsf{inr}\colon B \to F[C]\\ &= \nu\text{-}\mathsf{out} \circ \nu\text{-}\mathsf{rec}(g)\\ &F[[id_C, \nu\text{-}\mathsf{rec}(g)]] \circ [F[\mathsf{inl}] \circ \nu\text{-}\mathsf{out}, g] \circ \mathsf{inr}\colon B \to F[C]\\ &= F[[id_C, \nu\text{-}\mathsf{rec}(g)]] \circ g\end{aligned}$$

The defining equation for corecursion becomes then

$$\begin{aligned}&\nu\text{-}\mathsf{out} = \nu\text{-}\mathsf{out}\\ &\nu\text{-}\mathsf{out} \circ \nu\text{-}\mathsf{rec}(g) = F[[id_C, \nu\text{-}\mathsf{rec}(g)]] \circ g.\end{aligned}$$

As the principles of iteration and recursion are equivalent, so are the principles of coiteration and corecursion.

The use of the corecursion principle in place of the coiteration principle is parallel to the use of the *labelled anti-foundation axiom* in place of the simple anti-foundation axiom in the theory of non-well-founded sets (see [Acz88], pg. 10).

We use this categorical formulation as a guide to lay down the rules for coinductive types.

Definition 3.2.2 *Let $X\colon s \vdash F[X]\colon s$ be a strictly positive operator. The* coinductive type $\nu_X(F)$ *is defined by the following rules (where we write C for $\nu_X(F)$):*

formation

$$C\colon s$$

elimination

$$\frac{x\colon C}{(\nu\text{-out } x)\colon F[C]}$$

introduction (corecursion)

$$\frac{T\colon s \quad z\colon T \vdash w[z]\colon F[C+T]}{t\colon T \vdash (\nu\text{-rec } [z]w\ t)\colon C}$$

reduction

$$(\nu\text{-out } (\nu\text{-rec } [z]w\ t)) \rightsquigarrow F\left[[y\colon C+T]\mathsf{Cases}\ y\ \mathsf{of}\ \left\{\begin{array}{l}(\mathsf{inl}\ c) \mapsto c\\ (\mathsf{inr}\ z) \mapsto (\nu\text{-rec } [z]w\ z)\end{array}\right.\right](w[t])$$

The corecursion rule is the type theoretic version of the second categorical diagram for final coalgebras. As for inductive types, the rule for corecursion can be simplified to the rule for *coiteration*, corresponding to the first diagram for final coalgebras.

$$\frac{T\colon s \quad z\colon T \vdash u[z]\colon F[T]}{t\colon T \vdash (\nu\text{-it } [z]u\ t)\colon C}$$

The corresponding reduction relation is

$$(\nu\text{-out } (\nu\text{-it } [z]u\ t)) \rightsquigarrow (F[(\nu\text{-it } [z]u)]\ u[t]).$$

The coiterator ν-it can be defined from the corecursor ν-rec by

$$(\nu\text{-it } [z]u\ t) = (\nu\text{-rec } [z](F[\mathsf{inr}]\ u[z])\ t).$$

As recursion and iteration are equivalent for inductive types, corecursion and coiteration are equivalent for coinductive types.

Notice that the rules for coinductive types do not completely capture the categorical definition: We formalized the existence of the corecursor and imposed that it satisfies the defining equality by the recursion rule, but we did not formalize the uniqueness part of the categorical definition. This means that we actually formalized what is usually called a *weakly final coalgebra*. The reason to do this is that uniqueness cannot be characterized by reduction rules, and its imposition would make type-checking undecidable. Later, we define a stronger notion of equality on coinductive types, bisimilarity, by which ν-rec is unique. We do not impose that bisimilarity implies equality, but work directly with it. This means that the pair formed by a coinductive type and the corresponding bisimilarity relation is a setoid (see chapter 5). It is a final coalgebra in the category of setoids and setoid functions.

Remark 3.2.3 *The coiteration and corecursion principles for coinductive types are the dual of the iteration and recursion principles for inductive type. The duality can be seen immediately by taking one of the rules, changing the direction of the sequent signs, and exchanging products with sums:*

$$\textbf{iteration:}\quad \frac{F[B]\vdash B}{I\vdash B} \qquad \textbf{coiteration:}\quad \frac{B\vdash F[B]}{B\vdash C}$$

$$\textbf{recursion:}\quad \frac{F[I\times B]\vdash B}{I\vdash B} \qquad \textbf{corecursion:}\quad \frac{B\vdash F[C+B]}{B\vdash C}$$

On the other hand, there in no immediate way to dualize the general induction principle. It is possible of find a coinduction *principle for coinductive types that is the dual of a generalization of the induction principle for inductive types is that that uses relation instead of predicates. It is shown in [HJ98], extending the work on natural numbers by [RT93, RT94, TR98], that we can formulate a* binary induction principle *and the dual* binary coinduction principle *(see also [PA93]). Given a relation $F\colon X\to X\to s$, we can extend it through F: $R^F\colon F[X]\to F[X]\to s$ (see Definition 3.3.1). The induction principle can be extended to*

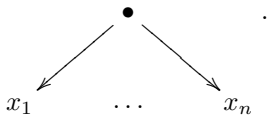
$$\frac{R\colon I\to I\to s \qquad (s,t\colon F[I])(R^F\ x\ y)\to(R\ (\mu\text{-intro}\ x)\ (\mu\text{-intro}y))}{(x,y\colon I)x=y\to(R\ x\ y)}$$

or, in words, every congruence on an initial algebra contains the equality relation. *The dual of this principle is*

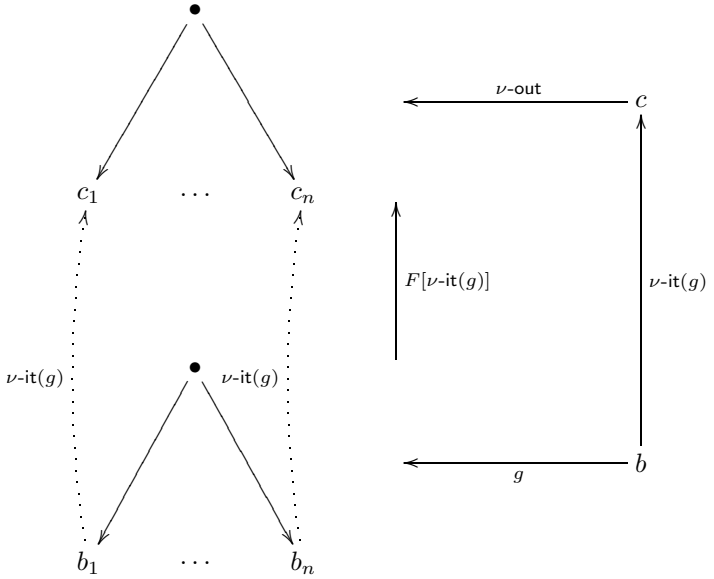
$$\frac{R\colon C\to C\to s \qquad (x,y\colon C)(R\ x\ y)\to(R^F\ (\nu\text{-out}\ x)\ (\nu\text{-out}\ y))}{(x,y\colon C)(R\ x\ y)\to x=y}$$

or, in words, every bisimulation on a terminal coalgebra is contained in the equality relation. *We talk about bisimulation relation in the next section. However, we do not assume this principle but, as already remarked, we would define the final coalgebra as a setoid having bisimilarity as the book equality. Therefore, the rule would become true by definition for setoids in place of types.*

Let us give a graphical explanation of the shape of the elements of a coinductive type and of the rules of coiteration and corecursion, similar to the depiction that we gave for inductive types. We represent, for a given type X, the elements of $F[X]$ as trees having elements of X as their leaves:

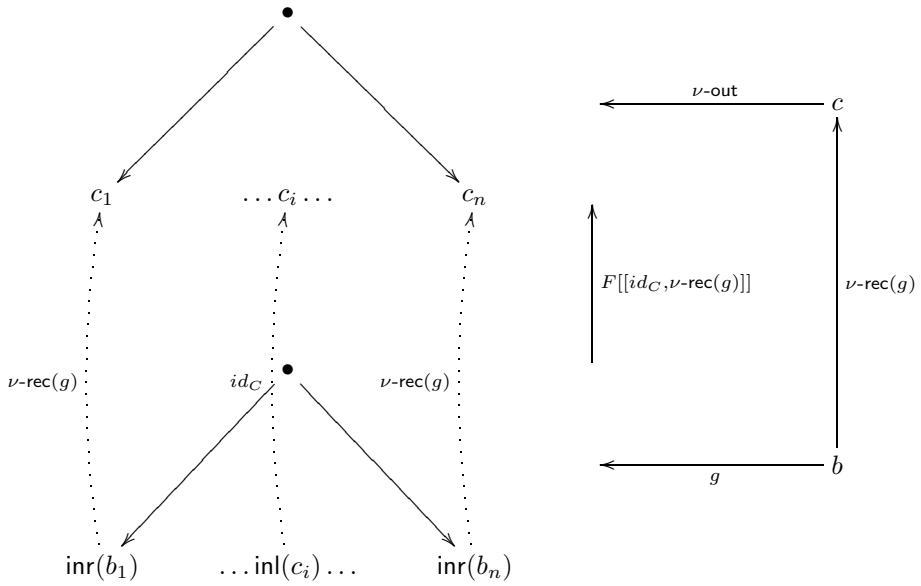


Then the rule of coiteration can be depicted as follows:



Given an element b of an F-coalgebra $\langle B, g\rangle$, the element c of the coinductive type corresponding to it is the one whose unfolding corresponds to the unfolding of b by g.

A similar depiction illustrates the rule of corecursion. Here a "shortcut" can be taken when evaluating b: the tree given by g may directly put an element c_i of C on a leaf.



Trees

We return to our example of trees of branching degree at most two. We take again the functor $F[X] := X^0 + X^1 + X^2$. We can now define the type of finite and infinite trees as $\mathbb{T}^\nu := \nu_X(X^0 + X^1 + X^2)$. This is, by definition, a type in which all F-coalgebras can be embedded by the coiteration rule.

Earlier we introduced the type $\mathbb{T}$ and proved informally that all trees can be represented by its elements. If that is true we should be able to prove that $\mathbb{T}$ and $\mathbb{T}^\nu$ are isomorphic.

Here we must stop for a moment to clarify what we mean by *isomorphic* when talking about coalgebras. In an F-coalgebra $\langle B, g\rangle$ a certain tree could be represented by different elements of the carrier type B. We need to define formally what it means for two elements $b_1, b_2 \colon B$ to be equal, in the sense of representing the same tree. In other words, we must give a formal definition of the relation $b_1 \approx b_2$ that we explained earlier as intuitively meaning $\lhd b_1 \rhd = \lhd b_2 \rhd$. It is hard to define such a relation without making any reference to the class $\mathfrak{T}$ of all trees. We want the relation to satisfy the property that, if $b_1 \approx b_2$, then their immediate structures are the same and their components are also in the relation $\approx$.

Definition 3.2.4 *Let $\langle B, g\rangle$ be an F-coalgebra. A relation $R \colon B \to B \to \mathsf{Prop}$ is said to be a* bisimulation relation *if, for every $b_1, b_2 \colon B$, if $(R\ b_1\ b_2)$ holds, then one of the following three conditions holds:*

$$(g\ b_1) = (b_b 2) = (\mathsf{in}_0\ \bullet)$$
$$(g\ b_1) = (\mathsf{in}_1\ b_1') \wedge (g\ b_2) = (\mathsf{in}_1\ b_2') \wedge (R\ b_1'\ b_2')$$
$$(g\ b_1) = (\mathsf{in}_2\ \langle b_1', b_1''\rangle) \wedge (g\ b_2) = (\mathsf{in}_2\ \langle b_2', b_2''\rangle) \wedge (R\ b_1'\ b_2') \wedge (R\ b_1''\ b_2'')$$

Ideally $\approx$ is the larges bisimulation relation (see [Acz88], Theorem 2.4, pg. 20, and [BM96], Theorem 7.3, pg. 82). In type theory, it could be defined by the impredicative characterization

$$b_1 \approx b_2 := \exists R \colon B \to B \to \mathsf{Prop}.\mathsf{Bisimulation}(R) \wedge (R\ b_1\ b_2)$$

where $\mathsf{Bisimulation}$ is the formalization of Definition 3.2.4.

Alternatively we may define $\approx$ as a *coinductive predicate*. This is done in the next section.

Proving that two F-coalgebras $\langle B_1, g_1\rangle$ and $\langle B_2, g_2\rangle$ are isomorphic consists in finding two functions $h_1 \colon B_1 \to B_2$ and $h_2 \colon B_2 \to B_1$ such that $\forall b_1 \colon B_1.b_1 \approx (h_2\ (h_1\ b_1))$ and $\forall b_2 \colon B_2.b_2 \approx (h_2\ (h_1\ b_2))$.

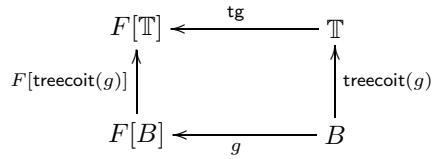
We will construct the isomorphism pair of function for $\mathbb{T}$ and $\mathbb{T}^\nu$ and leave the proof of the bisimulation property for later, when we define $\approx$ as a coinductive predicate.

The function $h_1^{\mathbb{T}} \colon \mathbb{T} \to \mathbb{T}^\nu$ is easily defined by using the coiteration principle, that states that every F-coalgebra, in particular $\mathbb{T}$, can be embedded in $\mathbb{T}^\nu$.

$$h_1^{\mathbb{T}} \colon \mathbb{T} \to \mathbb{T}^\nu$$
$$h_1^{\mathbb{T}}(t) := (\nu\text{-}\mathsf{it}\ [z](\mathsf{tg}\ z)\ t)$$

To construct $h_2^{\mathbb{T}}\colon \mathbb{T}^\nu \to \mathbb{T}$ we first prove that $\langle \mathbb{T}, \mathsf{tg}\rangle$ is a weakly final coalgebra, that is, that it satisfies the equivalent of the coiteration principle.

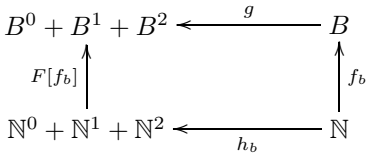
Lemma 3.2.5 *For every F-coalgebra $\langle B, g\rangle$, there exists a function* $\mathsf{treecoit}(g)\colon B \to \mathbb{T}$ *such that the diagram*



commutes, that is

$$\forall b\colon B.(F[\mathsf{treecoit}(g)]\ (g\ b)) = (\mathsf{tg}\ (\mathsf{treecoit}(g)\ b)).$$

Proof The proof is a formalized version of the proof of Theorem 3.1.5. Given $b\colon B$, we simultaneously define two functions $f_b\colon \mathbb{N} \to B$ and $h_b\colon \mathbb{N} \to \mathbb{N}^0 + \mathbb{N}^1 + \mathbb{N}^2$. Intuitively, f_b associates to every natural number an element of B that represents a subtree of the tree associated with b. We also want that $(f_b\ 0) = b$. The function h_b copies the structure of the F-coalgebra $\langle B, g\rangle$ on $\mathbb{N}$, that is, h_b is such that the diagram



commutes.

The two functions are constructed by recursion on natural numbers. During the construction we maintain two indexes m and n. The index m is the least natural number x such that $(f_b\ x)$ has been defined, but $(h_b\ x)$ hasn't been defined yet. The index n is the least natural number y such that $(f_b\ y)$ has not yet been defined. The definition is given in such a way that at every stage f_b and h_b are defined on an initial segment of $\mathbb{N}$. Therefore it is always true that $m \leq n$.

We define f_b and h_b by recursion on m.

The basis step consists in setting $(f_b\ 0) := b$, $n := 1$, and $m := 0$.

At any stage of the definition, it may happen that m is not well defined anymore, that is, h_b has been defined on all the arguments x on which f_b has been defined. In this case we stop and we give arbitrary values to f_b and h_b for arguments larger than m. For these values we don't require that the above diagram commutes.

In the recursive step we define $(h_b\ m)$ and we add some of the values of f_b. We proceed by cases on $(g\ (f_b\ m))$:

- If $(g\ (f_b\ m)) = (\mathsf{in}_0\ \bullet)$, then we put $(h_b\ m) := (\mathsf{in}_0\ \bullet)$;

- If $(g\ (f_b\ m)) = (\mathsf{in}_1\ b_m)$, then we put $(h_b\ m) := (\mathsf{in}_1\ n)$, $(f_b\ n) = b_m$, and we increment n by one, $n := n + 1$;
- If $(g\ (f_b\ m)) = (\mathsf{in}_2\ b'_m\ b''_m)$, then we put $(h_b\ m) := (\mathsf{in}_2\ n\ (n+1))$, $(f_b\ n) := b'_m$, $(f_b\ (n+1)) := b''_m$, and we increment n by two, $n := n + 2$.

The functions f_b and h_b are defined in a procedural way, but it is not difficult to turn the definition into a functional one that can be formalized in type theory. For example we can define an operator

$$\mathcal{F}\colon \mathbb{N} \to (\mathbb{N} \to B) \times (\mathbb{N} \to F[\mathbb{N}]) \times \mathbb{N}$$

by recursion, in such a way that $\mathcal{F}(m) = \langle f_{b,m}, h_{b,m}, n\rangle$, where $f_{b,m}$ and $h_{b,m}$ are the intermediate stages of the definition of f_b and h_b. Initially, we can put $(f_{b,0}\ x) := b$ and $(h_{b,0}\ x) := 0$ for every $x\colon \mathbb{N}$. We can use the trick of putting $n := 0$ as a signal to stop. Finally, we can define f_b and h_b by diagonalization: $(f_b\ n) := (f_{b,n}\ n)$ and $(h_b\ m) := (h_{b,m}\ m)$.

We can now define $(\mathsf{treecoit}(g)\ b) := \langle h_b, 0\rangle$. A routine check shows that, with this definition, the diagram in the statement commutes. □

Using Lemma 3.2.5 we can define the second function of the isomorphism pair.

$$h_2^{\mathbb{T}}\colon \mathbb{T}^\nu \to \mathbb{T}$$
$$h_2^{\mathbb{T}}(t) := (\mathsf{treecoit}(\nu\text{-}\mathsf{out})\ t)$$

Theorem 3.2.6 *The F-coalgebras $\mathbb{T}$ and $\mathbb{T}^\nu$ are isomorphic.*

Proof We have to check that $h_1^{\mathbb{T}}$ and $h_2^{\mathbb{T}}$ form an isomorphism pair. We can use the impredicative definition of equality as the largest bisimulation relation. But it is easier to prove the statement using the notion of equality as a coinductive predicate. Therefore we postpone the proof until the next section. □

Streams

The most widely used coinductive type is the type of streams, or infinite lists. (See [Rut01] and [HJ99] for good treatments of streams.) In lazy functional programming languages, the type of lists over a type A contains both finite and infinite lists. This type can be modeled in type theory by the coinductive type $\nu_X(\mathsf{Unit} + A \times X)$. More often, a type of purely infinite lists, or streams, is considered, $\mathbb{S}_A := \nu_X(A \times X)$. The elimination rule for streams is then

$$\frac{s\colon \mathbb{S}_A}{\nu\text{-}\mathsf{out}(s)\colon A \times \mathbb{S}_A}$$

In common practice the functions giving the head and tail of an infinite stream are used in place of ν-out:

$$\mathsf{head}\colon \mathbb{S}_A \to A \qquad \mathsf{tail}\colon \mathbb{S}_A \to \mathbb{S}_A$$
$$\mathsf{head}(s) := (\pi_1\ (\nu\text{-}\mathsf{out}\ s)) \qquad \mathsf{tail}(s) := (\pi_2\ (\nu\text{-}\mathsf{out}\ s))$$

We use the list notations $[h : t]$ for the stream with head h and tail t; $[h_1, h_2, h_3, \ldots]$ for the stream with head h_1, head of the tail h_2, head of the tail of the tail h_3, and so on.

We recall the rules of coiteration and corecursion in this particular case.

$$\frac{T\colon s \quad z\colon T \vdash u[z]\colon A \times T}{t\colon T \vdash (\nu\text{-}\mathsf{it}\ [z]u\ t)\colon \mathbb{S}_A} \qquad \frac{T\colon s \quad z\colon T \vdash w[z]\colon A \times (\mathbb{S}_A + T)}{t\colon T \vdash (\nu\text{-}\mathsf{rec}\ [z]w\ t)\colon \mathbb{S}_A}$$

As examples, we define some simple streams over natural numbers. We write just $\mathbb{S}$ for $\mathbb{S}_{\mathbb{N}}$.

The first example is the constantly zero stream $s_0 := [0, 0, 0, \ldots]$. It can be defined formally by

$$s_0 := (\nu\text{-}\mathsf{it}\ [z]\langle 0, z\rangle\ t)$$

where we can choose any nonempty T and any element $t\colon T$, for example $T := \mathsf{Unit}$ and $t := \bullet$.

Next, we define the stream $s_1 := [1, 0, 0, 0, \ldots]$. This is just the stream with head 1 and tail s_0. The corecursion principle allows us to give the tail directly:

$$s_1 := (\nu\text{-}\mathsf{rec}\ [z]\langle 1, (\mathsf{inl}\ s_0)\rangle\ t)$$

where again we can choose any non-empty type T and any element $t\colon T$.

The stream enumerating all natural numbers, $s_\omega := [0, 1, 2, 3, \ldots]$, can be defined by coiteration on the natural numbers:

$$s_\omega := (\nu\text{-}\mathsf{it}\ [x]\langle x, x+1\rangle\ 0)$$

with $T := \mathbb{N}$.

The stream of Fibonacci numbers can be defined by

$$\begin{array}{l} \mathsf{fib}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{S} \\ \mathsf{fib}(p) := (\nu\text{-}\mathsf{it}\ [z = \langle x, y\rangle]\langle x, \langle y, x + y\rangle\rangle\ p) \\ \\ \mathsf{Fib}\colon \mathbb{S} \\ \mathsf{Fib} := (\mathsf{fib}\ \langle 1, 1\rangle), \end{array}$$

where we used the simplified notation

$$\begin{array}{rcl} [z = \langle x, y\rangle] \ldots & := & [z](\mathsf{prod\text{-}case}\ [x, y] \ldots\ z) \\ & = & [z](\ldots [x := (\pi_1\ z), y := (\pi_2\ z)]). \end{array}$$

Let us now define some operations on streams. The first example is the zip function, that takes two streams $a = [a_0, a_1, \ldots]$ and $b = [b_0, b_1, \ldots]$, and gives the stream obtained by alternating the elements of a and b:

$$\begin{array}{l} \mathsf{zip}\colon \mathbb{S}_A \times \mathbb{S}_A \to \mathbb{S}_A \\ \mathsf{zip}(c) := (\nu\text{-}\mathsf{it}\ [z]\langle(\mathsf{head}\ (\pi_1\ z)), \langle(\pi_2 z), (\mathsf{tail}\ (\pi_1\ z))\rangle\rangle\ c) \end{array}$$

As in the case of trees of branching degree at most two, also streams can be defined without the help of coinductive types. For the functor $F[X] :=$

$A \times X$, we can construct a final F-coalgebra without the use of a coinductive definition. This final coalgebra is $\mathbb{SN}_A := \langle \mathbb{N} \to A, \mathsf{valshift} \rangle$, where $\mathsf{valshift} := [f]\langle (f\ 0), [n](f\ (n+1)) \rangle$. To show this fact we construct an isomorphism pair between $\mathbb{S}_A$ and $\mathbb{SN}$.

The first function, from $\mathbb{S}_A$ to $\mathbb{SN}_A$, is defined by recursion on the argument of its result:

$$\begin{array}{l} \mathsf{streamfun}\colon \mathbb{S}_A \to (\mathbb{N} \to A) \\ (\mathsf{streamfun}\ s\ 0) := (\mathsf{head}\ s) \\ (\mathsf{streamfun}\ s\ (\mathsf{S}\ n)) := (\mathsf{streamfun}\ (\mathsf{tail}\ s)\ n). \end{array}$$

The second function, from $\mathbb{SN}_A$ to $\mathbb{S}_A$, is defined by coiteration:

$$\begin{array}{l} \mathsf{funstream}\colon (\mathbb{N} \to A) \to \mathbb{S}_A \\ (\mathsf{funstream}\ g) := (\nu\text{-}\mathsf{it}\ [f]\langle (f\ 0), [n](f\ (n+1)) \rangle\ g) \end{array}$$

Theorem 3.2.7 *The coalgebras $\mathbb{SN}_A$ and $\mathbb{S}_A$ are isomorphic.*

Proof The functions streamfun and funstream form an isomorphism pair. The proof of the equality, as for trees, is postponed until the next section. □

The next example is a function parsums that computes the stream of the partial sums of a given stream, that is, given a stream $[a_0, a_1, a_2, \ldots, a_k, \ldots]$, the function computes the stream $[a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, \sum_{i=0}^{k} a_i, \ldots]$.

We have two ways of defining it.

We can translate the problem to the coalgebra $\mathbb{SN}$ and compute the function there,

$$\begin{array}{l} \mathsf{funparsums}\colon (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \\ \mathsf{funparsums}(f, 0) := (f\ 0) \\ \mathsf{funparsums}(f, (\mathsf{S}\ n)) := (\mathsf{funparsums}\ f\ n) + (f\ (\mathsf{S}n)), \end{array}$$

and then translate back to streams,

$$\begin{array}{l} \mathsf{parsums}\colon \mathbb{S} \to \mathbb{S} \\ \mathsf{parsums}(\sigma) := (\mathsf{funstream}\ (\mathsf{funparsums}\ (\mathsf{streamfun}\ \sigma))). \end{array}$$

Alternatively, we can define the function directly on streams, using coiteration. We first define an auxiliary function with an extra natural number argument to store the partial sum.

$$\begin{array}{l} \mathsf{auxsums}\colon \mathbb{S} \times \mathbb{N} \to \mathbb{S} \\ \mathsf{auxsums}(q) := (\nu\text{-}\mathsf{it}\ [p]\langle\ (\mathsf{head}\ (\pi_1\ p)) + (\pi_2\ p), \\ \qquad\qquad\qquad\qquad \langle (\mathsf{tail}\ (\pi_1\ p)), (\mathsf{head}\ (\pi_1\ p)) + (\pi_2\ p) \rangle\ \rangle\ q) \\ \\ \mathsf{parsums}\colon \mathbb{S} \to \mathbb{S} \\ \mathsf{parsums}(\sigma) := \mathsf{auxsums}\ \langle \sigma, 0 \rangle \end{array}$$

To understand how the function works, consider, in the definition of auxums, the step given by the function argument of ν-it. Assume that we already processed

the first k elements of the stream, and that we stored the remainder of the stream and the partial sum in an element $q = \langle \sigma', m \rangle \colon \mathbb{S} \times \mathbb{N}$, with $\sigma' = [a_{k+1}, a_{k+2}, \ldots]$ and $m = \sum_{i=0}^{k} a_i$. Then the step function produces the result

$$\begin{aligned}
&\langle (\mathsf{head}\ (\pi_1\ q)) + (\pi_2\ q), \langle (\mathsf{tail}\ (\pi_1\ q)), (\mathsf{head}\ (\pi_1\ q)) + (\pi_2\ q) \rangle \rangle \\
&= \langle (\mathsf{head}\ \sigma') + m, \langle (\mathsf{tail}\ \sigma'), (\mathsf{head}\ \sigma') + m \rangle \rangle \\
&= \langle a_{k+1} + \textstyle\sum_{i=0}^{k} a_i, \langle [a_{k+2}, \ldots], a_{k+1} + \sum_{i=0}^{k} a_i \rangle \rangle \\
&= \langle \textstyle\sum_{i=0}^{k+1} a_i, \langle [a_{k+2}, \ldots], \sum_{i=0}^{k+1} a_i \rangle \rangle,
\end{aligned}$$

which means that $\sum_{i=0}^{k+1} a_i$ is produced as the head of the result, and the computation continues with the stream $[a_{k+2}, \ldots]$ and the partial sum $\sum_{i=0}^{k+1} a_i$.

Two of the most widely used operators on streams are the mapping of a function on all elements of a stream and the generation of the stream of iterates of a function on an element.

The function map takes a function $f \colon A \to B$ and a stream $[a_0, a_1, a_2, \ldots] \colon \mathbb{S}_A$ as input and gives the stream $[(f\ a_0), (f\ a_1), (f\ a_2), \ldots] \colon \mathbb{S}_B$ as output.

$$\begin{aligned}
&\mathsf{map} \colon (A \to B) \to \mathbb{S}_A \to \mathbb{S}_B \\
&\mathsf{map}(f, \sigma) := (\nu\text{-}\mathsf{it}\ [\tau] \langle (f\ (\mathsf{head}\ \tau)), (\mathsf{tail}\ \tau) \rangle\ \sigma)
\end{aligned}$$

The function iter takes a function $f \colon A \to A$ and an element $a \colon A$, and produces the stream $[a, (f\ a), (f^2\ a), (f^3\ a), \ldots] \colon \mathbb{S}_A$.

$$\begin{aligned}
&\mathsf{iter} \colon (A \to A) \to A \to \mathbb{S}_A \\
&\mathsf{iter}(f, a) := (\nu\text{-}\mathsf{it}\ [x] \langle x, (f\ x) \rangle\ a)
\end{aligned}$$

A result by Miner and Marion (see [BM96], pg. 205) states that every function obtained by ν-it can be defined as the composition of an instance of map and iter. Suppose $h \colon C \to \mathbb{S}_A$ has been defined as $(h\ c) := (\nu\text{-}\mathsf{it}\ [y] w\ c)$, where $y \colon C \vdash w[y] \colon A \times C$. Define the two functions

$$\begin{array}{ll} f \colon C \to A & \\ f(y) := (\pi_1\ w[y]) & \text{and} \end{array} \qquad \begin{array}{l} g \colon C \to C \\ g(y) := (\pi_2\ w[y]). \end{array}$$

It is easy to prove that $\forall c \colon C.(h\ c) \approx (\mathsf{map}\ f\ (\mathsf{iter}\ g\ c))$.

We conclude this subsection on streams with a slightly more complex example, taken from [BM96], pg. 208. We want to define a *double map* function $\mathsf{dm} \colon (A \to A) \to \mathbb{S}_A \to \mathbb{S}_A$ such that the following equality holds:

$$(\mathsf{dm}_f\ \sigma) = [(f\ (\mathsf{head}\ \sigma)) : (\mathsf{dm}_f^2\ (\mathsf{tail}\ \sigma))],$$

where we put the function argument f as a subscript. First of all, we prove that such a function must satisfy the following generalized equation:

$$(\mathsf{dm}_f^n\ \sigma) = [(f^n\ (\mathsf{head}\ \sigma)) : (\mathsf{dm}_f^{2n}\ (\mathsf{tail}\ \sigma))]$$

by induction on n:

$$\begin{array}{ll} n = 0 & (\mathsf{dm}_f^0\ \sigma) = \sigma = [(\mathsf{head}\ \sigma) : (\mathsf{tail}\ \sigma)] \\ & = [(f^0\ (\mathsf{head}\ \sigma)) : (\mathsf{dm}_f^{2\cdot 0}\ (\mathsf{tail}\ \sigma))] \\ n = m+1 & (\mathsf{dm}_f^{m+1}\ \sigma) = (\mathsf{dm}_f\ (\mathsf{dm}_f^m\ \sigma)) \\ & = (\mathsf{dm}_f\ [(f^m\ (\mathsf{head}\ \sigma)) : (\mathsf{dm}_f^2\ (\mathsf{tail}\ \sigma))]) \quad \text{by I. H.} \\ & = [(f\ (f^m\ (\mathsf{head}\ \sigma))) : (\mathsf{dm}_f^2\ (\mathsf{dm}_f^{2m}\ (\mathsf{tail}\ \sigma)))] \\ & = [(f^{m+1}\ (\mathsf{head}\ \sigma)) : (\mathsf{dm}_f^{2(m+1)}\ (\mathsf{tail}\ \sigma)). \end{array}$$

We give a generalized definition of the function with the extra argument n:

$$\begin{array}{l} \mathsf{dmaux}_f : \mathbb{S}_A \times \mathbb{N} \to \mathbb{S}_A \\ \mathsf{dmaux}_f(p) := (\nu\text{-}\mathsf{it}\ [y = \langle \sigma, n\rangle]\langle (f^n\ (\mathsf{head}\ \sigma)), \langle (\mathsf{tail}\ \sigma), 2n\rangle\rangle), \end{array}$$

Finally, we can define

$$\begin{array}{l} \mathsf{dm}_f : \mathbb{S}_A \to \mathbb{S}_A \\ \mathsf{dm}_f(\sigma) := (\mathsf{dmaux}_f\ \langle \sigma, 1\rangle) \end{array}$$

3.3 Equality and bisimulation

We tackle now the problem of the definition of equality for coinductive types. The problem concerns coalgebras in general. Given a type operator F and an F-coalgebra $\mathcal{B} = \langle B, g\rangle$, we want to define what it means for two elements b_1 and b_2 to be equal. The type equality $b_1 = b_2$ is too strong, since there can be two distinct elements of B that represent the same infinite object. What we want to state is that the immediate structure of b_1 and b_2 is the same, and that the recursive substructures are also equal. To formalize this notion we must, first of all, define the F-extension of a relation, or *relation lifting*; see [HJ98, PA93].

Definition 3.3.1 *Let $X : s \vdash F[X] : s$ be a strictly positive type operator. Let $X, Y : s$ be two types and $R : X \to Y \to \mathsf{Prop}$ be a relation between X and Y. The F-*extension *of R is the relation*

$$R^F : F[X] \to F[Y] \to \mathsf{Prop}$$

defined by recursion on the structure of F, for every $u : F[X]$ and $w : F[Y]$:

- *If $F[X] = K$ is a constant, we put $(R^F\ u\ w) := (u =_K w)$;*
- *If $F[X] = X$, we put $(R^F\ u\ w) := (R\ u\ w)$;*
- *If $F[X] = F_1[X] \times F_2[X]$ for two strictly positive operators F_1 and F_2, we put $(R^F\ u\ w) := (R^{F_1}\ (\pi_1\ u)\ (\pi_1\ w)) \wedge (R^{F_2}\ (\pi_2\ u)\ (\pi_2\ w))$;*

- *If $F[X] = F_1[X] + F_2[X]$ for two strictly positive operators F_1 and F_2, we put*

$$(R^F\ u\ w) := \mathsf{Cases}\ u\ w\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ u_1)\ (\mathsf{inl}\ w_1) \mapsto (R^{F_1}\ u_1\ w_1) \\ (\mathsf{inl}\ u_1)\ (\mathsf{inr}\ w_2) \mapsto \bot \\ (\mathsf{inr}\ u_2)\ (\mathsf{inl}\ w_1) \mapsto \bot \\ (\mathsf{inr}\ u_2)\ (\mathsf{inr}\ w_2) \mapsto (R^{F_2}\ u_2\ w_2); \end{cases}$$

- *If $F[X] = K \to F'[X]$ with K a constant type and F' a strictly positive operator, we put $(R^F\ u\ w) := \forall k\colon K.(R^{F'}\ (u\ k)\ (w\ k))$.*

The functorial extension of relations is a monotone operator, that is, it is itself functorial.

Lemma 3.3.2 *If $R_1, R_2\colon X \to Y \to \mathsf{Prop}$ and we have a proof*

$$x\colon X, y\colon Y, p\colon (R_1\ x\ y) \vdash r[x,y,p]\colon (R_2\ x\ y),$$

then there is a proof

$$u\colon F[X], w\colon F[Y], q\colon (R_1^F\ u\ w) \vdash (\mathsf{relextmon}\ [x,y,p]r\ u\ w\ q)\colon (R_2^F\ u\ w).$$

Proof By induction on the structure of F. □

The first step in the definition of extensional equality for coalgebras is the notion of a bisimulation relation. A bisimulation is a relation that holds between two elements of a coalgebra whenever they have the same immediate structure and their components are also in the relation.

Definition 3.3.3 *Let F be a strictly positive type operator and $\langle B, g\rangle$ an F-coalgebra. We say that $R\colon B \to B \to \mathsf{Prop}$ is a* bisimulation relation *if it satisfies the property*

$$(\mathsf{Bisimulation}\ R) := \forall b_1, b_2\colon B.(R\ b_1\ b_2) \to (R^F\ (g\ b_1)\ (g\ b_2)).$$

Remark 3.3.4 *As we have already mentioned, the notion of bisimulation is the dual of the notion of congruence for F-algebras: A congruence on an F-algebra $\langle A, f\rangle$ is a relation R on A such that $R^F \subseteq (f \times f)^{-1}(R)$, that is,*

$$\forall a_1, a_2\colon B.(R^F\ a_1\ a_2) \to (R\ (f\ a_1)\ (f\ a_2)).$$

See [HJ98] for a categorical treatment of induction and coinduction expressed in terms of congruence and bisimulation.

Extensional equality on a coalgebra is the largest bisimulation relation. In type theory, it could be defined impredicatively by

$$b_1 \approx b_2 := \exists R\colon B \to B \to \mathsf{Prop}.(\mathsf{Bisimulation}\ R) \wedge (R\ b_1\ b_2).$$

Alternatively, if we desire to avoid impredicativity, it can be introduced as a new relation by the following rules.

Definition 3.3.5 *Given a strictly positive operator F and an F-coalgebra $\mathcal{B} = \langle B, g\rangle$, the* extensional equality *on $\mathcal{B}$ is the relation $\approx_{\mathcal{B}}$ on B with the rules:*

formation

$$\frac{b_1, b_2 \colon B}{b_1 \approx_{\mathcal{B}} b_2 \colon \mathsf{Prop}}$$

introduction

$$\frac{R \colon B \to B \to \mathsf{Prop} \quad p \colon (\mathsf{Bisimulation}\ R) \quad q \colon (R\ b_1\ b_2)}{(\mathsf{bisim}\ R\ p\ q) \colon b_1 \approx_{\mathcal{B}} b_2}$$

elimination

$$\frac{h \colon b_1 \approx_{\mathcal{B}} b_2}{(\mathsf{bisim\text{-}elim}\ h) \colon (g\ b_1) \approx^F_{\mathcal{B}} (g\ b_2)}$$

reduction

$$\begin{aligned}&(\mathsf{bisim\text{-}elim}\ (\mathsf{bisim}\ R\ p\ q))\\ &\quad \rightsquigarrow (\mathsf{relextmon}\ [b_1, b_2, q](\mathsf{bisim}\ R\ p\ q)\ (g\ b_1)\ (g\ b_2)\ (p\ b_1\ b_2\ q)).\end{aligned}$$

The uniqueness part of the definition of a final coalgebra would require that bisimulation implies equality. However, we cannot require that two bisimilar terms are convertible, since this would make type-checking undecidable. Our approach consists instead of using bisimulation as a *book equality*, that is, an equivalence relation that is chosen to represent semantic equality of terms. That is, we work with a pair $\langle X, \equiv\rangle$, called a *setoid*, where X is a type and $\equiv$ an equivalence relation on it. All functions, operations, predicates, and relations on X will have to be invariant under $\equiv$. In Chapter 5 we develop these ideas in full. For the moment, it must be understood that, when we talk about coinductive types, we say "equality" meaning "bisimulation", and we take care that all operations are invariant under it. (For a different approach, see [PA93], where parametricity conditions are used to ensure finality.)

We illustrate the use of bisimulation relation by giving a proof of a formalized version of Proposition 3.1.2.

Proposition 3.3.6 *In the F-coalgebra $\mathcal{N} := \langle \mathbb{N}, \mathsf{ng}\rangle$, for all elements $n \colon \mathbb{N}$ such that $n \geq 7$, we have that*

- *if n is odd, then $n \approx_{\mathcal{N}} 7$;*
- *if n is even, then $n \approx_{\mathcal{N}} 8$.*

Proof We just need to find a bisimulation relation R on $\mathcal{N}$ and use it to prove the statement. Let us define

$$(R\ n\ m) := n \leq 7 \wedge m \leq 7 \wedge n \equiv m (\mathrm{mod}\ 2).$$

The proof of $(\mathsf{Bisimulation}\ R)$ is easily constructed by looking at the proof of Proposition 3.1.2. By the introduction rule of $\approx_{\mathcal{N}}$, we have that $(R\ n\ m)$ implies $n \approx_{\mathcal{N}} m$. Therefore, we just need to prove that

- if n is odd, then $(R\ n\ 7)$ and
- if n is even, then $(R\ n\ 8)$,

and these statements are trivial. □

In the proof of Theorem 3.2.6, we postponed the proof that the two functions $h_1^{\mathbb{T}}$ and $h_2^{\mathbb{T}}$ are inverse of each other. Now we have the tools to formulate precisely this result.

Lemma 3.3.7

$$\forall x\colon \mathbb{T}.x \approx_{\mathbb{T}} (h_2^{\mathbb{T}}\ (h_1^{\mathbb{T}}\ x))$$
$$\forall y\colon \mathbb{T}^{\nu}.y \approx_{\mathbb{T}^{\nu}} (h_1^{\mathbb{T}}\ (h_2^{\mathbb{T}}\ y))$$

Proof We prove the first statement with the help of the bisimulation relation

$$(R\ x_1\ x_2) := x_2 = (h_2^{\mathbb{T}}\ (h_1^{\mathbb{T}}\ x_1)).$$

It is easy to prove that R is a bisimulation, using the definitions of $h_1^{\mathbb{T}}$ and $h_2^{\mathbb{T}}$. Then the statement follows immediately from the introduction rule for $\approx_{\mathbb{T}}$

The proof of the second statement is similar. □

In a similar way we can complete the proof of Theorem 3.2.7.

3.4 Impredicative formalization of coinduction

Coinductive types may be defined in an alternative way, by following the Martin-löf explanation of what a type is (see the beginning of Chapter 2): In giving the rules of a type we specify freely what the introduction rule is, and the elimination rule should follow automatically by just stating that we can do with an element of the type anything that could be done with the premises of the introduction rule.

In the case of a coinductive type $C := \nu_X(F)$, the introduction rule is the corecursion principle or, more simply, the coiteration principle

$$\frac{A\colon s \quad f\colon A \to F[A] \quad a\colon A}{(\nu\text{-intro}\ A\ f\ a)\colon C},$$

where $(\nu\text{-intro}\ A\ f\ a) = (\nu\text{-it}\ f\ a)$. Consequently, the automatic elimination rule would be

$$\frac{P\colon C \to s \quad X\colon s, y\colon X \to F[X], x\colon X \vdash e[X,y,x]\colon (P\ (\nu\text{-intro}\ A\ y\ x))}{c\colon C \vdash (\nu\text{-elim}\ [X,y,x]e\ c)\colon (P\ c)}$$

and the reduction rule

$$(\nu\text{-elim}\ [X,y,x]e\ (\nu\text{-intro}\ A\ f\ a)) \rightsquigarrow e[A,f,a].$$

Theorem 3.4.1 *Let C be a type satisfying the rules ν-intro and ν-elim with the corresponding reduction rule, then C is isomorphic to $\nu_X(F)$.*

Proof We prove that C must satisfy the rules of $\nu_X(F)$, that is, C is a final F-coalgebra. The result follows from the fact that all final F-coalgebras are isomorphic. (Once again, we are considering the final coalgebras modulo bisimulation, to guarantee unicity.) Assume that the type C satisfies the rules ν-intro and ν-elim, we show that also ν-it and ν-out are valid. We put

$$(\nu\text{-it } [z]u\ t) := (\nu\text{-intro } T\ [z]u\ t)$$
$$(\nu\text{-out } c) := (\nu\text{-elim } [X, y, x](F[(\nu\text{-intro } X\ y)]\ (y\ x))\ c)$$

and we verify the reduction rule,

$$\begin{aligned}&(\nu\text{-out } (\nu\text{-it } [z]u\ t))\\ &= (\nu\text{-elim } [X, y, x](F[(\nu\text{-intro } X\ y)]\ (y\ x))\ (\nu\text{-intro } T\ [z]u\ t))\\ &\rightsquigarrow (F[(\nu\text{-intro } T\ [z]u)]\ ([z]u\ t)) = (F[(\nu\text{-it } [z]u)]\ u[t]).\end{aligned}$$

□

Notice that this isomorphism theorem states that C and $\nu_X(F)$ are isomorphic as F-coalgebras, not that they are isomorphic types. This means that there are function $h_1 \colon C \to \nu_X(F)$ and $h_1 \colon \nu_X(F) \to C$ such that

$$\forall c \colon C.(h_2\ (h_1\ c)) \approx c \qquad \text{and}$$
$$\forall x \colon \nu_X(F).(h_1\ (h_2\ x)) \approx x.$$

The equivalent statements with reflexive equality replacing coinductive equality,

$$\forall c \colon C.(h_2\ (h_1\ c)) = c \qquad \text{and}$$
$$\forall x \colon \nu_X(F).(h_1\ (h_2\ x)) = x.$$

need not be true.

In fact, if we try to prove the converse of the main step of the proof, we fail: We try to show that $\nu_X(F)$ satisfies the rules ν-it and ν-out. We put

$$(\nu\text{-intro } A\ f\ a) := (\nu\text{-it } f\ a)$$
$$(\nu\text{-elim } [X, y, x]e\ c) := e[\nu_X(F), \nu\text{-out}, c],$$

but unfortunately the second definition does not type-check. The type of $e[\nu_X(F), \nu\text{-out}, c]$ is $(P\ (\nu\text{-it } \nu\text{-out } c))$, while the type of $(\nu\text{-elim } [X, y, x]e\ c)$ should be $(P\ c)$. It is true that $(\nu\text{-it } \nu\text{-out } c) \approx c$, but not that $(\nu\text{-it } \nu\text{-out } c) = c$. Therefore we can prove the statement only for those predicates P that are invariant under bisimulation, that is, such that

$$\forall c_1, c_2 \colon \nu_X(F).c_1 \approx c_2 \to (P\ c_1) \to (P\ c_2).$$

The rule ν-intro gives an idea for a second order definition of coinductive types. We just define the type that is the conjunction of all possible premises for the rule ν-intro (see [LPM93]):

$$C \cong \sum_{X\,:\,\mathsf{Set}} (X \to F[X]) \times X.$$

Theorem 3.4.2 *The type* $C := \sum_{X:\mathsf{Set}}(X \to F[X]) \times X$ *is a weakly final* F*-coalgebra.*

Proof Let $c\colon C$. Then $c = \langle A, f, a\rangle$ for some $A\colon \mathsf{Set}$, $f\colon A \to F[A]$, and $a\colon A$. We put $(\nu\text{-}\mathsf{out}\ c) := (F[\lambda x\colon A.\langle A, f, x\colon]\ (f\ a))$.

Let T be a type, $z\colon T \vdash u[z]\colon F[T]$, and $t\colon T$. We put $(\nu\text{-}\mathsf{it}\ [z]u) := \langle T, [z]u, t\rangle$.

The reduction rule is easily verified:

$$\begin{aligned}&(\nu\text{-}\mathsf{out}\ (\nu\text{-}\mathsf{it}\ [z]u\ t)) = (\nu\text{-}\mathsf{out}\ \langle T, [z]u, t\rangle)\\ &= (F[\lambda x\colon T.\langle T, [z]u, x\rangle]\ ([z]u\ t)) = (F[\lambda x\colon T.(\nu\text{-}\mathsf{it}\ [z]u\ x)]([z]u\ t))\\ &\rightsquigarrow (F[\lambda x\colon T.(\nu\text{-}\mathsf{it}\ [z]u\ x)]\ u[t]) = (F[(\nu\text{-}\mathsf{it}\ [z]u)]\ u[t]).\end{aligned}$$

□

If we compare this second order definition of coinductive types with the standard second order definition of inductive types, as found in [BB85] and [Gir89], we see a clear duality:

$$\mu_X F[X] \cong \prod_{X:\mathsf{Set}} (F[X] \to X) \to X$$

$$\nu_X F[X] \cong \sum_{X:\mathsf{Set}} (X \to F[X]) \times X.$$

In [Geu92], coinductive types are impredicatively defined as

$$\prod_{X:\mathsf{Set}} (\prod_{Y:\mathsf{Set}} (Y \to F[Y]) \to Y \to X) \to X.$$

This definition becomes exactly our definition if we use the second order definition of Σ-types and Currying

$$\sum_{X:\mathsf{Set}} T[X] := \prod_{X:\mathsf{Set}} (\prod_{Y:\mathsf{Set}} T[Y] \to X) \to X.$$

Part II

Constructions in Type Theory

Chapter 4

Recursive Families of Inductive Types

In this chapter, we consider the problem of defining a family of inductive types. Each member of the family is the fixed point of a strictly positive type operator, but the corresponding family of type operators does not satisfy the syntactic strict positivity condition. First, we look at an extension of the notion of strict positivity that partially solves the problem. Then, we propose a solution that does not require an extension of type theory, based on a syntactic encoding of type operators. The content of this article appeared previously in [Cap00b].

4.1 Introduction

In type theory we can define a new inductive type by giving its constructors (or introduction rules). For example, we define the types of natural numbers, binary trees, and lists over a type A as

$$\mathbb{N}\colon \quad \frac{}{0\colon \mathbb{N}} \quad \frac{n\colon \mathbb{N}}{(\mathsf{S}\ n)\colon \mathbb{N}}, \qquad \mathbb{T}_2\colon \quad \frac{}{\mathsf{leaf}\colon \mathbb{T}_2} \quad \frac{x_1\colon \mathbb{T}_2 \quad x_2\colon \mathbb{T}_2}{\mathsf{node}(x_1, x_2)\colon \mathbb{T}_2}$$

$$\text{and} \quad \mathsf{List}(A)\colon \quad \frac{}{\mathsf{nil}_A\colon \mathsf{List}(A)} \quad \frac{a\colon A \quad l\colon \mathsf{List}(A)}{\mathsf{cons}_A(a, l)\colon \mathsf{List}(A)},$$

respectively.

Consider the family $T\colon \mathbb{N} \to \mathsf{Set}$ (Set indicates the type of all small types,

or sets) of inductive types indexed on the natural numbers:

$$\begin{array}{lcccc} T_0\colon & & & & \dfrac{}{c_0^0\colon T_0} \\[2ex] T_1\colon & & \dfrac{}{c_1^1\colon T_1} & & \dfrac{x\colon T_1}{c_0^1(x)\colon T_1} \\[2ex] T_2\colon & \dfrac{}{c_2^2\colon T_2} & \dfrac{x\colon T_2}{c_1^2(x)\colon T_2} & & \dfrac{x_1\colon T_2 \quad x_2\colon T_2}{c_0^2(x_1,x_2)\colon T_2} \\[2ex] \vdots & & & & \end{array} \tag{4.1}$$

Every new type in the family is defined by a new constant and by the constructors of the previous type in the hierarchy with an extra recursive argument. Intuitively, T_n is the type of trees with branching degree at most n. In the standard formulation of inductive types, this definition is not allowed: The constructors and their types must be given directly at the moment of definition of the inductive type, whereas the number of constructors of T_n and their types are defined by recursion on n.

Families of this kind have not only theoretical interest. They arise in the course of formalization of mathematics in a proof tool. I first encountered them when I was working on the formalization of Universal Algebra in Coq (see [Cap99], [Cap00a], and Chapter 8 of this thesis). The family of term algebras on the type of signatures is one of them. The type of single-sorted signatures is $\mathsf{Sig} := \mathsf{List}(\mathbb{N})$. Given a signature $\sigma := [a_1, \ldots, a_n]$, the type of terms over σ is defined by

$$\mathsf{Term}_\sigma\colon \quad \frac{t_{11}\colon \mathsf{Term}_\sigma \ \cdots \ t_{1a_1}\colon \mathsf{Term}_\sigma}{(\mathsf{f}_1\ t_{11}\ \cdots\ t_{1a_1})\colon \mathsf{Term}_\sigma} \quad \cdots \quad \frac{t_{n1}\colon \mathsf{Term}_\sigma \ \cdots \ t_{na_n}\colon \mathsf{Term}_\sigma}{(\mathsf{f}_n\ t_{n1}\ \cdots\ t_{na_n})\colon \mathsf{Term}_\sigma}.$$

(One of the a_i's must be 0, so that Term_σ is nonempty.) We cannot obtain the family $\mathsf{Term}\colon \mathsf{Sig} \to \mathsf{Set}$ directly with an inductive definition, because the number and arity of the constructors depend on the signature σ: They are not fixed for the whole family. The situation is even more complicated when we consider many-sorted signatures, that require families of mutually inductive types. In [Cap99] we used Martin-Löf's $\mathbb{W}$-types to solve this instance of the problem. Here we formulate the general problem, we show that $\mathbb{W}$-types still provide a good model, but also propose a better solution (which, however, requires an extension of type theory). You can see the details of its application to many-sorted algebras in [Cap00a]. See also Chapter 8 of this thesis.

This chapter offers several different solutions to the problem, each with its advantages and disadvantages.

In Section 4.3 we formulate the general problem: We propose an extension of the notion of strictly positive operator, which is used to determine the admissibility of inductive definitions, using *positive type pointers*—that is, terms that specify the positive occurrence of parameters in recursive definitions.

In Section 4.4 we represent inductive types using Martin-Löf's type constructor for wellorderings ($\mathbb{W}$-types) (see [ML82, ML84] and chapter 15 of [NPS90]), extending the work by Dybjer [Dyb97]. This solution has the disadvantage that structurally equal elements of a $\mathbb{W}$-type are not always convertible, thus making the $\mathbb{W}$-type representation only *extensionally* isomorphic to the desired inductive type.

Alternatively, we can exploit the extension of the positivity condition implemented in the system Coq and described by Gimenez in [Gim98]. It allows an inductive definition to inherit a positive occurrence of a type variable from another inductive definition. To use this construction in our case, we need to give a translation of our recursive family of operators into an inductive family. In Section 4.5 we give such translation and we use it to solve our problem.

Finally, in Section 4.6 we use the two level approach (see [BRB95], [Bou97], [How88] and [BW00]): Positivity is a metapredicate; that is, it is not expressed inside type theory but it is an external syntactic property of type operators. This means that we cannot reason about positive operators and inductive definitions inside type theory. We internalize it by defining a type-theoretic predicate Positive expressing the metaproperty. We define a type of codes for inductive types and associate a code to every proof of an instance of the predicate Positive. We define a function that associates a type to every code. Then, we solve our problem by first, constructing a family of positive operators by recursion; second, proving their positivity inside type theory; third, obtaining the corresponding family of codes; finally, instantiating the codes to types. This last method has been completely formalized in the proof assistant Coq [BBC+99].

4.2 Inductive types

The results of this chapter hold in a type theory that is at least as expressive as the Pure Type System $\lambda P\bar{\omega}$ (see [Bar92]): There are two sorts of types, Set for small types and Type for large types. Sort Set is an element of Type. Moreover, we have sum and Σ types, which can be considered as special cases of inductive types, defined in Chapter 2. In $\lambda P\bar{\omega}$ every small type $T\colon \mathsf{Set}$ has an isomorphic version in Type. For simplicity, we identify the two; in other words, we consider Set and Type as the first two steps in a cumulative hierarchy of type universes. When we write type expressions that mix the two sorts, as $T \times \mathsf{Set}$ or $T + \mathsf{Set}$, the version of T in Type is used. Note, however, that if Set is impredicative (for example, if we work in the Calculus of Constructions) not all elements of Set can have a representation in Type, because this would lead to Girard's paradox (see [Coq86],[Coq90], and [Coq94]). Only if impredicativity was not used in the definition of the type, we can consider it as an element of Type. When we use small types in Type constructions, we implicitly assume that this condition is satisfied (as supported by the Coq implementation).

We use the notation $t[x]$ to denote a term t in which a variable x may occur. Thus t and $t[x]$ denote the same term, but in the second expression we stress the dependence on x. Do not confuse this notation with $(f\ x)$, which denotes

the application of a function f to x. If s is a term of the same type as x, $t[s]$ denotes the result of the substitution $t[x := s]$.

We briefly recall the notion of inductive type presented in Chapter 2, Section 2.11. In *extensional* type theory inductive types can be implemented as fixed points of type operators (see [Men87]). We are working in *intensional* type theory, in which inductive types are recursively defined by constructors. Following [CP90], [PM93], [PPM90] and [Ste99] an inductive type I is defined by a list of constructors:

$$\begin{array}{l}\mathsf{Inductive}\ I\ [x_1\colon P_1;\ldots;x_h\colon P_h]\colon (y_1\colon Q_1;\ldots;y_m\colon Q_m)\mathsf{Set} :=\\ \quad c_1\colon (z_{11}\colon R_{11})\cdots(z_{1k_1}\colon R_{1k_1})(I\ t_{11}\ \cdots\ t_{1m})\\ \qquad\vdots\\ \quad c_n\colon (z_{n1}\colon R_{n1})\cdots(z_{nk_n}\colon R_{nk_n})(I\ t_{n1}\ \cdots\ t_{nm})\\ \mathsf{end}.\end{array} \tag{4.2}$$

where I does not occur free in the Ps, Qs and ts and occurs only strictly positively in the Rs. The variables $x_1, \ldots, x_h$ are general parameters of I (such as the parameter X in $\mathsf{List}(X)$). See one of the cited references, Chapter 4 of the Coq manual [BBC+99], or Section 2.11 of this thesis for the definition of strict positivity and for the other rules.

If the types of the constructors do not use dependent product—that is, they are in the form $R_{i1} \to \cdots \to R_{ik_i} \to I$—we can use the alternative formulation of inductive types as fixed points of strictly positive type operators (see, for example, [Dyb97]). It is less intuitive but simpler for theoretical purposes, so we adopt it. Every strictly positive operator $X\colon \mathsf{Set} \vdash \Phi[X]\colon \mathsf{Set}$ has a functorial extension, which, for $X, Y\colon \mathsf{Set}$, maps every $f\colon X \to Y$ to a function $\Phi[f]\colon \Phi[X] \to \Phi[Y]$; preserving identities and composition (see [CP90] and [PR99]). This condition is sufficient to formulate the rules for inductive types (Matthes [Mat99] gives an extension of *system F* in which this is the only condition required for inductive types). In the next sections we consider extensions of the positivity condition that still have the functorial property. The rules for inductive types are then the same as in the following definition, with the corresponding property replacing *strictly positive*.

Definition 4.2.1 *Let $X\colon \mathsf{Set} \vdash \Phi[X]\colon \mathsf{Set}$ be a strictly positive operator. The* inductive type $\mu_X(\Phi)$ *is defined by the following rules (where we write I for $\mu_X(\Phi)$):*

formation

$$I\colon \mathsf{Set}$$

introduction

$$\frac{y\colon \Phi[I]}{(\mu\text{-intro}\ y)\colon I}$$

elimination

$$\frac{x\colon I \vdash (P\ x)\colon \mathsf{Set} \quad z\colon \Phi[(\Sigma\ I\ P)] \vdash w\colon (P\ (\mu\text{-intro}\ (\Phi[\pi_1]\ z)))}{(\mu\text{-ind}\ [z]w)\colon (x\colon I)(P\ x)}$$

conversion

$$(\mu\text{-ind } [z]w \ (\mu\text{-intro } y)) \rightsquigarrow w[(\Phi[[x]\langle x, (\mu\text{-ind } [z]w \ x)\rangle] \ y)]$$

We use this formulation to define our inductive types, since they are all non-dependent, but we use the notation of Formula 4.2 when it is intuitively clearer and when we need to define types whose constructors belong to dependent product types.

If the elimination predicate P is a constant type T, we obtain the recursion principle; if, furthermore, the recursion term u does not depend on the induction arguments, we obtain the iteration principle:

$$\frac{T\colon \mathsf{Set} \quad z\colon \Phi[I \times T] \vdash w\colon T}{(\mu\text{-rec } [z]w)\colon I \to T} \quad \text{and} \quad \frac{T\colon \mathsf{Set} \quad z\colon \Phi[T] \vdash u\colon T}{(\mu\text{-it } [z]u)\colon I \to T}.$$

It is well known that the recursion and iteration principles are equivalent, whereas the full induction principle is a proper extension of them (see, for example, [Geu92] or [PR99]).

The types of natural numbers, binary trees, and lists over a type A can be defined as $\mathbb{N} := \mu_X(\mathbb{N}_1 + X)$, where $\mathbb{N}_1$ is the type with only one element 0_1; $\mathbb{T}_2 := \mu_X(\mathbb{N}_1 + X \times X)$; and $\mathsf{List}(A) := \mathbb{N}_1 + A \times X$, respectively. Their constructors can be defined in terms of the single constructor μ-intro:

$$\begin{array}{rclrcl} 0 & := & (\mu\text{-intro } (\mathsf{inl}\ 0_1)), & S & := & [n](\mu\text{-intro } (\mathsf{inr}\ n)); \\ \mathsf{leaf} & := & (\mu\text{-intro } (\mathsf{inl}\ 0_1)), & \mathsf{node} & := & [x_1, x_2](\mu\text{-intro } (\mathsf{inr}\ \langle x_1, x_2\rangle)); \\ \mathsf{nil} & := & (\mu\text{-intro } (\mathsf{inl}\ 0_1)), & \mathsf{cons} & := & [a, l](\mu\text{-intro } (\mathsf{inr}\ \langle a, l\rangle)). \end{array}$$

The problem that we consider here is: Given a family of type operators $\Phi\colon A \to (\mathsf{Set} \to \mathsf{Set})$ such that every element of it is strictly positive, can we construct the corresponding family of inductive types? Observe that it is not possible to characterize such families in a decidable way. In fact, for every function $f : \mathbb{N} \to \mathbb{N}$ we can associate such a family:

$$\begin{array}{l} \Phi\colon \mathbb{N} \to (\mathsf{Set} \to \mathsf{Set}) \\ (\Phi \ n \ X) = \begin{cases} X & \text{if } (f \ n) = 0 \\ X \to X & \text{otherwise.} \end{cases} \end{array}$$

Deciding whether every element of this family is strictly positive is equivalent to deciding whether f is constantly 0. Since, in type theory, every primitive recursive function on the natural numbers is definable, we would be able to decide whether any such function is constantly 0, which is notoriously impossible.

The following section gives a decidable characterization of some of these families, which is wide enough for the examples that we are considering.

4.3 Inductive families by strong elimination

This section gives the first solution of the problem. We propose an extension of type theory with a stronger rule for inductive definitions.

Strong elimination is the elimination rule for inductive types in which the elimination predicate is allowed to be *big*; that is, we can have an elimination predicate $x\colon I \vdash (P\ x)\colon \mathsf{Type}$. If Set is impredicative, strong elimination results in inconsistency (see [Coq86], [Coq90], and [Coq94]). Nevertheless, it can still be admitted if the inductive type I is defined without the use of impredicativity—that is, as already mentioned, if there is a type in Type isomorphic to it. In such a case we allow strong elimination. This form of strong elimination is supported in Coq. We use strong elimination only in the form of iteration over the type Set: If $I = \mu_X(\Phi[X])$,

$$\frac{Z\colon \Phi[\mathsf{Set}] \vdash W[Z]\colon \mathsf{Set}}{(\mu\text{-}\mathsf{it}\ [Z]W)\colon I \to \mathsf{Set}}$$

with the corresponding reduction rule

$$(\mu\text{-}\mathsf{it}\ [Z]W\ (\mu\text{-}\mathsf{intro}\ t)) \rightsquigarrow W[(\Phi[(\mu\text{-}\mathsf{it}\ [Z]W)]\ t)].$$

We recast Example 4.1 as a family of inductive types defined by a family of type operators Ψ_n, such that the type T_n is the inductive type associated to the operator Ψ_n.

$$\begin{array}{rcl} X\colon \mathsf{Set} & \vdash & \Psi[X]\colon \mathbb{N} \to \mathsf{Set} \\ & & \Psi_0[X] := \mathbb{N}_1 \\ & & \Psi_{(\mathsf{S}\ n)}[X] := \mathbb{N}_1 + X \times \Psi_n \end{array}$$

To be completely formal, we must use strong elimination on the type of natural numbers $\mathbb{N}$. Remember that $\mathbb{N} = \mu_Y.\mathbb{N}_1 + Y$. The principle of strong iteration is then

$$\frac{Z\colon \mathbb{N}_1 + \mathsf{Set} \vdash W[Z]\colon \mathsf{Set}}{(\mu\text{-}\mathsf{it}\ [Z]W)\colon \mathbb{N} \to \mathsf{Set}}.$$

We use it to define formally the family of operators Ψ:

$$\frac{\begin{array}{rcl} X\colon \mathsf{Set}, Z\colon \mathbb{N}_1 + \mathsf{Set} & \vdash & W[X,Z]\colon \mathsf{Set} \\ & & := \mathsf{Cases}\ Z\ \mathsf{of}\ \left\{\begin{array}{l} (\mathsf{inl}\ \_\ ) \mapsto \mathbb{N}_1 \\ (\mathsf{inr}\ R) \mapsto \mathbb{N}_1 + X \times R \end{array}\right. \end{array}}{X\colon \mathsf{Set} \vdash \Psi[X] := (\mu\text{-}\mathsf{it}\ [Z]W)\colon \mathbb{N} \to \mathsf{Set}}. \quad (4.3)$$

The desired family of inductive types is now specified by $T_n := \mu_X(\Psi_n)$. Unfortunately, this is not an allowed definition, since Ψ does not satisfy the strict-positivity condition: Although Ψ_n reduces to a strictly positive operator for each numeral n, Ψ_x does not if $x\colon \mathbb{N}$ is a variable. Therefore, we cannot define the family $T\colon \mathbb{N} \to \mathsf{Set}$, even if every member of it is individually definable.

The case of term algebras over single-sorted signatures is similar. The operator associated to a signature $\sigma := [a_1, \ldots, a_n]$ is $\Psi_\sigma[X] := X^{a_1} + \cdots + X^{a_n}$. We can define, first, a single component $(n\colon \mathbb{N}, X\colon \mathsf{Set} \vdash X^n\colon \mathsf{Set})$ by strong elimination on the natural numbers, and then, Ψ_σ by strong elimination on $\mathsf{List}(\mathbb{N})$. The type of terms associated with the signature σ is then $\mathsf{Term}_\sigma := \mu_X(\Psi_\sigma)$. As in the previous example, this definition is not allowed in the standard implementation of inductive types, because Ψ_σ does not satisfy the strict-positivity condition.

Our purpose is to find ways to define families of inductive types in type theory. The first step is a formal description of the problem—that is, an abstract characterization of the definitions we are looking for. If we consider the first of the preceding examples, we see that the reason why every single element of the family is strictly positive is that, in the recursive step of the definition, not only the type parameter X, but also the recursive call Ψ_n (or R in the formalized version) occurs only strictly positively. It is enough to require that all such recursive calls occur strictly positively. Note, however, that, in the formal version of the definition, the recursive call is Z, not R, which is just a bound variable in the Cases construction. It doesn't mean anything to say that Z occurs positively and the variable R does not actually occur in W (being bound). So we need a finer notion than strict positivity. The following concept of *positive type pointer* solves the problem. To understand it intuitively, consider a generic premise for a definition by strong elimination $(X\colon \mathsf{Set}, Z\colon \Theta[\mathsf{Set}] \vdash U[X,Z]\colon \mathsf{Set})$ where Θ is a strictly positive operator. We want to define, first, the family of type operators $(X\colon \mathsf{Set} \vdash \Psi[X] := (\mu\text{-}\mathsf{it}\ [Z]U)\colon \mu_Y(\Theta[Y]) \to \mathsf{Set})$ and then the family of inductive types $(x\colon \mu_Y(\Theta[Y]) \vdash \mu_X((\Psi\ x))\colon \mathsf{Set})$. Imagine Z represented as a tree structure whose leaves are types representing recursive calls. We must require that each occurrence of a term pointing to such a leaf occurs only strictly positively in U.

In our specific example, Z has type $\mathbb{N}_1 + \mathsf{Set}$. We can think of Z as being a tree in one of two forms: A node inl with a single edge to a leaf of type $\mathbb{N}_1$, or a node inr with a single edge to a leaf of type Set. In the first case there are no recursive calls, in the second case the leaf is a recursive call. The term $W[Z]$ analyses the structure of Z and uses its leaves. In the second case, that is, when $Z = (\mathsf{inr}\ R)$ for some $R\colon \mathsf{Set}$, W produces the set $\mathbb{N}_1 + X \times R$. Since R occurs positively in this formula, we say that W is a positive type pointer.

We formalize the notions of strongly positive type operator and positive type pointer in Definition 4.3.1. But first, we give a depiction of type pointers.

A positive type pointer.

We show how a type pointer can be constructed for each of the type constructors that build new strongly positive operators.

If Z is in a product type (clause 3 in Definition 4.3.1), it is a pair, represented by a binary node with a subtree for each branch.

Clause 3: Z in a product type.

A positive type pointer first chooses one of the components and then uses a positive type pointer for that component.

If Z is in a function type (clause 5), the situation is similar, but the number of components is equal to the cardinality of the domain type (possibly infinite).

Clause 5: Z in a function type.

If Z is in a sum type (clause 4), then it is in one of the two forms specified by the component types.

Clause 4: Z in a sum type.

A positive type pointer must take into account both possibilities, so it prescribes a type pointer for each of the two components. Therefore, the picture for clause 3 shows two positive type pointers corresponding to the two components, whereas the picture for clause 4 shows only one positive type pointer that consists of two components.

The picture for clause 8 shows how a positive type pointer is used in a recursive definition of a family of strongly positive operators.

Clause 8: a strongly positive family.

The term $W[X, Z]$ is the iterator of the recursive definition. It contains some direct occurrences of the variable X and some recursive calls, here indicated by the leaves T_i and T_j of the iteration variable Z. When T_i and T_j are replaced with the values of the recursive call, new occurrences of X appear. The requirement that $W[X, Z]$, besides being strongly positive in X, is also a positive type pointer in Z causes all the new occurrences of X to be strictly positive.

Definition 4.3.1 *We simultaneously define the notions of* strongly positive operator *and* positive type pointer. *Let*

$$X\colon \mathsf{Set} \vdash \Phi[X]\colon \mathsf{Set} \quad \textit{and} \quad Z\colon \Phi[\mathsf{Set}] \vdash U[Z]\colon \mathsf{Set}.$$

We say that Φ *is a* strongly positive operator *in* X *and that* U *is a* positive type pointer *for* Φ *in* Z *if they have been constructed according to the following clauses.*

1. *If* K *is a type that does not depend on* X *(that is,* X *does not occur free in* K*), then* $\Phi[X] = K$ *is strongly positive and* $Z\colon K \vdash K\colon \mathsf{Set}$ *is a positive type pointer for* Φ*.*

2. *If* $\Phi[X] = X$ *then* Φ *is strongly positive and* $Z\colon \mathsf{Set} \vdash Z\colon \mathsf{Set}$ *is a positive type pointer for* Φ*.*

3. *If* $\Phi[X] = \Phi_1[X] \times \Phi_2[X]$ *and* Φ_1 *and* Φ_2 *are strongly positive, then* Φ *is strongly positive and, if* $(Z_1\colon \Phi_1[\mathsf{Set}] \vdash U_1[Z_1]\colon \mathsf{Set})$ *and* $(Z_2\colon \Phi_2[\mathsf{Set}] \vdash U_2[Z_2]\colon \mathsf{Set})$ *are positive type pointers for* Φ_1 *and* Φ_2*, respectively, then*

$$Z\colon \Phi_1[\mathsf{Set}] \times \Phi_2[\mathsf{Set}] \vdash (U_1\ (\pi_1\ Z))\colon \mathsf{Set} \quad \textit{and}$$
$$Z\colon \Phi_1[\mathsf{Set}] \times \Phi_2[\mathsf{Set}] \vdash (U_2\ (\pi_2\ Z))\colon \mathsf{Set}$$

   are positive type pointers for Φ*.*

4. *If* $\Phi[X] = \Phi_1[X] + \Phi_2[X]$ *and* Φ_1 *and* Φ_2 *are strongly positive, then* Φ *is strongly positive and if* $Z_1\colon \Phi_1[\mathsf{Set}] \vdash U_1[Z_1]\colon \mathsf{Set}$ *and* $Z_2\colon \Phi_2[\mathsf{Set}] \vdash U_2[Z_2]\colon \mathsf{Set}$ *are positive type pointers for* Φ_1 *and* Φ_2*, respectively, then*

$$Z\colon \Phi_1[\mathsf{Set}] + \Phi_2[\mathsf{Set}] \vdash \mathsf{Cases}\ Z\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ Z_1) \mapsto U_1[Z_1] \\ (\mathsf{inr}\ Z_2) \mapsto U_2[Z_2] \end{cases} : \mathsf{Set}$$

is a positive type pointer for Φ.

5. *If* $\Phi[X] = K \to \Phi'[X]$, *where* K *is a type that does not depend on* X, *and* Φ' *is a strongly positive type operator, then* Φ *is also strongly positive and if* $Z' \colon \Phi'[\mathsf{Set}] \vdash U'[Z'] \colon \mathsf{Set}$ *is a positive type pointer for* Φ', *then, for every* $k \colon K$,
$$Z \colon K \to \Phi'[\mathsf{Set}] \vdash U'[(Z\ k)] \colon \mathsf{Set}$$
is a positive type pointer for Φ.

6. *Suppose* $t \colon T_1 + T_2$ *for types* T_1 *and* $T_2 \colon \mathsf{Set}$. *If*
$$\Phi[X] = \mathsf{Cases}\ t\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ x_1) \mapsto \Phi_1[x_1, X] \\ (\mathsf{inl}\ x_2) \mapsto \Phi_2[x_2, X] \end{cases}$$
with X *not free in* t *and* Φ_1 *and* Φ_2 *are strongly positive, then* Φ *is also strongly positive. (If we consider sum types as instances of inductive types, then this is just an instance of clause 8.)*

7. *If* $(Y_1, \ldots, Y_n \colon \mathsf{Set} \vdash \Psi[Y_1, \ldots, Y_n] \colon Set)$ *is a strongly positive operator in* $Y_1, \ldots, Y_n$, $(X \colon \mathsf{Set} \vdash \Phi[X] \colon \mathsf{Set})$ *is a strongly positive operator in* X, *and* $(Z \colon \Phi[\mathsf{Set}] \vdash U_i[Z] \colon \mathsf{Set})$ *is a positive type pointer for* Φ *in* Z, *for* $1 \leq i \leq n$, *then*
$$Z \colon \Phi[\mathsf{Set}] \vdash \Psi[U_1[Z], \ldots, U_n[Z]] \colon \mathsf{Set}$$
is also a positive type pointer for Φ *in* Z.

8. *If* $(Y \colon \mathsf{Set} \vdash \Theta[Y] \colon \mathsf{Set})$ *is a strongly positive type operator,* $I = \mu_Y(\Theta)$, *and* $(X \colon \mathsf{Set}, Z \colon \Theta[\mathsf{Set}] \vdash W[X, Z] \colon \mathsf{Set})$ *is a positive type pointer for* Θ *in* Z *and is strongly positive in* X, *then every element of the family* $(X \colon \mathsf{Set} \vdash \Psi[X] := (\mu\text{-}\mathsf{it}\ [Z]W[X, Z]) \colon I \to \mathsf{Set})$ *is strongly positive; that is, for every* $i \colon I$, $(X \colon \mathsf{Set} \vdash (\Psi[X]\ i) \colon \mathsf{Set})$ *is a strongly positive type operator.*

We do not include a definition of positive type pointer corresponding to the strongly positive operator obtained in clause 8. This further complication is not necessary to define the families of types in which we are interested.

The definition of strongly positive type operator coincides with the definition of strictly positive type operator but for the last clause, which allows the definition of families of strongly positive type operators by recursion, using a positive type pointer as the recursion term.

Lemma 4.3.2 *Every strictly positive operator is strongly positive.*

For example, consider the family of type operators defined in Formula (4.3). We want to prove that $X \colon \mathsf{Set} \vdash (\Psi[X]\ n)$ is strongly positive for every $n \colon \mathbb{N}$. We show this in detail to illustrate how the definition work. Since $\mathbb{N} = \mu_Y(\Theta[Y])$ with $\Theta[Y] = \mathbb{N}_1 + Y$, we can use clause 8 of Definition 4.3.1. We have to prove that $X \colon \mathsf{Set}, Z \colon \mathbb{N}_1 + \mathsf{Set} \vdash W[X, Z] \colon \mathsf{Set}$ is a strongly positive type operator with respect to X and a positive type pointer for Θ with respect to Z.

First, we prove that W is a strongly positive operator in X.

$$W[X,Z] = \mathsf{Cases}\ Z\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ \_\ ) \mapsto \mathbb{N}_1 \\ (\mathsf{inr}\ R) \mapsto \mathbb{N}_1 + X \times R \end{cases}$$

By clause 6, this is a strongly positive operator in X if $(X\colon \mathsf{Set} \vdash \mathbb{N}_1)$ and $(R\colon \mathsf{Set}, X\colon \mathsf{Set} \vdash \mathbb{N}_1 + X \times R)$ are strongly positive operators in X. By clause 1 $(X\colon \mathsf{Set} \vdash \mathbb{N}_1)$ is strongly positive in X. We use clause 4 to show that $(R\colon \mathsf{Set}, X\colon \mathsf{Set} \vdash \mathbb{N}_1 + X \times R)$ is strongly positive in X. We have to prove that $(X\colon \mathsf{Set} \vdash \mathbb{N}_1\colon \mathsf{Set})$ and $(R\colon \mathsf{Set}, X\colon \mathsf{Set} \vdash X \times R)$ are strongly positive in X. The first operator is strongly positive by clause 1. For the second we use clause 3; then we have to prove that $(X\colon \mathsf{Set} \vdash X\colon X)$ and $(R\colon \mathsf{Set}, X\colon \mathsf{Set} \vdash R\colon \mathsf{Set})$ are strongly positive in X. The first is by clause 2 and the second by clause 1. So we concluded the proof that W is a strongly positive type operator in X.

Next, we have to prove that $W[X,Z]$ is a positive type pointer for Θ in Z. Since $\Theta[Y] = \Theta_1[Y] + \Theta_2[Y]$ with $\Theta_1[Y] := \mathbb{N}_1$ and $\Theta_2[Y] := Y$, we use clause 4. We have that

$$W[X,Z] = \mathsf{Cases}\ Z\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ Z_1) \mapsto U_1[Z_1] \\ (\mathsf{inr}\ Z_2) \mapsto U_2[Z_2] \end{cases}$$

with

$$Z_1\colon \Theta_1[\mathsf{Set}] = \mathbb{N}_1 \vdash U_1[Z_1] := \mathbb{N}_1\colon \mathsf{Set}$$

and

$$Z_2\colon \Theta_2[\mathsf{Set}] = \mathsf{Set} \vdash U_2[Z_2] := X \times Z_2\colon \mathsf{Set}.$$

We have to prove that these two terms are positive type pointers for Θ_1 in Z_1 and for Θ_2 in Z_2, respectively. The first statement is immediate from clause 1. For the second we use clause 7. In fact, we have that $U_2[Z_2] = \Psi[U_2'[Z_2]]$ with $\Psi[Y] := X \times Y$ and $U_2'[Z_2] = Z_2$. We have to prove that Ψ is a strongly positive operator in Y and U_2' is a positive type pointer for Θ_2 in Z_2. The first statement follows from clause 3, using clauses 1 and 2 for the components. The second statement is immediate from clause 2.

It follows that, in the system extended by Definition 4.3.1, we can define the family of inductive types

$$T := [n\colon \mathbb{N}]\mu_X(\Psi[X]\ n)\colon \mathbb{N} \to \mathsf{Set}.$$

The verification that W is a strongly positive operator was quite tedious. But if the notions of strong positivity and positive type operator are implemented in a proof assistant, there is no need for the user to do this verification: the user just gives the definition of the inductive family and the machine verifies that the condition holds.

Lemma 4.3.3 *The properties of being a strongly positive type operator and of being a positive type pointer are recursively decidable.*

Proof They are syntactic conditions that can be checked by just looking at the syntactic structure of the terms. □

The chief property of the notion of positive type operator consists in the fact that, when we substitute its variable with a family of strongly positive operators, we obtain a strongly positive operator. Let us make this notion precise.

Definition 4.3.4 *Let $T\colon \mathsf{Set}$ be a type and $\mathcal{P}$ be a metaproperty of terms of T, that is, $\mathcal{P}$ is not a predicate of type $T \to \mathsf{Prop}$, but an external predicate on the expressions that have type T. Let Φ be a strongly positive type operator. We say that a term $t\colon \Phi[T]$ is a* Φ-family of $\mathcal{P}$-terms *if the T components of t satisfy $\mathcal{P}$. To clarify the notion, if $\mathcal{P}$ were an internal predicate, we could formalize the notion by saying that t is a Φ-family of $\mathcal{P}$-terms if*

$$t = (\Phi[\pi_1]\ t')$$

for some $t'\colon \Phi[\Sigma x\colon X.(\mathcal{P}\ x)]$. For $\mathcal{P}$ a metaproperty, the notion can be defined by recursion on the structure of Φ.

We use this notion to prove metaproperties of inductive types.

Lemma 4.3.5 *Let $I = \mu_X(\Phi[X])$ be an inductive type and $\mathcal{P}$ a metaproperty on I closed under convertibility. If, for every Φ-family of $\mathcal{P}$-terms $t\colon \Phi[I]$, $\mathcal{P}$ holds for (μ-intro t); then $\mathcal{P}$ holds for every closed term of I.*

Here, we are specifically interested in the notion of a Φ-family of strictly positive type operators. If an operator reduces to a strictly positive one, we still call it strictly positive. In other words we work with the closure of the notion of strict positivity under convertibility. The reason, and the motivation for the notion of positive type pointer, is the following result.

Lemma 4.3.6 *Let Φ by a strictly positive type operator and*

$$Z\colon \Phi[\mathsf{Set}] \vdash U[Z]\colon \mathsf{Set}$$

be a positive type pointer for Φ in Z. Let $(X\colon \mathsf{Set} \vdash \Xi[X]\colon \Phi[\mathsf{Set}])$ be a Φ-family of strictly positive type operators. Then

$$X\colon \mathsf{Set} \vdash U[\Xi[X]]\colon \mathsf{Set}$$

is a strictly positive operator.

Proof By induction on the proof that U is a positive type pointer. □

Definition 4.3.1 does not add new inductive types to the system, but simply allows us to collect types in new families.

Theorem 4.3.7 *Every closed type $\mu_X(\Phi)$ definable by Definition 4.3.1 is definable by Definition 4.2.1 also.*

Proof We must prove that every strongly positive operator $X\colon \mathsf{Set} \vdash \Phi[X]\colon \mathsf{Set}$ in which no free variable except X occurs, is strictly positive (or, better, reduces to a strictly positive one). The proof is by induction on the definition of strongly positive operators. All clauses except clause 6 and clause 8 correspond to clauses in the definition of strictly positive operator. Therefore we just need to prove the statement for operators obtained by those two clauses.

Assume Φ has been obtained by clause 6. That means that Φ has the form

$$\Phi[X] = \mathsf{Cases}\ t\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ x_1) \mapsto \Phi_1[x_1, X] \\ (\mathsf{inr}\ x_2) \mapsto \Phi_2[x_2, X]. \end{cases}$$

Since the only free variable in $\Phi[X]$ is X and X is not allowed to occur free in t, t is a closed term and reduces to one of the two forms $(\mathsf{inl}\ t_1)$ or $(\mathsf{inr}\ t_2)$. In the first case we have that $\Phi[X] \rightsquigarrow \Phi_1[t_1, X]$, in the second $\Phi[X] \rightsquigarrow \Phi_2[t_2, X]$. In the first case, since $\Phi[x_1, X]$ is strongly positive in X, also $\Phi[t_1, X]$ is strongly positive in X, and the proof of this fact has the same number of steps. Therefore we can assume, by induction hypothesis, that $\Phi[t_1, X]$ reduces to a strictly positive operator. The second case is identical.

Assume that Φ has been obtained by clause 8—that is, $\Phi = (\Psi\ a)$, where Ψ is as in clause 8 and a is a closed term of type I. We assume that a is in normal form (otherwise we normalize it). We prove that $(\Psi\ a)$ is strictly positive by induction on the set of closed terms of I in normal form, that is, we apply Lemma 4.3.5. (Note that this is structural induction external to type theory, and not an internal application of the elimination rule. This explains why a must be a *closed* term for it to work.) Suppose $a = (\mu\text{-}\mathsf{intro}\ b)$ with $b\colon \Theta[I]$ closed. By induction hypothesis we assume that $(\Theta[\Phi[X]]\ b)$ is a Θ-family of strictly positive type operators in X. Then

$$\begin{array}{rcl} (\Psi\ a) & = & (\Psi\ (\mu\text{-}\mathsf{intro}\ b)) \\ & = & (\mu\text{-}\mathsf{it}\ [Z]W[X, Z]\ (\mu\text{-}\mathsf{intro}\ b)) \\ & \rightsquigarrow & W[X, (\Theta[(\mu\text{-}\mathsf{it}\ [Z]W[X, Z])]\ b)] \\ & = & W[X, (\Theta[\Phi[X]]\ b)] \end{array}$$

By Lemma 4.3.6, $(\Psi\ a)$ is a strictly positive type operator. □

4.4 Wellorderings

In the previous section we proposed an extension of the notion of inductive type. We see now that, without extending type theory, we can encode the desired types and families as wellorderings. We already gave a short introduction to wellorderings in Chapter 2, Section 2.11. Wellorderings (also called $\mathbb{W}$-types) are types of trees specified by a type of nodes A and, for every element a of A, a type of branches $(B\ a)$. This means that every node labelled with the element a has as many branches as the elements of $(B\ a)$.

Wellorderings were introduced by Martin-Löf [ML82, ML84] and used by Dybjer [Dyb97] to encode all inductive types obtained from strictly positive operators. Here we extend Dybjer's construction to strongly positive operators.

Definition 4.4.1 *Let* $A\colon \mathsf{Set}$ *and* $B\colon A \to \mathsf{Set}$. *The type* $(\mathbb{W}\ A\ B)$ *is defined by the rules*

formation

$$(\mathbb{W}\ A\ B)\colon \mathsf{Set}$$

introduction

$$\frac{a\colon A \quad f\colon (B\ a) \to (\mathbb{W}\ A\ B)}{(\mathsf{wtree}\ a\ f)\colon (\mathbb{W}\ A\ B)}$$

elimination *Let* $P\colon (\mathbb{W}\ A\ B) \to \mathsf{Set}$, *then*

$$\frac{\begin{array}{l} x\colon A, y\colon (B\ x) \to (\mathbb{W}\ A\ B), z\colon (u\colon (B\ x))(P\ (y\ u)) \\ \vdash e[x,y,z]\colon (P\ (\mathsf{wtree}\ x\ y)) \end{array}}{(\mathsf{wind}\ [x,y,z]e)\colon (w\colon (\mathbb{W}\ A\ B))(P\ w)}$$

reduction

$$\begin{array}{l} (\mathsf{wind}\ [x,y,z]e\ (\mathsf{wtree}\ a\ f)) \\ \quad \rightsquigarrow e[a,f,[u\colon (B\ a)](\mathsf{wind}\ [x,y,z]e\ (f\ u))] \end{array}$$

Wellorderings can be realized in type theory with the standard implementation of inductive types. Using Formula 4.2 we can define the $\mathbb{W}$ constructor as (see Section 2.11)

$$\begin{array}{l} \mathsf{Inductive}\ \mathbb{W}\ [A\colon \mathsf{Set}; B\colon A \to \mathsf{Set}]\colon \mathsf{Set} := \\ \quad \mathsf{wtree}\colon (a\colon A)((B\ a) \to (\mathbb{W}\ A\ B)) \to (\mathbb{W}\ A\ B) \\ \mathsf{end}. \end{array}$$

Dybjer showed in [Dyb97] that every strictly positive operator has an initial algebra constructed by a $\mathbb{W}$-type. This result holds if we take an extensional equality on the $\mathbb{W}$-type—that is, if we consider two elements $(\mathsf{wtree}\ a_1\ f_1)$ and $(\mathsf{wtree}\ a_2\ f_2)$ of $(\mathbb{W}\ A\ B)$ equal if a_1 and a_2 are convertible and if $(f_1\ b) = (f_2\ b)$ for every $b\colon (B\ a_1)$. In intensional type theory, which is the one we use, the second condition is not equivalent to the convertibility of f_1 and f_2. For this reason, when we use $\mathbb{W}$-types, we have to deal explicitly with extensional equality. They are, therefore, more cumbersome than direct inductive definitions. Once we have stressed this drawback, we can extend Dybjer's result to strongly positive operators.

Theorem 4.4.2 *For every strongly positive operator* $X\colon \mathsf{Set} \vdash \Phi[X]\colon \mathsf{Set}$ *there exist* $A\colon \mathsf{Set}$ *and* $B\colon A \to \mathsf{Set}$ *such that* $(\mathbb{W}\ A\ B)$ *is an initial algebra of* Φ. *(For a formal definition of initial algebras of type operators see, for example, [Geu92] or [PR99].)*

Proof The proof is by induction on the structure of Φ as in Dybjer [Dyb97]. Our Definition 4.3.1 contains two extra clauses that are not present in Dybjer's definition: clauses 6 and 8. Let us see how Dybjer's proof can be extended to include them.

Clause 6 is easily treated by defining A and B by cases on the term t in the definition of Φ and using the recursive results for the branches of the Cases expression. If

$$\Phi[X] = \mathsf{Cases}\ t\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ x_1) \mapsto \Phi_1[x_1, X] \\ (\mathsf{inl}\ x_2) \mapsto \Phi_2[x_2, X], \end{cases}$$

with $t\colon T_1 + T_2$; we assume, by inductive hypothesis, that there are

$$\begin{array}{l} x_1\colon T_1 \vdash A_1[x_1]\colon \mathsf{Set} \\ x_1\colon T_1 \vdash B_1[x_1]\colon A_1[x_1] \to \mathsf{Set} \\ x_2\colon T_2 \vdash A_2[x_2]\colon \mathsf{Set} \\ x_2\colon T_2 \vdash B_2[x_2]\colon A_2[x_2] \to \mathsf{Set} \end{array}$$

such that $(\mathbb{W}\ A_1\ B_1)$ is an initial $\Phi_1[x_1]$-algebra and $(\mathbb{W}\ A_2\ B_2)$ is an initial $\Phi_2[x_2]$-algebra. Then we put

$$\begin{array}{l} A\colon \mathsf{Set} \\ A := \mathsf{Cases}\ t\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ x_1) \mapsto A_1[x_1] \\ (\mathsf{inr}\ x_2) \mapsto A_2[x_2] \end{cases} \\ B\colon A \to \mathsf{Set} \\ B := \mathsf{Cases}\ t\ \mathsf{of}\ \begin{cases} (\mathsf{inl}\ x_1) \mapsto [a_1\colon A_1[x_1]](B_1[x_1]\ a_1) \\ (\mathsf{inr}\ x_2) \mapsto [a_2\colon A_2[x_2]](B_2[x_2]\ a_2) \end{cases} \end{array}$$

and we have that $(\mathbb{W}\ A\ B)$ is an initial Φ-algebra.

Now suppose that Φ has been defined according to clause 8. This means that $\Phi[X] = (\Psi[X]\ i)$ for some $i\colon I$, where $I = \mu_Y(\Theta[Y])$ and $\Psi[X] = (\mu\text{-}\mathsf{it}\ [Z]W)$. We define

$$\begin{array}{l} A\colon I \to \mathsf{Set} \\ B\colon (i\colon I)(A\ i) \to \mathsf{Set} \end{array}$$

by recursion on the inductive type I. Given $x = (\mu\text{-}\mathsf{intro}\ y)\colon I$, we assume by inductive hypothesis that A and B are defined for all the recursive occurrences of elements of I in y. We define the new A_x and B_x, by using Dybjer's construction for the occurrences of X and of the recursive calls Z on $W[X, Z]$. Formally, using Dybjer's method recursively on the clauses of Definition 4.3.1, we can construct from W two families of operators W_A and W_B and then apply the iteration principle to obtain A and the induction principle to obtain B:

$$\frac{Z_A\colon \Theta[\mathsf{Set}] \vdash W_A[Z_A]\colon \mathsf{Set}}{A := (\mu\text{-}\mathsf{it}\ [Z_A]W_A)\colon I \to \mathsf{Set}},$$

$$\frac{Z_B\colon \Theta[(\Sigma\ I\ [i : I]A_i \to \mathsf{Set})] \vdash W[Z_B]\colon A_{(\mu\text{-}\mathsf{intro}\ (\Theta[\pi_1]\ z))} \to \mathsf{Set}}{B := (\mu\text{-}\mathsf{ind}\ [Z_B]W_B)\colon (i\colon I)A_i \to \mathsf{Set}}.$$

Note the difference with the proof of Theorem 4.3.7: The assumption that a closed term $a\colon I$ is used was essential to that proof. That was necessary because we were proving an external predicate. But here we are constructing families of types internal to type theory, therefore we can use the elimination rule of type I to construct A (with elimination predicate $P_A = [x\colon I]\mathsf{Set}$) and B (with elimination predicate $P_B = [x\colon I]A_x \to \mathsf{Set}$. Therefore A_x and B_x are defined also for a free variable $x\colon I$. □

This construction gives, in the case of the family of operators of Formula 4.3, the following families of As and Bs:

$$
\begin{array}{lcl}
A\colon \mathbb{N} \to \mathsf{Set} & & \\
\quad A_0 & := & \mathbb{N}_1 \\
\quad A_{(\mathsf{S}\ n)} & := & \mathbb{N}_1 + A_n \\
 & & (\cong \mathbb{N}_1 + \mathbb{N}_1 \times A_n)
\end{array}
\qquad
\begin{array}{lcl}
B\colon (n\colon \mathbb{N})A_n \to \mathsf{Set} & & \\
\quad (B_0 \quad \_\ ) & := & \emptyset \\
\quad (B_{(\mathsf{S}\ n)} \quad (\mathsf{inl}\ \_)) & := & \emptyset \\
\quad (B_{(\mathsf{S}\ n)} \quad (\mathsf{inr}\ a)) & := & \mathbb{N}_1 + (B_n\ a)
\end{array}
$$

The $\mathbb{W}$ construction for terms over a signature in Sig is described in [Cap99], where it is extended to many-sorted signatures.

4.5 Recursive vs. Inductive families

We remarked that the $\mathbb{W}$ construction has the disadvantage that extensionally equal terms are not always convertible. This is unavoidable when we use transfinite types, but it could and should be avoided with finitary types. The solution proposed in this section exploits an extension of inductive types implemented in the proof tool Coq (see [Gim98]). This consists in extending the notion of strict positivity to that of *positivity* by a clause that allows operators to inherit positive occurrences of a parameter X from inductive definitions.

Definition 4.5.1 *A type operator* $X\colon \mathsf{Set} \vdash \Psi[X]\colon \mathsf{Set}$ *is* positive *if it satisfies the clauses of the definition of strict positivity where we substitute "positive" for "strictly positive" everywhere, and the new clause*

> *X is positive in $(J\ t_1 \cdots t_m)$ if J is an inductive type and, for every term t_i, either X does not occur in t_i or X is positive in t_i, t_i instantiates a general parameter of J and this parameter is positive in the arguments of the constructors of J.*

To apply this construction to our case we first need to replace the recursive definition of a family of type operators with an inductive one. We illustrate the method with the example of Formula 4.3. The family Ψ was defined by recursion on the natural numbers. Instead we use the following inductive definition

$$
\begin{array}{l}
\mathsf{Inductive}\ \mathsf{ind}(\Psi)\ [X\colon \mathsf{Set}]\colon \mathbb{N} \to \mathsf{Set} := \\
\quad \psi_0\colon \mathsf{ind}(\Psi)_0 \\
\quad \psi_1\colon (n\colon \mathbb{N})\mathsf{ind}(\Psi)_{(\mathsf{S}\ n)} \\
\quad \psi_2\colon (n\colon \mathbb{N})X \to \mathsf{ind}(\Psi)_n \to \mathsf{ind}(\Psi)_{(\mathsf{S}\ n)} \\
\mathsf{end}.
\end{array}
$$

(The constructors ψ_0 and ψ_1 could be unified in a single constructor ψ_{01}: $(n\colon \mathbb{N})\mathsf{ind}(\Psi)_n$, but we keep them separate to keep the parallel with the definition of Ψ in Formula 4.3.) X is a general parameter of $\mathsf{ind}(\Psi)$ and it is positive in the arguments of the constructors: It appears only as the type of the first argument of the constructor ψ_2. It follows from the clause in Definition 4.5.1 that $X\colon \mathsf{Set} \vdash (\mathsf{ind}(\Psi)\ X\ n)$ is a positive type operator for every $n\colon \mathbb{N}$. In the type system of Coq such positive operators can be used in the definition of inductive types, thus the family $T := [n]\mu_X((\mathsf{ind}(\Psi)\ X\ n))\colon \mathbb{N} \to \mathsf{Set}$ is admissible. Note that the condition expressed in clause 8 of Definition 4.3.1 by requiring W to be a positive type pointer corresponds to the fact that the recursive calls must occur positively in the definition of $\mathsf{ind}(\Psi)$. This translation can be done in general for every strongly positive operator.

Theorem 4.5.2 *For every strongly positive type operator* $X\colon \mathsf{Set} \vdash \Phi[X]\colon \mathsf{Set}$ *there exists a positive type operator* $X\colon \mathsf{Set} \vdash \mathsf{ind}(\Phi)[X]\colon \mathsf{Set}$ *such that, for every type* $X\colon \mathsf{Set}$, $\Phi[X] \cong \mathsf{ind}(\Phi)[X]$.

Proof As usual, the relevant case is clause 8 of Definition 4.3.1. If $X\colon \mathsf{Set} \vdash \Psi[X]\colon I \to \mathsf{Set}$ is defined as in that clause, then we replace it with the inductive family

$$\begin{array}{l}\mathsf{Inductive}\ \mathsf{ind}(\Psi)\ [X\colon \mathsf{Set}]\colon I \to \mathsf{Set} := \\ \quad \psi\colon (y\colon \Theta[I])W[X, (\Theta[\mathsf{ind}(\Psi)]\ y)] \to \mathsf{ind}(\Psi)_{(\mu\text{-}\mathsf{intro}\ y)} \\ \mathsf{end},\end{array}$$

which can be proved to be positive according to Definition 4.5.1, by induction on the proof that W is a positive type pointer. The general parameter X occurs only positively in the arguments of the constructor ψ because it occurs only positively in W (by induction hypothesis).

With this translation we always get inductive families with only one constructor. In practice it is intuitively easier to break it down into several constructors, as we did in the preceding example. □

4.6 Applying the two-level approach to inductive types

The *two-level approach* is a technique used for proof construction in type theory. A goal G is lifted to a syntactic level; that is, a term g, of a type $\mathsf{Goal}\colon \mathsf{Set}$ representing goals, is associated to G. Logical rules are reflected by functions or relations on Goal. To prove G we apply the functions or work with the relations on g. Once g is proved at the syntactic level, we can extract a proof of G.

The technique is described in [BRB95] and in Ruys' thesis [Ruy99]. It was used by Boutin, who calls it *reflection*, to implement the `Ring` tactic in Coq [Bou97]. Its furthest application consists in formalizing type theory inside type

theory itself and use it to do metareasoning. This was partially done by Howe in [How88] for Nuprl and by Barras and Werner in [BW00] for Coq.

We apply it to inductive definitions. First of all we define a type of codes for positive type operators PosOp. To every element ϕ: PosOp we associate a positive type operator (TypeOp ϕ): Set $\to$ Set and an inductive type (IndType ϕ): Set, using the technique of Section 4.5. Then we define an inductive predicate Positive on type operators, which is an internalization of the notion of strict positivity (note that we do not need to internalize strong positivity or positivity). We define a function that associates an element of PosOp to every operator Φ: Set $\to$ Set and proof p: (Positive Φ). So we can define an inductive type by proving that the corresponding type operator is strictly positive. This can be done for the families of operators defined by recursion, hence solving our initial problem.

Definition 4.6.1 *The type* PosOp: Type *is defined by the following introduction rules:*

$$\frac{K\colon \mathsf{Set}}{(\mathsf{op\text{-}const}\ K)\colon \mathsf{PosOp}} \qquad \frac{}{\mathsf{op\text{-}id}\colon \mathsf{PosOp}}$$

$$\frac{op_1\colon \mathsf{PosOp} \quad op_2\colon \mathsf{PosOp}}{(\mathsf{op\text{-}prod}\ op_1\ op_2)\colon \mathsf{PosOp}} \qquad \frac{op_1\colon \mathsf{PosOp} \quad op_2\colon \mathsf{PosOp}}{(\mathsf{op\text{-}sum}\ op_1\ op_2)\colon \mathsf{PosOp}}$$

$$\frac{K\colon \mathsf{Set} \quad op\colon \mathsf{PosOp}}{(\mathsf{op\text{-}fun}\ K\ op)}.$$

Formally

$$\begin{array}{l}\mathsf{Inductive}\ \ \mathsf{PosOp}\colon \mathsf{Type} :=\\ \quad \mathsf{op\text{-}const}\colon \mathsf{Set} \to \mathsf{PosOp}\\ \quad\quad \mathsf{op\text{-}id}\colon \mathsf{PosOp}\\ \quad\quad \mathsf{op\text{-}prod}\colon \mathsf{PosOp} \to \mathsf{PosOp} \to \mathsf{PosOp}\\ \quad\quad \mathsf{op\text{-}sum}\colon \mathsf{PosOp} \to \mathsf{PosOp} \to \mathsf{PosOp}\\ \quad\quad \mathsf{op\text{-}fun}\colon \mathsf{Set} \to \mathsf{PosOp}\\ \mathsf{end}.\end{array}$$

We can associate an actual type operator to every element of PosOp, by recursion on it:

$$\begin{array}{l}\mathsf{TypeOp}\colon \mathsf{PosOp} \to \mathsf{Set} \to \mathsf{Set}\\ \quad (\mathsf{op\text{-}const}\ K) \Rightarrow [X\colon \mathsf{Set}]K\\ \quad \mathsf{op\text{-}id} \Rightarrow [X\colon \mathsf{Set}]X\\ \quad (\mathsf{op\text{-}prod}\ op_1\ op_2) \Rightarrow [X\colon \mathsf{Set}](\mathsf{TypeOp}\ op_1\ X) \times (\mathsf{TypeOp}\ op_2\ X)\\ \quad (\mathsf{op\text{-}sum}\ op_1\ op_2) \Rightarrow [X\colon \mathsf{Set}](\mathsf{TypeOp}\ op_1\ X) + (\mathsf{TypeOp}\ op_2\ X)\\ \quad (\mathsf{op\text{-}fun}\ K\ op) \Rightarrow [X\colon \mathsf{Set}]K \to (\mathsf{TypeOp}\ op\ X).\end{array}$$

Unfortunately this approach leads us to a dead end, since the family of operators TypeOp is strongly positive but not positive, being obtained by recursion. In

fact, the formal definition of TypeOp, using the elimination predicate $P := \lambda op\colon \mathsf{PosOp}.\mathsf{Set} \to \mathsf{Set}\colon \mathsf{PosOp} \to \mathsf{Type}$, is

$$\begin{array}{l}\mathsf{TypeOp} := (\mu\text{-ind}\ \ [K\colon \mathsf{Set}]\\ \qquad\qquad [X\colon \mathsf{Set}]K\\ \qquad\qquad [X\colon \mathsf{Set}]X\\ \qquad\qquad [op_1\colon \mathsf{TypeOp}; OP_1\colon \mathsf{Set}\to\mathsf{Set}; op_2\colon \mathsf{TypeOp}; OP_2\colon \mathsf{Set}\to\mathsf{Set}]\\ \qquad\qquad\quad [X\colon \mathsf{Set}](OP_1\ X)\times(OP_2\ X)\\ \qquad\qquad [op_1\colon \mathsf{TypeOp}; OP_1\colon \mathsf{Set}\to\mathsf{Set}; op_2\colon \mathsf{TypeOp}; OP_2\colon \mathsf{Set}\to\mathsf{Set}]\\ \qquad\qquad\quad [X\colon \mathsf{Set}](OP_1\ X)+(OP_2\ X)\\ \qquad\qquad [K\colon \mathsf{Set}; op\colon \mathsf{TypeOp}; OP\colon \mathsf{Set}\to\mathsf{Set}]\\ \qquad\qquad\quad [X\colon \mathsf{Set}]K\to(OP\ X)\ ).\end{array}$$

This operator does not satisfy the strict positivity condition. It satisfies strong positivity, if we extend it to allow inductive definitions with several constructors. Alternatively we can redefine PosOp as the inductive type with one constructor define by

$$\mathsf{PosOp} := \mu_Y \mathsf{Set} + \mathbb{N}_1 + Y\times Y + Y\times Y + \mathsf{Set}\times Y.$$

The type operator would then become

$$\begin{array}{l}\mathsf{TypeOp} := \mu\text{-it}\ \ [Z\colon \mathsf{Set}+\mathbb{N}_1+(\mathsf{Set}\to\mathsf{Set})\times(\mathsf{Set}\to\mathsf{Set})\\ \qquad\qquad +(\mathsf{Set}\to\mathsf{Set})\times(\mathsf{Set}\to\mathsf{Set})+\mathsf{Set}\times(\mathsf{Set}\to\mathsf{Set})]\\ \qquad\qquad \mathsf{Cases}\ Z\ \mathsf{of}\ \left\{\begin{array}{ll}(\mathsf{in}_0\ K) & \mapsto [X\colon \mathsf{Set}]K\\ (\mathsf{in}_1\ \_) & \mapsto [X\colon \mathsf{Set}]X\\ (\mathsf{in}_2\ OP_1\ OP_2) & \mapsto [X\colon \mathsf{Set}](OP_1\ X)\times(OP_2\ X)\\ (\mathsf{in}_3\ OP_1\ OP_2) & \mapsto [X\colon \mathsf{Set}](OP_1\ X)+(OP_2\ X)\\ (\mathsf{in}_4\ K\ OP) & \mapsto [X\colon \mathsf{Set}]K\to(OP\ X)\end{array}\right.\end{array}$$

which is strongly positive. We want to be able to define the operators associated to the elements of PosOp without making use of strong positivity.

We apply the technique of Section 4.5 to transform TypeOp from a recursive family to an inductive one satisfying the positivity condition:

$$\begin{array}{l}\mathsf{Inductive\ IndOp}\ [X\colon \mathsf{Set}]\colon \mathsf{PosOp}\to\mathsf{Set} :=\\ \quad \mathsf{c_{const}}\colon (K\colon \mathsf{Set})K\to(\mathsf{IndOp}\ (\mathsf{op\text{-}const}\ K))\\ \quad \mathsf{c_{id}}\colon X\to(\mathsf{IndOp}\ \mathsf{op\text{-}id})\\ \quad \mathsf{c_{prod}}\colon (op_1, op_2\colon \mathsf{PosOp})\\ \qquad\qquad (\mathsf{IndOp}\ op_1)\to(\mathsf{IndOp}\ op_2)\to(\mathsf{IndOp}\ (\mathsf{op\text{-}prod}\ op_1\ op_2))\\ \quad \mathsf{c_{sum,l}}\colon (op_1, op_2\colon \mathsf{PosOp})(\mathsf{IndOp}\ op_1)\to(\mathsf{IndOp}\ (\mathsf{op\text{-}sum}\ op_1\ op_2))\\ \quad \mathsf{c_{sum,r}}\colon (op_1, op_2\colon \mathsf{PosOp})(\mathsf{IndOp}\ op_2)\to(\mathsf{IndOp}\ (\mathsf{op\text{-}sum}\ op_1\ op_2))\\ \quad \mathsf{c_{fun}}\colon (K\colon \mathsf{Set})(op\colon \mathsf{PosOp})(K\to(\mathsf{IndOp}\ op))\to(\mathsf{IndOp}\ (\mathsf{op\text{-}fun}\ K\ op))\\ \mathsf{end.}\end{array}$$

Lemma 4.6.2 *For every* $op\colon \mathsf{PosOp}$, $X\colon \mathsf{Set} \vdash (\mathsf{IndOp}\ X\ op)$ *is a positive operator.*

Proof Just check that the requirements of the new clause in Definition 4.5.1 are satisfied. □

Thus, in the type system of Coq, we can associate an inductive type to every element of PosOp:

$$\mathsf{IndType} := [op\colon \mathsf{PosOp}]\mu_X(\mathsf{IndOp}\ X\ op)\colon \mathsf{PosOp} \to \mathsf{Set}.$$

Whenever we have a family of type operators $X\colon \mathsf{Set} \vdash \Psi[X]\colon I \to \mathsf{Set}$ defined by recursion on an inductive type I, we can associate to it a function $f_\Psi\colon I \to \mathsf{PosOp}$ and obtain the family of inductive types as $[x\colon I](\mathsf{IndType}\ (f_\Psi\ x))$. For example, the family Ψ from Formula 4.3 is translated into the function

$$\begin{array}{l} f_\Psi\colon \mathbb{N} \to \mathsf{PosOp} \\ \quad (f_\Psi \quad 0 \quad) := \quad (\mathsf{op\text{-}const}\ \mathbb{N}_1) \\ \quad (f_\Psi \quad (\mathsf{S}\ n)) := \quad (\mathsf{op\text{-}sum}\ (\mathsf{op\text{-}const}\ \mathbb{N}_1)\ (\mathsf{op\text{-}prod}\ \mathsf{op\text{-}id}\ (f_\Psi\ n))) \end{array}$$

Moreover, we can avoid this translation by proving directly the positivity of the original operators inside type theory.

Definition 4.6.3 *The predicate* $\mathsf{Positive}\colon (\mathsf{Set} \to \mathsf{Set}) \to \mathsf{Set}$ *is inductively defined by the following rules:*

$$\frac{K\colon \mathsf{Set}}{(\mathsf{pos\text{-}const}\ K)\colon (\mathsf{Positive}\ [X\colon \mathsf{Set}]K)}$$

$$\frac{}{\mathsf{pos\text{-}id}\colon (\mathsf{Positive}\ [X\colon \mathsf{Set}]X)}$$

$$\frac{\Phi_1\colon \mathsf{Set} \to \mathsf{Set} \quad \Phi_2\colon \mathsf{Set} \to \mathsf{Set} \quad p_1\colon (\mathsf{Positive}\ \Phi_1) \quad p_2\colon (\mathsf{Positive}\ \Phi_2)}{(\mathsf{pos\text{-}prod}\ \Phi_1\ \Phi_2\ p_1\ p_2)\colon (\mathsf{Positive}\ [X\colon \mathsf{Set}](\Phi_1\ X) \times (\Phi_2\ X))}$$

$$\frac{\Phi_1\colon \mathsf{Set} \to \mathsf{Set} \quad \Phi_2\colon \mathsf{Set} \to \mathsf{Set} \quad p_1\colon (\mathsf{Positive}\ \Phi_1) \quad p_2\colon (\mathsf{Positive}\ \Phi_2)}{(\mathsf{pos\text{-}sum}\ \Phi_1\ \Phi_2\ p_1\ p_2)\colon (\mathsf{Positive}\ [X\colon \mathsf{Set}](\Phi_1\ X) + (\Phi_2\ X))}$$

$$\frac{K\colon \mathsf{Set} \quad \Phi\colon \mathsf{Set} \to \mathsf{Set} \quad p\colon (\mathsf{Positive}\ \Phi)}{(\mathsf{pos\text{-}fun}\ K\ \Phi\ p)\colon (\mathsf{Positive}\ [X\colon \mathsf{Set}]K \to (\Phi\ X))}$$

It is straightforward to define a function $\mathsf{pos\text{-}code}\colon (\Phi\colon \mathsf{Set} \to \mathsf{Set})(\mathsf{Positive}\ \Phi) \to \mathsf{PosOp}$ by recursion on the proof of $(\mathsf{Positive}\ \Phi)$:

$$\begin{array}{l} \mathsf{pos\text{-}code}\colon (\Phi\colon \mathsf{Set} \to \mathsf{Set})(\mathsf{Positive}\ \Phi) \to \mathsf{PosOp} \\ \mathsf{pos\text{-}code}([X\colon \mathsf{Set}]K, (\mathsf{pos\text{-}const}\ K)) := (\mathsf{op\text{-}const}\ K) \\ \mathsf{pos\text{-}code}([X\colon \mathsf{Set}]X, \mathsf{pos\text{-}id}) := \mathsf{op\text{-}id} \\ \mathsf{pos\text{-}code}([X\colon \mathsf{Set}](\Phi_1\ X) \times (\Phi_2\ X), (\mathsf{pos\text{-}prod}\ \Phi_1\ \Phi_2\ p_1\ p_2)) := \\ \quad (\mathsf{op\text{-}prod}\ (\mathsf{pos\text{-}code}\ \Phi_1\ p_1)\ (\mathsf{pos\text{-}code}\ \Phi_2\ p_2)) \\ \mathsf{pos\text{-}code}([X\colon \mathsf{Set}](\Phi_1\ X) + (\Phi_2\ X), (\mathsf{pos\text{-}sum}\ \Phi_1\ \Phi_2\ p_1\ p_2)) := \\ \quad (\mathsf{op\text{-}sum}\ (\mathsf{pos\text{-}code}\ \Phi_1\ p_1)\ (\mathsf{pos\text{-}code}\ \Phi_2\ p_2)) \\ \mathsf{pos\text{-}code}([X\colon \mathsf{Set}]K \to (\Phi\ X), (\mathsf{pos\text{-}fun}\ K\ \Phi\ p)) := \\ \quad (\mathsf{op\text{-}fun}\ K\ (\mathsf{pos\text{-}code}\ \Phi\ p)). \end{array}$$

In conclusion, given a recursive family of operators, we can prove by induction that every element of the family is positive and then obtain the recursive family of inductive types by composing IndType and pos-code.

Lemma 4.6.4 *Every type operator* $\Phi\colon \mathsf{Set} \to \mathsf{Set}$ *such that* $(\mathsf{Positive}\ \Phi)$ *is provable has an initial algebra. Let* $H\colon (\mathsf{Positive}\ \Phi)$. *Then the initial* Φ*-algebra is*

$$(\mathsf{Mu}\ \Phi\ H) := (\mathsf{IndType}\ (\mathsf{pos\text{-}code}\ \Phi\ H)).$$

Finally we can apply this method to the strongly positive operators.

Theorem 4.6.5 *If* $X\colon \mathsf{Set} \vdash \Phi[X]\colon \mathsf{Set}$ *is a strongly positive operator, then there is a proof of* $(\mathsf{Positive}\ [X]\Phi[X])$.

Proof We just formalize the proof of Theorem 4.3.7. Since we are now developing the proof inside type theory, the requirement that no free variable except X appears in Φ is no longer necessary. Hence the result holds for every strongly positive operator. □

As final illustration, we construct proofs of Positive for the family of operators of our example. We recall its definition:

$$\begin{array}{lll} X\colon \mathsf{Set} & \vdash & \Psi[X]\colon \mathbb{N} \to \mathsf{Set} \\ & & \Psi_0[X] := \mathbb{N}_1 \\ & & \Psi_{(\mathsf{S}\ n)}[X] := \mathbb{N}_1 + X \times \Psi_n. \end{array}$$

We prove $\forall n\colon \mathbb{N}.(\mathsf{Positive}\ [X\colon \mathsf{Set}]\Psi_n)$ by induction on n.

If $n = 0$, we have to prove $(\mathsf{Positive}\ [X\colon \mathsf{Set}]\Psi_0)$, that is, $(\mathsf{Positive}\ [X\colon \mathsf{Set}]\mathbb{N}_1)$. A proof of it is $(\mathsf{pos\text{-}const}\ \mathbb{N}_1)$.

If $n = (\mathsf{S}\ n')$, we have to prove $(\mathsf{Positive}\ [X\colon \mathsf{Set}]\Psi_{(\mathsf{S}\ n')})$, that is the same as $(\mathsf{Positive}\ [X\colon \mathsf{Set}]\mathbb{N}_1 + X \times \Psi_{n'})$. We assume, by inductive hypothesis, that we have a proof $H\colon (\mathsf{Positive}\ [X\colon \mathsf{Set}]\Psi_{n'})$. Then we construct the proof for $(\mathsf{S}\ n')$:

$$\begin{array}{ll} (\mathsf{pos\text{-}sum} & [X]\mathbb{N}_1\ [X]X \times \Psi_{n'} \\ & (\mathsf{pos\text{-}const}\ \mathbb{N}_1) \\ & (\mathsf{pos\text{-}prod}\ [X]X\ [X]\Psi_{n'}\ \mathsf{pos\text{-}id}\ H)) \end{array}$$

Let us call the proof H_Ψ, that is

$$H_\Psi\colon \forall n\colon \mathbb{N}.(\mathsf{Positive}\ [X\colon \mathsf{Set}]\Psi_n).$$

Then we get the family of inductive types

$$\begin{array}{l} T\colon \mathbb{N} \to \mathsf{Set} \\ T(n) := (\mathsf{Mu}\ [X]\Psi_n\ (H_\Psi\ n)). \end{array}$$

4.7 Conclusion

We have considered the problem of defining families of inductive types whose constructors are given by recursion. These families occur naturally in some developments of abstract mathematics in type theory. We characterized them with the notion of strongly positive operator. We described a model of them in type theory that uses wellorderings. We showed that a more manageable model can be constructed in a type theory with an extended notion of inductive definition. Finally we generalized the later model to a complete internalization of inductive definitions. This last part was completely formalized in Coq.

An interesting topic of future research is whether it is possible to obtain a more elegant solution to the problem of defining recursive families of inductive types using Dybjer and Setzer's axiomatization of induction-recursion [DS99, DS01].

Chapter 5

Setoids

coauthors: Gilles Barthe and Olivier Pons

This chapter is common work in collaboration with Gilles Barthe and Olivier Pons of the Lemme research group in Sophia-Antipolis, France. I worked with them during a stay in INRIA, Sophia-Antipolis, in October and November 2000, made possible by a grant from the Dutch Organization for Scientific Research (NWO).

5.1 Introduction

Proof-development systems such as Agda [CC99], Coq [BBC$^+$99] and Lego [LP92] rely on powerful type systems and have been successfully used in the formalisation of mathematics [Acz93, Bai93, Bai98, Bar95b, Cap99, Ced97, Per99, Saï98]. Nevertheless, their underlying type theories—Martin-Löf's intensional type theory [NPS90] and the Calculus of Inductive Constructions [Wer94]—fail to support extensional concepts such as quotients and subsets, which play a fundamental role in mathematics. While significant efforts have been devoted to embed subset and quotient types in intensional type theory, see for example, [Bar95a, Hof93, Hof95a, Hof95b, Jac99, Mai99, SS88], all proposals to date are unsatisfactory, in that they introduce non-canonical elements or lead to undecidable type-checking, except for a recent proposal by T. Altenkirch [Alt99] which avoids both pitfalls but for which some theoretical details remain to be unveiled. Thus, current versions of Agda, Coq and Lego do not implement subset or quotient types. Instead, mathematical formalisations usually rely on *setoids*, that is, mathematical structures packaging a carrier, the "set"; its equality, the "book equality"; and a proof component ensuring that the book equality is well-behaved.

While setoids have been extensively used in the formalisation of mathematics, there does not seem to be any consensus on their precise definition. Instead, setoids come in several flavours: for example, they can be total (the

book equality is an equivalence relation) or partial (the book equality is a partial equivalence relation); classical (apartness is defined as the logical negation of equality) or constructive (setoids come equipped with an apartness relation that is disjoint from the equality relation). Worse, literature about setoids fails to compare the respective merits of existing approaches, especially from the viewpoint of formalising mathematics.

This chapter has four aims:

- in Section 5.3, we review existing approaches to define (the category of) setoids. It turns out that there are several alternatives to define morphisms of partial setoids, leading to different definitions of the category of partial setoids;
- in Section 5.5, we show that not all resulting categories are equivalent. In particular, we show that one possible approach to partial setoids leads to a category that is not cartesian closed;
- in Section 5.6, we assess the suitability of the different approaches by considering choice principles. We show that total setoids may be turned into a model of intuitionistic set theory by assuming the axiom of unique choice, whereas the situation with partial setoids is less satisfactory.
- in Section 5.7, we introduce some basic constructions on setoids, such as subsets and quotients, and assess the relative advantages of existing approaches with respect to these constructions.

Setting To fix ideas, we shall be working with an extension of the Calculus of Constructions with dependent record types and universes. Dependent record types are used to formalise mathematical structures and universes are used to form the type of categories. We do not need record subtyping and cumulativity between universes and equality between records is neither extensional nor typed. However, our results are to a large extent independent of the choice of a type system.

Notations Following [Luo94], we use Prop for the universe of propositions, Type_i for the i-th universe of types. By abuse of notation, we write Type for Type_0 so we have $\mathsf{Prop} : \mathsf{Type}$ and $\mathsf{Type}_i : \mathsf{Type}_{i+1}$. Moreover, we use the notation $\langle \mathsf{l} : L,\ \mathsf{r} : R\rangle$ for a record type with two fields l of type L and r of type R and $\langle \mathsf{l} = a,\ \mathsf{r} = b\rangle$ for an inhabitant of that type. Translated to the notation of Chapter 2, this means that the declaration

$$\mathsf{Rec} := \langle \mathsf{l} : L,\ \mathsf{r} : R\rangle$$

means

$$\mathsf{Record\ Rec} \ := \ \mathsf{Rec\text{-}make} \left\{ \begin{array}{ll} \mathsf{l}: & L \\ \mathsf{r}: & R \end{array} \right.$$

and

$$\langle \mathsf{l} = a,\ \mathsf{r} = b\rangle := (\mathsf{Rec\text{-}make}\ a\ b).$$

If $\mathsf{rec}\colon \mathsf{Rec}$ is a record, we also use the notations $\mathsf{rec}\cdot\mathsf{l}$ and $\mathsf{rec}\cdot\mathsf{r}$ for $(\mathsf{l}\ \mathsf{rec})$ and $(\mathsf{r}\ \mathsf{rec})$, respectively.

Finally, we let $\doteq$ denote Leibniz equality, defined as

$$\lambda A : \mathsf{Type}.\lambda x, y : A.\ \Pi P : A \to \mathsf{Prop}.\ (P\ x) \to (P\ y).$$

5.2 On the definition of category

In the next section we define several categories of total and partial setoids. It is necessary, then, to define precisely what we mean by a category. Standard category theory assumes some form of set or class theory as a base. Let us look at the standard definition of category.

Definition 5.2.1 *A* category *is a structure* $\mathcal{C} = \langle \mathcal{C}_0, \mathcal{C}_1, \mathsf{id}, \circ \rangle$ *consisting in*

- *a collection* $\mathcal{C}_0$ *of* objects*;*
- *for every two objects* $A, B \in \mathcal{C}_0$*, a collection* $\mathcal{C}_1(A, B)$ *of* morphisms *from* A *to* B*;*
- *for every object* $A \in \mathcal{C}_0$*, a denoted* identity *morphism* $\mathsf{id}_A \in \mathcal{C}_1(A, A)$*;*
- *for every three objects* $A, B, C \in \mathcal{C}_0$ *and every two morphism* $f \in \mathcal{C}_1(A, B)$*,* $g \in \mathcal{C}_1(B, C)$*, a* composite *morphism* $g \circ f \in \mathcal{C}_1(A, C)$*;*

such that the following axioms are satisfied for all objects $A, B, C, D \in \mathcal{C}_0$ *and all morphisms* $f \in \mathcal{C}_1(A, B)$*,* $g \in \mathcal{C}_1(B, C)$*, and* $h \in \mathcal{C}_1(C, D)$*:*

- $f \circ \mathsf{id}_A = f$;
- $\mathsf{id}_B \circ f = f$;
- $f \circ (g \circ h) = (f \circ g) \circ h$.

This definition presupposes the notions of collection, because objects and morphisms form collections, and of equality of morphisms, for the axioms to have meaning. Usually, collections are intended as sets or classes from ZFC or some other foundation for set theory. Equality is then (extensional) set equality.

In our case, we do not use axiomatic set theory, but type theory as a foundation. Collections of objects and morphisms must be entities definable in type theory and equality must be a relation definable in type theory. The most direct approach would be to use types as collections and Leibniz equality as equality between morphisms.

However, this choice would be too restrictive: We need to define categories in which the collections of morphisms do not form a type. Specifically, we want more freedom in the definition of equality of morphisms. In categories in which morphisms are functions, we want to use extensional equality, which is weaker that Leibniz equality. Therefore, the natural choice is to let morphisms form a

setoid rather than a type. That way we are free to choose a book equality for morphisms.

Since there are several different choices for the formalization of setoids, there are several distinct notions of categories.

The study of the category of setoids is for us the equivalent of the study of the category of sets in standard category theory; with the difference that we have five categories of setoids. To sum up, we show that we have five possible notions of setoids, that we call T-setoid, P-setoid, Q-setoid, R-setoid, and S-setoid. To each of these notions corresponds a different formulation of category theory: T-category theory, P-category theory, Q-category theory, R-category theory, and S-category theory. Within each of these formulations of category theory we may study the categories of setoids. For example we may choose to use T-category theory to study the categories of T-setoids, of P-setoids, of Q-setoids, of R-setoids, and of S-setoids and their relations.

The results of category theory are general enough not to depend on the choice of which version of category theory we choose. What we want to stress is that we do not need to use P-category theory to study the category of P-setoids, we may use T-category theory for all five categories. The reader may feel cheated here, because we are already showing a preference for T-category theory. However, the reader can easily check for herself that our results hold also in the other versions of category theory.

An important property of the category of sets in standard category theory is that the morphisms, that is, functions, between two objects, that is, sets, A and B, is still a set, that is, an object of the category, $A \to B$. This is called the *object of exponents*. The properties of an object that make it a good choice for an object of morphisms between two other objects is formalized in the notion of cartesian closed category. In the category of sets this notion is even stronger: Not only there is an object of morphisms between two other objects A and B, but this object is exactly the set of all morphisms $A \to B$.

In the case of setoids, we are interested in checking whether, for every two setoids $\mathcal{A}$ and $\mathcal{B}$, there is an exponent setoid $\mathcal{A} \to \mathcal{B}$. We even have a stronger requirement: $\mathcal{A} \to \mathcal{B}$ should be exactly the setoid of all setoid functions from $\mathcal{A}$ to $\mathcal{B}$. We will see that, even if this setoid is definable for each of the five categories of setoids, the choice of it as the exponent object does not always lead to a cartesian closed category.

For each of the notions of setoid we define the associated notion of function setoid. Then we ask if the corresponding category is cartesian closed when we choose the function setoid as exponent. This differs from standard category theory, in which we say that a category is cartesian closed when there exists a exponent object. We ask that the exponent is explicitly given and that it is the function setoid.

5.3 Definitions

This section gathers some existing definitions of setoids. Here we focus on classical setoids, that is, setoids that do not carry an apartness relation. Similarly, we ignore issues related to the decidability of equality and do not require equality to be decidable. In this setting, there is a single reasonable definition for total setoids and morphisms of total setoids. Also, there is a single reasonable definition for partial setoids, but there are at least four possible definitions for morphisms of partial setoids.

Below we give these possible definitions of setoids. None of them is original. The first definition has been used, for example, in the formalisation of Galois theory [Acz93, Bar95b] and of constructive category theory [HS00]. The second definition has been used, for example, in the formalisation of polynomials [Bai93]. The other definitions do not seem to appear in the context of formal proofs but have been used by M. Hofmann to interpret extensional concepts in intensional type theory [Hof93, Hof95a].

Total setoids

A total setoid consists of a type T (the carrier), a binary relation R on T (the book equality), and a proof that R is an equivalence relation over T.

Definition 5.3.1 *The type of* total setoids *is defined as the record type*

$$\textsc{Set}_t := \langle \mathsf{el}_t : \mathsf{Type},\ \mathsf{eq}_t : \mathsf{el}_t \to \mathsf{el}_t \to \mathsf{Prop},\ \mathsf{er} : \textsc{Er}\ \mathsf{el}_t\ \mathsf{eq}_t \rangle,$$

where

$$\begin{array}{lrcl} \textsc{Er} := & \lambda A : \mathsf{Type}.\lambda R : A \to A \to \mathsf{Prop}. & & \\ & \langle \quad \mathsf{refl}_t & : & \forall x : A.\ R\ x\ x, \\ & \quad\ \mathsf{sym}_t & : & \forall x, y : A.\ (R\ x\ y) \to (R\ y\ x), \\ & \quad\ \mathsf{trans}_t & : & \forall x, y, z : A.\ (R\ x\ y) \to (R\ y\ z) \to (R\ x\ z) \rangle. \end{array}$$

Each type T induces a setoid $\dagger T$, in which all elements of T are identified, defined as

$$\langle \mathsf{el}_t = T,\ \mathsf{eq}_T = \lambda x, y : T.\top,\ \mathsf{er} = \ldots \rangle$$

and a setoid $\ddagger T$, with the Leibniz equality, defined as

$$\langle \mathsf{el}_t = T,\ \mathsf{eq}_T = \lambda x, y : T.x \doteq y,\ \mathsf{er} = \ldots \rangle.$$

A map of total setoids is a map between the underlying carriers which preserves equality. So, if A and B are two total setoids, a map of total setoids from A to B consists of a function $f : \mathsf{el}_t\ A \to \mathsf{el}_t\ B$ and a proof that f preserves equality.

Definition 5.3.2 *Let A and B be two total setoids.*

- *The type* $\text{MAP}_t\ A\ B$ *of* morphisms of total setoids *from* A *to* B *is defined as the record type*

$$\begin{array}{rcl}\text{MAP}_t\ A\ B := \langle & \mathsf{ap}_t & : \quad \mathsf{el}_t\ A \to \mathsf{el}_t\ B, \\ & \mathsf{ext}_t & : \quad \forall x, y : \mathsf{el}_t\ A.\ (x =_A y) \to (\mathsf{ap}_t\ x =_B \mathsf{ap}_t\ y)\rangle.\end{array}$$

- *The* function space setoid $\text{MAP}_t\ A\ B$ *of maps from* A *to* B *is defined as the record*

$$\begin{array}{rcl}\text{MAP}_t\ A\ B := \langle\ \mathsf{el}_t & = & \text{MAP}_t\ A\ B, \\ \mathsf{eq}_t & = & \lambda f, g : \text{MAP}_t\ A\ B.\ \forall x : \mathsf{el}_t\ A. \\ & & \quad (\mathsf{ap}_t\ f\ x) =_B (\mathsf{ap}_t\ g\ x), \\ \mathsf{er} & = & \ldots\rangle.\end{array}$$

Total setoids can be made into a category.

Definition 5.3.3 *The category* **TSet** *of total setoids takes as objects elements of* SET_t *and as homset between* A *and* B *the setoid* $\text{MAP}_t\ A\ B$.

We conclude this paragraph by observing that it would have been equivalent to define equality between morphisms from A to B as

$$\lambda f, g : \text{MAP}_t\ A\ B.\ \forall x, y : \mathsf{el}_t\ A.\ (x =_A y) \to (\mathsf{ap}_t\ f\ x) =_B (\mathsf{ap}_t\ g\ y).$$

This alternative definition will be used later for partial setoids—in that case the two definitions will not be equivalent.

Partial setoids

A partial setoid consists of a type T (the carrier), a binary relation R on T (the book equality) and a proof that R is a partial equivalence relation over T.

Definition 5.3.4 *The type of* partial setoids *is defined as the record type*

$$\text{SET}_p := \langle \mathsf{el}_p : \mathsf{Type},\ \mathsf{eq}_p : \mathsf{el}_p \to \mathsf{el}_p \to \mathsf{Prop},\ \mathsf{per} : \text{PER}\ \mathsf{el}_p\ \mathsf{eq}_p \rangle$$

where

$$\begin{array}{rcl}\text{PER} := & \lambda A : \mathsf{Type}.\lambda R : A \to A \to \mathsf{Prop}. \\ & \langle\ \mathsf{sym}_p & : \quad \forall x, y : A.\ (R\ x\ y) \to (R\ y\ x), \\ & \quad \mathsf{trans}_p & : \quad \forall x, y, z : A.\ (R\ x\ y) \to (R\ y\ z) \to (R\ x\ z)\rangle.\end{array}$$

In the framework of partial setoids, one distinguishes between defined and undefined elements. The defined elements of a partial setoid A are those expressions $x : \mathsf{el}_p\ A$ such that $x =_A x$; they form the domain of the partial setoid.

Definition 5.3.5

- *The* domain *of a partial setoid* A *is defined as the record type*

$$\text{DOMAIN}\ A := \langle \mathsf{cont} : \mathsf{el}_p\ A, \mathsf{def} : \mathsf{cont} =_A \mathsf{cont} \rangle.$$

- *The* domain setoid *of a partial setoid A is defined as*

$$\begin{array}{rcl} \textsc{Domain}\ A := \langle\ \mathsf{el}_p & = & \textsc{domain}\ A, \\ \mathsf{eq}_p & = & \lambda x, y : \textsc{domain}\ A.\ x \cdot \mathsf{cont} =_A y \cdot \mathsf{cont}, \\ \mathsf{per} & = & \ldots\rangle. \end{array}$$

Note that the underlying equality of domain setoids is a total equivalence relation. In Section 5.5, we will use domain setoids to relate partial setoids to total setoids.

We now turn to the definition of morphism of partial setoids. It turns out that there are several possible alternatives for this notion; below we present four alternatives that appear in the literature. The alternatives are determined by the following two issues:

- Issue 1: what is the domain of the morphism? A morphism of partial setoids from A to B may either take as inputs elements of A, or elements of $\textsc{Domain}\ A$. So a morphism from A to B is an element

$$f : \mathsf{el}_p\ A \to \mathsf{el}_p\ B$$

  if we make the first choice, it is an element

$$f : \Pi x : \mathsf{el}_p\ A.x =_A x \to B$$

  if we make the second choice. In the latter case, one will require that the morphism is constant in the def field of the record.

- Issue 2: what is the status of extensionality? How do we restrict the allowed functions so that only the ones that preserve equality are admitted? We can make this restriction directly in the carrier type of the function setoid, or allow any type-theoretic function in the carrier and define the equality of functions in such a way that only functions that preserve equality are equal to themselves.

  The first option follows the definition of morphism of total setoids: A morphism of partial setoids from A to B can be defined as a pair $\langle f, \phi_f \rangle$ where f is a type-theoretical function mapping "elements" of A to "elements" of B and ϕ_f is a proof that f preserves equality; this definition is similar to the one for total setoids.

  The second option takes advantage of the possibility to restrict the defined elements by using a partial equivalence relation and choose (1) to define morphisms of setoids as type-theoretical functions, (2) to embed extensionality in the definition of equality for morphisms of setoids—in such a way that a morphism is defined w.r.t. the equality of the setoid $\textsc{Map}\ A\ B$ iff it preserves equality;

This leaves us with four alternatives, which are summarised in the following table

Extensionality vs. Inputs	Elements of A	Elements of DOMAIN A
In the definition of morphism	SET_p	SET_q
In the definition of equality	SET_r	SET_s

and described below.

- The first alternative, which appears in [Bai93], is to adapt to partial setoids the definition of map of total setoids. Indeed, one can define a map of partial setoids as a map between the underlying carriers which preserves equality.

  Definition 5.3.6 *Let A and B be two partial setoids.*

  - *The type* $\text{MAP}_p\ A\ B$ *of* P-morphisms of partial setoids *from A to B is defined as the record type*

$$\text{MAP}_p\ A\ B := \langle \begin{array}{rcl} \mathsf{ap}_p & : & \mathsf{el}_p\ A \to \mathsf{el}_p\ B, \\ \mathsf{ext}_p & : & \forall x, y : \mathsf{el}_p\ A. \\ & & (x =_A y) \to (\mathsf{ap}_p\ x =_B \mathsf{ap}_p\ y)\rangle. \end{array}$$

  - *The* P-function space setoid $\text{MAP}_p\ A\ B$ *of maps from A to B is defined as the record*

$$\text{MAP}_p\ A\ B := \langle \begin{array}{rcl} \mathsf{el}_p & = & \text{MAP}_p\ A\ B, \\ \mathsf{eq}_p & = & \lambda f, g : \text{MAP}_p\ A\ B.\ \forall x : \mathsf{el}_p\ A. \\ & & (x =_A x) \to (\mathsf{ap}_p\ f\ x) =_B (\mathsf{ap}_p\ g\ x), \\ \mathsf{per} & = & \ldots\rangle. \end{array}$$

  Note that $f =_{\text{MAP}_p\ A\ B} f$ for every $A, B : \text{SET}_p$ and $f : \text{MAP}_p\ A\ B$.

  Definition 5.3.7 *The category* **PSet** *of partial setoids takes as objects elements of* SET_p *and as homset between A and B the setoid* $\text{MAP}_p\ A\ B$.

- The second alternative requires that a function from A to B takes two arguments, namely an element $a : \mathsf{el}_p\ A$ and a proof $\phi : a =_A a$. The second argument is here to prevent some anomalies with empty sets (see below) but the result of the function does not depend on it.

  Definition 5.3.8 *Let A and B be two partial setoids.*

  - *The type* $\text{MAP}_q\ A\ B$ *of* Q-morphisms of partial setoids *from A to B is defined as the record type*

$$\text{MAP}_q\ A\ B := \langle \begin{array}{rcl} \mathsf{ap}_q & : & \Pi a : \mathsf{el}_p\ A.\ (a =_A a) \to \mathsf{el}_p\ B, \\ \mathsf{ext}_q & : & \forall x, y : \mathsf{el}_p\ A. \forall \phi : x =_A x. \forall \psi : y =_A y. \\ & & (x =_A y) \to (\mathsf{ap}_q\ x\ \phi =_B \mathsf{ap}_q\ y\ \psi)\rangle. \end{array}$$

– *The Q-*function space setoid $\textsc{Map}_q\ A\ B$ *of maps from A to B is defined as the record*

$$\begin{array}{rcl}\textsc{Map}_q\ A\ B := \langle\ \mathsf{el}_p & = & \textsc{map}_q\ A\ B, \\ \mathsf{eq}_p & = & \lambda f, g : \textsc{map}_q\ A\ B. \\ & & \forall x : \mathsf{el}_p\ A.\ \forall \phi : (x =_A x). \\ & & (\mathsf{ap}_q\ f\ x\ \phi) =_B (\mathsf{ap}_q\ g\ x\ \phi), \\ \mathsf{per} & = & \ldots\rangle.\end{array}$$

This approach makes function application awkward; perhaps for this reason it has never been used in practice. Note that $f =_{\textsc{Map}_q\ A\ B} f$ for every $A, B : \textsc{Set}_p$ and $f : \textsc{map}_q\ A\ B$.

Definition 5.3.9 *The category* **QSet** *of partial setoids takes as objects elements of* $\textsc{Set}_p$ *and as homset between A and B the setoid* $\textsc{Map}_q\ A\ B$.

- The third alternative, which appears in [Hof95a] and has been used extensively in [ČDS98, Qia00], does not require maps to preserve equality: instead, the function space between A and B is defined as a partial setoid with carrier $\mathsf{el}_p\ A \to \mathsf{el}_p\ B$. Equality is defined in the obvious way; as a consequence, the defined elements of this partial setoids are those type-theoretical functions preserving equality.

Definition 5.3.10 *Let A and B be two partial setoids.*

– *The type* $\textsc{map}_r\ A\ B$ *of* R-morphisms of partial setoids *from A to B is defined as the type*

$$\textsc{map}_r\ A\ B := \mathsf{el}_p\ A \to \mathsf{el}_p\ B.$$

– *The R-*function space setoid $\textsc{Map}_r\ A\ B$ *of maps from A to B is defined as the record*

$$\begin{array}{rcl}\textsc{Map}_r\ A\ B := \langle\ \mathsf{el}_p & = & \textsc{map}_r\ A\ B, \\ \mathsf{eq}_p & = & \lambda f, g : \textsc{map}_r\ A\ B.\ \forall x, y : \mathsf{el}_p\ A. \\ & & (x =_A y) \to f\ x =_B g\ y, \\ \mathsf{per} & = & \ldots\rangle.\end{array}$$

Note that we need not have $f =_{\textsc{Map}_r\ A\ B} f$ for $A, B : \textsc{Set}_p$ and $f : \textsc{map}_r\ A\ B$: in other words, $\textsc{Map}_r\ A\ B$ may be a partial setoid. Also note that transitivity of equality for $\textsc{Map}_r\ A\ B$ uses that

$$\forall x, x' : \mathsf{el}_p\ A.\ (x =_A x') \to x =_A x$$

which is provable from the symmetry and transitivity of $=_A$.

Definition 5.3.11 *The category* **RSet** *of partial setoids takes as objects elements of* $\textsc{Set}_p$ *and as homset between A and B the setoid* $\textsc{Map}_r\ A\ B$.

- The fourth alternative, which appears in [Hof93], takes as inputs defined elements of A and does not require maps to preserve equality.

  Definition 5.3.12 *Let A and B be two partial setoids.*

  - *The type* $\textsc{map}_s\ A\ B$ *of* S-morphisms of partial setoids *from A to B is defined as the type*

    $$\textsc{map}_s\ A\ B := \Pi a : \mathsf{el}_p\ A.\ x =_A x \to \mathsf{el}_p\ B.$$

  - *The* S-function space setoid $\textsc{Map}_s\ A\ B$ *of maps from A to B is defined as the record*

    $$\begin{array}{rcl} \textsc{Map}_s\ A\ B := \langle\ \mathsf{el}_p & = & \textsc{map}_s\ A\ B, \\ \mathsf{eq}_p & = & \lambda f, g : \textsc{map}_s\ A\ B. \\ & & \forall x, y : \mathsf{el}_p\ A.\ \forall \phi : x =_A x.\ \forall \psi : y =_A y. \\ & & (x =_A y) \to f\ x\ \phi =_B g\ y\ \psi, \\ \mathsf{per} & = & \ldots\rangle. \end{array}$$

  Definition 5.3.13 *The category* **SSet** *of partial setoids takes as objects elements of* $\textsc{Set}_p$ *and as homset between A and B the setoid* $\textsc{Map}_s\ A\ B$.

Total functional relations as morphisms?

All previous definitions introduce morphisms of setoids as (structures with underlying) type-theoretical functions. In contrast, set theory views morphisms of sets as graphs. One may therefore wonder about this departure from mainstream mathematics. Two points need to be emphasised:

- first, our type theory is expressive enough to formalise the notion of total functional relation and thus one needs not depart, at least in principle, from mainstream mathematics;

- second, our type theory does make a difference between the two approaches: every function has an associated total functional relation but the converse needs not be true.

In Section 5.6 we provide some choice axioms under which the two approaches coincide, and briefly discuss the validity of our results/claims in other type-theoretical settings, but for the time being, let us focus on the relative benefits of the two approaches:

1. Using type-theoretical functions as the underlying concept for morphisms of setoids is very much in line with the philosophy of type theory because it provides a computational meaning to functions. In effect, most formalisations of mathematics in type theory follow the first approach.

2. Using total functional relations as the underlying concept for morphisms of setoids avoids some of the difficulties with choice principles, see Section 5.6. On the other hand, total functional relations do not have a computational meaning, which is a weakness from a type-theoretical perspective, and their use complicates the presentation of formal proofs, because it becomes impossible to write $f\ a$ for the result of applying the function f to a.

While we are strongly in favour of approach 1, we would like to conclude this section by observing that it is possible to use a monadic style to manipulate total functional relations. To our knowledge, this approach has not been pursued before, and we have no practical experience with it; yet we feel that it is likely to be less cumbersome than manipulating total functional relations directly. Concretely, our suggestion is to use the ι-monad, which assigns to every setoid $A : \textsc{Set}_t$ the setoid $\iota\ A$ defined as the predicates over A that are satisfied by exactly one element.

Let us recall the notion of monad. A *monad* in a category $\mathcal{C}$ is a triple $\langle M, \eta, \mu\rangle$ where $M\colon \mathcal{C} \to \mathcal{C}$ is a functor, $\eta\colon \mathsf{Id}_{\mathcal{C}} \to M$ and $\mu\colon M \circ M \to M$ are natural transformations such that $\mu \circ \mu M = \mu \circ M\mu$ and $\mu \circ \eta M = \mu \circ T\eta = \mathsf{id}_M$.

Formally, we need to introduce the setoid Ω of propositions

$$\langle \mathsf{el}_t = \mathsf{Prop},\ \mathsf{eq}_t = \lambda P, Q : \mathsf{Prop}.\ P \leftrightarrow Q,\ \mathsf{er} = \ldots\rangle$$

the quantifier $\exists!\ x \in A.\ P\ x$ where A is a total setoid

$$\exists x : \mathsf{el}_t\ A.\ (P\ x) \wedge (\forall y : \mathsf{el}_t\ A.\ P\ y\ \to x =_A y)$$

the type of predicates over A that are satisfied by exactly one element

$$\underline{\iota}\ A := \langle \mathsf{up} :\ \textsc{map}_t\ A\ \Omega,\ \mathsf{pp} :\ \exists!\ x \in A.\ \mathsf{ap}_t\ \mathsf{up}\ x\rangle$$

and finally the setoid $\iota\ A$ itself

$$\langle \mathsf{el}_t = \underline{\iota}\ A,\ \mathsf{eq}_t = \lambda P, Q : \underline{\iota}\ A.\ P \cdot \mathsf{up} =_{\textsc{Map}_t\ A\ \Omega} Q \cdot \mathsf{up},\ \mathsf{er} = \ldots\rangle.$$

It is easy to turn ι into a monad. For example, the unit η_ι of the monad is defined as

$$\begin{array}{l}\lambda A : \textsc{Set}_t.\ \langle\ \mathsf{ap}_t = \lambda x : \mathsf{el}_t\ A.\ \langle\ \mathsf{up} = \langle \mathsf{ap}_t = \lambda y : \mathsf{el}_t\ A.\ x =_A y,\ \mathsf{ext}_t = \ldots\rangle,\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{pp} = \ldots\rangle,\\ \qquad\quad \mathsf{ext}_t = \ldots\rangle.\end{array}$$

Finally, observe that it is possible to define likewise an ϵ-monad which maps every setoid A to the setoid $\epsilon\ A$ of non-empty predicates over A, where the type of non-empty predicates over A is defined as

$$\underline{\epsilon}\ A := \langle \mathsf{np} :\ \textsc{map}_t\ A\ \Omega,\ \mathsf{ne} :\ \exists x :\ \mathsf{el}_t\ A.\ \mathsf{ap}_t\ \mathsf{np}\ x\rangle$$

and the setoid $\epsilon\ A$ is defined as

$$\langle \mathsf{el}_t = \underline{\epsilon}\ A,\ \mathsf{eq}_t = \lambda P, Q : \underline{\epsilon}\ A.P \cdot \mathsf{np} =_{\textsc{Map}_t\ A\ \Omega} Q \cdot \mathsf{np},\ \mathsf{er} = \ldots\rangle.$$

5.4 What is a (cartesian closed) category?

In Section 5.3, we introduced several categories of setoids. Yet we have not specified formally what is meant by a category and, more precisely, what is the exact nature of homsets. We will take the objects of a category to form a type, but the morphisms between two objects are required to form a set, a setoid in our setting, because we need to identify equal morphisms. Therefore we will have different notions of category according to what notion of setoid we assume. For the category **TSet**, it is natural to define homsets as total setoids; for the categories **PSet**, **QSet** and **RSet**, it is natural to define homsets as partial setoids—there are some alternatives, for example, for the category **TSet**, one can define homsets as partial setoids, and for the categories **PSet** and **QSet**, homsets can also be defined as total setoids. Here we will define the formalisation of category theory associated with **TSet**, T-category theory. According to what notion of setoid we choose for the hom setoid, we could in the same way define P-category theory, Q-category theory, and S-category theory.

Following [DR94, Lac95] and previous work on the formalisation of category theory in type theory [Acz93, HS00, Saï98], a T-category consists of

- a type obj of objects (in Type_1);
- a polymorphic setoid of morphisms $\mathsf{hom} : \mathsf{obj} \to \mathsf{obj} \to \textsc{Set}_t$;
- a polymorphic composition operator

  $$\bullet : \Pi A, B, C : \mathsf{obj}.\ \textsc{bmap}_t\ (\mathsf{hom}\ A\ B)\ (\mathsf{hom}\ B\ C)\ (\mathsf{hom}\ A\ C),$$

  where $\textsc{bmap}_t\ X\ Y\ Z$ is defined as $\textsc{map}_t\ X\ (\textsc{Map}_t\ Y\ Z)$;
- a polymorphic identity $\mathsf{id} : \Pi A : \mathsf{obj}.\ (\underline{\mathsf{hom}}\ A\ A)$;
- a proof that composition is associative and identity acts as a unit.

This definition and the definitions below focus on total setoids, but it is possible to adapt them to partial setoids.

Definition 5.4.1 *The type* $\textsc{Cat}_t$ *of* T-categories *is defined as the record type*

$$\begin{array}{rcl} \langle \mathsf{obj} & : & \mathsf{Type}_1, \\ \mathsf{hom} & : & \mathsf{obj} \to \mathsf{obj} \to \textsc{Set}_t, \\ \bullet & : & \Pi A, B, C : \mathsf{obj}.\ \textsc{bmap}_t\ (\mathsf{hom}\ A\ B)\ (\mathsf{hom}\ B\ C)\ (\mathsf{hom}\ A\ C), \\ \mathsf{id} & : & \Pi A : \mathsf{obj}.\ \underline{\mathsf{hom}}\ A\ A, \\ \mathsf{catlaw} & : & \phi_{\mathrm{cat}} \rangle \end{array}$$

where ϕ_{cat} *is*

$$\begin{array}{ll} & (\forall A, B, C, D : \mathsf{obj}.\ \forall f : \underline{\mathsf{hom}}\ A\ B.\ \forall g : \underline{\mathsf{hom}}\ B\ C.\ \forall h : \underline{\mathsf{hom}}\ C\ D. \\ & \quad f \mathbin{\underline{\bullet}} (g \mathbin{\underline{\bullet}} h) =_{(\mathsf{hom}\ A\ D)} (f \mathbin{\underline{\bullet}} g) \mathbin{\underline{\bullet}} h) \\ \wedge & (\forall A, B : \mathsf{obj}.\ \forall f : \underline{\mathsf{hom}}\ A\ B. \\ & \quad (\mathsf{id}\ A) \mathbin{\underline{\bullet}} f =_{(\mathsf{hom}\ A\ B)} f \wedge f \mathbin{\underline{\bullet}} (\mathsf{id}\ B) =_{(\mathsf{hom}\ A\ B)} f) \end{array}$$

using $\underline{\mathsf{hom}}\ Y\ Z$ *as a shorthand for* $\mathsf{el}_t\ (\mathsf{hom}\ Y\ Z)$ *and* $x \bullet y$ *as a shorthand for* $\mathsf{ap}_t\ (\bullet\ x)\ y$. *In the sequel, we use* obj_C, hom_C *and* $\underline{\mathsf{hom}}_C$ *as shorthand for* $C \cdot \mathsf{obj}$, $C \cdot \mathsf{hom}$, *and* $\mathsf{el}_t\ (\mathsf{hom}_C\ Y\ Z)$, *respectively.*

As stated above, the notion of T-category can be modified to accommodate partial setoids, but the modification involves some subtle choices, see for example, [ČDS98, Hof95a, Qia00]. For example we cannot require that $(\mathsf{id}\ A) \bullet f =_{(\mathsf{hom}\ A\ B)} f$ for all $f : \underline{\mathsf{hom}}\ A\ B$ since it would entail that $\mathsf{hom}\ A\ B$ is always total; instead one requires that

$$f =_{(\mathsf{hom}\ A\ B)} f' \quad \Rightarrow \quad (\mathsf{id}\ A) \bullet f =_{(\mathsf{hom}\ A\ B)} f'$$

Similar issues arise with composition, and similar solutions exist. In [ČDS98], Čubrić, Dybjer and Scott develop a theory of categories based on partial setoids, which they coin $\mathcal{P}$-category theory. Their framework takes R-morphisms as the notion of morphisms between partial setoids, but their definition readily adapts to P-morphisms, Q-morphisms and S-morphisms.

We now turn to functors. Informally, a functor from $C : \textsc{Cat}_t$ to $C' : \textsc{Cat}_t$ consists of

- a function $\mathsf{fobj} : \mathsf{obj}_C \to \mathsf{obj}_{C'}$;
- a polymorphic map of setoids

$$\mathsf{fmor} : \Pi o, o' : \mathsf{obj}_C.\ \textsc{map}_t\ (\textsc{hom}_C\ o\ o')\ (\textsc{hom}_{C'}\ (\mathsf{fobj}\ o)\ (\mathsf{fobj}\ o'));$$

- a proof that fobj preserves identities and composition.

Definition 5.4.2 *The parametric type* $\textsc{Func}_t$ *of* T-functors *is defined as*

$$\begin{array}{lll} \lambda C, C' : \textsc{Cat}_t.\ \langle & \mathsf{fobj} & : \mathsf{obj}_C \to \mathsf{obj}_{C'}, \\ & \mathsf{fmor} & : \Pi o, o' : \mathsf{obj}_C. \\ & & \quad \textsc{map}_t\ (\mathsf{hom}_C\ o\ o') \\ & & \qquad (\mathsf{hom}_{C'}\ (\mathsf{fobj}\ o)\ (\mathsf{fobj}\ o')), \\ & \mathsf{flaw} & : \ldots\rangle. \end{array}$$

The parametric type $\textsc{BFunc}_t$ *of* T-bifunctors *is defined as*

$$\begin{array}{lll} \lambda C, C', C'' : \textsc{Cat}_t.\ \langle & \mathsf{bfobj} & : \mathsf{obj}_C \to \mathsf{obj}_{C'} \to \mathsf{obj}_{C''}, \\ & \mathsf{bfmor} & : \Pi o, o' : \mathsf{obj}_C.\ \Pi u, u' : \mathsf{obj}_{C'}. \\ & & \quad \textsc{bmap}_t\ (\mathsf{hom}_C\ o\ o') \\ & & \qquad (\mathsf{hom}_{C'}\ u\ u') \\ & & \qquad (\mathsf{hom}_{C''}\ (\mathsf{bfobj}\ o\ u)\ (\mathsf{bfobj}\ o'\ u')), \\ & \mathsf{bflaw} & : \ldots\rangle. \end{array}$$

We now proceed towards the definition of cartesian closedness and define the notions of terminal object, products and exponents. Recall that o is a terminal object if for every object o' there exists a unique morphism from o' to o.

Definition 5.4.3 *The parametric type* $\textsc{Obj}_t$ *of* terminal objects *is defined as:*

$$\begin{array}{lrcl}\lambda C : \textsc{Cat}_t.\ \langle & \mathsf{tobj} & : & \mathsf{obj}_C, \\ & \mathsf{tarr} & : & \Pi o : \mathsf{obj}_C.\ \underline{\mathsf{hom}}_C\ o\ \mathsf{tobj}, \\ & \mathsf{tlaw} & : & \forall o : \mathsf{obj}_C.\ \forall f : \underline{\mathsf{hom}}_C\ o\ \mathsf{tobj}.\ f =_{\mathsf{hom}_C\ o\ \mathsf{tobj}} \mathsf{tarr}\ o\rangle.\end{array}$$

As appears from the above definition, terminal objects are understood constructively. This constructive reading of categorical notions is in line with for example, [HS00, Saï98] and is more appropriate for the issues tackled here.

Initial objects are defined similarly.

Definition 5.4.4 *The parametric type* $\textsc{Iobj}_t$ *of* initial objects *is defined as:*

$$\begin{array}{lrcl}\lambda C : \textsc{Cat}_t.\ \langle & \mathsf{iobj} & : & \mathsf{obj}_C, \\ & \mathsf{iarr} & : & \Pi o : \mathsf{obj}_C.\ \underline{\mathsf{hom}}_C\ \mathsf{iobj}\ o, \\ & \mathsf{ilaw} & : & \forall o : \mathsf{obj}_C.\ \forall f : \underline{\mathsf{hom}}_C\ \mathsf{iobj}\ o.\ f =_{\mathsf{hom}_C\ \mathsf{iobj}\ o\ \mathsf{iarr}\ o}\rangle.\end{array}$$

The structure of binary products on a category is defined as a record type.

Definition 5.4.5 *The parametric type* $\textsc{Prod}_t$ *is defined as*

$$\begin{array}{lrcl}\lambda C : \textsc{Cat}_t.\ \langle & \mathsf{prodo} & : & \mathsf{obj}_C \to \mathsf{obj}_C \to \mathsf{obj}_C, \\ & \mathsf{proda} & : & \Pi o, o', o'' : \mathsf{obj}_C.\ \textsc{bmap}_t\ (\mathsf{hom}_C\ o\ o') \\ & & & \qquad\qquad\qquad\qquad\qquad\quad (\mathsf{hom}_C\ o\ o'') \\ & & & \qquad\qquad\qquad\qquad\qquad\quad (\mathsf{hom}_C\ o\ (\mathsf{prodo}\ o'\ o'')), \\ & \mathsf{prodl} & : & \Pi o, o' : \mathsf{obj}_C.\ \underline{\mathsf{hom}}_C\ (\mathsf{prodo}\ o\ o')\ o \\ & \mathsf{prodr} & : & \Pi o, o' : \mathsf{obj}_C.\ \underline{\mathsf{hom}}_C\ (\mathsf{prodo}\ o\ o')\ o', \\ & \mathsf{prodlaw} & : & \cdots\rangle.\end{array}$$

Given a category $\mathcal{C}$ with a product structure $\mathsf{prod} : \textsc{Prod}_t\ \mathcal{C}$, we use the notation $o \times o'$ for $\mathsf{prod} \cdot \mathsf{prodo}\ o\ o'$.

Given two morphisms of $\mathcal{C}$, $f : \underline{\mathcal{C} \cdot \mathsf{hom}}\ o_1\ o_2$ and $f' : \underline{\mathcal{C} \cdot \mathsf{hom}}\ o'_1\ o'_2$, we can define a product morphism $f \times f' : \underline{\mathcal{C} \cdot \mathsf{hom}}\ o_1 \times o'_1\ o_2 \times o'_2$.

We can now define a *cartesian closed category* (CCC) as a structure consisting of ($\mathcal{C}^{\mathrm{op}}$ is the *opposite category* of $\mathcal{C}$, that is, the category with the same objects and morphisms as $\mathcal{C}$ but with source and target of morphisms exchanged):

- a category $\mathcal{C}$;
- a terminal object $\bullet$;
- a bifunctor for products $\times : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$;
- a bifunctor for exponents $\Rightarrow : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{C}$;
- an evaluation map for every pair of objects o and o':

$$\mathsf{eval}_{o,o'} : \underline{\mathsf{hom}}\ ((o \Rightarrow o') \times o)\ o';$$

- an abstraction map for every three objects o, o', and o'':

$$\mathsf{abst}_{o,o',o''} : \mathrm{MAP}_t\ (\mathsf{hom}\ (o \times o')\ o'')\ (\mathsf{hom}\ o\ (o' \Rightarrow o''));$$

- a proof that for every $f : \underline{\mathsf{hom}}\ (o \times o')\ o''$, $\mathsf{ap}_t\ \mathsf{abst}_{o,o',o''}\ f$ is the unique morphism that gives back f when composed with $\mathsf{eval}_{o',o''}$.

Definition 5.4.6 *The type of* cartesian closed categories CCC *is the record type*

$$\begin{array}{rcl}
\langle \mathsf{cccat} & : & \mathrm{CAT}_t, \\
\mathsf{terminal} & : & \mathrm{TOBJ}_t\ \mathsf{cccat}, \\
\mathsf{ccprod} & : & \mathrm{PROD}_t\ \mathsf{cccat}, \\
\mathsf{ccexp} & : & \mathsf{cccat} \cdot \mathsf{obj} \to \mathsf{cccat} \cdot \mathsf{obj} \to \mathsf{cccat} \cdot \mathsf{obj}, \\
\mathsf{cceval} & : & \Pi o, o' : \mathsf{cccat} \cdot \mathsf{obj}.\underline{\mathsf{cccat} \cdot \mathsf{hom}}\ ((o \Rightarrow o') \times o)\ o', \\
\mathsf{ccabst} & : & \Pi o, o', o'' : \mathsf{cccat} \cdot \mathsf{obj}.\mathrm{MAP}_t\ (\mathsf{cccat} \cdot \mathsf{hom}\ (o \times o')\ o'') \\
 & & \qquad\qquad (\mathsf{ccat} \cdot \mathsf{hom}\ o\ (o' \Rightarrow o'')), \\
\mathsf{cceq} & : & \forall o, o', o'' : \mathsf{cccat} \cdot \mathsf{obj}.\forall f : \underline{\mathsf{cccat} \cdot \mathsf{hom}}\ (o \times o')\ o''. \\
 & & ((\mathsf{ap}_t\ \mathsf{ccabst}_{o,o',o''}\ f) \times (\mathsf{cccat} \cdot \mathsf{id}\ o')) \bullet \mathsf{cceval}_{o',o''} \\
 & & \quad =_{(\mathsf{cccat} \cdot \mathsf{hom}\ (o \times o')\ o'')} f, \\
\mathsf{ccunique} & : & \forall o, o', o'' : \mathsf{cccat} \cdot \mathsf{obj}. \\
 & & \forall f : \underline{\mathsf{cccat} \cdot \mathsf{hom}}\ (o \times o')\ o''.\forall g : \underline{\mathsf{cccat} \cdot \mathsf{hom}}\ o\ (o' \Rightarrow o''). \\
 & & (g \times (\mathsf{cccat} \cdot \mathsf{id}\ o')) \bullet \mathsf{cceval}_{o',o''} =_{(\mathsf{cccat} \cdot \mathsf{hom}\ (o \times o')\ o'')} f \\
 & & \to g =_{(\mathsf{cccat} \cdot \mathsf{hom}\ o\ (o' \Rightarrow o''))} \mathsf{ccabst}_{o,o',o''}\ f\rangle
\end{array}$$

where we used the notations $o \times o'$ *for* $(\mathsf{cccprod} \cdot \mathsf{prodo}\ o\ o')$ *and* $o \Rightarrow o'$ *for* $(\mathsf{ccexp}\ o\ o')$ *and we wrote the object parameters as indexes, for example we wrote* $\mathsf{cceval}_{o',o''}$ *for* $(\mathsf{cceval}\ o'\ o'')$.

Note that this is a constructive definition of cartesian closed category: the existence of products and exponents is not just postulated, but it must be given by two functors. In defining a category we must then specify what its exponents are. It is possible to have two categories with the same objects and morphisms, but with different notions of exponent. Such categories are considered distinct in our approach. In general, we need the axiom of choice to infer the existence of a cartesian closed category structure from the existence of an exponent object for every pair of objects. It is shown in [Jac99] that, in developing *internal category theory*, that is, category theory inside a given category, this requirement is not always met.

Given a category $\mathcal{C}$, it is not formally correct, in our approach, to ask if it is cartesian closed, because the notion of a CCC is not a predicate over categories, but a structure containing a category as one of its fields. It may be understood to indicate the predicate stating the existence of such structure. But in the following discussion, since we deal with categories that already have a candidate for the exponent object (the function setoid), we mean to investigate whether this definition of exponent can be the corresponding field of a cartesian closed category structure.

In Section 5.5 we investigate if the categories **TSet**, **PSet**, **QSet**, **RSet** and **SSet** are cartesian closed or not. This must be intended in the following strict sense: Can the structure

$$\langle \mathbf{TSet}, \dagger\mathsf{Unit}, \times_t, \textsc{Map}_t \rangle$$

be completed to a cartesian closed category? Here, $\times_t$ is the natural definition for product of setoids, that is, $S_1 \times_t S_2$ is the setoid having the type-theoretic product of the carriers of S_1 and S_2 has carrier, componentwise book equality, and the type-theoretic projections as projection functions. In other words: Is there a cartesian closed category $\mathcal{TS} \colon CCC$ such that

- $\mathcal{TS} \cdot \mathsf{cccat} = \mathbf{TSet}$;
- $\mathcal{TS} \cdot \mathsf{terminal} = \dagger\mathsf{Unit}$;
- $\mathcal{TS} \cdot \mathsf{ccprod} = \times_t$;
- $\mathcal{TS} \cdot \mathsf{ccexp} = \textsc{Map}_t$?

5.5 Basic results on the categories of setoids

The purpose of this section is two-fold:

- in the next subsection, we perform a sanity check and check whether all categories defined in the previous section form a model of the simply typed λ-calculus. It turns out that **PSet** does not form such a model and, hence, may be considered unsuitable to serve as the basis for formalising mathematics;
- in the following subsection, we compare the categories **TSet**, **PSet**, **QSet**, **RSet** and **SSet**. It turns out that in most cases the obvious functors between these categories do not yield an equivalence.

Many of the results presented in this section involve empty setoids. For total setoids, an empty setoid is a setoid whose underlying type is empty. For partial setoids an empty setoid is a setoid A such that there is no $a : \mathsf{el}_p\ A$ verifying $x =_A x$; hence every type has an associated empty partial setoid, as defined below.

Definition 5.5.1 *For any type T, the empty partial setoid $\emptyset\ T$ over T is defined by*

$$\begin{array}{rcl} \emptyset\ T := \langle\ \mathsf{el}_p & = & T, \\ \mathsf{eq}_p & = & \lambda x, y : T.\ \bot, \\ \mathsf{per} & = & \langle\ \mathsf{sym}_p = \lambda x, y : T. \lambda p : \bot.\ p, \\ & & \quad \mathsf{trans}_p = \lambda x, y : T. \lambda p, q : \bot.\ p \rangle\rangle. \end{array}$$

Cartesian closedness

In this subsection, we examine which categories are cartesian closed, that is, form a model of simply typed λ-calculus (see, for example, [LS86]).

We are interested in determining whether the categories **TSet**, **PSet**, **QSet**, **RSet** and **SSet** with their associated function space setoid, as defined in the previous section, are cartesian closed.

Lemma 5.5.2
- **TSet**, **QSet**, **RSet** *and* **SSet** *are cartesian closed categories.*
- **PSet** *is not cartesian closed.*

Proof We only show that the category **PSet** is not cartesian closed. This is done by exploiting anomalies related to "empty" partial setoids. Let

$$\mathbf{1}_p := \langle \mathsf{el}_p = \mathsf{Unit}, \mathsf{eq}_p = \lambda x, y : \mathsf{Unit}.\mathsf{true}, \mathsf{per} = \cdots \rangle$$

and $A = \emptyset\ \mathsf{Unit}$. We prove that the type $\textsc{map}_p\ A\ (\textsc{Map}_p\ \mathbf{1}_p\ A)$ is empty: indeed, let $f : \textsc{map}_p\ A\ (\textsc{Map}_p\ \mathbf{1}_p\ A)$. Then $((f \cdot \mathsf{ap}_p\ *) \cdot \mathsf{ext}_p\ *\ *\ \mathsf{triv}) : \mathsf{false}$, where $*$ is the only inhabitant of Unit and triv is the proof of true; this is impossible by consistency of the system.

On the other hand, if **PSet** were a cartesian closed category with $\textsc{Map}_p$ as exponent then the type $\textsc{map}_p\ A\ (\textsc{Map}_p\ \mathbf{1}_p\ A)$ would be inhabited. Hence **PSet** is not cartesian closed. □

Summing up, **PSet** is not a model of simply typed λ-calculus, and hence does not provide a suitable basis for formalising mathematics in type theory. On the other hand, the categories **TSet**, **QSet**, **RSet** and **SSet** form a cartesian closed category and thus pass our sanity check.

One may wonder about more stringent sanity checks. For example, one could check which of the categories **TSet**, **QSet**, **RSet** and **SSet** do form a model of dependent type theory. This issue has been investigated in depth by M. Hofmann, see for example, [Hof95a]. It turns out that the category **RSet** does form a model of dependent type theory, whereas there are some difficulties with **TSet**. In particular, it is problematic to define a family of setoids depending on a setoid.

Alternately, one could check which of the categories **TSet**, **QSet**, **RSet** and **SSet** do form a model of intuitionistic set theory, that is, a topos [LS86]. It turns out that none of the categories forms a topos because of the distinction between total functional relations and functions. In Section 5.6, we study choice principles which turn these categories into toposes.

Equivalence between categories

The purpose of this paragraph is to establish (in-)equivalences between categories. Informally, two categories are equivalent if there are mutually inverse, up to a natural isomorphism, functors from one to another. More precisely, an equivalence between two categories $\mathcal{C}$ and $\mathcal{D}$ is a tuple $\langle F, G, \eta, \epsilon \rangle$ where

- F is a functor from $\mathcal{C}$ to $\mathcal{D}$;
- G is a functor from $\mathcal{D}$ to $\mathcal{C}$;
- η is a natural isomorphism from $\mathsf{id}_{\mathcal{C}}$ to $G \circ F$;
- ϵ is a natural isomorphism from $F \circ G$ to $\mathsf{id}_{\mathcal{D}}$.

The components of the natural transformations η and ϵ are morphisms, so the notion of equivalence depend on the chosen notion of setoid morphism. In particular, the above definition is not appropriate to compare categories that do not belong to the same framework, for example, **TSet** which is a T-category and **RSet** which is an R-category. One remedy to this problem is to define transformations that map categories in one formalism to categories in another formalism. Going back to **TSet** and **RSet**, one can for example transform **TSet** into an R-category $\mathbf{TSet}_r$ and compare it to **RSet**. Here we prefer to remain at an informal level of discussion, since our main purpose is to stress that not all existing alternatives are equivalent.

In the sequel, only some of the most interesting comparisons are detailed, but Figure 5.1 summarises all comparisons w.r.t. the obvious functors. Note that, despite the fact that **PSet** cannot be completed to a cartesian closed category, it is still interesting to compare it to the other four categories. In fact, since the notion of equivalence does not take into account the exponent objects, the functors F and G do not need to preserve the function setoids. We will see that, if we do not require that the functors preserve function setoids, there is in fact a isomorphism between **PSet** and **RSet**.

Comparing TSet and PSet

There are two canonical functors $\mathcal{TP} : \mathbf{TSet} \to \mathbf{PSet}$ and $\mathcal{PT} : \mathbf{PSet} \to \mathbf{TSet}$. As expected, neither $\mathcal{TP}$ nor $\mathcal{PT}$ yields an equivalence of categories.

The functor $\mathcal{TP} : \mathbf{TSet} \to \mathbf{PSet}$ is defined in the obvious way: its object part turns a total setoid into a partial setoid simply by forgetting about the reflexivity of equality, whereas its arrow part is the "identity".

Definition 5.5.3 *The functor $\mathcal{TP} : \mathbf{TSet} \to \mathbf{PSet}$ is defined as follows:*

- *Object part: if A is a total setoid, then*

$$\mathcal{TP}\ A := \textsc{Partial}\ A := \langle \begin{array}{rcl} \mathsf{el}_p & = & A \cdot \mathsf{el}_t, \\ \mathsf{eq}_p & = & A \cdot \mathsf{eq}_t, \\ \mathsf{per} & = & \ldots \rangle \end{array}$$

  is its corresponding partial setoid.

- *Arrow part: the arrow part of $\mathcal{TP}$ is the "identity".*

$\mathcal{TP}$ cannot induce any equivalence between **TSet** and **PSet**, as shown by the following lemma.

Lemma 5.5.4 *There exists a partial setoid $A : \text{SET}_p$ that is not isomorphic to the image of any total setoid.*

Proof We exhibit a partial setoid $A : \text{SET}_p$ such that for every $B : \text{SET}_t$, either $\text{MAP}_p\ A\ (\mathcal{TP}\ B)$ or $\text{MAP}_p\ (\mathcal{TP}\ B)\ A$ is empty. Indeed, assume $\vdash t : T$ and let $A = \emptyset\ T$. Now assume $B : \text{SET}_t$. We claim that in the empty context:

- $\text{MAP}_p\ (\mathcal{TP}\ B)\ A$ is not empty iff B is empty. Indeed, if $b : \mathsf{el}_t\ B$ and $f : \text{MAP}_p\ (\mathcal{TP}\ B)\ A$ then $f \cdot \mathsf{ext}_p\ b\ b\ (B \cdot \mathsf{er} \cdot \mathsf{refl}_t\ b) : \bot$, which is impossible by consistency of the system.
- $\text{MAP}_p\ A\ (\mathcal{TP}\ B)$ is not empty iff B is not empty.

The result follows. □

The functor $\mathcal{PT}$ mapping partial setoids to total setoids takes as object part the function mapping a partial setoid to its domain setoid, and as function part the corresponding transformation described below.

Definition 5.5.5 *The functor $\mathcal{PT} : \mathbf{PSet} \to \mathbf{TSet}$ is defined as follows:*

- *Object part: if A is a partial setoid, then*

$$\begin{array}{l}\mathcal{PT}\ A := \text{TOTAL}\ A := \\ \quad \langle\ \mathsf{el}_t\ =\ \text{DOMAIN}\ A, \\ \quad\ \mathsf{eq}_t\ =\ \lambda x, y : \text{DOMAIN}\ A.\ x \cdot \mathsf{cont} =_A y \cdot \mathsf{cont}, \\ \quad\ \mathsf{er}\ =\ \langle\ \mathsf{refl}_t\ =\ \lambda x : \text{DOMAIN}\ A.\ x \cdot \mathsf{def}, \\ \qquad\qquad \mathsf{sym}_t\ =\ \lambda x, y : \text{DOMAIN}\ A. \\ \qquad\qquad\qquad\qquad A \cdot \mathsf{per} \cdot \mathsf{sym}_p\ x \cdot \mathsf{cont}\ y \cdot \mathsf{cont}, \\ \qquad\qquad \mathsf{trans}_t\ =\ \lambda x, y, z : \text{DOMAIN}\ A. \\ \qquad\qquad\qquad\qquad A \cdot \mathsf{per} \cdot \mathsf{trans}_p\ x \cdot \mathsf{cont}\ y \cdot \mathsf{cont}\ z \cdot \mathsf{cont}\rangle\rangle\end{array}$$

  *is its corresponding total setoid—*DOMAIN *A differs from* TOTAL *A by the name of its fields.*

- *Arrow part: if g is a morphism of partial setoids from A to B, that is, $g : \text{MAP}_p\ A\ B$ then $\mathcal{PT}\ g$ is defined as the record*

$$\begin{array}{rcl}\mathcal{PT}\ g := \langle\ \mathsf{ap}_t & = & \lambda x : \text{DOMAIN}\ A. \\ & & \langle\ \mathsf{cont}\ =\ \mathsf{ap}_p\ g\ x \cdot \mathsf{cont}, \\ & & \quad \mathsf{def}\ =\ g \cdot \mathsf{ext}_p\ x \cdot \mathsf{cont}\ x \cdot \mathsf{cont}\ x \cdot \mathsf{def}\rangle, \\ \mathsf{ext}_t & = & \lambda x, y : \text{DOMAIN}\ A.\ g \cdot \mathsf{ext}_p\ x \cdot \mathsf{cont}\ y \cdot \mathsf{cont}\rangle.\end{array}$$

$\mathcal{PT}$ cannot induce any equivalence between **TSet** and **PSet**, as shown by the following lemma.

Lemma 5.5.6 *$\mathcal{PT}$ is not full, that is, there exist two partial setoids A and B such that $\text{MAP}_p\ A\ B$ is empty but $\text{MAP}_t\ (\mathcal{PT}\ A)\ (\mathcal{PT}\ B)$ is not.*

Proof Assume $\vdash t : T$ and let $A := \emptyset\ T$ and $B := \emptyset\ (\text{DOMAIN}\ A)$. We claim that in the empty context:

- there is no $g : \text{MAP}_p\ A\ B$. If there were such a g, then $(\mathsf{ap}_p\ g\ t) \cdot \mathsf{def} : \bot$. This is impossible by consistency of the system;
- there is a $g : \text{MAP}_t\ (\text{TOTAL}\ A)\ (\text{TOTAL}\ B)$. Indeed, consider

$$\begin{array}{rcl} g := \langle\ \mathsf{ap}_t & = & \lambda x : \text{DOMAIN}\ A.\ \langle \mathsf{cont} = x,\ \mathsf{def} = x \cdot \mathsf{def}\rangle, \\ \mathsf{ext}_t & = & \lambda x, y : \text{DOMAIN}\ A.\ \lambda p : \bot.\ p\rangle. \end{array}$$

  The expression g is indeed of type $\text{MAP}_t\ (\text{TOTAL}\ A)\ (\text{TOTAL}\ B)$ since we have

$$\begin{array}{rcl} \mathsf{el}_t\ (\text{TOTAL}\ A) & =_\beta & \text{DOMAIN}\ A \\ \mathsf{el}_t\ (\text{TOTAL}\ B) & =_\beta & \text{DOMAIN}\ (\text{DOMAIN}\ A) \\ (x =_{(\text{TOTAL}\ A)} y) & =_\beta & \bot \\ (\mathsf{ap}_t\ x =_{(\text{TOTAL}\ B)} \mathsf{ap}_t\ y) & =_\beta & \bot. \end{array}$$

□

Comparing TSet and QSet

There are two canonical functors $\mathcal{TQ} : \mathbf{TSet} \to \mathbf{QSet}$ and $\mathcal{QT} : \mathbf{QSet} \to \mathbf{TSet}$. They form an equivalence pair.

Definition 5.5.7 *The functor $\mathcal{TQ} : \mathbf{TSet} \to \mathbf{QSet}$ is defined as follows:*

- *Object part: if A is a total setoid, then $\mathcal{TQ}\ A = \text{PARTIAL}\ A$ is its corresponding partial setoid.*
- *Arrow part: let A and B be total setoids and $f : \text{MAP}_t\ A\ B$. Then we define $\mathcal{TQ}\ f : \text{MAP}_q\ (\mathcal{TQ}\ A)\ (\mathcal{TQ}\ B)$ by the components*

$$\begin{array}{rcl} (\mathcal{TQ}\ f) \cdot \mathsf{ap}_q & : & \Pi a : \mathsf{el}_p\ (\mathcal{TQ}\ A).a =_{(\mathcal{TQ}\ A)} a \to \mathsf{el}_p\ (\mathcal{TQ}\ B) \\ & : & \Pi a : \mathsf{el}_t\ A.a =_A a \to \mathsf{el}_t\ B \\ (\mathcal{TQ}\ f) \cdot \mathsf{ap}_q & := & \lambda a : \mathsf{el}_t\ A.\lambda\phi : a =_A a.\mathsf{ap}_t\ f\ a \end{array}$$

$$\begin{array}{rcl} (\mathcal{TQ}\ f) \cdot \mathsf{ext}_q & : & \forall x, y : \mathsf{el}_p\ (\mathcal{TQ}\ A). \\ & & \forall\phi : x =_{(\mathcal{TQ}\ A)} x.\forall\psi : y =_{(\mathcal{TQ}\ A)} y. \\ & & x =_{(\mathcal{TQ}\ A)} y \to \\ & & (\mathsf{ap}_q\ (\mathcal{TQ}\ f)\ x\ \phi) =_{(\mathcal{TQ}\ B)} (\mathsf{ap}_q\ (\mathcal{TQ}\ f)\ y\ \psi) \\ & : & \forall x, y : \mathsf{el}_t\ A.\forall\phi : x =_A x.\forall\phi : y =_A y \\ & & x =_A y \to (\mathsf{ap}_t\ f\ x) =_B (\mathsf{ap}_t\ f\ y) \\ (\mathcal{TQ}\ f) \cdot \mathsf{ext}_q & := & \lambda x, y : \mathsf{el}_p\ (\mathcal{TQ}\ A). \\ & & \lambda\phi : x =_{(\mathcal{TQ}\ A)} x.\lambda\psi : y =_{(\mathcal{TQ}\ A)} y. \\ & & f \cdot \mathsf{ext}_t\ x\ y. \end{array}$$

Definition 5.5.8 *The functor $\mathcal{QT} : \mathbf{QSet} \to \mathbf{TSet}$ is defined as follows:*

- *Object part: if A is a partial setoid, then $\mathcal{QT}\ A = \textsc{Total}\ A$ is its corresponding total setoid.*

- *Arrow part: let A and B be partial setoids and $f : \textsc{map}_q\ A\ B$. Then we define $\mathcal{QT}\ f : \textsc{map}_t\ (\mathcal{QT}\ A)\ (\mathcal{QT}\ B)$ by the components*

$$\begin{array}{rcl}
(\mathcal{QT}\ f)\cdot \mathsf{ap}_t & : & \mathsf{el}_t\ (\mathcal{QT}\ A) \to \mathsf{el}_t\ (\mathcal{QT}\ B) \\
 & : & \textsc{domain}\ A \to \textsc{domain}\ B \\
(\mathcal{QT}\ f)\cdot \mathsf{ap}_t & := & \lambda x : \textsc{domain}\ A. \\
 & & \langle\ \mathsf{cont} = \mathsf{ap}_q\ f\ x\cdot\mathsf{cont}\ y\cdot\mathsf{cont}, \\
 & & \quad \mathsf{def} = f\cdot\mathsf{ext}_q\ x\cdot\mathsf{cont}\ x\cdot\mathsf{cont}\ y\cdot\mathsf{cont}\ y\cdot\mathsf{cont}\rangle
\end{array}$$

$$\begin{array}{rcl}
(\mathcal{QT}\ f)\cdot \mathsf{ext}_t & : & \forall x, y : \mathsf{el}_t\ (\mathcal{QT}\ A).(x =_{(\mathcal{QT}\ A)} y) \\
 & & \to \mathsf{ap}_t\ (\mathcal{QT}\ f)\ x =_{(\mathcal{QT}\ B)} \mathsf{ap}_t\ (\mathcal{QT}\ f)\ y \\
 & : & \forall x, y : \textsc{domain}\ A. x\cdot\mathsf{cont} =_A x\cdot\mathsf{cont} \to \\
 & & (\mathsf{ap}_q\ f\ x\cdot\mathsf{cont}\ x\cdot\mathsf{def}) =_B (\mathsf{ap}_q\ f\ y\cdot\mathsf{cont}\ y\cdot\mathsf{def}) \\
(\mathcal{QT}\ f)\cdot \mathsf{ext}_t & := & \lambda x, y : \textsc{domain}\ A. \\
 & & f\cdot\mathsf{ext}_q\ x\cdot\mathsf{cont}\ y\cdot\mathsf{cont}\ x\cdot\mathsf{def}\ y\cdot\mathsf{def}.
\end{array}$$

To show that these functors form an equivalence we look at their two compositions and show that they are naturally isomorphic to the identity functor in the respective categories.

Let A : **TSet**, then we have

$$\begin{array}{rcl}
\mathcal{QT}\ (\mathcal{TQ}\ A) & : & \mathbf{TSet} \\
 & = & \textsc{Total}\ (\textsc{Partial}\ A) \\
 & = & \langle\ \mathsf{el}_t = \textsc{domain}\ (\textsc{Partial}\ A) \\
 & & \quad\ = \langle \mathsf{cont} : \mathsf{el}_t\ A, \mathsf{def} : \mathsf{cont} =_A \mathsf{cont}\rangle, \\
 & & \mathsf{eq}_t = \cdots \\
 & & \mathsf{er} = \cdots\rangle.
\end{array}$$

Let B : **QSet**, then we have

$$\begin{array}{rcl}
\mathcal{TQ}\ (\mathcal{QT}\ B) & : & \mathbf{QSet} \\
 & = & \textsc{Partial}\ (\textsc{Total}\ B) \\
 & = & \langle\ \mathsf{el}_p = \mathsf{el}_t\ (\textsc{Total}\ A) = \textsc{domain}\ B, \\
 & & \mathsf{eq}_p = \cdots \\
 & & \mathsf{per} = \cdots\rangle.
\end{array}$$

The unit of the equivalence is then defined, for every A : **TSet** as the morphism

$$\begin{array}{rcl}
\eta_A & : & \textsc{map}_t\ A\ (\mathcal{QT}\ (\mathcal{TQ}\ A)) \\
\eta_A & := & \langle\ \mathsf{ap}_t : \mathsf{el}_t\ A \to \mathsf{el}_t\ (\mathcal{QT}\ (\mathcal{TQ}\ A)) \\
 & & \quad\ : \mathsf{el}_t\ A \to \langle \mathsf{cont} : \mathsf{el}_t\ A, \mathsf{def} : \mathsf{cont} =_A \mathsf{cont}\rangle \\
 & & \quad\ = \lambda x : \mathsf{el}_t\ A.\langle \mathsf{cont} = x, \mathsf{def} : A\cdot\mathsf{er}\cdot\mathsf{refl}_t\ x\rangle, \\
 & & \mathsf{ext}_t = \cdots\rangle.
\end{array}$$

η is a natural isomorphism between the identity functor $\mathcal{I}_{\mathbf{TSet}}$ and the functor $\mathcal{QT} \circ \mathcal{TQ}$.

The counit of the equivalence is defined, for every $B : \mathbf{QSet}$ as the morphism

$$\begin{array}{lcll} \epsilon_B & : & \text{MAP}_q\ (\mathcal{TQ}\ (\mathcal{QT}\ B))\ B & \\ \epsilon_B & := & \langle\ \mathsf{ap}_q & : \Pi x : \mathsf{el}_p\ (\mathcal{TQ}\ (\mathcal{QT}\ B)).x =_{(\mathcal{TQ}\ (\mathcal{QT}\ B))}\ x \rightarrow_p B \\ & & & : \Pi x : \text{DOMAIN}\ B.x =_{(\mathcal{TQ}\ (\mathcal{QT}\ B))}\ x \rightarrow_p B \\ & & & = \lambda x : \text{DOMAIN}\ B.\lambda\phi : x =_{(\mathcal{TQ}\ (\mathcal{QT}\ B))}\ x.x, \\ & & \mathsf{ext}_q & = \cdots\rangle. \end{array}$$

ϵ is a natural isomorphism between the functor $\mathcal{TQ} \circ \mathcal{QT}$ and the identity functor $\mathcal{I}_{\mathbf{QSet}}$.

We conclude that the two categories are equivalent.

Theorem 5.5.9 *The categories* **TSet** *and* **QSet** *are equivalent.* $\langle \mathcal{TQ}, \mathcal{QT}, \eta, \epsilon\rangle$ *is an equivalence between them.*

Comparing TSet and RSet

There are two canonical functors $\mathcal{TR} : \mathbf{TSet} \rightarrow \mathbf{RSet}$ and $\mathcal{RT} : \mathbf{RSet} \rightarrow \mathbf{TSet}$. As expected, neither $\mathcal{TR}$ nor $\mathcal{RT}$ yields an equivalence of categories.

The functor $\mathcal{TR}$ is defined in the obvious way: its object part forgets the reflexivity of equality, and its arrow part forgets the extensionality of the morphism. As to $\mathcal{RT}$, it has the same object part as $\mathcal{PT}$, but the function part needs to be modified accordingly.

Definition 5.5.10 *The functor* $\mathcal{RT} : \mathbf{RSet} \rightarrow \mathbf{TSet}$ *is defined as follows:*

- *Object part: if A is a partial setoid, then* TOTAL A *is its corresponding total setoid;*
- *Arrow Part: if g is a defined morphism of partial setoids from A to B, that is, $g : \text{MAP}_r\ A\ B$ and $\phi : g =_{\text{MAP}_r\ A\ B} g$ then $\mathcal{RT}\ g$ is defined as the record*

$$\begin{array}{lcll} \mathcal{RT}\ g := \langle\ \mathsf{ap}_t & = & \lambda x : \text{DOMAIN}\ A.\ \langle\ \mathsf{cont} & = & \mathsf{ap}_p\ g\ x \cdot \mathsf{cont}, \\ & & \mathsf{def} & = & g \cdot \mathsf{ext}_p\ (x \cdot \mathsf{def})\rangle, \\ \mathsf{ext}_t & = & \phi\rangle. & & \end{array}$$

One can show that the canonical functors between **TSet** and **RSet** cannot give an equivalence—for example, we can prove that $\mathcal{RT}$ is not full as in Lemma 5.5.6. We can strengthen this result by proving that there are no functors giving an equivalence of the two categories.

Theorem 5.5.11 *The categories* **TSet** *and* **RSet** *are not equivalent.*

Proof Assume that there are functors $\mathcal{F} : \mathbf{TSet} \rightarrow \mathbf{RSet}$ and $\mathcal{G} : \mathbf{RSet} \rightarrow \mathbf{TSet}$ that make the two categories equivalent. An equivalence maps initial objects to initial objects. The initial object of **RSet** is $\emptyset$ Empty, where Empty

is the empty type and so $\mathcal{G}\ (\emptyset\ \mathsf{Empty})$ is an initial object of **TSet**. On the other hand $\emptyset\ \mathsf{Unit}$ is not an initial object of **RSet** and therefore $\mathcal{G}\ (\emptyset\ \mathsf{Unit})$ is not an initial object of **TSet**.

But in **RSet** there is at most one morphism from $\emptyset\ \mathsf{Unit}$ to any other object A, precisely none if A is an initial object, one otherwise. This property must then hold in **TSet** for $\mathcal{G}\ (\emptyset\ \mathsf{Unit})$. In particular, there can be at most one morphism from $\mathcal{G}\ (\emptyset\ \mathsf{Unit})$ to $\ddagger\,\mathsf{Bool}$, where Bool is the type with exactly two elements true and false. For any total setoid B, we can define the constant functions $c_{\mathsf{true}}, c_{\mathsf{false}} : \mathrm{MAP}_t\ B\ (\ddagger\,\mathsf{Bool})$. It is easy to prove that $c_{\mathsf{true}} =_{(\mathrm{MAP}_t\ B\ (\ddagger\,\mathsf{Bool}))} c_{\mathsf{false}}$ if and only if B is an initial object. In the case $B = \mathcal{G}\ (\emptyset\ \mathsf{Unit})$ there can be at most one morphism to any setoid, so c_{true} and c_{false} must be equal.

We conclude that $\mathcal{G}\ (\emptyset\ \mathsf{Unit})$ is an initial object of **TSet**, and therefore it is isomorphic to $\mathcal{G}\ (\emptyset\ \mathsf{Empty})$. But this is impossible because $\emptyset\ \mathsf{Unit}$ and $\emptyset\ \mathsf{Empty}$ are not isomorphic in **RSet**. □

Comparing TSet and SSet

The categories **TSet** and **SSet** are equivalent. The shortest way to show this fact is to prove that **SSet** is equivalent to **QSet**, and then we obtain the equivalence to **TSet** by Theorem 5.5.9. There are two canonical functors $\mathcal{QS} : \mathbf{QSet} \to \mathbf{SSet}$ and $\mathcal{SQ} : \mathbf{SSet} \to \mathbf{QSet}$. They are simply the identity on objects. On morphisms, $\mathcal{QS}$ just forgets the proof component ext_q, while $\mathcal{SQ}$ on the defined elements of $\mathrm{MAP}_s\ A\ B$, that is the functions $f : \mathrm{MAP}_s\ A\ B$ for which there is a proof $\xi : f =_{(\mathrm{MAP}_s\ A\ B)} f$, is

$$\mathcal{SQ}\ f\ \xi := \langle \mathsf{ap}_q = f, \mathsf{ext}_q = \xi \rangle$$

$\mathcal{QS}$ and $\mathcal{QS}$ are inverse of each other, so

Theorem 5.5.12 *The categories* **SSet** *and* **QSet** *are isomorphic. The categories* **SSet** *and* **TSet** *are equivalent.*

Comparing PSet and RSet

We proved in Lemma 5.5.2 that **PSet** cannot be completed to a cartesian closed category having $\mathrm{MAP}_p\ A\ B$ as exponent object. Therefore there are no equivalences between **PSet** and any of the other four categories that preserves the setoid of functions. Anyway, there is an equivalence between **PSet** and **RSet** that does not preserve the setoid of functions. This is yet another argument against the use of $\mathrm{MAP}_p\ A\ B$ as function setoid.

Definition 5.5.13 *The functor* $\mathcal{PR} : \mathbf{PSet} \to \mathbf{RSet}$ *is defined as follows:*

- *Object part: the identity.*
- *Arrow Part: if* f *is a morphism of partial setoids from* A *to* B *in* **PSet**, *i.e.* $f : \mathrm{MAP}_p\ A\ B$, *then* $\mathcal{PR}\ f$ *is simply* $f \cdot \mathsf{ap}_p$.

To From	**TSet**	**PSet**	**QSet**	**RSet**	**SSet**
TSet		n.f.	e.	n.f.	e.
PSet	n.e.		n.e.	is.*	n.e.
QSet	e.	n.f.		n.f.	is.
RSet	n.e.	is.*	n.e.		n.e.
SSet	e.	n.e.	is.	n.e.	

n.e. = not an equivalence
n.f. = not full (implies n.e.)
e. = equivalence
is. = isomorphism (implies e.)

Figure 5.1: COMPARISONS BETWEEN CATEGORIES

Definition 5.5.14 *The functor* $\mathcal{RP} : \mathbf{RSet} \to \mathbf{PSet}$ *is defined as follows:*

- *Object part: the identity.*
- *Arrow part: if* f *is a morphism of partial setoids from* A *to* B *in* **PSet**, *i.e.* $f : \text{MAP}_r\ A\ B$ *such that* $f =_{\text{MAP}_r\ A\ B} f$, *then* $\mathcal{RP}\ f$ *is the record*

$$\mathcal{RP}\ f = \langle \mathsf{ap}_p = f, \mathsf{ext}_p = \xi \rangle$$

where ξ *is a proof that* f *is a defined element of* $\text{MAP}_r\ A\ B$, *i.e.* $\xi : (f =_{\text{MAP}_r\ A\ B} f) = \forall x : \mathsf{el}_p\ A.\ (x =_A x) \to (\mathsf{ap}_p\ f\ x) =_B (\mathsf{ap}_p\ f\ x)$.

It is trivial to verify that $\mathcal{PR}$ and $\mathcal{RP}$ with the identity natural transformations form an equivalence, actually even an isomorphism, between **PSet** and **RSet**. Since **RSet** is a cartesian closed category, so is **PSet**. However, the exponent objects obtained through the equivalence are different from and not in general isomorphic to the function setoids $\text{MAP}_p\ A\ B$. In fact, if we start with two objects A and B of **PSet**, map them to $\mathcal{PR}\ A$ and $\mathcal{PR}\ B$ in **RSet**, form the exponent object $(\mathcal{PR}\ A) \Rightarrow_r (\mathcal{PR}\ B)$, and map it back to $\mathcal{RP}\ ((\mathcal{PR}\ A) \Rightarrow_r (\mathcal{PR}\ B))$ in **PSet**, the final result, considering that $\mathcal{PR}$ and $\mathcal{RP}$ are the identity on objects, is

$$A \Rightarrow_p B = \text{MAP}_r\ A\ B$$

which is not in general isomorphic to $\text{MAP}_p\ A\ B$. The conclusion is that $\text{MAP}_p\ A\ B$ is incorrect as a setoid of functions, and that one should use **RSet** in place of **PSet**. For this reason we put a star on the isomorphism between these two categories in Figure 5.1.

We draw the conclusion that there are three categories of setoids up to equivalence: **PSet**, that is not cartesian closed and therefore is rejected immediately; **TSet**, that is equivalent to **QSet** and **SSet**; and **RSet**. Therefore, from now on we will talk only about **TSet** and **RSet**. We will abuse terminology in speaking about *total setoids* when referring to objects of **TSet** and about *partial setoids* when referring to objects of **RSet**. We recall, however, that this choice is not precise, since one could use partial setoids in the formalizations given by **QSet** or **SSet**, and this choice may be more convenient for some practical reasons.

5.6 Choice principles

To complete the analysis of setoids as a type-theoretic formalisation of the notion of sets, we study their behaviour in relation to choice principles. The problem is mathematically relevant, since these principles are not satisfied by either total or partial setoids, and their addition has different effects on **TSet** and **RSet**. Before proceeding any further, let us dispose of a red herring. It may be objected that the assumption of an axiom is methodologically unjustified: setoids were devised to develop mathematics in type theory without external assumptions; if we are to assume axioms, we may as well assume all the axioms of set theory and dispense with setoids. While we agree that one should try to develop mathematics using only the constructions available in type theory, we still defend the importance of knowing the relation of the notion of setoids with choice principles, because this relation tells us much about the nature of setoids.

More specifically, we focus on the axiom of choice for setoids and the axiom of unique choice for setoids—the former leads to classical logic whereas the second is constructively valid. In particular, we show that the latter is consistent in both **TSet** and **RSet**, but it gives stronger results in **TSet**. In particular, we show that the *axiom of descriptions*, also called *axiom of unique choice*, for partial setoids is not powerful enough to ensure the existence of some very natural functions on partial setoids. Our conclusion is that **TSet** behaves better than **RSet** with respect to choice principles, and that is a good reason to chose it for the formalisation of mathematics, even if we have no intention to actually assume choice axioms.

The axiom of choice for types

The axiom of choice for types expresses that every total relation from U to V yields a type-theoretic function of type $U \to V$.

Definition 5.6.1 *Let U and V be two types. The type* $\mathrm{TR}\ U\ V$ *of total relations from U to V is defined as the record type*

$$\mathrm{TR}\ U\ V := \langle \quad \begin{array}{rcl} \mathsf{rel} & : & U \to V \to \mathsf{Prop}, \\ \mathsf{total} & : & \forall x : U. \exists y : V.\ \mathsf{rel}\ x\ y \end{array} \rangle.$$

The axiom of choice for types is given by the context Γ_{ACT}

$$\mathrm{ACT}_{make} \quad : \quad \Pi U, V : \mathsf{Type}.\ (\mathrm{TR}\ U\ V) \to U \to V,$$

$$\begin{array}{rcl} \mathrm{ACT}_{check} & : & \forall U, V : \mathsf{Type}. \forall R : \mathrm{TR}\ U\ V. \forall x : U. \\ & & \quad R \cdot \mathsf{rel}\ x\ (\mathrm{ACT}_{make}\ U\ V\ R\ x). \end{array}$$

The following result is well-known, see for example, [Coq90] and also [Wer97], where an easy model of the Calculus of Inductive Constructions is given in which the Γ_{ACT} is valid.

Proposition 5.6.2 Γ_{ACT} *is consistent, but not inhabited in the Calculus of Inductive Constructions.*

We conclude this subsection with two observations on the validity of the axiom of choice in other type-theoretical frameworks. First, the context Γ_{ACT} is instantiable in Martin-Löf's type theory [NPS90]. Second, it is also instantiable in the Calculus of Inductive Constructions if we use the impredicative universe both for data types and propositions. Let us clarify this last point. The Calculus of Inductive Constructions has two sorts, an impredicative sort $*$ and a predicative one $\Box$. The usual practice in developing mathematics in the Calculus of Inductive Constructions consists in using $*$ for propositions and $\Box$ for data types and mathematical structures. In this case $*$ and $\Box$ are renamed Prop and Type, respectively. In this case we cannot eliminate a proposition, an element of the impredicative sort, over a data type, an element of the predicative sort. The assumption that we can is inconsistent [Coq86]. So we cannot extract computational information (that is, functions) from logical information (proofs). An equally fruitful methodology consists in using $*$ both for data types and for propositions. In this case there is no type-theoretic distinction between propositions and data types so there is no problem in using logical information to obtain computational objects. A similar way is followed in the proof assistant Coq, where, however, there are two impredicative sorts Set and Prop, for data types and propositions, respectively. Elimination of elements of Prop over Set is not allowed, but only to preserve the distinction between computational and non-computational types. In principle such elimination would be possible without causing inconsistencies.

The axiom of choice for total setoids

The axiom of choice for total setoids states that every total relation between total setoids induces a map of total setoids. Recall that a relation from A to B, where A and B are total setoids, consists of a type-theoretical relation $R : (\mathsf{el}_t\ A) \to (\mathsf{el}_t\ B) \to \mathsf{Prop}$ and a proof that R is compatible.

Definition 5.6.3 *Let A and B be two total setoids. The type* $\textsc{Rel}_t\ A\ B$ *of relations from A to B is defined as the record type*

$$\begin{array}{rlcl} \textsc{Rel}_t\ A\ B := \langle & \mathsf{rel}_t & : & \mathsf{el}_t\ A \to \mathsf{el}_t\ B \to \mathsf{Prop}, \\ & \mathsf{compat}_t & : & \forall x, x' : \mathsf{el}_t\ A. \forall y, y' : \mathsf{el}_t\ B. \\ & & & x =_A x' \to y =_B y' \to \mathsf{rel}_t\ x\ y \to \mathsf{rel}_t\ x'\ y' \rangle. \end{array}$$

An alternative definition can be given using Ω as defined in the previous section.

A total relation from A to B is a relation R such that for every $a : \mathsf{el}_t\ A$, there exists $b : \mathsf{el}_t\ B$ satisfying $R \cdot \mathsf{rel}_t\ a\ b$.

Definition 5.6.4 *Let A and B be two total setoids. The type* $\textsc{Trel}_t\ A\ B$ *of total relations from A to B is defined as the record type*

$$\begin{array}{rlcl} \textsc{Trel}_t\ A\ B := \langle & \mathsf{trel}_t & : & \textsc{Rel}_t\ A\ B, \\ & \mathsf{total}_t & : & \forall x : \mathsf{el}_t\ A. \exists y : \mathsf{el}_t\ B.\ \mathsf{trel}_t \cdot \mathsf{rel}_t\ x\ y \rangle. \end{array}$$

The axiom of choice for total setoids is given by the context Γ_{AC}

$$\begin{array}{rcl} \mathrm{AC}_{make} & : & \Pi A, B : \textsc{Set}_t.\ (\textsc{trel}_t\ A\ B) \to (\textsc{map}_t\ A\ B), \\ \\ \mathrm{AC}_{check} & : & \forall A, B : \textsc{Set}_t. \forall R : \textsc{trel}_t\ A\ B. \forall x : \mathsf{el}_t\ A. \\ & & \quad R \cdot \mathsf{trel}_t \cdot \mathsf{rel}_t\ x\ (\mathsf{ap}_t\ (\mathrm{AC}_{make}\ A\ B\ R)\ x). \end{array}$$

The following result is well-known. For example its first statement is derivable from the fact that the axiom of choice for setoids holds in the proof-irrelevance model, see also for example, [Hof95a]; its second statement is derivable from Proposition 5.6.9 and the non-provability of classical logic in the context Γ_{ACT}.

Lemma 5.6.5 *The context Γ_{AC} is consistent but not instantiable in the context Γ_{ACT}.*

Proof We only prove the second fact by constructing a total relation on a setoid that respects equality, but from which we cannot extract a setoid function, even in the context Γ_{ACT}.

Let us consider the setoid of infinite bounded sequences of natural numbers with extensional equality. A sequence is bounded if all its elements are smaller than some natural number:

$$\begin{array}{rcl} \textsc{BSeq} & = & \langle \mathsf{seq} : \mathsf{N} \to \mathsf{N},\ \mathsf{bound} : \mathsf{N},\ \mathsf{bp} : \forall i : \mathsf{N}.\ \mathsf{seq}\ i \leq \mathsf{bound} \rangle, \\ \\ \textsc{BS} & = & \langle \mathsf{el}_t = \textsc{BSeq}, \\ & & \mathsf{eq}_t = \lambda x, y : \textsc{BSeq}.\ \forall i : \mathsf{N}.\ x \cdot \mathsf{seq}\ i \doteq_{\mathsf{N}} y \cdot \mathsf{seq}\ i, \\ & & \mathsf{er}_t = \ldots \rangle. \end{array}$$

where $\doteq_{\mathsf{N}}$ is Leibniz equality on natural numbers.

The relation "the sequence s is bounded by the number m" yields a total relation between bounded sequences and natural numbers. Formally, we define:

$$\begin{array}{rcl} \textsc{RBound} & : & \textsc{BSeq} \to \mathsf{N} \to \mathsf{Prop} \\ & = & \lambda s : \textsc{BSeq}. \lambda m : \mathsf{N}. \forall i : \mathsf{N}.\ (s \cdot \mathsf{seq})\ i \leq m. \end{array}$$

The relation RBound is compatible with respect to the extensional equality and total. Therefore we obtain an element

$$\mathrm{RB} : \textsc{TRel}_t\ \mathrm{BS}\ \mathbb{N}_t.$$

where $\mathbb{N}_t = \ddagger \mathsf{N} = \langle \mathsf{el}_t = \mathsf{N},\ \mathsf{eq}_t = \lambda x, y : \mathsf{N}.\ x \doteq y,\ \mathsf{er} = \ldots \rangle$ is the setoid of natural numbers with Leibniz equality.

Now we show that, in context Γ_{ACT}, we cannot construct a function $f : \textsc{map}_t\ \mathrm{BS}\ \mathbb{N}_t$ realizing RB. In fact, suppose that such a function exists:

$$\forall s : \mathsf{el}_t\ \mathrm{BS}. \textsc{RBound}\ s\ (\mathsf{ap}_t\ f\ s).$$

Let $\bar{0} : \mathrm{BS} \cdot \mathsf{el}_t$ be the constantly 0 sequence and set $m = \mathsf{ap}_t\ f\ \bar{0}$. Given any sequence $\alpha : \mathsf{N} \to \mathsf{N}$, we define the bounded sequence σ_α as

$$\sigma_\alpha \cdot \mathsf{seq}\ i = \begin{cases} 0 & \text{if } \alpha\ i = 0 \\ m + 1 & \text{otherwise.} \end{cases}$$

A formal definition in type theory can easily be obtained by cases on the decidability of equality for natural numbers. Now, α is constantly 0 if and only if $\sigma_\alpha =_{\mathrm{BS}} \bar{0}$. In that case, we have $\mathsf{ap}_t\ f\ \sigma_\alpha = m$. On the other hand, if α is not constantly 0, there will be an index i such that $\alpha\ i \neq 0$ and therefore $\sigma_\alpha \cdot \mathsf{seq}\ i = m + 1$. It follows that m cannot be a bound for σ_α and, therefore, $\mathsf{ap}_t\ f\ \sigma_\alpha \neq m$. Hence we have that

$$\alpha \text{ is constantly } 0 \quad \Leftrightarrow \quad \mathsf{ap}_t\ f\ \sigma_\alpha = m.$$

We obtain a method to decide whether an arbitrary sequence of natural numbers is constantly zero. Now, type theory with the axiom of choice for types is valid in a realisability model in which function types are interpreted as sets of total recursive functions. In such model there is no decision procedure to test whether a sequence is constantly zero, since this is equivalent to the halting problem—in fact, from the decidability of the halting problem we can derive the principle of excluded middle. We must therefore conclude that the function f cannot be constructed in type theory and, therefore, that the axiom of choice for setoids cannot be derived from the axiom of choice for types. □

An equivalent formulation of the axiom of choice makes use of the ϵ-monad defined in subsection 5.3. In this form the axiom states that we can find an element of every non-empty predicate. Formally the context $\Gamma_{\epsilon\mathrm{AC}}$ consists of the terms

$$\begin{array}{lcl} \epsilon\mathrm{AC}_{make} & : & \Pi A : \mathsf{Set}.\ \mathrm{MAP}_t\ (\epsilon\ A)\ A \\ \epsilon\mathrm{AC}_{check} & : & \Pi A : \mathsf{Set}.\ \Pi P : \mathsf{el}_t\ (\epsilon\ A).\mathsf{ap}_t\ P \cdot \mathsf{np}\ (\mathsf{ap}_t\ (\epsilon\mathrm{AC}_{make}\ A)\ P). \end{array}$$

Theorem 5.6.6 *The contexts* Γ_{AC} *and* $\Gamma_{\epsilon\mathrm{AC}}$ *are equivalent.*

Proof First we prove that in the context Γ_{AC} we can instantiate $\epsilon\mathrm{AC}_{make}$ and $\epsilon\mathrm{AC}_{check}$. Given $A : \mathsf{Set}$ we use the axiom of choice to find an element of $\mathrm{MAP}_t\ (\epsilon\ A)\ A$. We first construct a total functional relation $R_\epsilon : \mathrm{TREL}_t\ (\epsilon A)\ A$. We put

$$\begin{array}{rcl} R_\epsilon \cdot \mathsf{trel}_t \cdot \mathsf{rel}_t & : & \mathsf{el}_t\ (\epsilon\ A) \to \mathsf{el}_t\ A \to \mathsf{Prop} \\ & := & \lambda P.\lambda x.\mathsf{ap}_t\ P \cdot \mathsf{np}\ x. \end{array}$$

The proof components

$$\begin{array}{rcl} R_\epsilon \cdot \mathsf{trel}_t \cdot \mathsf{compat}_t & : & \forall P, P' : \mathsf{el}_t\ (\epsilon\ A).\forall x, x' : \mathsf{el}_t\ A. \\ & & P =_{\epsilon\ A} P' \to x =_A x' \to \\ & & \mathsf{ap}_t\ P \cdot \mathsf{np}\ x \to \mathsf{ap}_t\ P' \cdot \mathsf{np}\ x', \\ R_\epsilon \cdot \mathsf{total}_t & : & \forall P : \mathsf{el}_t\ (\epsilon\ A).\exists x : \mathsf{el}_t\ A.\mathsf{ap}_t\ P \cdot \mathsf{np}\ x. \end{array}$$

follow directly from the definition of $\epsilon\ A$. We can now instantiate $\Gamma_{\epsilon\mathrm{AC}}$:

$$\begin{array}{rcl} \epsilon\mathrm{AC}_{make}\ A & := & \mathrm{AC}_{make}\ (\epsilon A)\ A\ R_\epsilon, \\ \epsilon\mathrm{AC}_{check}\ A & := & \mathrm{AC}_{check}\ (\epsilon A)\ A\ R_\epsilon. \end{array}$$

In the other direction we show that the context Γ_{AC} can be instantiated in the context $\Gamma_{\epsilon\mathrm{AC}}$. Assume $\epsilon\mathrm{AC}_{make}$ and $\epsilon\mathrm{AC}_{check}$. We must define

$$\mathrm{AC}_{make} : \Pi A, B : \mathsf{Set}.(\textsc{trel}_t\ A\ B) \to (\textsc{map}_t\ A\ B).$$

Assume $A, B : \mathsf{Set}$ and $R : \textsc{trel}_t\ A\ B$. We begin by constructing a map between ϵ-setoids, $F : \textsc{map}_t\ (\epsilon\ A)\ (\epsilon\ B)$. Let $P\colon \underline{\epsilon}\ A$. We define $(\mathsf{ap}_t\ F\ P)\cdot\mathsf{np} : \textsc{map}_t\ B\ \Omega$ by

$$\begin{array}{rcl}(\mathsf{ap}_t\ F\ P)\cdot\mathsf{np}\cdot\mathsf{ap}_t & : & \mathsf{el}_t \to \mathsf{Prop}\\ (\mathsf{ap}_t\ F\ P)\cdot\mathsf{np}\cdot\mathsf{ap}_t & := & \lambda y : \mathsf{el}_t\ B.\exists x : \mathsf{el}_t\ A.(R\cdot\mathsf{trel}_t\cdot\mathsf{rel}_t\ x\ y)\wedge(P\cdot\mathsf{np}\ x).\end{array}$$

where

$$\begin{array}{rcl}(\mathsf{ap}_t\ F\ P)\cdot\mathsf{np}\cdot\mathsf{ext}_t & : & \forall y_1, y_2 : \mathsf{el}_t\ B.y_1 =_B y_2 \to\\ & & \mathsf{ap}_t\ (\mathsf{ap}_t\ F\ P)\ y_1 =_\Omega \mathsf{ap}_t\ (\mathsf{ap}_t\ F\ P)\ y_2\\ & := & \forall y_1, y_2 : \mathsf{el}_t\ B.y_1 =_B y_2 \to\\ & & \exists x : \mathsf{el}_t\ A.(R\cdot\mathsf{trel}_t\cdot\mathsf{rel}_t\ x\ y_1)\wedge(P\cdot\mathsf{np}\ x)\leftrightarrow\\ & & \exists x : \mathsf{el}_t\ A.(R\cdot\mathsf{trel}_t\cdot\mathsf{rel}_t\ x\ y_2)\wedge(P\cdot\mathsf{np}\ x).\end{array}$$

follows from

$$\begin{array}{rcl}R\cdot\mathsf{trel}_t\cdot\mathsf{compat}_t & : & \forall x_1, x_2 : \mathsf{el}_t\ A.\forall y_1, y_2 : \mathsf{el}_t\ B.\\ & & x_1 =_A x_2 \to y_1 =_B y_2 \to\\ & & R\cdot\mathsf{trel}_t\cdot\mathsf{rel}_t\ x_1\ y_1 \to R\cdot\mathsf{trel}_t\cdot\mathsf{rel}_t\ x_2\ y_2.\end{array}$$

and the non-emptiness of the image predicate

$$\begin{array}{rcl}(\mathsf{ap}_t\ F\ P)\cdot\mathsf{ne} & : & \exists y : \mathsf{el}_t\ B.\mathsf{ap}_t\ (\mathsf{ap}_t\ F\ P)\cdot\mathsf{np}\ y\\ & := & \exists y : \mathsf{el}_t\ B.\exists x : \mathsf{el}_t\ A.(R\cdot\mathsf{trel}_t\cdot\mathsf{rel}_t\ x\ y)\wedge(P\cdot\mathsf{np}\ x).\end{array}$$

follows from

$$\begin{array}{rcl}P\cdot\mathsf{ne} & : & \exists x : \mathsf{el}_t\ A.\mathsf{ap}_t\ P\cdot\mathsf{np}\ x \quad \text{and}\\ R\cdot\mathsf{total}_t & : & \forall x : \mathsf{el}_t\ A.\exists y : \mathsf{el}_t\ B.R\cdot\mathsf{trel}_t\cdot\mathsf{rel}_t\ x\ y.\end{array}$$

Finally

$$F\cdot\mathsf{ext}_t : \forall P_1, P_2 : \underline{\epsilon}\ A.P_1 =_{\epsilon\ A} P_2 \to \mathsf{ap}_t\ F\ P_1 =_{\epsilon\ B} \mathsf{ap}_t\ F\ P_1.$$

can be easily checked, because we used only the extension of P in the definition of $\mathsf{ap}_t\ F\ P$.

We can define an injection map from a setoid to the corresponding ϵ-setoid:

$$\begin{array}{rcl}\eta_\epsilon & : & \Pi A : \mathsf{Set}.\textsc{map}_t\ A\ (\epsilon\ A)\\ (\eta_\epsilon\ A)\cdot\mathsf{ap}_t & : & \mathsf{el}_t\ A \to \underline{\epsilon}\ A\\ (\eta_\epsilon\ A)\cdot\mathsf{ap}_t & := & \lambda x : \mathsf{el}_t\ A.\langle\ \mathsf{np} = \langle\ \mathsf{ap}_t = \lambda x' : \mathsf{el}_t\ A.x' =_A x,\\ & & \qquad\qquad \mathsf{ext}_t = \cdots\rangle,\\ & & \qquad \mathsf{ne} = \cdots\rangle.\end{array}$$

We have that $\epsilon\mathrm{AC}_{make}\ B : \mathrm{MAP}_t\ (\epsilon\ B)\ B$. So we can compose F with these two maps to obtain

$$\begin{array}{lcl}(\mathrm{AC}_{make}\ A\ B\ R) & : & \mathrm{MAP}_t\ A\ B \\ (\mathrm{AC}_{make}\ A\ B\ R) & := & (\epsilon\mathrm{AC}_{make}\ B) \circ F \circ (\eta_\epsilon\ A).\end{array}$$

We still have to prove

$$\begin{array}{rcl}\mathrm{AC}_{check}\ A\ B\ R & : & \forall x : \mathsf{el}_t\ A.R \cdot \mathsf{trel}_t \cdot \mathsf{rel}_t\ x\ (\mathsf{ap}_t\ (\mathrm{AC}_{make}\ A\ B\ R)\ x) \\ & : & \forall x : \mathsf{el}_t\ A.R \cdot \mathsf{trel}_t \cdot \mathsf{rel}_t\ x(\mathsf{ap}_t\ (\epsilon\mathrm{AC}_{make}\ B) \\ & & \qquad\qquad (\mathsf{ap}_t\ F\ (\mathsf{ap}_t\ (\eta_\epsilon\ A)\ x))).\end{array}$$

Given $x : \mathsf{el}_t\ A$, to simplify the terms, we write P_x for $\mathsf{ap}_t\ (\eta_\epsilon\ A)\ x$ and y_x for $\mathsf{ap}_t\ (\epsilon\mathrm{AC}_{make}\ B)\ (\mathsf{ap}_t\ F\ (\mathsf{ap}_t\ (\eta_\epsilon\ A)\ x))$. We have to prove

$$\mathrm{AC}_{check}\ A\ B\ R\ x : R \cdot \mathsf{trel}_t \cdot \mathsf{rel}_t\ x\ y_x.$$

For this purpose we use the axiom $\epsilon\mathrm{AC}_{check}$:

$$\epsilon\mathrm{AC}_{check}\ B\ (\mathsf{ap}_t\ F\ P_x) : \exists x' : \mathsf{el}_t\ A.(R \cdot \mathsf{trel}_t \cdot \mathsf{rel}_t\ x'\ y_x) \wedge (P_x \cdot \mathsf{np}\ x').$$

By the definition of P_x, we have that $(P_x \cdot \mathsf{np}\ x')$ reduces to $x' =_A x$. Thus, by $R \cdot \mathsf{trel}_t \cdot \mathsf{compat}_t$, we conclude that $R \cdot \mathsf{trel}_t \cdot \mathsf{rel}_t\ x'\ y_x$ implies $R \cdot \mathsf{trel}_t \cdot \mathsf{rel}_t\ x\ y_x$. This completes the proof. □

We conclude this subsection by summarising a few well-known properties of the axiom of choice for setoids. These properties are related to categorical models of intuitionistic set theory known as toposes, see for example, [LS86]. Informally, a topos $\mathcal{T}$ is a cartesian closed category with a subobject classifier, that is, with an object that acts as a set of truth values. The first item of the proposition below states that **TSet** is a topos with Ω as subobject classifier and is proved by a simple calculation. The second item states the provability of classical logic from the axiom of choice for setoids and follows from Diaconescu's construction, see for example, [LS86] and also [LW99]. The third item establishes that proof-irrelevance, that is, the property that all proofs of a proposition are equal (the property was first considered by de Bruijn in the AUTOMATH project, see for example, [NGV94]), is derivable from the axiom of choice for setoids and can be established from [BB96]. It should also be pointed that these results were probably discovered independently by authors not cited above, for example, the results are also implicit in [Hof95a].

Recall that a topos $\mathcal{T}$ is a cartesian closed category with a subobject classifier, which can be described informally as the categorical equivalent of the set of truth values. The notion of topos is important because toposes are models of intuitionistic set theory [LS86].

We use the following two lemmas, that do not depend on the axiom of choice.

Lemma 5.6.7 TSet *has equalizers.*

Proof Given two morphisms $h_1, h_2 \colon \textsc{map}_t\ A\ B$ in **TSet**, define the C as the setoid having as carrier the type of elements $a \colon \mathsf{el}_t\ A$ such that $\mathsf{ap}_t\ h_1\ a =_B \mathsf{ap}_t\ h_2\ a$, and the book equality inherited from A. The injection function $i \colon \textsc{map}_t\ C\ A$ is the obvious one. It is easy to show that C and i are an equalizer of h_1 and h_2. □

Lemma 5.6.8 *The monomorphisms of* **TSet** *are characterised by*

$$\left(\begin{array}{l} m : \textsc{map}_t\ B\ A \\ \textit{is a monomorphism} \end{array}\right) \Longleftrightarrow \left(\begin{array}{l} \forall b, b' : \mathsf{el}_t\ B. \\ \mathsf{ap}_t\ m\ b =_A \mathsf{ap}_t\ m\ b' \to b =_B b' \end{array}\right),$$

that is, the monomorphisms are the injective setoid functions.

Proposition 5.6.9 *In the context* Γ_{AC}*, one can prove the following:*

- **TSet** *is a topos with* Ω *as subobject classifier;*
- *excluded middle:* $\forall A : \mathsf{Prop}.\ A \vee \neg A$*;*
- *proof-irrelevance:* $\forall A : \mathsf{Prop}. \forall x, y : A.\ x \doteq y$*.*

Proof We only prove the first item. Define $\top_A : \textsc{map}_t\ A\ \Omega$ as $\langle \mathsf{ap}_t = \lambda x : \mathsf{el}_t\ A.\ \top, \mathsf{ext}_p = \ldots \rangle$, and $\hat{\top} : \textsc{map}_t\ \dagger\,\mathsf{Unit}\ \Omega$ as $\top_{\dagger\mathsf{Unit}}$. It is easy to show that for every $A : \textsc{Set}_t$ we have $\top_A =_{\textsc{Map}_t\ A\ \Omega} \hat{\top} \circ !_A$ where $\circ$ denotes composition of morphisms of setoids and $!_A$ denotes the unique morphism of setoids from A to $\dagger\mathsf{Unit}$.

Then one has to show that:

- for every arrow $h : \textsc{map}_t\ A\ \Omega$, there exists an equaliser $\textsc{ker}\ h$ of h and $\top_A$;
- for every monomorphism $m : \textsc{map}_t\ B\ A$ there exists a unique arrow $\textsc{char}\ m : \textsc{map}_t\ A\ \Omega$ such that m is a kernel of $\textsc{char}\ m$.

Equalisers exist by Lemma 5.6.7; specifically, set $\textsc{ker}\ h$ to be the injection function from those elements of A to satisfy $\mathsf{ap}_t\ h$—the setoid is defined formally but in a slightly different form as $\textsc{subSetoid}_t\ A\ h$ in the next section; the definition of the injection is trivial. Then every arrow i that equates h and $\top_A$ by right-composition factors uniquely through $\textsc{ker}\ h$.

Let us now turn to the existence of characteristic functions. Set $\textsc{char}\ m$ to be

$$\langle \mathsf{ap}_t = \lambda a : \mathsf{el}_t\ A.\ \exists b : \mathsf{el}_t\ B.\ \mathsf{ap}_t\ m\ b =_A a,\ \mathsf{ext}_p = \ldots \rangle.$$

We need to show that m is a kernel for $\textsc{char}\ m$. So let $i : \textsc{map}_t\ C\ A$ such that $\top_A \circ i =_{\textsc{Map}_t\ C\ \Omega} (\textsc{char}\ m) \circ i$. By simple logical manipulations, one concludes that

$$\forall c : \mathsf{el}_t\ C. \exists b : \mathsf{el}_t\ B.\ \mathsf{ap}_t\ m\ b =_A \mathsf{ap}_t\ i\ c. \qquad (\heartsuit)$$

By the axiom of choice for setoids, one obtains $j : \text{MAP}_t\ C\ B$ such that $i =_{\text{MAP}_t\ C\ A} m \circ j$. Uniqueness of j follows from the characterization of monomorphisms given in Lemma 5.6.8. □

Note that the second statement of Lemma 5.6.5 also follows from Proposition 5.6.9 since the axiom of excluded middle does not follow from the axiom of choice for types.

The axiom of descriptions

As seen above, the axiom of choice for setoids is extremely powerful. A weaker form of choice principle, acceptable intuitionistically, is the axiom of descriptions, which states that every total functional relation induces a map.

The axiom and its consistency

Recall that a total functional relation from A to B, where A and B are total setoids, consists of a relation R on A and B and proofs that R is total and functional.

Definition 5.6.10 *Let A and B be two total setoids. The type $\text{TFREL}_t\ A\ B$ of total functional relations from A to B is defined as the record type*

$$\begin{array}{rrcl} \text{TFREL}_t\ A\ B := \langle & \mathsf{tfrel}_t & : & \text{REL}_t\ A\ B, \\ & \mathsf{ttotal}_t & : & \forall x : \mathsf{el}_t\ A.\exists y : \mathsf{el}_t\ B.\ \mathsf{tfrel}_t \cdot \mathsf{rel}_t\ x\ y, \\ & \mathsf{fun}_t & : & \forall x : \mathsf{el}_t\ A.\forall y, y' : \mathsf{el}_t\ B. \\ & & & (\mathsf{tfrel}_t \cdot \mathsf{rel}_t\ x\ y) \to (\mathsf{tfrel}_t \cdot \mathsf{rel}_t\ x\ y') \to y =_B y' \rangle. \end{array}$$

The axiom of descriptions for total setoids is given by the context Γ_{AD}

$$\text{AD}_{make} : \Pi A, B : \text{SET}_t.\ (\text{TFREL}_t\ A\ B) \to (\text{MAP}_t\ A\ B),$$

$$\begin{array}{rcl} \text{AD}_{check} & : & \forall A, B : \text{SET}_t.\forall R : \text{TFREL}_t\ A\ B.\forall x : \mathsf{el}_t\ A. \\ & & R \cdot \mathsf{tfrel}_t \cdot \mathsf{rel}_t\ x\ (\mathsf{ap}_t\ (\text{AD}_{make}\ A\ B\ R)\ x). \end{array}$$

The context is consistent, since it follows from the axiom of choice for types.

Lemma 5.6.11 *The context Γ_{AD} is instantiable in the context Γ_{ACT}, and hence consistent.*

A corollary of this lemma is that Γ_{AD} is inhabited in Martin-Löf's type theory and in the Calculus of Inductive Constructions with identification of data types and propositions.

In the previous subsection, we have seen that in the context Γ_{AC}, one can prove that **TSet** is a topos. In fact, it is sufficient to assume Γ_{AD}; however, in the context Γ_{AD}, neither classical logic nor proof-irrelevance are provable, since they do not follow from Γ_{ACT}.

Proposition 5.6.12 *In the context Γ_{AD}, **TSet** is a topos.*

Proof The proof is similar to that of Proposition 5.6.9. Indeed, observe that in the situation of Proposition 5.6.9, one can conclude from ($\heartsuit$) and from the injectivity of m, or better said from the fact that m is a monomorphism, that

$$\forall c : \mathsf{el}_t\ C.\exists!\ b \in B.\ \mathsf{ap}_t\ m\ b =_A \mathsf{ap}_t\ i\ c.$$

Hence one can apply the axiom of descriptions for setoids instead of the axiom of choice for setoids and proceed as before. □

Equivalent principles

As with the axiom of choice, we can express the axiom of descriptions in terms of the ι-monad. We define the context $\Gamma_{\iota\mathrm{AD}}$ as

$$\begin{array}{rcl} \iota\mathrm{AD}_{make} & : & \Pi A : \mathsf{Set}.\ \textsc{map}_t\ (\iota\ A)\ A, \\ \iota\mathrm{AD}_{check} & : & \Pi A : \mathsf{Set}.\ \Pi P : \mathsf{el}_t\ (\iota\ A).\mathsf{ap}_t\ P \cdot \mathsf{up}\ (\mathsf{ap}_t\ (\iota\mathrm{AD}_{make}\ A)\ P). \end{array}$$

Theorem 5.6.13 *The contexts* Γ_{AD} *and* $\Gamma_{\iota\mathrm{AD}}$ *are equivalent.*

Proof Similar to the proof of Theorem 5.6.6. □

While we have not checked the details, we believe that $\iota\mathrm{AD}_{check}$ is also equivalent to stating that $\iota\mathrm{AD}_{make}$ is left-inverse (or right-inverse) to the unit η_ι of the ι-monad. Below we also present another axiom, which we coin the axiom of invertible bijections, that is equivalent to the axiom of descriptions.

One peculiarity of setoids is that bijections, that is, maps that are both injective and surjective, need not have an inverse. It is therefore natural to consider as a choice principle the axiom of invertible bijections, which states that every map that is bijective has an inverse.

Definition 5.6.14 *Let A and B be two total setoids. The type* $\textsc{bij}_t\ A\ B$ *of bijections from A to B is defined as the record type*

$$\begin{array}{rcl} \textsc{bij}_t\ A\ B := \langle\ \mathsf{mapb}_t & : & \textsc{map}_t\ A\ B, \\ \mathsf{surj}_t & : & \forall y : \mathsf{el}_t\ B.\exists x : \mathsf{el}_t\ A.\ (\mathsf{ap}_t\ \mathsf{mapb}_t\ x) =_B y, \\ \mathsf{inj}_t & : & \forall x, x' : \mathsf{el}_t\ A. \\ & & (\mathsf{ap}_t\ \mathsf{mapb}_t\ x) =_B (\mathsf{ap}_t\ \mathsf{mapb}_t\ x') \to x =_A x' \rangle. \end{array}$$

The axiom of invertible bijections for total setoids is given by the context Γ_{AIB}

$$\begin{array}{rcl} \mathrm{AIB}_{make} & : & \Pi A, B : \textsc{Set}_t.\ (\textsc{bij}_t\ A\ B) \to (\textsc{map}_t\ B\ A), \\ \mathrm{AIB}_{checkleft} & : & \forall A, B : \textsc{Set}_t.\forall f : \textsc{bij}_t\ A\ B.\forall x : \mathsf{el}_t\ A. \\ & & (\mathsf{ap}_t\ (\mathrm{AIB}_{make}\ A\ B\ f)\ (\mathsf{ap}_t\ f \cdot \mathsf{mapb}_t\ x)) =_A x, \\ \mathrm{AIB}_{checkright} & : & \forall A, B : \textsc{Set}_t.\forall f : \textsc{bij}_t\ A\ B.\forall y : \mathsf{el}_t\ B. \\ & & (\mathsf{ap}_t\ f \cdot \mathsf{mapb}_t\ (\mathsf{ap}_t\ (\mathrm{AIB}_{make}\ A\ B\ f)\ y)) =_B y. \end{array}$$

The context is consistent, as shown by the following result.

Proposition 5.6.15 *The contexts* Γ_{AD} *and* Γ_{AIB} *are equivalent.*

Proof [Sketch] There are two implications to treat:

- The axiom of descriptions implies the axiom of invertible bijections. Assume $f : \textsc{bij}_t\ A\ B$. Then the inverse graph of f yields a total functional relation from B to A. More precisely, we have $\textsc{invgraph} : \textsc{tfrel}_t\ B\ A$ where

$$\begin{array}{rcl} \textsc{invgraphrel} := \langle \quad \mathsf{rel}_t & = & \lambda y : \mathsf{el}_t\ B.\lambda x : \mathsf{el}_t\ A. \\ & & \quad (\mathsf{ap}_t\ f \cdot \mathsf{mapb}_t\ x) =_B y, \\ \mathsf{compat}_t & = & \ldots\ \rangle, \end{array}$$

$$\begin{array}{rcl} \textsc{invgraph} = \langle \quad \mathsf{tfrel}_t & = & \textsc{invgraphrel}, \\ \mathsf{ttotal}_t & = & f \cdot \mathsf{surj}_t, \\ \mathsf{fun}_t & = & \ldots\rangle. \end{array}$$

  By the axiom of descriptions, we obtain a map from B to A, namely $\mathrm{AD}_{make}\ \textsc{invgraph} : \textsc{map}_t\ B\ A$. It is then routine to check that the function $\mathrm{AD}_{make}\ \textsc{invgraph}$ is (right and left) inverse to f.

- The axiom of invertible bijections implies the axiom of descriptions. Assume $r : \textsc{tfrel}_t\ A\ B$. We build up the "graph" type $\textsc{graph}$ of r as triples (a, b, ϕ) where $a : \mathsf{el}_t\ A$, $b : \mathsf{el}_t\ B$ and ϕ is a proof that r relates a and b. $\textsc{graph}$ is turned into a setoid by setting (a, b, ϕ) equal to (a', b', ϕ') iff $a =_A a'$. Formally, we let

$$\begin{array}{rcl} \textsc{graph} := \langle \ \mathsf{arg} & : & \mathsf{el}_t\ A, \\ \mathsf{val} & : & \mathsf{el}_t\ B, \\ \mathsf{fit} & : & (r \cdot \mathsf{tfrel}_t \cdot \mathsf{rel}_t\ \mathsf{arg}\ \mathsf{val})\rangle \end{array}$$

  and

$$\begin{array}{rcl} \textsc{Graph} := \langle \ \mathsf{el}_t & = & \textsc{graph}, \\ \mathsf{eq}_t & = & \lambda x, y : \textsc{graph}.\ x \cdot \mathsf{arg} =_A y \cdot \mathsf{arg}, \\ \mathsf{er} & = & \ldots\rangle. \end{array}$$

  We have $\textsc{Graph} : \textsc{Set}_t$. The field selection function arg yields a bijection $\Pi : \textsc{bij}_t\ \textsc{Graph}\ A$. Formally, we have $\pi : \mathsf{map}_t\ \textsc{Graph}\ A$ with

$$\begin{array}{rcl} \pi := \langle \ \mathsf{ap}_t & = & \lambda g : \textsc{graph}.\ g \cdot \mathsf{arg}, \\ \mathsf{ext}_t & = & \lambda x, y : \textsc{graph}.\lambda h : (x =_{\textsc{Graph}} y).\ h\rangle \end{array}$$

  and $\Pi : \textsc{bij}_t\ \textsc{Graph}\ A$ with

$$\begin{array}{rcl} \Pi := \langle \ \mathsf{mapb}_t & = & \pi, \\ \mathsf{surj}_t & = & \ldots, \\ \mathsf{inj}_t & = & \ldots\rangle. \end{array}$$

  The axiom of invertible bijections yields a map $f = (\mathrm{AIB}_{make}\ \Pi) : \textsc{map}_t\ A\ \textsc{Graph}$. Moreover, f may be composed with the field selection

function val to yield a map $g : \textsc{map}_t\ A\ B$. Formally, we have:

$$g := \langle \begin{array}{rcl} \mathsf{ap}_t & = & \lambda x : \mathsf{el}_t\ A.\ (\mathsf{ap}_t\ f\ x) \cdot \mathsf{val}, \\ \mathsf{ext}_t & = & \ldots \rangle. \end{array}$$

It is then routine to check that for every $x : \mathsf{el}_t\ A$, one has

$$r \cdot \mathsf{trel}_t \cdot \mathsf{rel}_t\ x\ (\mathsf{ap}_t\ g\ x).$$

□

A different and shorter proof of the proposition can be obtained by proving that Γ_{AIB} is equivalent to $\Gamma_{\iota\mathrm{AD}}$ and concluding by Theorem 5.6.13.

Choice principles for partial setoids

For completeness, we briefly discuss the axiom of descriptions for partial setoids. Formulating the axioms requires to decide about the notion of total relation. A total relation from a partial setoid A to a partial setoid B is a relation R such that:

- (definedness-irrelevant) for every $a : \mathsf{el}_p\ A$ there exists $b : \mathsf{el}_p\ B$ such that $R\ a\ b$;
- or: (definedness-relevant) for every $a : \mathsf{el}_p\ A$ such that a is defined, that is, $a =_A a$, there exists $b : \mathsf{el}_p\ B$ such that $R\ a\ b$ and $b =_B b$.

The purpose of this subsection is to show that the axiom of descriptions based on the latter notion of total relation is inconsistent, and that the axiom of descriptions based on the former notion of total relation is too weak. More precisely, we show that some very natural functions that can be defined on a total setoid with the axiom of descriptions cannot be defined in the corresponding partial setoid, even in presence of the (definedness-irrelevant) axiom of descriptions. This observation leads us to the position that total setoids are better suited for the development of mathematics in type theory than partial setoids. Notice that in the following we will not use exponents for partial setoids, so our results hold both for **PSet** and **RSet**.

Axiom of descriptions, definedness-relevant version

Definition 5.6.16 *Let A and B be two partial setoids.*

1. *The type $\textsc{rel}_p\ A\ B$ of relations from A to B is defined as the record type*

$$\begin{array}{rcl} \textsc{rel}_p\ A\ B := \langle \quad \mathsf{rel}_p & : & \mathsf{el}_p\ A \to \mathsf{el}_p\ B \to \mathsf{Prop}, \\ \mathsf{compat}_p & : & \forall x, x' : \mathsf{el}_p\ A. \forall y, y' : \mathsf{el}_p\ B. \\ & & x =_A x' \to y =_B y' \to \mathsf{rel}_p\ x\ y \to \mathsf{rel}_p\ x'\ y' \rangle. \end{array}$$

2. *The type* $\textsc{tfrel}_p\ A\ B$ *of total functional relations from* A *to* B *is defined as the record type*

$$\begin{array}{rcl}\textsc{tfrel}_p\ A\ B = \langle\ \mathsf{tfrel}_p & : & \textsc{Rel}_p\ A\ B, \\ \mathsf{ttotal}_p & : & \forall x : \mathsf{el}_p\ A.(x =_A x) \to \\ & & \exists y : \mathsf{el}_p\ B.\ (\mathsf{tfrel}_p \cdot \mathsf{rel}_p\ x\ y \wedge y =_B y), \\ \mathsf{fun}_p & : & \forall x : \mathsf{el}_p\ A.\forall y, y' : \mathsf{el}_p\ B. \\ & & (\mathsf{tfrel}_p \cdot \mathsf{rel}_p\ x\ y) \to (\mathsf{tfrel}_p \cdot \mathsf{rel}_p\ x\ y') \to \\ & & x =_A x \to y =_B y' \rangle.\end{array}$$

The axiom of descriptions for partial setoids is given by the context Γ_{AD}

$$\begin{array}{rcl}\mathrm{AD}_{make} & : & \Pi A, B : \textsc{Set}_p.\ (\textsc{tfrel}_p\ A\ B) \to (\textsc{map}_p\ A\ B),\end{array}$$

$$\begin{array}{rcl}\mathrm{AD}_{check} & : & \forall A, B : \textsc{Set}_p.\forall R : \textsc{tfrel}_p\ A\ B.\forall x : \mathsf{el}_p\ A. \\ & & R \cdot \mathsf{tfrel}_p \cdot \mathsf{rel}_p\ x\ (\mathsf{ap}_p\ (\mathrm{AD}_{make}\ A\ B\ R)\ x).\end{array}$$

The context is inconsistent, as shown below.

Lemma 5.6.17 *The context* Γ_{AD} *is inconsistent.*

Proof For every type A and partial setoid B, one can prove that the empty relation $r : \textsc{rel}_p\ (\emptyset\ A)\ B$ yields a total functional relation $r' : \textsc{tfrel}_p\ (\emptyset\ A)\ B$ and by the axiom of descriptions, a map $f : \textsc{map}_p\ (\emptyset\ A)\ B$. Now take A to be inhabited. It follows that every $\mathsf{el}_p\ B$ is inhabited, so every type B is inhabited. □

Axiom of descriptions, definedness-irrelevant version

In this paragraph we adopt the definedness-irrelevant definition of total functional relations.

Definition 5.6.18 *Let* A *and* B *be two partial setoids. The type* $\textsc{tfrel}_p\ A\ B$ *of total functional relations from* A *to* B *is defined as the record type*

$$\begin{array}{rcl}\textsc{tfrel}_p\ A\ B := \langle\ \mathsf{tfrel}_p & : & \textsc{Rel}_p\ A\ B, \\ \mathsf{ttotal}_p & : & \forall x : \mathsf{el}_p\ A.\exists y : \mathsf{el}_p\ B.\ \mathsf{tfrel}_p \cdot \mathsf{rel}_p\ x\ y, \\ \mathsf{fun}_p & : & \forall x : \mathsf{el}_p\ A.\forall y, y' : \mathsf{el}_p\ B. \\ & & (\mathsf{tfrel}_p \cdot \mathsf{rel}_p\ x\ y) \to (\mathsf{tfrel}_p \cdot \mathsf{rel}_p\ x\ y') \to \\ & & x =_A x \to y =_B y' \rangle.\end{array}$$

Then the context Γ_{AD} is defined exactly as in the previous paragraph.

Lemma 5.6.19 *The context* Γ_{AD} *is instantiable in the context* Γ_{ACT}*, and hence consistent. Further, in context* Γ_{AD}*,* **RSet** *does form a topos.*

However, we show that some very natural functions that can be defined on a total setoid with the axiom of descriptions cannot be defined in the corresponding partial setoid with the axiom of descriptions. We will construct a counterexample, that is, a function that is definable on a total setoid but not on the corresponding partial setoid. We will use the following principle in showing that the function is not definable on the partial setoid.

Theorem 5.6.20 (Continuity Principle) *In type theory with or without the axiom of choice for types. For every operator* $F : (\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$ *and for every sequence* $\alpha : \mathsf{N} \to \mathsf{N}$*, there exists a natural number* m *such that, for every other sequence* $\beta : \mathsf{N} \to \mathsf{N}$ *that is equal to* α *for indexes up to* m*, that is,* $(\beta\ i) = (\alpha\ i)$ *for* $i \leq m$*, we have that* $F(\beta) = F(\alpha)$*.*

Proof This is a well-known principle in intuitionistic mathematics. In constructive recursion theory, it follows from the Kreisel-Lacombe-Shoenfield theorem (see [TvD88] and Chapter 16 of [Bee85]).

Notice that the Continuity Principle is not provable in type theory, but it is a meta-result that holds for the operators definable in type theory. □

Our counterexample is a function on the setoid of eventually null sequences of natural numbers. We use the extensional equality on sequences: If $s_1, s_2 : \mathsf{N} \to \mathsf{N}$, then $(s_1 =_{ext} s_2) = (\forall i : \mathsf{N}.s_1\ i = s_2\ i)$. The total version of the setoid is

$$\begin{array}{rcl} \text{ZSEQ}_t := \langle\ \mathsf{el}_t & = & \langle \mathsf{seq} : \mathsf{N} \to \mathsf{N}, \mathsf{evz} : \exists m : \mathsf{N}.\forall i : \mathsf{N}.\ i > m \to \mathsf{seq}\ i = 0\rangle, \\ \mathsf{eq}_t & = & \lambda s_1, s_2 : \mathsf{el}_t.s_1 \cdot \mathsf{seq} =_{ext} s_2 \cdot \mathsf{seq}, \\ \mathsf{er}_t & = & \cdots\rangle \end{array}$$

while its partial version is

$$\begin{array}{rcl} \text{ZSEQ}_p := \langle\ \mathsf{el}_p & = & \mathsf{N} \to \mathsf{N}, \\ \mathsf{eq}_p & = & \lambda\sigma_1, \sigma_2 : \mathsf{N} \to \mathsf{N}.\exists m : \mathsf{N}.\forall i : \mathsf{N}. \\ & & (i \leq m \to (\sigma_1\ i) = (\sigma_2\ i)) \\ & & \wedge\ (i > m \to (\sigma_1\ i) = 0 \wedge (\sigma_2\ i) = 0), \\ \mathsf{per} & = & \cdots\rangle. \end{array}$$

Of course we could define ZSEQ_p from ZSEQ_t by simply forgetting the proof of reflexivity, but what we want to stress here is that the idea of using the book equality to restrict the domain, which is the main advantage of partial setoids, does not always work as desired.

Now, consider the function that gives the length of the part of a sequence that is nonzero:

$$\text{LENGTH}_t : \text{MAP}_t\ \text{ZSEQ}_t\ \mathbb{N}_t.$$

The function can be defined in the context Γ_{AD}.

In contrast, we claim that there cannot be a version of this function for ZSEQ_p. Let $\mathbb{N}_p = \langle \mathsf{el}_p = \mathsf{N},\ \mathsf{eq}_p = \lambda x, y : \mathsf{N}.\ x \doteq y,\ \mathsf{per} = \ldots\rangle$ be the partial setoid of natural numbers. If there were a function $\text{LENGTH}_p : \text{MAP}_p\ \text{ZSEQ}_p\ \mathbb{N}_p$, then $\text{LENGTH}_p \cdot \mathsf{ap}_p : (\mathsf{N} \to \mathsf{N}) \to \mathsf{N}$ would be an extension of LENGTH to all sequences. We can apply the Continuity Principle to the operator $\text{LENGTH}_p \cdot \mathsf{ap}_p$ and the constantly zero sequence $\bar{0}$. Hence, there must exist a natural number m such that, for all $\beta : \mathsf{N} \to \mathsf{N}$ satisfying $(\forall i : \mathsf{N}.\ (i \leq m) \to \beta\ i = \bar{0}\ i = 0)$, $(\mathsf{ap}_p\ \text{LENGTH}_p\ \beta = \mathsf{ap}_p\ \text{LENGTH}_p\ \bar{0} = 0)$. Now consider the sequence γ defined

by

$$\gamma\ i := \begin{cases} 0 & \text{if } i \leq m \\ 1 & \text{if } i = m+1 \\ 0 & \text{if } i > m+1. \end{cases}$$

The sequence γ coincides with $\bar{0}$ on the first m elements, and thus, for the consequence of the continuity principle $\mathsf{ap}_p\ \textsc{length}_p\ \gamma = 0$. On the other hand, γ becomes eventually zero only after the element $m+1$, so we should have $\mathsf{ap}_p\ \textsc{length}_p\ \gamma = m+1$. We reached a contradiction, so our assumption that $\textsc{length}_p$ could be constructed is confuted.

Notice that the crux of the counterexample is that, in the total setoid $\textsc{ZSeq}_t$, the carrier type already contains information on when the sequence becomes eventually zero. We can use this information to define $\textsc{length}_t$. This information is not present in the domain of the partial setoid $\textsc{ZSeq}_p$, making it impossible to define the function $\textsc{length}_p$.

We can summarise our counterexample by stating that, in the context of the continuity principle, the function $\textsc{length}_p$ is provably undefinable. Let us formalise the principle as the type-theoretic context Γ_{CP}

$$\begin{array}{lcl} \mathrm{CP}_{make} & : & ((\mathsf{N} \to \mathsf{N}) \to \mathsf{N}) \to (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \\ \\ \mathrm{CP}_{check} & : & \forall F : (\mathsf{N} \to \mathsf{N}) \to \mathsf{N}. \forall \alpha, \beta : \mathsf{N} \to \mathsf{N}. \\ & & (\forall i : \mathsf{N}. (i \leq \mathrm{CP}_{make}\ F\ \alpha) \to \beta\ i = \alpha\ i) \to F\ \beta = F\ \alpha. \end{array}$$

Theorem 5.6.21 *In the context Γ_{CP} there does not exists a length function on the setoid of eventually zero sequences.*

$$\begin{array}{ll} \Gamma_{\mathrm{CP}} \vdash & \neg\exists \textsc{length}_p : (\textsc{map}_p\ \textsc{ZSeq}_p\ \mathbb{N}_p). \forall \sigma : (\textsc{ZSeq}_p \cdot \mathsf{el}_p). \sigma =_{\textsc{ZSeq}_p} \sigma \to \\ & \sigma\ (\textsc{length}_p \cdot \mathsf{app}_p\ \sigma) \neq 0 \wedge \\ & \forall i : \mathsf{N}. i > (\textsc{length}_p \cdot \mathsf{app}_p\ \sigma) \to \sigma\ i = 0. \end{array}$$

Notice that Γ_{CP} is valid in some models such as the realisability model. Hence the function $\textsc{length}_p$ does not exist in those models and therefore is not definable in type theory—if it were definable, it would exist in every model.

5.7 Mathematical constructions with setoids

We study the development of mathematical theories using setoids as an implementation of the intuitive idea of sets. We will show that even if we work with partial setoids, we are forced to restrict the carrier type of the setoid to obtain the desired constructions. In particular, we will show that the partial setoid methodology runs into serious practical problems when dealing with subsetoids and quotients.

Subsetoids

Total setoids, as embodied by **TSet**, and partial setoids, as embodied in **RSet**, are based on two distinct ways of restricting the domain of a structure, that is,

of defining subsetoids. In the first case, restriction is achieved by modifying the underlying carrier of the setoid, while in the second case, restriction is achieved by modifying the setoid's underlying equality relation. However, we will show that total setoids are unavoidable, in the sense that, even if we use partial setoids, we will be forced to restrict the carrier type of a setoid to obtain certain subsetoids.

We begin by reviewing the definition of subsetoids in the context of total setoids. Intuitively a subsetoid is that part of a setoid whose elements satisfy a predicate. Predicates on setoids are defined as type-theoretic predicates on the carrier sets that are invariant for the setoid equality.

Definition 5.7.1 *Let A be a total setoid.* Setoid predicates *over A are the elements of the record type*

$$\begin{array}{rcl} \textsc{pred}_t\ A := \langle & \mathsf{pf}_t & : \mathsf{el}_t\ A \to \mathsf{Prop}, \\ & \mathsf{inv}_t & : \forall x, y : \mathsf{el}_t\ A.\ (x =_A y) \to (\mathsf{pf}_t\ x \to \mathsf{pf}_t\ y)\rangle. \end{array}$$

An equivalent definition would be $\textsc{pred}_t\ A = \textsc{map}_t\ A\ \Omega$.

In **TSet** the subsetoid of a setoid A defined by a predicate $P : \textsc{pred}_t\ A$ is obtained by first restricting the carrier type, and then constructing the setoid over this carrier by projecting the equality of A on the first component.

Definition 5.7.2 *Let A : **TSet** and $P : \textsc{pred}_t\ A$. The carrier of the subsetoid selected by P from A is*

$$\textsc{subCarrier}\ A\ P := \langle \mathsf{subel} : \mathsf{el}_t\ A, \mathsf{insub} : P \cdot \mathsf{pf}_t\ \mathsf{subel}\rangle$$

and the subsetoid is

$$\begin{array}{rcl} \textsc{subSetoid}_t\ A\ P := \langle\ \mathsf{el}_t & = & \textsc{subCarrier}\ A\ P, \\ \mathsf{eq}_t & = & \lambda x, y : \mathsf{el}_t.(\mathsf{subel}\ x) =_A (\mathsf{subel}\ y), \\ \mathsf{er} & = & \cdots\rangle. \end{array}$$

On the other hand, when using partial setoids, we do not change the underlying type, but we modify the equality. Predicates over partial setoids are defined in the same way as predicates over total setoids.

Definition 5.7.3 *Let A be a partial setoid.* Setoid predicates *over A are the elements of the record type*

$$\begin{array}{rcl} \textsc{pred}_p\ A := \langle & \mathsf{pf}_p & : \mathsf{el}_p\ A \to \mathsf{Prop}, \\ & \mathsf{inv}_p & : \forall x, y : \mathsf{el}_p\ A.\ (x =_A y) \to (\mathsf{pf}_p\ x \to \mathsf{pf}_p\ y)\rangle. \end{array}$$

The propositional function pf_p must be defined on the whole carrier type $\mathsf{el}_p\ A$, even on elements x for which $x =_A x$ is not true.

Definition 5.7.4 *If A is a partial setoid and $P : \textsc{pred}_p\ A$, then we define the subsetoid of A selected by P as*

$$\begin{array}{rcl} \textsc{subSetoid}_p\ A\ P := \langle\ \mathsf{el}_p & = & \mathsf{el}_p\ A, \\ \mathsf{eq}_p & = & \lambda x, y : \mathsf{el}_p\ A.(P \cdot \mathsf{pf}_p\ x) \wedge x =_A y, \\ \mathsf{per} & = & \cdots\rangle. \end{array}$$

In the definition of eq_p we do not require $(P \cdot \mathsf{pf}_p\ y)$ because it is derivable from $(P \cdot \mathsf{pf}_p\ x)$, $x =_A y$, and $P \cdot \mathsf{inv}_p$.

This definition has the nice property that an element of the carrier of the subsetoid is automatically an element of the carrier of the setoid. However, a serious drawback of this approach consists in the fact that a function defined on a subsetoid of A must be a type-theoretic function defined on the entire carrier type of A. In some cases, this cannot be done and the use of $\textsc{subSetoid}_t\ A\ P$ is unavoidable.

A first example of the above is given by the example of eventually null sequences that we developed in Subsection 5.6. Indeed, one can define the setoid of sequences as $\textsc{Map}_r\ \mathbb{N}_p\ \mathbb{N}_p$, define the predicate of being eventually zero and then form the subsetoid of eventually zero sequences. Using the results of the previous section, this gives us a first example of a function that cannot be defined using subsetoids *à la* partial setoid.

Below we develop a second example based on the real numbers. Here the idea is to define a setoid of real numbers and then restrict the setoid to smaller systems, say for example, the rationals or the natural numbers. Of course, one would hope that the number systems defined in this fashion enjoy the same properties and have the same definable functions as their more standard counterparts. It turns out that this is not possible in the framework of partial setoids.

It is well-known that in a constructive setting there are several possible implementations of real numbers, see for example, [CH92, CG00, GPWZ01, Har98, Jon93] for some works on the formalisation of reals in type theory. Here we choose to define the setoid of real numbers $\mathbb{R}$ using Cauchy sequences. The total setoid is defined as

$$\begin{array}{rcl} \mathbb{R}_t := \langle\ \mathsf{el}_t & = & \langle \mathsf{seq} : \mathsf{N} \to \mathsf{Q}, \mathsf{con} : \textsc{Cauchy}\ \mathsf{seq} \rangle, \\ \mathsf{eq}_t & = & \lambda r_1, r_2 : \mathsf{el}_t . r_1 \cdot \mathsf{seq} =_{\mathsf{conv}} r_2 \cdot \mathsf{seq}, \\ \mathsf{er} & = & \cdots \rangle \end{array}$$

where Q is the type of rational numbers—while rationals are usually defined as a setoid it is possible to define rationals as an inductive type with one 0-ary constructor and two unary constructors, as observed by Y. Bertot; in any case the exact definition of rationals below is irrelevant—, $\textsc{Cauchy}$ is the the property of being a Cauchy sequence of rationals:

$$\begin{array}{rcl} \textsc{Cauchy}\ s := & \forall i : \mathsf{N} . \exists k : \mathsf{N} . \forall j_1, j_2 : \mathsf{N} . \\ & j_1 > k \to j_2 > k \to |(s\ j_1) - (s\ j_2)| < 1/i \end{array}$$

and $=_{\mathsf{conv}}$ is the equality on sequences of rational numbers that holds whenever two sequences are co-convergent:

$$(s_1 =_{\mathsf{conv}} s_2) := \forall i : \mathsf{N} . \exists k : \mathsf{N} . \forall j : \mathsf{N} . j > k \to |(s_1\ j) - (s_2\ j)| < 1/i.$$

The corresponding partial setoid is

$$\begin{array}{rcl}\mathbb{R}_p := \langle\ \mathsf{el}_p & = & \mathsf{N} \to \mathsf{Q}, \\ \mathsf{eq}_p & = & \lambda r_1, r_2 : \mathsf{el}_t.\ (\textsc{Cauchy}\ r_1) \wedge (\textsc{Cauchy}\ r_2) \\ & & \qquad \wedge (r_1 \cdot \mathsf{seq} =_{\mathsf{conv}} r_2 \cdot \mathsf{seq}), \\ \mathsf{er} & = & \cdots \rangle.\end{array}$$

As emphasised above, it is convenient to consider smaller number systems, like the natural or rational numbers, as subsetoids of the real numbers—an alternative would be to consider implicit coercions, see for example, [Saï98] but this falls beyond the scope of this paper. We have a type Q of rational numbers, that is not a subsetoid of $\mathbb{R}$. We are going to define the subset of real numbers corresponding to the rationals. To this end we use the relation $\rightsquigarrow$ between $\mathsf{N} \to \mathsf{Q}$ and Q, such that $s \rightsquigarrow q$ holds if s converges to q:

$$s \rightsquigarrow q := \forall i : \mathsf{N}.\exists k : \mathsf{N}.\forall j : \mathsf{N}.i > k \to |(s\ j) - q| < 1/i.$$

We define the predicate $\textsc{isRational}_t : \textsc{pred}_t\ \mathbb{R}_t$ by $(\textsc{isRational}_t \cdot \mathsf{pf}_t\ r) = \exists q : \mathsf{Q}.\ s \cdot \mathsf{seq} \rightsquigarrow q$. The predicate $\textsc{isRational}_p : \textsc{pred}_p\ \mathbb{R}_p$ is defined correspondingly. Now we want to define the subsetoid of the reals whose elements are the real numbers that satisfy $\textsc{isRational}_t$. This is given by $\mathbb{Q}_t = (\textsc{subSetoid}_t\ \mathbb{R}_\mathsf{t}\ \textsc{isRational}_t)$ in the category **TSet** and by $\mathbb{Q}_p = (\textsc{subSetoid}_p\ \mathbb{R}_p\ \textsc{isRational}_p)$ in **RSet**. The problem now arises if we want to define a function on these subsetoids that depends strongly on the satisfaction of the condition. For example consider the function Num that gives the numerator of the reduced fraction representing a rational number.

In the framework of partial setoids, defining Num on $\mathbb{Q}_p$ requires that we define it on the whole type $\mathsf{N} \to \mathsf{Q}$, without any information on convergence. This is impossible because we cannot constructively compute whether a sequence converges to a rational value and, in such case, to which one. Further, it is also impossible to define Num with the axiom of descriptions. On the other hand, if we work in the framework of total setoids and use the axiom of descriptions for total setoids, we can easily define such a function for $\mathbb{Q}_t$ since we can extract from one of its elements r the proof $\mathsf{insub}\ r$ containing the rational value of r.

We conclude this subsection with a brief discussion on the definability of Num in the empty context. Two observations can be made:

- the function Num is definable without any axiom if we work in a predicative type theory that satisfies the axiom of choice, for example, Martin-Löf's type theory, since in this case we can extract a witness from a proof of existence;
- working in an impredicative type theory such as the Calculus of Inductive Constructions—where elimination of an existential quantification over a type is not possible and even leads to inconsistency [Coq86]—one can still define Num in the empty context by modifying the definition of subsetoid to include the witness directly.

$$\mathbb{Q}_t \cdot \mathsf{el}_t := \langle \mathsf{rval} : \mathbb{R}_t, \mathsf{qval} : \mathsf{Q}, \mathsf{convergence} : \mathsf{rval} \rightsquigarrow \mathsf{qval} \rangle.$$

In this case we can extract the value qval and define the function NUM.

Remark 5.7.5 *We did not formulate formal rules that the notion of subsetoid must satisfy. These rules are stated in [Jac99]. It can be verified that the notion of subsetoid given above satisfies those rules.*

Quotients

Both when working with total and partial setoids, quotients can be realized by just substituting the setoid equality with a stronger equivalence relation—below we refer to the latter as the quotienting relation. In either case the quotienting relation must preserve the setoid equality: if two elements are equal according to the book equality, they must be equivalent w.r.t the quotienting relation. However, when working with partial setoids, a problem arises: the quotienting relation may hold for elements that are not equal to themselves according to the setoid equality, that is, that are not in the domain of the setoid. In this case, taking the equivalence relation as the book equality of the quotient would add elements to the domain, which is incorrect. A possible solution is to take as book equality in the quotient setoid the restriction of the equivalence relation of the original setoid to the domain of the setoid. This definition of quotients, which essentially amounts to the one given by M. Hofmann in [Hof95b], is a bit cumbersome because the equality on the quotient is not the relation by which we quotiented. In our view, this should be another point in favour of total setoids.

For the sake of completeness, we briefly review the definition of quotients for partial setoids and illustrate in a concrete case some of the difficulties that arise by using these quotients in practice. Recall that an equivalence relation R over a setoid A is an element of $\mathrm{REL}_t\ A\ A$ that satisfies reflexivity, symmetry, and transitivity. Now assume that A is a total setoid and that R is such an equivalence relation with ϕ_R to witness that R is indeed an equivalence relation. For the sake of readability, we write $x \equiv_R y$ as a shorthand for $R \cdot \mathsf{rel}_t\ x\ y$. In **TSet** the quotient setoid A/R is defined as

$$\langle \mathsf{el}_t = \mathsf{el}_t\ A,\ \mathsf{eq}_t = \lambda x, y : \mathsf{el}_t\ A.\ x \equiv_R y, \mathsf{er} = \phi_R \rangle.$$

But in the same situation in **RSet**, that is, with A is a partial setoid, R a partial equivalence relation with ϕ_R to witness that R is indeed an partial equivalence relation—and using $x \equiv_R y$ as a shorthand for $R \cdot \mathsf{rel}_p\ x\ y$ it would be wrong to define (A/R) as

$$\langle \mathsf{el}_p = \mathsf{el}_p\ A,\ \mathsf{eq}_p = \lambda x, y : \mathsf{el}_p\ A.\ x \equiv_R y, \mathsf{per} = \phi_R \rangle$$

because it may happen that $x \equiv_R x$ holds when $x =_A x$ does not hold, so we are introducing new elements in the setoid. Instead, we must define first $R' : \mathrm{REL}_p\ A\ A$ as

$$\langle \mathsf{rel}_p = \lambda x, y : \mathsf{el}_p\ A. x =_A x \wedge y =_A y \wedge x \equiv_R y,\ \mathsf{compat}_p = \cdots \rangle$$

and then define (A/R) as

$$\langle \mathsf{el}_p = \mathsf{el}_p\ A,\ \mathsf{eq}_p = \lambda x, y : \mathsf{el}_p\ A.\ x \equiv_{R'} y, \mathsf{per} = \ldots \rangle.$$

If we go back to our example of eventually zero sequences, we may want to quotient the setoid by the relation that equates two sequences when they are proportional, that is, when all elements of one of the two can be obtained from the other by multiplication by a nonzero natural number:

$$\begin{array}{lcl} \textsc{proportional} & : & (\mathsf{N} \to \mathsf{N}) \to (\mathsf{N} \to \mathsf{N}) \to \mathsf{Prop} \\ & := & \lambda s_1, s_2 : \mathsf{N} \to \mathsf{N}. \\ & & \exists k_1, k_2 : \mathsf{N}. \forall i : \mathsf{N}. \mathsf{mult}\ k_1\ (s_1\ i) = \mathsf{mult}\ k_2\ (s_2\ i). \end{array}$$

This time the existential quantification does not give any constructivity problem: we do not need to extract k_1 and k_2 from a proof of $\textsc{proportional}\ s_1\ s_2$, because we can always choose $k_1 = (s_2\ 0)$ and $k_2 = (s_1\ 0)$.

This relation is invariant under the extensional equality of the sequences. It gives a binary setoid relation on the bounded sequences.

$$\begin{array}{lcl} \textsc{Proportional}_t & : & \textsc{rel}_t\ \textsc{ZSeq}_t\ \textsc{ZSeq}_t \\ & := & \langle \quad \mathsf{rel}_t = \lambda \sigma_1, \sigma_2 : \mathsf{el}_t\ \textsc{ZSeq}_t. \\ & & \qquad\qquad \textsc{proportional}\ \sigma_1 \cdot \mathsf{seq}\ \sigma_2 \cdot \mathsf{seq}, \\ & & \quad \mathsf{compat}_t = \cdots \rangle, \end{array}$$

$$\begin{array}{lcl} \textsc{Proportional}_p & : & \textsc{rel}_p\ \textsc{ZSeq}_t\ \textsc{ZSeq}_p \\ & := & \langle \quad \mathsf{rel}_p = \lambda s_1, s_2 : \mathsf{el}_p\ \textsc{ZSeq}_t. \\ & & \qquad\qquad \textsc{proportional}\ s_1\ s_2, \\ & & \quad \mathsf{compat}_p = \cdots \rangle. \end{array}$$

We can easily prove that $\textsc{Proportional}_t$ and $\textsc{Proportional}_p$ are equivalence relations on the corresponding total and partial setoids. Then we can define $\textsc{ZSeq}_t/\textsc{Proportional}_t$ and $\textsc{ZSeq}_p/\textsc{Proportional}_p$. Notice now that while the book equality on $\textsc{ZSeq}_t/\textsc{Proportional}_t$ is exactly $\textsc{Proportional}_t \cdot \mathsf{rel}_t$, the book equality on $\textsc{ZSeq}_p/\textsc{Proportional}_p$ is not even equivalent to $\textsc{Proportional}_p \cdot \mathsf{rel}_p$, but requires the additional property of the arguments being eventually zero. This is not a serious defect from a theoretical point of view, but in practice makes the use of quotients more awkward in **RSet** and in **TSet**.

Remark 5.7.6 *As for the notion of subsetoid, we did not formulate formal rules for quotients. Once again, see [Jac99] for the rules. It is a routine exercise to verify them with our definition.*

5.8 Conclusion

Type-theoretical frameworks are used as a foundation for mathematics in several ongoing efforts to develop large libraries of formalised mathematics with proof-assistants such as Agda, Coq and Lego. It is therefore natural to study the

relationship between type theory and the standard foundational framework for mathematics, that is, set theory. Recently, several authors, see for example, [Acz99, Wer97], have undertaken a systematic comparison between set theory and intensional type theory—see also [Acz78, Acz82, Acz86] for earlier work.

This paper focuses on a related issue, namely the relationship between existing approaches to formalise sets in type theory. We have shown how two widely used approaches to formalise sets in type theory, partial setoids and total setoids, are in fact not equivalent. In addition, we have gathered some evidence that partial setoids are not suitable for serving as a basis for the formalisation of mathematics. We thus conclude total setoids are better suited for the formalisation of mathematics.

Chapter 6

Nested General Recursion in Type Theory

coauthor: Ana Bove

This chapter is the fruit of collaboration with Ana Bove, of Chalmers University of Technology in Göteborg, Sweden. It was originally published in [BC01].

The subject of our research is the definition of general recursive functions in type theory, that is, functions that do not satisfy any restriction on recursive calls. These functions are not directly definable in type theory. However, a method developed by Ana Bove [Bov99, Bov01] extends the class of representable functions. Here, we present the method and extend it to general and nested recursive definitions.

Contrary to the rest of the thesis, the type theoretic computer tool of reference is not Coq but Alf [AGNvS94]. This is necessary, because we use features of type theory that are not available in Coq, but are implemented in Alf.

Another distinctive characteristic is that we do not use the sort Prop for propositions, but rather use Set both for data types and propositions.

6.1 Introduction

General recursive algorithms are defined by cases where the recursive calls are performed on objects that satisfy no syntactic condition guaranteeing termination. As a consequence, there is no direct way of formalising them in type theory. The standard way of handling general recursion in type theory uses a well-founded recursion principle derived from the accessibility predicate Acc (see [Acz77, Nor88, BB00]). The idea behind the accessibility predicate is that an element a is accessible by a relation $\prec$ if there exists no infinite decreasing sequence starting from a. A set A is said to be well-founded with respect to $\prec$ if all its elements are accessible by $\prec$. Formally, given a set A, a binary relation $\prec$ on A and an element a in A, we can form the set $\mathsf{Acc}(A, \prec, a)$. The only

introduction rule for the accessibility predicate is

$$\frac{a\colon A \quad p\colon (x\colon A;\, h\colon x \prec a)\mathsf{Acc}(A, \prec, x)}{\mathsf{acc}(a, p)\colon \mathsf{Acc}(A, \prec, a)}.$$

The corresponding elimination rule, also known as the rule of well-founded recursion, is

$$\frac{\begin{array}{c} a\colon A \\ h\colon \mathsf{Acc}(A, \prec, a) \\ e\colon (x\colon A;\, h_x\colon \mathsf{Acc}(A, \prec, x);\, p_x\colon (y\colon A;\, q\colon y \prec x)P(y))P(x) \end{array}}{\mathsf{wfrec}(a, h, e)\colon P(a)}$$

and its computation rule is

$$\mathsf{wfrec}(a, \mathsf{acc}(a, p), e) = e(a, \mathsf{acc}(a, p), [y, q]\mathsf{wfrec}(y, p(y, q), e))\colon P(a).$$

Hence, to guarantee that a general recursive algorithm that performs the recursive calls on elements of type A terminates, we have to prove that A is well-founded and that the arguments supplied to the recursive calls are smaller than the input.

Since Acc is a general predicate, it gives no information that can help us in the formalization of a specific recursive algorithm. As a consequence, its use in the formalization of general recursive algorithms often results in long and complicated code. On the other hand, functional programming languages like Haskell [JHA+99] impose no restrictions on recursive programs; therefore, writing general recursive algorithms in Haskell is straightforward. In addition, functional programs are usually short and self-explanatory. However, there is no powerful framework to reason about the correctness of Haskell-like programs.

From [Bov99] we can extract a method to formalise simple general recursive algorithms in type theory (by simple we mean non-nested and non-mutually recursive) in a clear and compact way. We believe that this technique helps to close the gap between programming in a functional language and programming in type theory.

This work is similar to that of Paulson in [Pau86]. He defines an ordering associated with the recursive steps of an algorithm, such that the inputs on which the algorithm terminates are the objects accessible by the order. Then he defines the algorithm by induction on the order. The proof of termination for the algorithm reduces to a proof that the order is wellfounded. Bove's idea is a translation of this in the framework of type theory in a way more convenient than the straightforward translation. Given the Haskell version of an algorithm f_alg, the method in [Bov99] uses an inductive special-purpose accessibility predicate called fAcc. We construct this predicate directly from f_alg, and we regard it as a characterization of the collection of inputs on which f_alg terminates. It has an introduction rule for each case in the algorithm and provides a syntactic condition that guarantees termination. In this way, we can formalise f_alg in type theory by structural recursion on the proof that the input of f_alg satisfies fAcc, obtaining a compact and readable formalization of the algorithm.

However, the technique in [Bov99] cannot be immediately applied to nested recursive algorithms. Here, we present a method for formalising nested recursive algorithms in type theory in a similar way to the one used in [Bov99]. Thus, we obtain short and clear formalisations of nested recursive algorithms in type theory. This technique uses the schema for simultaneous inductive-recursive definitions presented by Dybjer in [Dyb00]; hence, it can be used only in type theories extended with such schema.

The rest of the paper is organised as follows. In section 6.2, we illustrate the method used in [Bov99] on a simple example. In addition, we point out the advantages of this technique over the standard way of defining general recursive algorithms in type theory by using the predicate Acc. In section 6.3, we adapt the method to nested recursive algorithms, using Dybjer's schema. In section 6.4, we show how the method can be put to use also in the formalisation of partial functions. Finally, in section 6.5, we present some conclusions and related work.

6.2 Simple General Recursion in Type Theory

Here, we illustrate the technique used in [Bov99] on a simple example: the modulo algorithm on natural numbers. In addition, we point out the advantages of this technique over the standard way of defining general recursive algorithms in type theory by using the accessibility predicate Acc.

First, we give the Haskell version of the modulo algorithm. Second, we define the type-theoretic version of it that uses the standard accessibility predicate Acc to handle the recursive call, and we point out the problems of this formalisation. Third, we introduce a special-purpose accessibility predicate, ModAcc, specifically defined for this case study. Intuitively, this predicate defines the collection of pairs of natural numbers on which the modulo algorithm terminates. Fourth, we present a formalisation of the modulo algorithm in type theory by structural recursion on the proof that the input pair of natural numbers satisfies the predicate ModAcc. Finally, we show that all pairs of natural numbers satisfy ModAcc, which implies that the modulo algorithm terminates on all inputs.

In the Haskell definition of the modulo algorithm we use the set `N` of natural numbers, the subtraction operation `<->` and the less-than relation `<<` over `N`, defined in Haskell in the usual way. We also use Haskell's data type `Maybe A`, whose elements are `Nothing` and `Just a`, for any `a` of type `A`. Here is the Haskell code for the modulo algorithm[1]:

```
mod :: N -> N -> Maybe N
mod n 0 = Nothing
mod n m | n << m = Just n
        | not(n << m) = mod (n <-> m) m.
```

It is evident that this algorithm terminates on all inputs. However, the recursive call is made on the argument $n - m$, which is not structurally smaller than the

[1] For the sake of simplicity, we ignore efficiency aspects such as the fact that the expression `n << m` is computed twice.

argument n, although the value of $n - m$ is smaller than n.

Before introducing the type-theoretic version of the algorithm that uses the standard accessibility predicate, we give the types of two operators and two lemmas[2]:

$$\begin{array}{lcllcl} - & : & (n, m\colon \mathbb{N})\mathbb{N} & \mathsf{less\text{-}dec} & : & (n, m\colon \mathbb{N})\mathsf{Dec}(n < m) \\ < & : & (n, m\colon \mathbb{N})\mathsf{Set} & \mathsf{min\text{-}less} & : & (n, m\colon \mathbb{N}; \neg(n < \mathsf{S}(m)))(n - \mathsf{S}(m) < n). \end{array}$$

On the left side we have the types of the subtraction operation and the less-than relation over natural numbers. On the right side we have the types of two lemmas that we use later on. The first lemma states that it is decidable whether a natural number is less than another. The second lemma establishes that if the natural number n is not less than the natural number $\mathsf{S}(m)$, then the result of subtracting $\mathsf{S}(m)$ from n is less than n. The hypothesis is necessary because the subtraction of a larger number from a smaller one is set to be 0 by default.

In place of Haskell's `Maybe` type, we use the type-theoretic disjunction of the set $\mathbb{N}$ of natural numbers and the singleton set Error whose only element is error. The type-theoretic version of the modulo algorithm that uses the standard accessibility predicate Acc to handle the recursive call is[3]

$$\begin{array}{l} \mathsf{mod_{acc}}\colon (n, m\colon \mathbb{N}; \mathsf{Acc}(\mathbb{N}, <, n))\mathbb{N} \vee \mathsf{Error} \\ \quad \mathsf{mod_{acc}}(n, 0, \mathsf{acc}(n, p)) = \mathsf{inr}(\mathsf{error}) \\ \quad \mathsf{mod_{acc}}(n, \mathsf{S}(m_1), \mathsf{acc}(n, p)) = \\ \qquad \textbf{case } \mathsf{less\text{-}dec}(n, \mathsf{S}(m_1))\colon \mathsf{Dec}(n < \mathsf{S}(m_1)) \textbf{ of} \\ \qquad\quad \mathsf{inl}(q_1) \;\Rightarrow\; \mathsf{inl}(n) \\ \qquad\quad \mathsf{inr}(q_2) \;\Rightarrow\; \mathsf{mod_{acc}}(n - \mathsf{S}(m_1), \mathsf{S}(m_1), p(n - \mathsf{S}(m_1), \mathsf{min\text{-}less}(n, m_1, q_2))) \\ \qquad \textbf{end}. \end{array}$$

This algorithm is defined by recursion on the proof that the first argument of the modulo operator is accessible by $<$. We first distinguish cases on m. If m is zero, we return an error, because the modulo zero operation is not defined. If m is equal to $\mathsf{S}(m_1)$ for some natural number m_1, we distinguish cases on whether n is smaller than $\mathsf{S}(m_1)$. If so, we return the value n. Otherwise, we subtract $\mathsf{S}(m_1)$ from n and we call the modulo algorithm recursively on the values $n - \mathsf{S}(m_1)$ and $\mathsf{S}(m_1)$. The recursive call needs a proof that the value $n - \mathsf{S}(m_1)$ is accessible. This proof is given by the expression $p(n - \mathsf{S}(m_1), \mathsf{min\text{-}less}(n, m_1, q_2))$, which is structurally smaller than $\mathsf{acc}(n, p)$. We can easily define a function $\mathsf{allacc}_{\mathbb{N}}$ that, applied to a natural number n, returns a proof that n is accessible by $<$. We use this function to define the desired modulo algorithm:

$$\begin{array}{l} \mathsf{Mod_{acc}}\colon (n, m\colon \mathbb{N})\mathbb{N} \vee \mathsf{Error} \\ \quad \mathsf{Mod_{acc}}(n, m) = \mathsf{mod_{acc}}(n, m, \mathsf{allacc}_{\mathbb{N}}(n)). \end{array}$$

The main disadvantage of this formalisation of the modulo algorithm is that we have to supply a proof that $n - \mathsf{S}(m_1)$ is accessible by $<$ to the recursive call.

[2] Dec is the decidability predicate: given a proposition P, $\mathsf{Dec}(P) \equiv P \vee \neg P$.

[3] The set former $\vee$ represents the disjunction of two sets, and inl and inr the two constructors of the set.

This proof has no computational content and its only purpose is to serve as a structurally smaller argument on which to perform the recursion. Notice that, even for such a small example, this accessibility proof distracts our attention and enlarges the code of the algorithm.

To overcome this problem, we define a special-purpose accessibility predicate, ModAcc, containing information that helps us to write a new type-theoretic version of the algorithm. To construct this predicate, we ask ourselves the following question: on which inputs does the modulo algorithm terminate? To find the answer, we inspect closely the Haskell version of the modulo algorithm. We can directly extract from its structure the conditions that the input values should satisfy to produce a basic (that is, non recursive) result or to perform a terminating recursive call. In other words, we formulate the property that an input value must satisfy for the computation to terminate: either the algorithm does not perform any recursive call, or the values on which the recursive calls are performed have themselves the property. We distinguish three cases:

- if the input numbers are n and zero, then the algorithm terminates;
- if the input number n is less than the input number m, then the algorithm terminates;
- if the number n is not less than the number m and m is not zero[4], then the algorithm terminates on the inputs n and m if it terminates on the inputs $n - m$ and m.

Following this description, we define the inductive predicate ModAcc over pairs of natural numbers by the introduction rules (for n and m natural numbers)

$$\frac{}{\mathsf{ModAcc}(n,0)}, \quad \frac{n < m}{\mathsf{ModAcc}(n,m)}, \quad \frac{\neg(m = 0) \quad \neg(n < m) \quad \mathsf{ModAcc}(n-m,m)}{\mathsf{ModAcc}(n,m)}.$$

This predicate can easily be formalised in type theory:

$$\begin{array}{l}
\mathsf{ModAcc}\colon (n,m\colon \mathbb{N})\mathsf{Set} \\
\quad \begin{array}{lcl}
\mathsf{modacc}_0 & : & (n\colon \mathbb{N})\mathsf{ModAcc}(n,0) \\
\mathsf{modacc}_< & : & (n,m\colon \mathbb{N}; n < m)\mathsf{ModAcc}(n,m) \\
\mathsf{modacc}_\geq & : & (n,m\colon \mathbb{N}; \neg(m=0); \neg(n<m); \mathsf{ModAcc}(n-m,m)) \\
 & & \mathsf{ModAcc}(n,m).
\end{array}
\end{array}$$

We now use this predicate to formalise the modulo algorithm in type theory:

$$\begin{array}{l}
\mathsf{mod}\colon (n,m\colon \mathbb{N}; \mathsf{ModAcc}(n,m))\mathbb{N} \vee \mathsf{Error} \\
\quad \mathsf{mod}(n,0,\mathsf{modacc}_0(n)) = \mathsf{inr}(\mathsf{error}) \\
\quad \mathsf{mod}(n,m,\mathsf{modacc}_<(n,m,q)) = \mathsf{inl}(n) \\
\quad \mathsf{mod}(n,m,\mathsf{modacc}_\geq(n,m,q_1,q_2,h)) = \mathsf{mod}(n-m,m,h).
\end{array}$$

[4]Observe that this condition is not needed in the Haskell version of the algorithm due to the order in which Haskell processes the equations that define an algorithm.

This algorithm is defined by structural recursion on the proof that the input pair of numbers satisfies the predicate ModAcc. The first two equations are straightforward. The last equation considers the case where n is not less than m; here q_1 is a proof that m is different from zero, q_2 is a proof that n is not less than m and h is a proof that the pair $(n-m,\, m)$ satisfies the predicate ModAcc. In this case, we call the algorithm recursively on the values $n-m$ and m. We have to supply a proof that the pair $(n-m,\, m)$ satisfies the predicate ModAcc to the recursive call, which is given by the argument h.

To prove that the modulo algorithm terminates on all inputs, we use the auxiliary lemma $\mathsf{modacc_{aux}}$. Given a natural number m, this lemma proves $\mathsf{ModAcc}(i, m)$, for i an accessible natural number, from the assumption that $\mathsf{ModAcc}(j, m)$ holds for every natural number j smaller than i. The proof proceeds by case analysis on m and, when m is equal to $\mathsf{S}(m_1)$ for some natural number m_1, by cases on whether i is smaller than $\mathsf{S}(m_1)$. The term $\mathsf{nots0}(m_1)$ is a proof that $\mathsf{S}(m_1)$ is different from 0.

$$
\begin{array}{l}
\mathsf{modacc_{aux}} \quad : \quad (m, i\colon \mathbb{N}; \mathsf{Acc}(\mathbb{N}, <, i); f\colon (j\colon \mathbb{N}; j < i)\mathsf{ModAcc}(j, m)) \\
\qquad\qquad\qquad\quad \mathsf{ModAcc}(i, m) \\
\quad \mathsf{modacc_{aux}}(0, i, h, f) = \mathsf{modacc}_0(i) \\
\quad \mathsf{modacc_{aux}}(\mathsf{S}(m_1), i, h, f) = \\
\qquad \textbf{case}\ \mathsf{less\text{-}dec}(i, \mathsf{S}(m_1))\colon \mathsf{Dec}(i < \mathsf{S}(m_1))\ \textbf{of} \\
\qquad\quad \mathsf{inl}(q_1) \quad \Rightarrow \quad \mathsf{modacc}_<(i, \mathsf{S}(m_1), q_1) \\
\qquad\quad \mathsf{inr}(q_2) \quad \Rightarrow \quad \mathsf{modacc}_\geq(i, \mathsf{S}(m_1), \mathsf{nots0}(m_1), q_2, \\
\qquad\qquad\qquad\qquad\qquad\qquad f(i - \mathsf{S}(m_1), \mathsf{min\text{-}less}(i, m_1, q_2))) \\
\qquad \textbf{end}
\end{array}
$$

Now, we prove that the modulo algorithm terminates on all inputs, that is, we prove that all pairs of natural numbers satisfy ModAcc[5]:

$$
\begin{array}{l}
\mathsf{allModAcc}\colon (n, m\colon \mathbb{N})\mathsf{ModAcc}(n, m) \\
\quad \mathsf{allModAcc}(n, m) = \mathsf{wfrec}(n, \mathsf{allacc}_\mathbb{N}(n), \mathsf{modacc_{aux}}(m)).
\end{array}
$$

Notice that the skeleton of the proof of the function $\mathsf{modacc_{aux}}$ is very similar to the skeleton of the algorithm $\mathsf{mod_{acc}}$.

Finally, we can use the previous function to write the final modulo algorithm:

$$
\begin{array}{l}
\mathsf{Mod}\colon (n, m\colon \mathbb{N})\mathbb{N} \vee \mathsf{Error} \\
\quad \mathsf{Mod}(n, m) = \mathsf{mod}(n, m, \mathsf{allModAcc}(n, m)).
\end{array}
$$

Observe that, even for such a small example, the version of the algorithm that uses our special predicate is slightly shorter and more readable than the type-theoretic version of the algorithm that is defined by using the predicate Acc. Notice also that we were able to move the non-computational parts from the code of $\mathsf{mod_{acc}}$ into the proof that the predicate ModAcc holds for all possible inputs, thus separating the actual algorithm from the proof of its termination.

[5]Here, we use the general recursor wfrec with the elimination predicate $P(n) \equiv \mathsf{ModAcc}(n, m)$.

We hope that, by now, the reader is quite familiar with our notation. So, in the following sections, we will not explain the type-theoretic codes in detail.

6.3 Nested Recursion in Type Theory

The technique we have just described to formalise simple general recursion cannot be applied to nested general recursive algorithms in a straightforward way. We illustrate the problem on a simple nested recursive algorithm over natural numbers. Its Haskell definition is

```
nest :: N -> N
nest 0 = 0
nest (S n) = nest(nest n).
```

Clearly, this is a total algorithm returning 0 on every input.

If we want to use the technique described in the previous section to formalise this algorithm, we need to define an inductive special-purpose accessibility predicate NestAcc over the natural numbers. To construct NestAcc, we ask ourselves the following question: on which inputs does the nest algorithm terminate? By inspecting the Haskell version of the nest algorithm, we distinguish two cases:

- if the input number is 0, then the algorithm terminates;
- if the input number is $\mathsf{S}(n)$ for some natural number n, then the algorithm terminates if it terminates on the inputs n and $\mathsf{nest}(n)$.

Following this description, we define the inductive predicate NestAcc over natural numbers by the introduction rules (for n natural number)

$$\frac{}{\mathsf{NestAcc}(0)}, \qquad \frac{\mathsf{NestAcc}(n) \quad \mathsf{NestAcc}(\mathsf{nest}(n))}{\mathsf{NestAcc}(\mathsf{S}(n))}.$$

Unfortunately, this definition is not correct since nest is not yet defined. Moreover, the purpose of defining the predicate NestAcc is to be able to define the algorithm nest by structural recursion on the proof that its input value satisfies NestAcc. Hence, the definitions of NestAcc and nest are locked in a vicious circle.

However, there is an extension of type theory that gives us the means to define the predicate NestAcc inductively generated by two constructors corresponding to the two introduction rules of the previous paragraph. This extension has been introduced by Dybjer in [Dyb00] and it allows the simultaneous definition of an inductive predicate P and a function f, where f has the predicate P as part of its domain and is defined by recursion on P. In our case, given the input value n, nest requires an argument of type $\mathsf{NestAcc}(n)$. Using Dybjer's

schema, we can simultaneously define NestAcc and nest:

$$\begin{array}{lcl}
\mathsf{NestAcc} & : & (n\colon \mathbb{N})\mathsf{Set} \\
\mathsf{nest} & : & (n\colon \mathbb{N}; \mathsf{NestAcc}(n))\mathbb{N}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{nestacc0} & : & \mathsf{NestAcc}(0) \\
\mathsf{nestaccs} & : & (n\colon \mathbb{N}; h_1\colon \mathsf{NestAcc}(n); h_2\colon \mathsf{NestAcc}(\mathsf{nest}(n, h_1))) \\
 & & \mathsf{NestAcc}(\mathsf{S}(n))
\end{array}$$

$$\mathsf{nest}(0, \mathsf{nestacc0}) = 0$$
$$\mathsf{nest}(\mathsf{S}(n), \mathsf{nestaccs}(n, h_1, h_2)) = \mathsf{nest}(\mathsf{nest}(n, h_1), h_2).$$

This definition may at first look circular: the type of nest requires that the predicate NestAcc is defined, while the type of the constructor nestaccs of the predicate NestAcc requires that nest is defined. However, we can see that it is not so by analysing how the elements in NestAcc and the values of nest are generated. First of all, $\mathsf{NestAcc}(0)$ is well defined because it does not depend on any assumption and its only element is nestacc0. Once $\mathsf{NestAcc}(0)$ is defined, the result of nest on the inputs 0 and nestacc0 becomes defined and its value is 0. Now, we can apply the constructor nestaccs to the arguments $n = 0$, $h_1 = \mathsf{nestacc0}$ and $h_2 = \mathsf{nestacc0}$. This application is well typed since h_2 must be an element in $\mathsf{NestAcc}(\mathsf{nest}(0, \mathsf{nestacc0}))$, that is, $\mathsf{NestAcc}(0)$. At this point, we can compute the value of $\mathsf{nest}(\mathsf{S}(0), \mathsf{nestaccs}(0, \mathsf{nestacc0}, \mathsf{nestacc0}))$ and obtain the value zero[6], and so on. Circularity is avoided because the values of nest can be computed at the moment a new proof of the predicate NestAcc is generated; in turn, each constructor of NestAcc calls nest only on those arguments that appear previously in its assumptions, for which we can assume that nest has already been computed.

The next step consists in proving that the predicate NestAcc is satisfied by all natural numbers:

$$\mathsf{allNestAcc}\colon (n\colon \mathbb{N})\mathsf{NestAcc}(n).$$

This can be done by first proving that, given a natural number n and a proof h of $\mathsf{NestAcc}(n)$, $\mathsf{nest}(n, h) \leq n$ (by structural recursion on h), and then using well-founded recursion on the set of natural numbers.

Now, we define Nest as a function from natural numbers to natural numbers:

$$\begin{array}{l}
\mathsf{Nest}\colon (n\colon \mathbb{N})\mathbb{N} \\
\quad \mathsf{Nest}(n) = \mathsf{nest}(n, \mathsf{allNestAcc}(n)).
\end{array}$$

Notice that by making the simultaneous definition of NestAcc and nest we can treat nested recursion similarly to how we treat simple recursion. In this way, we obtain a short and clear formalisation of the nest algorithm.

To illustrate our technique for nested general recursive algorithms in more interesting situations, we present a slightly more complicated example: Paulson's

[6] Since $\mathsf{nest}(\mathsf{S}(0), \mathsf{nestaccs}(0, \mathsf{nestacc0}, \mathsf{nestacc0})) = \mathsf{nest}(\mathsf{nest}(0, \mathsf{nestacc0}), \mathsf{nestacc0}) = \mathsf{nest}(0, \mathsf{nestacc0}) = 0$

normalisation function for conditional expressions [Pau86]. Its Haskell definition is

```
data CExp = At | If CExp CExp CExp

nm :: CExp -> CExp
nm At = At
nm (If At y z) = If At (nm y) (nm z)
nm (If (If u v w) y z) = nm (If u (nm (If v y z)) (nm (If w y z))).
```

To define the special-purpose accessibility predicate, we study the different equations in the Haskell version of the algorithm, putting the emphasis on the input expressions and the expressions on which the recursive calls are performed. We obtain the following introduction rules for the inductive predicate nmAcc (for y, z, u, v and w conditional expressions):

$$\frac{}{\mathsf{nmAcc}(\mathsf{At})}, \qquad \frac{\mathsf{nmAcc}(y) \quad \mathsf{nmAcc}(z)}{\mathsf{nmAcc}(\mathsf{If}(\mathsf{At}, y, z))}, \qquad \frac{\begin{array}{c}\mathsf{nmAcc}(\mathsf{If}(v, y, z)) \\ \mathsf{nmAcc}(\mathsf{If}(w, y, z)) \\ \mathsf{nmAcc}(\mathsf{If}(u, \mathsf{nm}(\mathsf{If}(v, y, z)), \mathsf{nm}(\mathsf{If}(w, y, z))))\end{array}}{\mathsf{nmAcc}(\mathsf{If}(\mathsf{If}(u, v, w), y, z))}.$$

In type theory, we define the inductive predicate nmAcc simultaneously with the function nm, recursively defined on nmAcc:

$$\begin{array}{lcl} \mathsf{nmAcc} & : & (e\colon \mathsf{CExp})\mathsf{Set} \\ \mathsf{nm} & : & (e\colon \mathsf{CExp}; \mathsf{nmAcc}(e))\mathsf{CExp} \end{array}$$

$$\begin{array}{lcl} \mathsf{nmacc}_1 & : & \mathsf{nmAcc}(\mathsf{At}) \\ \mathsf{nmacc}_2 & : & (y, z\colon \mathsf{CExp}; \mathsf{nmAcc}(y); \mathsf{nmAcc}(z))\mathsf{nmAcc}(\mathsf{If}(\mathsf{At}, y, z)) \\ \mathsf{nmacc}_3 & : & (u, v, w, y, z\colon \mathsf{CExp}; \\ & & h_1\colon \mathsf{nmAcc}(\mathsf{If}(v, y, z)); h_2\colon \mathsf{nmAcc}(\mathsf{If}(w, y, z)); \\ & & h_3\colon \mathsf{nmAcc}(\mathsf{If}(u, \mathsf{nm}(\mathsf{If}(v, y, z), h_1), \mathsf{nm}(\mathsf{If}(w, y, z), h_2)))) \\ & & \mathsf{nmAcc}(\mathsf{If}(\mathsf{If}(u, v, w), y, z)) \end{array}$$

$$\begin{array}{l} \mathsf{nm}(\mathsf{At}, \mathsf{nmacc}_1) = \mathsf{At} \\ \mathsf{nm}(\mathsf{If}(\mathsf{At}, y, z), \mathsf{nmacc}_2(y, z, h_1, h_2)) = \mathsf{If}(\mathsf{At}, \mathsf{nm}(y, h_1), \mathsf{nm}(z, h_2)) \\ \mathsf{nm}(\mathsf{If}(\mathsf{If}(u, v, w), y, z), \mathsf{nmacc}_3(u, v, w, y, z, h_1, h_2, h_3)) = \\ \quad \mathsf{nm}(\mathsf{If}(u, \mathsf{nm}(\mathsf{If}(v, y, z), h_1), \mathsf{nm}(\mathsf{If}(w, y, z), h_2)), h_3). \end{array}$$

We can justify this definition as we did for the nest algorithm, reasoning about the well-foundedness of the recursive calls: the function nm takes a proof that the input expression satisfies the predicate nmAcc as an extra argument and it is defined by structural recursion on that proof, and each constructor of nmAcc calls nm only on those proofs that appear previously in its assumptions, for which we can assume that nm has already been computed.

Once again, the next step consists in proving that the predicate nmAcc is satisfied by all conditional expressions:

$$\mathsf{allnmAcc}\colon (e\colon \mathsf{CExp})\mathsf{nmAcc}(e).$$

To do this, we first show that the constructors of the predicate nmAcc use inductive assumptions on smaller arguments, though not necessarily structurally smaller ones. To that end, we define a measure that assigns a natural number to each conditional expression:

$$|\mathsf{At}| = 1 \quad \text{and} \quad |\mathsf{If}(x, y, z)| = |x| * (1 + |y| + |z|).$$

With this measure, it is easy to prove that

$$|\mathsf{If}(v, y, z)| < |\mathsf{If}(\mathsf{If}(u, v, w), y, z)|, \quad |\mathsf{If}(w, y, z)| < |\mathsf{If}(\mathsf{If}(u, v, w), y, z)| \\ \text{and} \quad |\mathsf{If}(u, v', w')| < |\mathsf{If}(\mathsf{If}(u, v, w), y, z)|$$

for every v', w' such that $|v'| \leq |\mathsf{If}(v, y, z)|$ and $|w'| \leq |\mathsf{If}(w, y, z)|$. Therefore, to prove that the predicate nmAcc holds for a certain $e\colon \mathsf{CExp}$, we need to call nm only on those arguments that have smaller measure than e[7].

Now, we can prove that every conditional expression satisfies nmAcc by first proving that, given a conditional expression e and a proof h of $\mathsf{nmAcc}(e)$, $|\mathsf{nm}(e, h)| \leq |e|$ (by structural recursion on h), and then using well-founded recursion on the set of natural numbers.

We can then define NM as a function from conditional expressions to conditional expressions:

$$\begin{array}{l} \mathsf{NM}\colon (e\colon \mathsf{CExp})\mathsf{CExp} \\ \quad \mathsf{NM}(e) = \mathsf{nm}(e, \mathsf{allnmAcc}(e)). \end{array}$$

6.4 Partial Functions in Type Theory

Until now we have applied our technique to total functions for which totality could not be proven easily by structural recursion. However, it can also be put to use in the formalisation of partial functions. A standard way to formalise partial functions in type theory is to define them as relations rather than objects of a function type. For example, the minimization operator for natural numbers, which takes a function $f\colon (\mathbb{N})\mathbb{N}$ as input and gives the least $n\colon \mathbb{N}$ such that $f(n) = 0$ as output, cannot be represented as an object of type $((\mathbb{N})\mathbb{N})\mathbb{N}$ because it does not terminate on all inputs. A standard representation of this operator in type theory is the inductive relation

$$\begin{array}{lcl} \mu\colon (f\colon (\mathbb{N})\mathbb{N}; n\colon \mathbb{N})\mathsf{Set} & & \\ \quad \mu_0 & : & (f\colon (\mathbb{N})\mathbb{N}; f(0) = 0)\mu(f, 0) \\ \quad \mu_1 & : & (f\colon (\mathbb{N})\mathbb{N}; f(0) \neq 0; n\colon \mathbb{N}; \mu([m]f(\mathsf{S}(m)), n))\mu(f, \mathsf{S}(n)). \end{array}$$

The relation μ represents the graph of the minimization operator. If we indicate the minimization function by min, then $\mu(f, n)$ is inhabited if and only if $\mathsf{min}(f) = n$. The fact that min may be undefined on some function f is expressed by $\mu(f, n)$ being empty for every natural number n.

[7] We could have done something similar in the case of the algorithm nest by defining the measure $|x| = x$ and proving the inequality $y < \mathsf{S}(x)$ for every $y \leq x$

There are reasons to be unhappy with this approach. First, for a relation to really define a partial function, we must prove that it is univocal: in our case, that for all $n, m\colon \mathbb{N}$, if $\mu(f, n)$ and $\mu(f, m)$ are both nonempty then $n = m$. Second, there is no computational content in this representation, that is, we cannot actually compute the value of $\mathsf{min}(f)$ for any f.

Let us try to apply our technique to this example and start with the Haskell definition of min:

```
min :: (N -> N) -> N
min f | f 0 == 0 = 0
      | f 0 /= 0 = s (min (\m -> f (s m))).
```

We observe that the computation of min on the input f terminates if $f(0) = 0$ or if $f(0) \neq 0$ and min terminates on the input $[m]f(\mathsf{S}(m))$. This leads to the inductive definition of the special predicate minAcc on functions defined by the introduction rules (for f a function from natural numbers to natural numbers and m a natural number)

$$\frac{f(0) = 0}{\mathsf{minAcc}(f)}, \qquad \frac{f(0) \neq 0 \quad \mathsf{minAcc}([m]f(\mathsf{S}(m)))}{\mathsf{minAcc}(f)}.$$

We can directly translate these rules into type theory:

$$\begin{array}{l} \mathsf{minAcc}\colon (f\colon (\mathbb{N})\mathbb{N})\mathsf{Set} \\ \quad \mathsf{minacc0} \quad : \quad (f\colon (\mathbb{N})\mathbb{N}; f(0) = 0)\mathsf{minAcc}(f) \\ \quad \mathsf{minacc1} \quad : \quad (f\colon (\mathbb{N})\mathbb{N}; f(0) \neq 0; \mathsf{minAcc}([m]f(\mathsf{S}(m))))\mathsf{minAcc}(f). \end{array}$$

Now, we define min for those inputs that satisfy minAcc:

$$\begin{array}{l} \mathsf{min}\colon (f\colon (\mathbb{N})\mathbb{N}; \mathsf{minAcc}(f))\mathbb{N} \\ \quad \mathsf{min}(f, \mathsf{minacc0}(f, q)) = 0 \\ \quad \mathsf{min}(f, \mathsf{minacc1}(f, q, h)) = \mathsf{S}(\mathsf{min}([m]f(\mathsf{S}(m)), h)). \end{array}$$

In this case, it is not possible to prove that all elements in $(\mathbb{N})\mathbb{N}$ satisfy the special predicate, simply because it is not true. However, given a function f, we may first prove $\mathsf{minAcc}(f)$ (that is, that the recursive calls in the definition of min are well-founded and, thus, that the function min terminates for the input f) and then use min to actually compute the value of the minimization of f.

Partial functions can also be defined by occurrences of nested recursive calls, in which case we need to use simultaneous inductive-recursive definitions. We show how this works on the example of a reduction algorithm for terms of the untyped λ-calculus. This algorithm gives a normal form of the lambda term in input, when it terminates. However, there are terms that have a normal form, but on which this algorithm does not terminate. There are other reduction strategies that always produce the normal form, if there is one (see [Bar84]). However, we chose the weaker reduction strategy because it gives a good illustration of the role of nested recursion in the definition of partial recursive functions. The Haskell program that implements the algorithm is

```
data Lambda = Var N | Abst N Lambda | App Lambda Lambda

sub :: Lambda -> N -> Lambda -> Lambda

nf :: Lambda -> Lambda
nf (Var i) = Var i
nf (Abst i a) = Abst i (nf a)
nf (App a b) = case (nf a) of
                  Var i -> App (Var i) (nf b)
                  Abst i a' -> nf (sub a' i b)
                  App a' a'' -> App (App a' a'') (nf b).
```

The elements of `Lambda` denote λ-terms: `Var i`, `Abst i a` and `App a b` denote the variable x_i, the term $(\lambda x_i.a)$ and the term $a(b)$, respectively. We assume that a substitution algorithm `sub` is given, such that `(sub a i b)` computes the term $a[x_i := b]$.

Notice that the algorithm contains a hidden nested recursion: in the second sub-case of the case expression, the term `a'`, produced by the call `(nf a)`, appears inside the call `nf (sub a' i b)`. This sub-case could be written in the following way, where we abuse notation to make the nested calls explicit:

```
nf (App a b) = nf (let (Abst i a') = nf a in (sub a' i b)).
```

Let Λ be the type-theoretic definition of `Lambda`. To formalise the algorithm, we use the method described in the previous section with simultaneous induction-recursion definitions. The introduction rules for the special predicate nfAcc, some of which use nf in their premises, are (for i natural number, and a, a', a'' and b λ-terms)

$$\frac{}{\mathsf{nfAcc}(\mathsf{Var}(i))}, \qquad \frac{\mathsf{nfAcc}(a) \quad \mathsf{nfAcc}(b) \quad \mathsf{nf}(a) = \mathsf{Var}(i)}{\mathsf{nfAcc}(\mathsf{App}(a,b))},$$

$$\frac{\mathsf{nfAcc}(a)}{\mathsf{nfAcc}(\mathsf{Abst}(i,a))}, \qquad \frac{\mathsf{nfAcc}(a) \quad \mathsf{nf}(a) = \mathsf{Abst}(i,a') \quad \mathsf{nfAcc}(\mathsf{sub}(a',i,b))}{\mathsf{nfAcc}(\mathsf{App}(a,b))},$$

$$\frac{\mathsf{nfAcc}(a) \quad \mathsf{nfAcc}(b) \quad \mathsf{nf}(a) = \mathsf{App}(a',a'')}{\mathsf{nfAcc}(\mathsf{App}(a,b))}.$$

To write a correct type-theoretic definition, we must define the inductive predicate nfAcc simultaneously with the function nf, recursively defined on

nfAcc:

$$
\begin{array}{lcl}
\mathsf{nfAcc} & : & (x\colon \Lambda)\mathsf{Set} \\
\mathsf{nf} & : & (x\colon \Lambda; \mathsf{nfAcc}(x))\Lambda
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{nfacc}_1 & : & (i\colon \mathbb{N})\mathsf{nfAcc}(\mathsf{Var}(i)) \\
\mathsf{nfacc}_2 & : & (i\colon \mathbb{N}; a\colon \Lambda; h_a\colon \mathsf{nfAcc}(a))\mathsf{nfAcc}(\mathsf{Abst}(i,a)) \\
\mathsf{nfacc}_3 & : & (a,b\colon \Lambda; h_a\colon \mathsf{nfAcc}(a); h_b\colon \mathsf{nfAcc}(b); i\colon \mathbb{N}; \mathsf{nf}(a,h_a) = \mathsf{Var}(i)) \\
 & & \mathsf{nfAcc}(\mathsf{App}(a,b)) \\
\mathsf{nfacc}_4 & : & (a,b\colon \Lambda; h_a\colon \mathsf{nfAcc}(a); i\colon \mathbb{N}; a'\colon \Lambda; \\
 & & \mathsf{nf}(a,h_a) = \mathsf{Abst}(i,a'); \mathsf{nfAcc}(\mathsf{sub}(a',i,b))) \\
 & & \mathsf{nfAcc}(\mathsf{App}(a,b)) \\
\mathsf{nfacc}_5 & : & (a,b\colon \Lambda; h_a\colon \mathsf{nfAcc}(a); h_b\colon \mathsf{nfAcc}(b); \\
 & & a', a''\colon \Lambda; \mathsf{nf}(a,h_a) = \mathsf{App}(a',a'')) \\
 & & \mathsf{nfAcc}(\mathsf{App}(a,b))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{nf}(\mathsf{Var}(i), \mathsf{nfacc}_1(i)) = \mathsf{Var}(i) \\
\mathsf{nf}(\mathsf{Abst}(i,a), \mathsf{nfacc}_2(i,a,h_a)) = \mathsf{Abst}(i, \mathsf{nf}(a,h_a)) \\
\mathsf{nf}(\mathsf{App}(a,b), \mathsf{nfacc}_3(a,b,h_a,h_b,i,q)) = \mathsf{App}(\mathsf{Var}(i), \mathsf{nf}(b,h_b)) \\
\mathsf{nf}(\mathsf{App}(a,b), \mathsf{nfacc}_4(a,b,h_a,i,a',q,h)) = \mathsf{nf}(\mathsf{sub}(a',i,b),h) \\
\mathsf{nf}(\mathsf{App}(a,b), \mathsf{nfacc}_5(a,b,h_a,h_b,a',a'',q)) = \mathsf{App}(\mathsf{App}(a',a''), \mathsf{nf}(b,h_b)).
\end{array}
$$

6.5 Conclusions and related work

We describe a technique to formalise algorithms in type theory that separates the computational and logical parts of the definition. As a consequence, the resulting type-theoretic algorithms are compact and easy to understand. They are as simple as their Haskell versions, where there is no restriction on the recursive calls. The technique was originally developed by Bove for simple general recursive algorithms. Here, we extend it to nested recursion using Dybjer's schema for simultaneous inductive-recursive definitions. We also show how we can use this technique to formalise partial functions. Notice that the proof of the special predicate for a particular input is a trace of the computation of the original algorithm, therefore its structural complexity is proportional to the number of steps of the algorithm.

We believe that our technique simplifies the task of formal verification. Often, in the process of verifying complex algorithms, the formalisation of the algorithm is so complicated and clouded with logical information, that the formal verification of its properties becomes very difficult. If the algorithm is formalised as we propose, the simplicity of its definition would make the task of formal verification dramatically easier.

Most of the examples we presented have been formally checked using the proof assistant Alf (see [AGNvS94, MN93]), which supports Dybjer's schema.

There are not many studies on formalising general recursion in type theory, as far as we know. In [Nor88], Nordström uses the predicate Acc for that purpose. Balaa and Bertot [BB00] use fix-point equations to obtain the desired

equalities for the recursive definitions, but one still has to mix the actual algorithm with proofs concerning the well-foundedness of the recursive calls. In any case, their methods do not provide simple definitions for nested recursive algorithms. Both Giesl [Gie97], from where we took some of our examples, and Slind [Sli00] have methods to define nested recursive algorithms independently of their proofs of termination. However, neither of them works in the framework of constructive type theory. Giesl works in first order logic and his main concern is to prove termination of nested recursive algorithms automatically. Slind works in classical higher order logic and uses strong inductive principles not available in type theory.

Some work has been done in the area of formalising partial functions. Usually type theory is extended with partial objects or nonterminating computations. This is different from our method, in which partiality is realized by adding a new argument that restricts the domain of the original input; the function is still total in the two arguments. In [Con83], Constable associates a domain to every partial function. This domain is automatically generated from the function definition and contains basically the same information as our special-purpose predicates. However, the definition of the function does not depend on its domain as in our case. Based on this work, Constable and Mendler [CM85] introduce the type of partial functions as a new type constructor. In [CS87], Constable and Smith develop a partial type theory in which every type has twin containing diverging objects. Inspired by the work in [CS87], Audebaud [Aud91] introduces fix-points to the Calculus of Constructions [CH88], obtaining a conservative extension of it where the desired properties still hold.

Chapter 7

General Recursion via Coinductive Types

Not all recursive functions can be represented in type theory as elements of a function type. First of all, partial functions are not representable. Second, not even all total recursive functions can be represented. This problem is related with decidability of type-checking: Given a type-theoretic function $f : \mathbb{N} \to \mathbb{N}$, we can decide whether $(f\ 0) = 0$ by just checking if the term $(\mathsf{refl}\ 0)$ is an element of the type $(f\ 0) = 0$ (see pg. 59). On the other hand, it is well-known that this problem is undecidable for total recursive functions.

This is a problem if we want to use type theory to implement algorithms and prove their properties. One solutions consists in representing functions as relations, in a set-theoretical style. This causes the complete loss of computational information, since relations cannot be executed. Another solution consists in restricting the domain of definition of a function by a predicate characterizing the arguments on which it terminates. The previous chapter gave one version of this method.

Here, we show that it is possible to represent every partial recursive function directly as an element of a function type, without using any predicate to restrict the domain. Coinductive types (see Chapter 3) can be used to formalize general partial recursive functions in type theory. We can define a type $\mathbb{N}^\nu$ such that $\mathbb{N}$ can be embedded in $\mathbb{N}^\nu$ and every general recursive function on natural numbers can be implemented as an element of $\mathbb{N}^\nu \to \mathbb{N}^\nu$. The main idea is that $\mathbb{N}^\nu$ contains possibly diverging elements and that partial functions will produce diverging results on inputs on which they are not defined.

7.1 Coinductive Natural Numbers

We define *coinductive natural numbers* as the following coinductive type.

Definition 7.1.1 *The type of* coinductive natural numbers *is*

$$\mathbb{N}^\nu := \nu_X(\mathsf{Unit} + X + X).$$

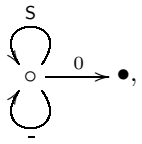
Thus, $\mathbb{N}^\nu$ is the coinductive type associated with the functor $F[X] := \mathsf{Unit} + X + X$. The inductive type associated with F is the type $\mathbb{N}^\mu : \mu_X.F$ with three constructors that we indicate by

$$\frac{}{0 : \mathbb{N}^\mu} \qquad \frac{x : \mathbb{N}^\mu}{(\mathsf{S}\ x) : \mathbb{N}^\mu} \qquad \frac{x : \mathbb{N}^\mu}{(\_\ x) : \mathbb{N}^\mu}.$$

We interpret the constructors 0 and S as usual as the zero constant and the successor function. The third constructor $\_$ is interpreted as an empty function, that is, we consider x and $\_x$ equal. Therefore the term $(\mathsf{S}\ \_\ \_\ \mathsf{S}\ \mathsf{S}\ \_\ 0)$ is just a representation of the natural number 3.

Remark 7.1.2 *The most obvious definition for a type of possibly infinite natural numbers would seem to be* $\mathbb{N}^\omega := \nu_X(\mathsf{Unit} + X)$*, that is, just the coinductive type on the same functor as the inductive type of natural numbers. The type* $\mathbb{N}^\omega$ *contains copies of all the natural numbers plus an infinite natural number* ω *consisting of an infinite sequence of* S *constructors. Unfortunately, this type does not obtain for our purposes: Not all recursive functions can be represented as elements of the function type* $\mathbb{N}^\omega \to \mathbb{N}^\omega$*. In fact, if* $f : \mathbb{N}^\omega \to \mathbb{N}^\omega$*, then we can decide whether* $(f\ 0) = 0$ *by just applying* ν-out *to it once and check if it is in the form* $(\mathsf{inl}\ \bullet)$*. But we already recalled that this is an undecidable problem for recursive functions. In general, in* $\mathbb{N}^\omega$ *it is decidable whether* $(f\ n) = m$ *for any finite numerals* n *and* m*. The only undecidable statements are in the form* $(f\ x) = \omega$*. For our purposes, it is necessary that* $(f\ n) = m$ *is in general undecidable. The type* $\mathbb{N}^\nu$ *satisfies this requirement.*

The coinductive version $\mathbb{N}^\nu$ of $\mathbb{N}^\mu$ can be thought of as the type of possibly infinite terms built up with these three constructors. In other words the elements of $\mathbb{N}^\nu$ are the words of the infinitary language associated with the automaton



that is the language whose words are the possibly infinite sequences of labels of possibly infinite paths starting from $\circ$ and terminating, when they terminate, in $\bullet$.

Let us recall the rules for coinductive types in this case.

elimination

$$\frac{x : \mathbb{N}^\nu}{(\nu\text{-}\mathsf{out}\ x) : \mathsf{Unit} + \mathbb{N}^\nu + \mathbb{N}^\nu}$$

coiteration

$$\frac{T\colon \mathsf{Set} \quad z\colon T \vdash u[z]\colon \mathsf{Unit} + T + T}{t\colon T \vdash (\nu\text{-}\mathsf{it}\ [z]u\ t)\colon \mathbb{N}^\nu}$$

corecursion

$$\frac{T\colon \mathsf{Set} \quad z\colon T \vdash w[z]\colon \mathsf{Unit} + (\mathbb{N}^\nu + T) + (\mathbb{N}^\nu + T)}{t\colon T \vdash (\nu\text{-}\mathsf{rec}\ [z]w\ t)\colon \mathbb{N}^\nu}$$

reduction

$$(\nu\text{-}\mathsf{out}\ (\nu\text{-}\mathsf{it}\ [z]u\ t)) \rightsquigarrow (F[(\nu\text{-}\mathsf{it}\ [z]u)]\ u[t])$$

$$(\nu\text{-}\mathsf{out}\ (\nu\text{-}\mathsf{rec}\ [z]w\ t)) \rightsquigarrow F\left[[y\colon \mathbb{N}^\nu + T]\mathsf{Cases}\ y\ \mathsf{of}\ \left\{\begin{array}{l}(\mathsf{inl}\ c) \mapsto c\\ (\mathsf{inr}\ z) \mapsto (\nu\text{-}\mathsf{rec}\ [z]w\ z)\end{array}\right.\right](w[t])$$

where the functorial extension of F is defined, on a function $f\colon X \to Y$, as the function

$$F[f]\colon \mathsf{Unit} + X + X \to \mathsf{Unit} + Y + Y$$
$$F[f](z) := \mathsf{Cases}\ z\ \mathsf{of}\ \left\{\begin{array}{l}(\mathsf{in}_0\ v) \mapsto (\mathsf{in}_0\ v)\\ (\mathsf{in}_1\ x) \mapsto (\mathsf{in}_1\ (f\ x))\\ (\mathsf{in}_2\ x) \mapsto (\mathsf{in}_2\ (f\ x)).\end{array}\right.$$

Let us give an intuitive explanation of the rules.

First of all, we show how the elements of $\mathbb{N}^\nu$ can be thought as being in one of the three forms 0, $(\mathsf{S}\ n)$, and $(\_\ n)$. The elimination rule states that from an element $m\colon \mathbb{N}^\nu$ we can extract an element $(\nu\text{-}\mathsf{out}\ m)\colon \mathsf{Unit} + \mathbb{N}^\nu + \mathbb{N}^\nu$. This implies that $(\nu\text{-}\mathsf{out}\ m)$ is in one of the three forms $(\mathsf{in}_0\ x)$, $(\mathsf{in}_1\ n)$, and $(\mathsf{in}_2\ n)$. Intuitively, we say that if $(\nu\text{-}\mathsf{out}\ m) = (\mathsf{in}_0\ x)$, m is 0; if $(\nu\text{-}\mathsf{out}\ m) = (\mathsf{in}_1\ n)$, m is $(\mathsf{S}\ n)$; and if $(\nu\text{-}\mathsf{out}\ m) = (\mathsf{in}_2\ n)$, m is $(\_\ n)$.

Suppose we define a function from a type T to $\mathbb{N}^\nu$ by an application of coiteration, $f := (\nu\text{-}\mathsf{it}\ [z]u)\colon T \to \mathbb{N}^\nu$, where $z\colon T \vdash u[z]\colon \mathsf{Unit} + T + T$. Given an object $t\colon T$, we want to compute $(f\ t)$. This is done in the following way: We check $u[t]$, and, according to its form, we define the result.

- If $u[t] = (\mathsf{in}_0\ \bullet)$, then $(f\ t) = 0$;
- If $u[t] = (\mathsf{in}_1\ t')$, then $(f\ t) = \mathsf{S}\ (f\ t')$;
- If $u[t] = (\mathsf{in}_2\ t')$, then $(f\ t) = \_\ (f\ t')$.

The corecursion principle gives more freedom than the coiteration principle: We can give the whole result without recomputing it by coiteration. We define a function by corecursion, $f := (\nu\text{-}\mathsf{rec}\ [z]w)\colon T \to \mathbb{N}^\nu$, with $z\colon T \vdash w[z]\colon \mathsf{Unit} + (\mathbb{N}^\nu + T) + (\mathbb{N}^\nu + T)$. The value of f on an object t can be computed by computing $w[t]$ and, according to its form, defining the result.

- If $w[t] = (\mathsf{in}_0\ \bullet)$, then $(f\ t) = 0$;

- If $w[t] = (\mathsf{in}_1\ q)$, then
  - if $q = (\mathsf{inl}\ n)$ then $(f\ t) = \mathsf{S}\ n$,
  - if $q = (\mathsf{inr}\ t')$ then $(f\ t) = \mathsf{S}\ (f\ t')$;
- If $w[t] = (\mathsf{in}_2\ q)$, then
  - if $q = (\mathsf{inl}\ n)$ then $(f\ t) = \_\ n$,
  - if $q = (\mathsf{inr}\ t')$ then $(f\ t) = \_\ (f\ t')$.

We can formally define the constructors as (we call them constructors because they correspond to the constructor of $\mathbb{N}^\mu$, but they are actually functions defined by corecursion)

$$\begin{array}{l} 0 := (\nu\text{-}\mathsf{it}\ [x\colon \mathsf{Unit}](\mathsf{in}_0\ x)\ \bullet) \\ \mathsf{S} := (\nu\text{-}\mathsf{rec}\ [n\colon \mathbb{N}^\nu](\mathsf{in}_1\ (\mathsf{inl}\ n))) \\ \_ := (\nu\text{-}\mathsf{rec}\ [n\colon \mathbb{N}^\nu](\mathsf{in}_2\ (\mathsf{inl}\ n))). \end{array}$$

These definitions and the explanation of the working of the functions defined by coiteration and corecursion are consistent with the explanation of an element $m\colon \mathbb{N}^\nu$ being zero, a successor, or an empty step, according to the shape of $(\nu\text{-}\mathsf{out}\ m)$. This can be verified using the reduction rules.

Sometimes we give the definition of an object by cases on these constructors. If $n\colon \mathbb{N}^\nu$, $T\colon s$ for some sort s, $e_0\colon T$ with x not occurring free in it, $z\colon \mathbb{N}^\nu \vdash e_1[z]\colon T$, and $z\colon \mathbb{N}^\nu \vdash e_2[z]\colon T$, we write

$$\mathsf{Cases}\ n\ \mathsf{of}\ \begin{cases} 0 & \mapsto e_0 \\ (\mathsf{S}\ n') & \mapsto e_1[n'] \\ (\_\ n') & \mapsto e_2[n'] \end{cases}$$

as a notation for

$$\mathsf{Cases}\ (\nu\text{-}\mathsf{out}\ n)\ \mathsf{of}\ \begin{cases} (\mathsf{in}_0\ x) & \mapsto e_0 \\ (\mathsf{in}_1\ n') & \mapsto e_1[n'] \\ (\mathsf{in}_2\ n') & \mapsto e_2[n']. \end{cases}$$

The standard inductive type of natural numbers $\mathbb{N}$ can be injected in $\mathbb{N}^\nu$. We define the injection function by recursion on $\mathbb{N}$:

$$\begin{array}{l} \ulcorner \_ \urcorner^\nu \colon \mathbb{N} \to \mathbb{N}^\nu \\ \ulcorner 0 \urcorner^\nu := 0 \\ \ulcorner (\mathsf{S}\ n) \urcorner^\nu := (\mathsf{S}\ \ulcorner n \urcorner^\nu). \end{array}$$

7.2 Equality

We need to define an equality on $\mathbb{N}^\nu$ that captures the intuition that its finite elements represent natural numbers and its infinite elements represent diverging computations. We formalize the notions of finite and infinite element in Definition 7.2.5 and Definition 7.2.10, respectively. (See [Jac02] for a characterization

of finite an infinite elements that uses temporal logic.) The basic idea is that we ignore the _ constructor. We can distinguish three kinds of elements of $\mathbb{N}^\nu$. First, we have the finite elements, the ones that end with a 0. Equality must identify finite elements that have the same number of occurrences of S. For example

$$\begin{array}{rcl} \ulcorner 3 \urcorner^\nu & \simeq & \mathsf{S}\ \mathsf{S}\ \mathsf{S}\ 0 \\ & \simeq & \mathsf{S}\ \_\ \_\ \mathsf{S}\ \mathsf{S}\ \_\ 0 \\ & \simeq & \_\ \_\ \_\ \mathsf{S}\ \_\ \mathsf{S}\ \mathsf{S}\ 0. \end{array}$$

There are two ways in which an element can be infinite: by having an infinite number of occurrences of S or by having a finite number of occurrences of S and an infinite number of occurrences of _. We denote by ∞ an element with an infinite number of Ss. It can be defined, for example, as

$$\infty := \mathsf{S}\ \infty;$$

or, formally, as

$$\infty := (\nu\text{-}\mathsf{it}\ [x\colon \mathsf{Unit}](\mathsf{in}_1\ x)\ \bullet).$$

If an element has a finite number n of Ss and an infinite number of _s, we indicate it by $\uparrow_n$:

$$\begin{array}{l} \uparrow_0 := \_\ \uparrow_0 \\ \uparrow_{(\mathsf{S}\ n)} := \mathsf{S}\ \uparrow_n. \end{array}$$

Formally, we can define it either by iteration on $\mathbb{N}$,

$$\begin{array}{l} \uparrow\colon \mathbb{N} \to \mathbb{N}^\nu \\ \uparrow_0 := (\nu\text{-}\mathsf{it}\ [x\colon \mathsf{Unit}](\mathsf{in}_2\ x)\ \bullet) \\ \uparrow_{(\mathsf{S}\ n)} := (\nu\text{-}\mathsf{rec}\ [x\colon \mathbb{N}^\nu](\mathsf{in}_1\ (\mathsf{inl}\ x))\ \uparrow_n), \end{array}$$

or by coiteration on $\mathbb{N}^\nu$,

$$\begin{array}{l} \uparrow\colon \mathbb{N} \to \mathbb{N}^\nu \\ \uparrow := \left(\nu\text{-}\mathsf{it}\ [n\colon \mathbb{N}]\mathsf{Cases}\ n\ \mathsf{of}\ \left\{ \begin{array}{ll} 0 & \mapsto (\mathsf{in}_2\ 0) \\ (\mathsf{S}\ n') & \mapsto (\mathsf{in}_1\ n') \end{array} \right.\right). \end{array}$$

Notice, however, that it is not possible to prove constructively that every element of $\mathbb{N}^\nu$ is in one of these three forms, since it is undecidable whether an object contains a finite number of occurrences of S or _.

We recall the definition of the extensional equality on a coalgebra from Chapter 3, Section 3.3. It is the largest bisimulation relation.

formation

$$\frac{n_1, n_2\colon \mathbb{N}^\nu}{n_1 \approx n_2\colon \mathsf{Prop}}$$

introduction

$$\frac{R\colon \mathbb{N}^\nu \to \mathbb{N}^\nu \to \mathsf{Prop} \quad p\colon (\mathsf{Bisimulation}\ R) \quad n_1, n_2\colon \mathbb{N}^\nu \quad q\colon (R\ n_1\ n_2)}{(\mathsf{bisim}\ R\ p\ n_1\ n_2\ q)\colon n_1 \approx n_2}$$

elimination

$$\frac{n_1, n_2 \colon \mathbb{N}^\nu \quad h \colon n_1 \approx n_2}{(\mathsf{bisim\text{-}elim}\ n_1\ n_2\ h) \colon (\nu\text{-}\mathsf{out}\ n_1) \approx^F (\nu\text{-}\mathsf{out}\ n_2)}$$

reduction

$$\begin{array}{ll} & (\mathsf{bisim\text{-}elim}\ n_1\ n_2\ (\mathsf{bisim}\ R\ p\ n_1\ n_2\ q)) \\ \rightsquigarrow & (\mathsf{relextmon}\quad [x_1, x_2, r](\mathsf{bisim}\ R\ p\ x_1\ x_2\ r) \\ & \qquad\qquad (\nu\text{-}\mathsf{out}\ n_1)\ (\nu\text{-}\mathsf{out}\ n_2)\ (p\ n_1\ n_2\ q)). \end{array}$$

We explain the notion of bisimulation in our specific case. A relation R on $\mathbb{N}^\nu$ is a bisimulation if, whenever $(R\ n_1\ n_2)$ holds, one of the following three cases holds:

- $n_1 = n_2 = 0$;
- $n_1 = \mathsf{S}\ n_1' \wedge n_2 = \mathsf{S}\ n_2' \wedge (R\ n_1'\ n_2')$;
- $n_1 = \_\ n_1' \wedge n_2 = \_\ n_2' \wedge (R\ n_1'\ n_2')$.

For our purposes, the notion of bisimulation is too restrictive. We want two terms that differ only by the number of occurrences of $\_$ to be equal. In other words, we want $\_$ to be ignored by the equality. Therefore, we use the weaker equality $\overset{\mathsf{bc}}{=}$. This equality is similar to the τ-bisimulation notion in Process Algebra (see [BW90], pg. 162, Definition 5.8.1). Intuitively $n_1 \overset{\mathsf{bc}}{=} n_2$ holds when one of the following conditions is verified:

- $n_1 = n_2 = 0$;
- $n_1 = \mathsf{S}\ n_1' \wedge n_2 = \mathsf{S}\ n_2' \wedge n_1' \overset{\mathsf{bc}}{=} n_2'$;
- $n_1 = \_\ n_1' \wedge n_1' \overset{\mathsf{bc}}{=} n_2$;
- $n_2 = \_\ n_2' \wedge n_1 \overset{\mathsf{bc}}{=} n_2'$

We define the equality $\overset{\mathsf{bc}}{=}$ exactly as $\approx$, with the only difference that the concept of bisimulation is substituted by a weaker notion of *bicounting* relation.

Definition 7.2.1 *Let $\langle B, g \colon \mathsf{Unit}{+}B{+}B\rangle$ an F-coalgebra. We say that $R \colon B \to B \to \mathsf{Prop}$ is a* bicounting relation *if it satisfies the property*

$$\begin{array}{l} (\mathsf{Bicounting}\ R) := \\ \forall b_1, b_2 \colon B.(R\ b_1\ b_2) \to \\ \mathsf{Cases}\ (g\ b_1)\ \mathsf{of} \left\{ \begin{array}{l} (\mathsf{in}_0\ x) \mapsto \mathsf{Cases}\ (g\ b_2)\ \mathsf{of} \left\{ \begin{array}{l} (\mathsf{in}_0\ y) \mapsto \top \\ (\mathsf{in}_1\ b_2') \mapsto \bot \\ (\mathsf{in}_2\ b_2') \mapsto (R\ b_1\ b_2') \end{array} \right. \\ (\mathsf{in}_1\ b_1') \mapsto \mathsf{Cases}\ (g\ b_2)\ \mathsf{of} \left\{ \begin{array}{l} (\mathsf{in}_0\ y) \mapsto \bot \\ (\mathsf{in}_1\ b_2') \mapsto (R\ b_1'\ b_2') \\ (\mathsf{in}_2\ b_2') \mapsto (R\ b_1\ b_2') \end{array} \right. \\ (\mathsf{in}_2\ b_1') \mapsto (R\ b_1'\ b_2) \end{array} \right. \end{array}$$

Definition 7.2.2 *Two elements $n_1, n_2 \colon \mathbb{N}^\nu$ are said to be* bicounting equal, *$n_1 \overset{\mathsf{bc}}{=} n_2$, when there exists a bicounting relation R such that $(R\ n_1\ n_2)$. Formally $\overset{\mathsf{bc}}{=}$ is the relation defined by the rules*

formation

$$\frac{n_1, n_2 \colon \mathbb{N}^\nu}{n_1 \overset{\mathsf{bc}}{=} n_2 \colon \mathsf{Prop}}$$

introduction

$$\frac{R \colon \mathbb{N}^\nu \to \mathbb{N}^\nu \to \mathsf{Prop} \quad p \colon (\mathsf{Bicounting}\ R) \quad n_1, n_2 \colon \mathbb{N}^\nu \quad q \colon (R\ n_1\ n_2)}{(\mathsf{bicount}\ R\ p\ n_1\ n_2\ q) \colon n_1 \overset{\mathsf{bc}}{=} n_2}$$

elimination

$$\frac{n_1, n_2 \colon \mathbb{N}^\nu \quad h \colon n_1 \overset{\mathsf{bc}}{=} n_2}{(\mathsf{bicount\text{-}elim}\ n_1\ n_2\ h) \colon (\nu\text{-}\mathsf{out}\ n_1) \overset{\mathsf{bc}^F}{=} (\nu\text{-}\mathsf{out}\ n_2)}$$

reduction

$$\begin{array}{l}(\mathsf{bicount\text{-}elim}\ n_1\ n_2\ (\mathsf{bicount}\ R\ p\ n_1\ n_2\ q)) \\ \rightsquigarrow \quad (\mathsf{relextmon} \quad [x_1, x_2, r](\mathsf{bicount}\ R\ p\ x_1\ x_2\ r) \\ \qquad\qquad\qquad\quad (\nu\text{-}\mathsf{out}\ n_1)\ (\nu\text{-}\mathsf{out}\ n_2)\ (p\ n_1\ n_2\ q)).\end{array}$$

Unfortunately, this equality is too strong, because it identifies every finite element with every infinite element that has at most the same number of occurrences of S. Intuitively, a bicounting relation can always identify 0 and $\uparrow_0$. More precisely, if R is a bicounting relation, then

$$(Q\ n_1\ n_2) := (R\ n_1\ n_2) \vee (n_1 = 0 \wedge n_2 = \uparrow_0) \vee (n_1 = \uparrow_0 \wedge n_2 = 0)$$

is also a bicounting relation. From this follows that $0 \overset{\mathsf{bc}}{=} \uparrow_0$. In general, we have the following paradoxical result.

Lemma 7.2.3 *For every $m_1, m_2 \colon \mathbb{N}$ with $m_2 \leq m_1$, $\ulcorner m_1 \urcorner^\nu \overset{\mathsf{bc}}{=} \uparrow_{m_2}$.*

Proof We prove it using the relation

$$(R\ n_1\ n_2) := \exists m_1, m_2 \colon \mathbb{N}. m_2 \leq m_1 \wedge n_1 = \ulcorner m_1 \urcorner^\nu \wedge n_2 = \uparrow_{m_2}.$$

We prove that it is a bicounting relation. We assume that $(R\ n_1\ n_2)$ holds, we have to prove the corresponding conclusion, depending on the form of $(\nu\text{-}\mathsf{out}\ n_1)$ and $(\nu\text{-}\mathsf{out}\ n_2)$:

- If $(\nu\text{-}\mathsf{out}\ n_1) = (\mathsf{in}_0\ x)$, then $m_1 = 0$ and:
  - The case $(\nu\text{-}\mathsf{out}\ n_2) = (\mathsf{in}_0\ y)$ does not occur, because $n_2 = \uparrow_{m_2}$ is an infinite term, and cannot be 0;

  - Also the case $(\nu\text{-}\mathsf{out}\ n_2) = (\mathsf{in}_1\ n_2')$ does not occur, because it would mean that $m_2 = (\mathsf{S}\ m_2')$ for some $m_2' \colon \mathbb{N}$ and the relation $m_2 \leq m_1$ would become $(\mathsf{S}\ m_2') \leq 0$, which is false;
  - If $(\nu\text{-}\mathsf{out}\ n_2) = (\mathsf{in}_2\ n_2')$, then it must be $n_2 = \uparrow_0$ (otherwise it would start with an S) and also $n_2' = \uparrow_0$; we have to prove $(R\ n_1\ n_2')$; in fact $n_1 = \ulcorner 0 \urcorner^\nu$, $n_2' = \uparrow_0$ and $0 \leq 0$;

- If $(\nu\text{-}\mathsf{out}\ n_1) = (\mathsf{in}_1\ n_1')$, then $m_1 = (\mathsf{S}\ m_1')$ for some $m_1' \colon \mathbb{N}$ with $n_1' = \ulcorner m_1' \urcorner^\nu$, and:
  - The case $(\nu\text{-}\mathsf{out}\ n_2) = (\mathsf{in}_0\ y)$ does not occur, because $n_2 = \uparrow_{m_2}$ cannot be zero;
  - If $(\nu\text{-}\mathsf{out}\ n_2) = (\mathsf{in}_1\ n_2')$, then $m_2 = (\mathsf{S}\ m_2')$ for some $m_2' \colon \mathbb{N}$ with $n_2' = \ulcorner m_2' \urcorner^\nu$; we have to prove $(R\ n_1'\ n_2')$; in fact $n_1' = \ulcorner m_1' \urcorner^\nu$, $n_2' = \uparrow_{m_2'}$ and $m_2' \leq m_1'$ follows from $(\mathsf{S}\ m_1') = m_1 \leq m_2 = (\mathsf{S}\ m_2')$;
  - If $(\nu\text{-}\mathsf{out}\ n_2) = (\mathsf{in}_2\ n_2')$, then it must be $n_2 = \uparrow_0$ (otherwise it would start with an S) and also $n_2' = \uparrow_0$; we have to prove $(R\ n_1\ n_2')$; in fact $n_1 = \ulcorner m_1 \urcorner^\nu$, $n_2' = \uparrow_0$ and $0 \leq m_1$;

- The case $(\nu\text{-}\mathsf{out}\ n_1) = (\mathsf{in}_2\ n_1')$ does not occur, because the constructor $\_$ does not occur in $n_1 = \ulcorner m_1 \urcorner^\nu$.

We proved that R is a bicounting relation. Let $m_1, m_2 \colon \mathbb{N}$ be such that $m_2 \leq m_1$. Then $(R\ \ulcorner m_1 \urcorner^\nu\ \uparrow_{m_2})$ holds. Therefore, $\ulcorner m_1 \urcorner^\nu \overset{\mathsf{bc}}{=} \uparrow_{m_2}$. □

Even worse, the relation is not even an equivalence relation: Transitivity fails.

Lemma 7.2.4 *For every $n, m \colon \mathbb{N}$ with $n \neq m$, we have that $\ulcorner n \urcorner^\nu \overset{\mathsf{bc}}{=} \uparrow_0$ and $\ulcorner m \urcorner^\nu \overset{\mathsf{bc}}{=} \uparrow_0$, but $\neg(\ulcorner n \urcorner^\nu \overset{\mathsf{bc}}{=} \ulcorner m \urcorner^\nu)$.*

Proof Lemma 7.2.3 shows that $\ulcorner n \urcorner^\nu \overset{\mathsf{bc}}{=} \uparrow_0$ and $\ulcorner m \urcorner^\nu \overset{\mathsf{bc}}{=} \uparrow_0$.

To show that $\neg(\ulcorner n \urcorner^\nu \overset{\mathsf{bc}}{=} \ulcorner m \urcorner^\nu)$, we have to prove that no bicounting relation holds between $\ulcorner n \urcorner^\nu$ and $\ulcorner m \urcorner^\nu$. Let R be a bicounting relation. We show that $(R\ \ulcorner n \urcorner^\nu\ \ulcorner m \urcorner^\nu) \to \bot$ by induction on n.

- If $n = 0$, we must prove that $(R\ \ulcorner 0 \urcorner^\nu\ \ulcorner m \urcorner^\nu) \to \bot$. The number m cannot be 0 because n and m are different. Therefore, $m = (\mathsf{S}\ m')$ for some natural number m'. From the definition of bicounting relation we deduce that
$$(R\ \ulcorner 0 \urcorner^\nu\ \ulcorner (\mathsf{S}\ m') \urcorner^\nu) \to \bot;$$

- If $n = (\mathsf{S}\ n')$ for some natural number n', we assume the induction hypothesis $(R\ \ulcorner n' \urcorner^\nu\ \ulcorner m' \urcorner^\nu) \to \bot$ for every natural number m' such that $n' \neq m'$, and we must prove that $(R\ \ulcorner (\mathsf{S}\ n') \urcorner^\nu\ \ulcorner m \urcorner^\nu) \to \bot$. We proceed by cases on m:

– If $m = 0$, then the definition of bicounting relation states that

$$(R\ \ulcorner(\mathsf{S}\ n')^{\urcorner\nu}\ \ulcorner 0^{\urcorner\nu}) \to \bot;$$

– If $m = (\mathsf{S}\ m')$, then the definition of bicounting relation states that

$$(R\ \ulcorner(\mathsf{S}\ n')^{\urcorner\nu}\ \ulcorner(\mathsf{S}\ m')^{\urcorner\nu}) \to (R\ \ulcorner n'^{\urcorner\nu}\ \ulcorner m'^{\urcorner\nu})$$

and, by induction hypothesis, $(R\ \ulcorner n'^{\urcorner\nu}\ \ulcorner m'^{\urcorner\nu}) \to \bot$, from which follows that

$$(R\ \ulcorner(\mathsf{S}\ n')^{\urcorner\nu}\ \ulcorner(\mathsf{S}\ m')^{\urcorner\nu}) \to \bot.$$

□

Infinite elements are the cause of the problem, because they automatically satisfy the relation with an infinite empty proof. If we make a distinction between the finite and infinite terms, everything falls into place. First of all, we define the finiteness predicate. This is an inductive predicate on the coinductive type.

Definition 7.2.5 *An element of* $\mathbb{N}^\nu$ *is said to be* finite *if it satisfies the following inductive predicate.*

$$\begin{array}{l}\mathsf{Inductive\ Finite}\ :\ \mathbb{N}^\nu \to \mathsf{Prop} :=\\ \quad \mathsf{fin}_0 \colon (\mathsf{Finite}\ 0)\\ \quad \mathsf{fin}_\mathsf{S} \colon (n \colon \mathbb{N}^\nu)(\mathsf{Finite}\ n) \to (\mathsf{Finite}\ (\mathsf{S}\ n))\\ \quad \mathsf{fin}_\_ \colon (n \colon \mathbb{N}^\nu)(\mathsf{Finite}\ n) \to (\mathsf{Finite}\ (\_\ n))\\ \mathsf{end.}\end{array}$$

Formally

$$\begin{array}{l}\mathsf{Inductive\ Finite}\ :\ \mathbb{N}^\nu \to \mathsf{Prop} :=\\ \quad \mathsf{fin}_0 \colon (n \colon \mathbb{N}^\nu)(\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_0\ \bullet) \to (\mathsf{Finite}\ n)\\ \quad \mathsf{fin}_\mathsf{S} \colon (n, n' \colon \mathbb{N}^\nu)(\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_1\ n') \to (\mathsf{Finite}\ n') \to (\mathsf{Finite}\ n)\\ \quad \mathsf{fin}_\_ \colon (n, n' \colon \mathbb{N}^\nu)(\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_2\ n') \to (\mathsf{Finite}\ n') \to (\mathsf{Finite}\ n)\\ \mathsf{end.}\end{array}$$

Then we can define the equality

Definition 7.2.6 *Two elements* $n_1, n_2 \colon \mathbb{N}^\nu$ *are said to be equal if they satisfy the relation*

$$n_1 \simeq n_2 := ((\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_2)) \wedge n_1 \overset{\mathsf{bc}}{=} n_2.$$

This is the standard equality that we are going to use.

There exists a *finite* characterization of $\simeq$ by defining an inductive version of bicounting.

$$
\begin{array}{l}
\mathsf{Inductive}\ (\overset{\mathsf{f}}{=}) : \mathbb{N}^\nu \to \mathbb{N}^\nu \to \mathsf{Prop} := \\
\quad \mathsf{fin\text{-}eq}_0 : 0 \overset{\mathsf{f}}{=} 0 \\
\quad \mathsf{fin\text{-}eq}_\mathsf{S} : (n_1, n_2 : \mathbb{N}^\nu) n_1 \overset{\mathsf{f}}{=} n_2 \to (\mathsf{S}\ n_1) \overset{\mathsf{f}}{=} (\mathsf{S}\ n_2) \\
\quad \mathsf{fin\text{-}eq}_{\_\mathsf{l}} : (n_1, n_2 : \mathbb{N}^\nu) n_1 \overset{\mathsf{f}}{=} n_2 \to (\_\ n_1) \overset{\mathsf{f}}{=} n_2 \\
\quad \mathsf{fin\text{-}eq}_{\_\mathsf{r}} : (n_1, n_2 : \mathbb{N}^\nu) n_1 \overset{\mathsf{f}}{=} n_2 \to n_1 \overset{\mathsf{f}}{=} (\_\ n_2) \\
\mathsf{end}.
\end{array}
$$

(It is clear how to obtain the formal version, so we will not repeat the definition.) Since this is an inductive relation, only a finite number of applications of constructors $\mathsf{fin\text{-}eq}_\mathsf{l}$ and $\mathsf{fin\text{-}eq}_\mathsf{r}$ can occur in a proof. Therefore, only finite elements can be proved equal. This means that $\overset{\mathsf{f}}{=}$ never holds for infinite elements. Therefore, $\overset{\mathsf{f}}{=}$ is not reflexive; for example, $\infty \overset{\mathsf{f}}{=} \infty$ is not provable. To transform $\overset{\mathsf{f}}{=}$ into an equivalence relation, we need to deal separately with infinite elements.

Definition 7.2.7 *We define the* finite equality *on* $\mathbb{N}^\nu$ *as the relation* $\doteq$*:*

$$
\begin{array}{rcl}
(n_1 \doteq n_2) & := & ((\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_2)) \wedge \\
& & ((\mathsf{Finite}\ n_1) \to (\mathsf{Finite}\ n_2) \to n_1 \overset{\mathsf{f}}{=} n_2).
\end{array}
$$

Proposition 7.2.8 $\forall n_1, n_2 : \mathbb{N}^\nu . n_1 \simeq n_2 \leftrightarrow n_1 \doteq n_2$.

Proof First we prove that $n_1 \simeq n_2 \to n_1 \doteq n_2$. Assume $n_1 \simeq n_2$. We must prove $n_1 \doteq n_2$, that is,

$$((\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_2)) \wedge ((\mathsf{Finite}\ n_1) \to (\mathsf{Finite}\ n_2) \to n_1 \overset{\mathsf{f}}{=} n_2).$$

The first statement, $(\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_2)$, is immediate from the definition of $n_1 \doteq n_2$. Now, assume that $h_1 : (\mathsf{Finite}\ n_1)$ and $h_2 : (\mathsf{Finite}\ n_2)$ hold. We must prove $n_1 \overset{\mathsf{f}}{=} n_2$. By hypothesis $n_1 \simeq n_2$, so $n_1 \overset{\mathsf{bc}}{=} n_2$. This means that there is a bicounting relation R such that $(R\ n_1\ n_2)$ holds. We proceed by double induction on the proofs $h_1 : (\mathsf{Finite}\ n_1)$ and $h_2 : (\mathsf{Finite}\ n_2)$:

- If $h_1 = \mathsf{fin}_0$, then $n_1 = 0$. Next, we proceed by cases on the proof $h_2 : (\mathsf{Finite}\ n_2)$:
  - If $h_2 = \mathsf{fin}_0$, then $n_2 = 0$ and we prove $0 \overset{\mathsf{f}}{=} 0$ by $\mathsf{fin\text{-}eq}_0$;
  - The case $h_2 = (\mathsf{fin}_\mathsf{S}\ n_2'\ h_2')$ does not occur, since it would imply that $n_2 = (\mathsf{S}\ n_2')$; but in that case, the definition of bicounting would give $(R\ n_1\ n_2) \to \bot$, contradicting the hypothesis $(R\ n_1\ n_2)$;

  - If $h_2 = (\mathsf{fin}\_\ n_2'\ h_2')$, then $n_2 = (\_\ n_2')$; by definition of bicounting we have $(R\ n_1\ n_2) \to (R\ n_1\ n_2')$, that, together with the hypothesis, gives $(R\ n_1\ n_2')$; since $h_2' \colon (\mathsf{Finite}\ n_2')$ is structurally simpler that h_2, by induction hypothesis we can assume that $n_1 \stackrel{\mathsf{f}}{=} n_2'$; then $n_1 \stackrel{\mathsf{f}}{=} n_2$ holds by $\mathsf{fin\text{-}eq}\_\mathsf{r}$.

- If $h_1 = (\mathsf{fin}\mathsf{s}\ n_1'\ h_1')$, then $n_1 = (\mathsf{S}\ n_1')$. We proceed by cases on the proof $h_2 \colon (\mathsf{Finite}\ n_2)$:

  - The case $h_2 = \mathsf{fin}_0$ does not occur, since it would imply that $n_2 = 0$; but in that case, the definition of bicounting would give $(R\ n_1\ n_2) \to \bot$, contradicting the hypothesis $(R\ n_1\ n_2)$;
  - If $h_2 = (\mathsf{fin}\_\ n_2'\ h_2')$, then $n_2 = (\_\ n_2')$; by definition of bicounting we have $(R\ n_1\ n_2) \to (R\ n_1'\ n_2')$, that, together with the hypothesis, gives $(R\ n_1'\ n_2')$; since $h_1' \colon (\mathsf{Finite}\ n_1')$ is structurally simpler that h_1, by induction hypothesis we can assume that $n_1' \stackrel{\mathsf{f}}{=} n_2'$; then $n_1 \stackrel{\mathsf{f}}{=} n_2$ holds by $\mathsf{fin\text{-}eq}_\mathsf{S}$;
  - If $h_2 = (\mathsf{fin}\_\ n_2'\ h_2')$, then $n_2 = (\_\ n_2')$; by definition of bicounting we have $(R\ n_1\ n_2) \to (R\ n_1\ n_2')$, that, together with the hypothesis, gives $(R\ n_1\ n_2')$; since $h_2' \colon (\mathsf{Finite}\ n_2')$ is structurally simpler that h_2, by induction hypothesis we can assume that $n_1 \stackrel{\mathsf{f}}{=} n_2'$; then $n_1 \stackrel{\mathsf{f}}{=} n_2$ holds by $\mathsf{fin\text{-}eq}\_\mathsf{r}$.

- If $h_1 = (\mathsf{fin}\_\ n_1'\ h_1')$, then $n_1 = (\_\ n_1')$; by definition of bicounting we have $(R\ n_1\ n_2) \to (R\ n_1'\ n_2)$, that, together with the hypothesis, gives $(R\ n_1'\ n_2)$; since $h_1' \colon (\mathsf{Finite}\ n_1')$ is structurally simpler that h_1, by induction hypothesis we can assume that $n_1' \stackrel{\mathsf{f}}{=} n_2$; then $n_1 \stackrel{\mathsf{f}}{=} n_2$ holds by $\mathsf{fin\text{-}eq}\_\mathsf{r}$.

Vice versa, we have to prove that $n_1 \doteq n_2 \to n_1 \simeq n_2$. The conjunct $(\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_2)$ is immediate.

We still have to prove that $n_1 \doteq n_2 \to n_1 \stackrel{\mathsf{bc}}{=} n_2$. We do it by showing that $\doteq$ is a bicounting relation. Assume that $n_1 \doteq n_2$ holds. We have to prove the consequence required by the definition of bicounting relation. According to the shape of n_1 and n_2 we have:

- If $(\nu\text{-}\mathsf{out}\ n_1) = (\mathsf{in}_0\ x)$, that is, $n_1 = 0$, n_1 is finite and so must be n_2 by the first conjunct of the hypothesis; then, by the hypothesis, it must be $n_1 \stackrel{\mathsf{f}}{=} n_2$ and, since $n_1 = 0$, this can happen only if $n_2 = 0$; the definition of bicounting gives then $0 \doteq 0 \to \top$, so there is nothing to prove.

- If $(\nu\text{-}\mathsf{out}\ n_1) = (\mathsf{in}_1\ n_1')$, that is, $n_1 = (\mathsf{S}\ n_1')$ we have:

  - The case $(\nu\text{-}\mathsf{out}\ n_2) = (\mathsf{in}_0\ x)$, that is, $n_2 = 0$, cannot occur, because it would imply that also n_1 is finite and $n_1 \stackrel{\mathsf{f}}{=} 0$, which is impossible because n_1 is a successor;

  - If $(\nu\text{-}\mathsf{out}\ n_2) = (\mathsf{in}_1\ n_2')$, that is, $n_2 = (\mathsf{S}\ n_2')$; by the definition of bicounting it must be $n_1 \doteq n_2 \to n_1' \doteq n_2'$, therefore we must prove $n_1' \doteq n_2'$; we have that $(\mathsf{Finite}\ n_2) \leftrightarrow (\mathsf{Finite} n_2')$ and $(\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite} n_1')$; by hypothesis $(\mathsf{Finite}\, n_1) \leftrightarrow (\mathsf{Finite}\, n_2)$, and so $(\mathsf{Finite}\, n_1') \leftrightarrow (\mathsf{Finite}\ n_2')$; assume $(\mathsf{Finite}\ n_1')$ and $(\mathsf{Finite}\ n_2')$, then $(\mathsf{Finite}\ n_1)$ and $(\mathsf{Finite} n_2)$ hold as well; by the hypothesis $n_1 \overset{\mathsf{f}}{=} n_2$ holds; the proof of this fact must be in the form $(\mathsf{fin\text{-}eq}_{\mathsf{S}}\ n_1'\ n_2'\ h)$, and so we have a proof $h\colon n_1' \overset{\mathsf{f}}{=} n_2'$;
  - If $(\nu\text{-}\mathsf{out}\ n_2) = (\mathsf{in}_2\ n_2')$, that is, $n_2 = (\_\ n_2')$; by the definition of bicounting it must be $n_1 \doteq n_2 \to n_1 \doteq n_2'$, therefore we must prove $n_1 \doteq n_2'$; we have that $(\mathsf{Finite}\ n_2) \leftrightarrow (\mathsf{Finite} n_2')$; by hypothesis $(\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_2)$, and so $(\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_2')$; assume $(\mathsf{Finite}\ n_1)$ and $(\mathsf{Finite}\ n_2')$, then $(\mathsf{Finite} n_2)$ hold as well; by the hypothesis $n_1 \overset{\mathsf{f}}{=} n_2$ holds; the proof of this fact must be in the form $(\mathsf{fin\text{-}eq}_{\mathsf{\_r}}\ n_1\ n_2'\ h)$, and so we have a proof $h\colon n_1 \overset{\mathsf{f}}{=} n_2'$;

- If $(\nu\text{-}\mathsf{out}\ n_1) = (\mathsf{in}_2\ n_1')$, that is, $n_1 = (\_\ n_1')$; by the definition of bicounting it must be $n_1 \doteq n_2 \to n_1' \doteq n_2$, therefore we must prove $n_1' \doteq n_2$; we have that $(\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_1')$; by hypothesis $(\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_2)$, and so $(\mathsf{Finite}\ n_1') \leftrightarrow (\mathsf{Finite}\ n_2)$; assume $(\mathsf{Finite}\ n_1')$ and $(\mathsf{Finite}\ n_2)$, then $(\mathsf{Finite} n_1)$ hold as well; by the hypothesis $n_1 \overset{\mathsf{f}}{=} n_2$ holds; from this it is easy to show that $n_1' \overset{\mathsf{f}}{=} n_2$ must also hold. (Here, we cannot assume that the proof uses $\mathsf{fin\text{-}eq}_{\mathsf{\_l}}$, because $\mathsf{fin\text{-}eq}_{\mathsf{\_r}}$ could also have been used; but it is easy to prove that $(\_\ n_1') \overset{\mathsf{f}}{=} n_2 \to n_1' \overset{\mathsf{f}}{=} n_2$.)

Therefore, $\doteq$ is a bicounting relation and so it is contained in $\overset{\mathsf{bc}}{=}$. □

The property of being an infinite element can be captured by the negation of the predicate Finite or, directly, by a coinductive predicate. We define a coinductive predicate Infinite that holds for infinite elements. It is defined as the largest *infinity predicate*.

Definition 7.2.9 *Let $\langle B, g\colon \mathsf{Unit} + B + B\rangle$ be a an F-coalgebra. We say that $P\colon B \to \mathsf{Prop}$ is an* infinity predicate *if it satisfies the property*

$$(\mathsf{Infinity}\ P) := \forall n\colon \mathbb{N}^{\nu}.(P\ n) \to \mathsf{Cases}\ (\nu\text{-}\mathsf{out}\ n)\ \mathsf{of}\ \begin{cases} (\mathsf{in}_0\ x) & \mapsto \bot \\ (\mathsf{in}_1\ n') & \mapsto (P\ n') \\ (\mathsf{in}_2\ n') & \mapsto (P\ n'). \end{cases}$$

The desired predicate Infinite is the union of all infinity predicates.

Definition 7.2.10 *An element $n\colon \mathbb{N}^{\nu}$ is said to be* infinite *if there is an infinity predicate P such that $(P\ n)$ holds. In that case we write* $(\mathsf{Infinite}\ n)$.

Lemma 7.2.11 $\forall n\colon \mathbb{N}^{\nu}.(\mathsf{Infinite}\ n) \leftrightarrow \neg(\mathsf{Finite}\ n)$.

Proof To prove that $\neg(\mathsf{Finite}\ n) \to (\mathsf{Infinite}\ n)$, we show that $\lambda n\colon \mathbb{N}^\nu.\neg(\mathsf{Finite}\ n)$ is an infinity predicate. The verification of this fact is immediate.

To show the converse, $(\mathsf{Infinite}\ n) \to \neg(\mathsf{Finite}\ n)$, we show that $(\mathsf{Finite}\ n) \to (\mathsf{Infinite}\ n) \to \bot$ by induction on the proof of $(\mathsf{Finite}\ n)$. This is also immediate. □

Remark 7.2.12 *We are making an effort of defining an equality that is an equivalence relation. In that way,* $\mathbb{N}^\nu$ *together with this equality is a total setoid. In this case it might be more elegant to use a partial setoid. Notice that* $\overset{\mathsf{f}}{=}$ *is a partial equivalence relation. If we use the partial setoid defined by* $\mathbb{N}^\nu$ *and* $\overset{\mathsf{f}}{=}$*, then its elements are exactly the finite elements of* $\mathbb{N}^\nu$*. We should then explicitly introduce the setoid of partial functions between two partial setoids. The condition of preservation of equality must then be weakened. A total function* f *between partial setoids is such that if* $x = x$*, then* $(f\ x) = (f\ x)$*. For partial functions this is not required,* f *may be undefined on defined elements. But we still need some notion of preservation of equality. A sufficient weaker condition is that if* $x = y$ *and* $(f\ x) = (f\ x)$*, then* $(f\ x) = (f\ y)$*. If we want to consider only strict functions, we need to add the condition that if* $(f\ x) = (f\ x)$*, then* $x = x$*.*

We define a last equivalence relation on $\mathbb{N}^\nu$, that is finer that the ones considered so far. It coincides with $\overset{\mathsf{f}}{=}$ on finite elements. On infinite elements, it distinguishes them according to the number of Ss, so ∞ is not equal to any of the $\uparrow_n$s and $\uparrow_n$ is equal to $\uparrow_m$ if and only if $n = m$. Once again, it is a coinductive relation defined as the union of all relations having a certain property, that we call *countsimulation*.

To define countsimulation relations, we need two predicates and a function. The first predicate states that an object is zero. It is an inductive predicate.

$$\begin{array}{l}\mathsf{Inductive\ Is\text{-}zero} : \mathbb{N}^\nu \to \mathsf{Prop} :=\\ \quad \mathsf{iszero}_0 \colon (\mathsf{Is\text{-}zero}\ 0)\\ \quad \mathsf{iszero}\_ \colon (n\colon \mathbb{N}^\nu)(\mathsf{Is\text{-}zero}\ n) \to (\mathsf{Is\text{-}zero}\ (\_\ n))\\ \mathsf{end.}\end{array}$$

The second predicate states that an object is a successor. It is also an inductive predicate.

$$\begin{array}{l}\mathsf{Inductive\ Is\text{-}succ} : \mathbb{N}^\nu \to \mathsf{Prop} :=\\ \quad \mathsf{issucc}_\mathsf{S} \colon (n\colon \mathbb{N}^\nu)(\mathsf{Is\text{-}succ}\ (\mathsf{S}\ n))\\ \quad \mathsf{issucc}\_ \colon (n\colon \mathbb{N}^\nu)(\mathsf{Is\text{-}succ}\ n) \to (\mathsf{Is\text{-}succ}\ (\_\ n))\\ \mathsf{end.}\end{array}$$

Note that $\mathsf{Is\text{-}zero}$ and $\mathsf{Is\text{-}succ}$ are not exactly the negation of each other, because the object $\uparrow_0$ does not satisfy either of them.

The function is the predecessor function. On 0 it gives 0, on $\uparrow_0$ it gives $\uparrow_0$.

$$\begin{array}{l}\mathsf{pred}\colon \mathbb{N}^\nu \to \mathbb{N}^\nu\\ \mathsf{pred}(0) := 0\\ \mathsf{pred}((\mathsf{S}\ n)) := n\\ \mathsf{pred}((\_\ n)) := (\_\ \mathsf{pred}(n))\end{array}$$

Definition 7.2.13 *Let $\langle B, g\colon \mathsf{Unit} + B + B\rangle$ be an F-coalgebra. We say that $R\colon B \to B \to \mathsf{Prop}$ is a* countsimulation relation *if it satisfies the property*

$$\begin{array}{l}(\mathsf{Countsimulation}\ R) :=\\ \forall b_1, b_2\colon B.(R\ b_1\ b_2) \to\\ \mathsf{Cases}\ (g\ b_1)\ \mathsf{of}\ \left\{\begin{array}{l}(\mathsf{in}_0\ x) \mapsto (\mathsf{Is\text{-}zero}\ b_2)\\ (\mathsf{in}_1\ b_1') \mapsto (\mathsf{Is\text{-}succ}\ b_2) \wedge (R\ b_1'\ (\mathsf{pred}\ b_2))\\ (\mathsf{in}_2\ b_1') \mapsto (R\ b_2\ b_1')\end{array}\right.\end{array}$$

Definition 7.2.14 *Two elements $n_1, n_2\colon \mathbb{N}^\nu$ are said to be* countsimulation equal*, $n_1 \overset{\infty}{=} n_2$, when there exists a countsimulation relation R such that $(R\ n_1\ n_2)$ holds.*

Countsimulation equality is finer that bicounting equality.

Lemma 7.2.15 $\forall n_1, n_2\colon \mathbb{N}^\nu. n_1 \overset{\infty}{=} n_2 \to n_1 \simeq n_2$.

Proof It is easy to prove that $n_1 \overset{\infty}{=} n_2 \to ((\mathsf{Finite}\ n_1) \leftrightarrow (\mathsf{Finite}\ n_2))$ (each direction by induction on the proof of the hypothesis).

We still have to prove that $n_1 \overset{\infty}{=} n_2 \to n_1 \overset{\mathsf{bc}}{=} n_2$. It is easy to prove that every countsimulation relation is a bicounting relation. Since $\overset{\infty}{=}$ is the union of all countsimulation relations, it is contained in the union of all bicounting relations, $\overset{\mathsf{bc}}{=}$.

Alternatively, we can just prove that $\overset{\infty}{=}$ is a bicounting relation. □

The reason we formulate this equality is that it may be useful in reasoning about lazy functions. A *lazy function* is a function that does not need to read the whole input before it gives an output. That means that it may give a finite result on an infinite object. Let us, for example, consider the function that tests whether an element of $\mathbb{N}^\nu$ is larger or equal to 3, $f\colon \mathbb{N}^\nu \to \mathbb{N}^\nu$ such that $(f\ n) = 0$ if and only if $n \geq 3$. What result should it give on $\uparrow_3$? After reading the first three symbols of $\uparrow_3$, which are three occurrences of S, it is certain that this number must be larger or equal to 3 and therefore we can ignore the rest of it and return 0. Unfortunately, with the equality $\simeq$, $\uparrow_3$ is equal to $\uparrow_0$ and we cannot put $(f\ \uparrow_0) = 0$. So f would not preserve equality.

On the other hand, using $\overset{\infty}{=}$ as our equality, the problem is solved, because $\uparrow_3$ would not be equal to $\uparrow_0$.

A *strict function* is a function that diverges whenever one of its arguments diverges. The problem stated above does not arise with strict functions, and so we can safely use $(\simeq)$ when dealing only with strict functions. In standard

recursion theory, all recursive functions are strict. For our purposes, then, $(\simeq)$ is the appropriate equality.

However, the study of recursive functions using $\overset{\infty}{=}$ as equality is a very interesting subject that needs to be investigated further.

We do not intend to impose that $(\simeq)$ coincides with Leibniz equality, but rather to work with the setoid whose carrier is $\mathbb{N}^\nu$ and whose book equality is $(\simeq)$. It is easy to check that the functions that we define in the next section are invariant under this equality, that is, they are setoid functions.

7.3 Recursive Functions

We want to show that every recursive function with n arguments can be represented by a term $f\colon (\mathbb{N}^\nu)^n \to \mathbb{N}^\nu$. There are generally two ways of defining such a function, one strict and one lazy. The strict definition requires that the result of the application of the function is undefined if one of its arguments is undefined. On the other hand, the lazy version of the function may produce a finite result even if some of the arguments diverge. The strict version is the standard one in recursion theory. However, in many cases the lazy version is more efficient and often easier to define. Therefore, in the following, we give both versions.

The Zero Function

The zero function in recursion theory is defined as

$$\begin{aligned}&\underline{0}\colon \mathbb{N} \to \mathbb{N}\\ &\underline{0}(x) := 0.\end{aligned}$$

We could define it as the lazy constant 0:

$$\begin{aligned}&\underline{0}\colon \mathbb{N}^\nu \to \mathbb{N}^\nu\\ &\underline{0}(x) := 0.\end{aligned}$$

However, recursive functions are assumed to diverge on divergent arguments. This means that we expect $\underline{0}(\uparrow_0) = \uparrow_0$, while, with our definition $\underline{0}(\uparrow_0) = 0$. Therefore, we change the definition to its strict version

$$\begin{aligned}&\underline{0}\colon \mathbb{N}^\nu \to \mathbb{N}^\nu\\ &\underline{0}(n) := \mathsf{Cases}\ n\ \mathsf{of}\ \left\{\begin{array}{ll} 0 & \mapsto 0\\ (\mathsf{S}\ n') & \mapsto (\_\ \underline{0}(n'))\\ (\_\ n') & \mapsto (\_\ \underline{0}(n')).\end{array}\right.\end{aligned}$$

More formally, we would define

$$\underline{0} := \left(\nu\text{-}\mathsf{it}\ [n]\mathsf{Cases}\ (\nu\text{-}\mathsf{out}\ n)\ \mathsf{of}\ \left\{\begin{array}{ll}(\mathsf{in}_0\ x) & \mapsto (\mathsf{in}_0\ x)\\ (\mathsf{in}_1\ n') & \mapsto (\mathsf{in}_2\ n')\\ (\mathsf{in}_2\ n') & \mapsto (\mathsf{in}_2\ n')\end{array}\right.\right).$$

This strict version of the function simply substitutes every occurrence of S in the argument with an occurrence of $\_$. This implies that $\underline{0}(\infty) = \underline{0}(\uparrow_n) = \uparrow_0$. It is inefficient compared to the lazy version, because it always reads the input. For practical purposes it may be more sensible to use the lazy version.

The Successor Function

We already defined the successor function as

$$\begin{array}{l}\mathsf{S}\colon \mathbb{N}^\nu \to \mathbb{N}^\nu\\ \mathsf{S}(n) := (\nu\text{-}\mathsf{rec}\ [x\colon \mathbb{N}^\nu](\mathsf{in}_1\ (\mathsf{inl}\ x))\ n).\end{array}$$

This is the lazy version. The strict version can be defined by putting the extra S constructor at the end of the term in place of the beginning:

$$\begin{array}{l}\mathsf{succ}\colon \mathbb{N}^\nu \to \mathbb{N}^\nu\\ \mathsf{succ}(0) := (\mathsf{S}\ 0)\\ \mathsf{succ}(\mathsf{S}\ n) := (\mathsf{S}\ (\mathsf{succ}\ n))\\ \mathsf{succ}(\_\ n) := (\_\ (\mathsf{succ}\ n)).\end{array}$$

Formally

$$\begin{array}{l}\mathsf{succ}\colon \mathbb{N}^\nu \to \mathbb{N}^\nu\\ \mathsf{succ}(m) := \left(\nu\text{-}\mathsf{it}\ [n]\mathsf{Cases}\ (\nu\text{-}\mathsf{out}\ n)\ \mathsf{of}\ \left\{\begin{array}{ll}(\mathsf{in}_0\ x) & \mapsto (\mathsf{in}_1\ n)\\ (\mathsf{in}_1\ n') & \mapsto (\mathsf{in}_1\ n')\\ (\mathsf{in}_2\ n') & \mapsto (\mathsf{in}_2\ n')\end{array}\right.\ m\right).\end{array}$$

Notice, however, that, in this case, the distinction between the lazy and the strict version is not relevant, because even the lazy version always returns an infinite object on an infinite input. Therefore, the two versions of the function are equivalent, in the sense that they give equal results (by the equality $\simeq$) on the same input.

Projection functions

The projection functions could be defined as simply the projection functions of type theory,

$$\pi_i^k\colon (\mathbb{N}^\nu)^k \to \mathbb{N}^\nu,$$

for $0 \leq i < k$. Once again, we obtain a lazy function, that is, $(\pi_i^k\ \langle x_0, \ldots, x_{k-1}\rangle)$ converges if x_i converges, even if some of the x_j's with $j \neq i$ diverge. Later, we give a general method to obtain the strict version of a lazy function. Let us do it explicitly in this case.

To define the strict version of the operation, we consider the following generalization. Given a function $f\colon \mathbb{N}^\nu \to \mathbb{N}^\nu$, we define a new function $(\Box \times f)\colon \mathbb{N}^\nu \times \mathbb{N}^\nu \to \mathbb{N}^\nu$ such that $(\Box \times f)(\langle n, m\rangle) = (f\ m)$ if n converges, but diverges if n

diverges.

$$
\begin{array}{l}
(\Box \times f)\colon \mathbb{N}^\nu \times \mathbb{N}^\nu \to \mathbb{N}^\nu \\
(\Box \times f)(\langle 0, m\rangle) := \_\ (f\ m) \\
(\Box \times f)(\langle (\mathsf{S}\ n'), m\rangle) := \_\ (\Box \times f)(\langle n', m\rangle) \\
(\Box \times f)(\langle (\_\ n'), m\rangle) := \_\ (\Box \times f)(\langle n', m\rangle)
\end{array}
$$

Formally

$$(\Box \times f) := (\nu\text{-}\mathsf{rec}\ [p = \langle n, m\rangle\colon \mathbb{N}^\nu \times \mathbb{N}^\nu] w[n, m])$$

where

$$
\begin{array}{rl}
n, m\colon \mathbb{N}^\nu \vdash & w[n, m]\colon \mathsf{Unit} + (\mathbb{N}^\nu + \mathbb{N}^\nu \times \mathbb{N}^\nu) + (\mathbb{N}^\nu + \mathbb{N}^\nu \times \mathbb{N}^\nu) \\
& := \mathsf{Cases}\ (\nu\text{-}\mathsf{out}\ n)\ \mathsf{of} \left\{ \begin{array}{l}
(\mathsf{in}_0\ x) \mapsto (\mathsf{in}_2\ (\mathsf{inl}\ (f\ m))) \\
(\mathsf{in}_1\ n') \mapsto (\mathsf{in}_2\ (\mathsf{inr}\ \langle n', m\rangle)) \\
(\mathsf{in}_2\ n') \mapsto (\mathsf{in}_2\ (\mathsf{inr}\ \langle n', m\rangle))
\end{array} \right.
\end{array}
$$

Similarly we define $(f \times \Box)$.

Finally, we can define the strict version of the projections as

$$\pi_i^k := \underbrace{\Box \times \cdots \times \Box}_{i-1} \times \mathsf{id}_{\mathbb{N}^\nu} \times \underbrace{\Box \times \cdots \times \Box}_{k-i}.$$

Composition

Composition is just the type-theoretic composition. If $f\colon (\mathbb{N}^\nu)^k \to \mathbb{N}^\nu$ and $g_i\colon (\mathbb{N}^\nu)^h \to \mathbb{N}^\nu$ for $i = 0, \ldots, k-1$, then

$$
\begin{array}{l}
f \circ \langle g_0, \ldots, g_{k-1}\rangle\colon (\mathbb{N}^\nu)^h \to \mathbb{N}^\nu \\
f \circ \langle g_0, \ldots, g_{k-1}\rangle(\overline{x}) := (f\ \langle (g_1\ \overline{x}), \ldots, (g_{k-1}\ \overline{x})\rangle).
\end{array}
$$

Primitive Recursion

Primitive recursion is available in type theory for every inductively defined type. However, $\mathbb{N}^\nu$ is not and inductive type, and the principle of primitive recursion is not immediate for it. Let us begin with the simpler case of a function of one argument. Given a constant $c_0\colon \mathbb{N}^\nu$ and a binary function $g\colon (\mathbb{N}^\nu)^2 \to \mathbb{N}^\nu$, we want to define the function

$$
\begin{array}{l}
h\colon \mathbb{N}^\nu \to \mathbb{N}^\nu \\
h(0) := c_0 \\
h(\mathsf{S}\ n) := (g\ \langle n, h(n)\rangle)
\end{array}
$$

There are two problems here: There is no specification of what the function should do on an object of the form $(\_\ n)$ and the right hand side of the clause for $(\mathsf{S}\ n)$ is not productive, that is, it is not guaranteed to produce a canonical form. Notice here, that we cannot argue that the function is guaranteed to produce a canonical form because the recursive call $h(n)$ has a structurally smaller argument than the input $(\mathsf{S}\ n)$: In $\mathbb{N}^\nu$, an object n is not necessarily simpler that $(\mathsf{S}\ n)$; in fact, we could have $(\mathsf{S}\ n) \simeq n$. The trick to define the

function is to add a $\_$ constructor for every step of primitive recursion. We start by defining an auxiliary function

$$\begin{array}{l}\mathcal{R}_{c_0,g}\colon (\mathbb{N}^\nu \to \mathbb{N}^\nu) \times \mathbb{N}^\nu \to \mathbb{N}^\nu\\ \mathcal{R}_{c_0,g}(\langle f, 0\rangle) := \_\ (f\ c_0)\\ \mathcal{R}_{c_0,g}(\langle f, (\mathsf{S}\ n)\rangle) := \_\ \mathcal{R}_{c_0,g}(\langle [x](f\ (g\ \langle n, x\rangle)), n\rangle)\\ \mathcal{R}_{c_0,g}(\langle f, (\_\ n)\rangle) := \_\ \mathcal{R}_{c_0,g}(\langle f, n\rangle)\end{array}$$

and then

$$h := [n]\mathcal{R}_{c_0,g}(\langle \mathsf{id}_{\mathbb{N}^\nu}, n\rangle).$$

Let us explain how the auxiliary function $\mathcal{R}$ works. Since we are using $\mathbb{N}^\nu$, we cannot define the function h starting with its value on 0 and working our way up the the input n, since we ignore whether n is finite or infinite. Let us assume that n has the form

$$n = \_\ \_\ \mathsf{S}\ \_\ \mathsf{S}\ \_\ \_\ \cdots.$$

If we knew that this sequence ends with a 0, then we could first compute h on 0, then on $(\mathsf{S}\ 0)$, and so on, ignoring the $\_$ steps, until we reach the value of h on n. But it is undecidable whether n ends with a 0 or goes on forever. A correct function on $\mathbb{N}^\nu$ must be definable by generating a finite portion of the result from a finite portion of the input. In other words, when we read the first S, we must produce part of the result, without knowing how the computation will proceed. We solve this problem by passing a function containing the information obtained so far to the next stage. In order to compute $h(n)$ we generate functions $f^m\colon \mathbb{N}^\nu \to \mathbb{N}^\nu$ for every component m' of n, such that

$$h(n) = f^m(h(m')).$$

In other words, since we do not know the value of $h(m')$ yet, we store the function f^m to be used later, when $h(m')$ becomes known. It is clear that $f^n := \mathsf{id}_{\mathbb{N}^\nu}$ works. Suppose we know $f^{(\mathsf{S}\ m')}$, we can compute $f^{m'}$ by

$$f^{m'} := [x](f^{(\mathsf{S}\ m')}\ (g\ \langle m', x\rangle)).$$

In fact, we have, by definition of $f^{(\mathsf{S}\ m')}$ and of h,

$$\begin{array}{rl}h(n) & = f^{(\mathsf{S}\ m')}(h(\mathsf{S}\ m')) = f^{(\mathsf{S}\ m')}(g\ \langle m', h(m')\rangle)\\ & = ([x](f^{(\mathsf{S}\ m')}\ (g\ \langle m', x\rangle))\ h(m'))\\ & = (f^{m'}\ h(m')).\end{array}$$

Notice how this coinductive process works exactly in the opposite direction to the inductive definition of h: While we define $h(\mathsf{S}\ m')$ in terms of $h(m')$, we define $f^{m'}$ in terms of $f^{(\mathsf{S}\ m')}$.

It is then immediate from the definition of $\mathcal{R}$ that

$$h(n) = \mathcal{R}_{c_0,g}(\langle f^m, m\rangle).$$

The auxiliary function $\mathcal{R}_{c_0,g}$ can be defined formally as (remember, from Chapter 2, Section 2.7, that the notation $[z = \langle f, n\rangle]w[f, n]$ denotes the term $[z]w[(\pi_1\ z), (\pi_2\ z)]$)

$$\mathcal{R}_{c_0,g} := (\nu\text{-}\mathsf{rec}\ [z = \langle f, n\rangle]w[f, n])$$

with

$$f\colon \mathbb{N}^\nu \to \mathbb{N}^\nu, n\colon \mathbb{N}^\nu \vdash w[f, n]\colon \mathsf{Unit} + (\mathbb{N}^\nu + (\mathbb{N}^\nu \to \mathbb{N}^\nu) \times \mathbb{N}^\nu) + (\mathbb{N}^\nu + (\mathbb{N}^\nu \to \mathbb{N}^\nu) \times \mathbb{N}^\nu)$$

$$w[f, n] := \mathsf{Cases}\ (\nu\text{-}\mathsf{out}\ n)\ \mathsf{of}\ \begin{cases} (\mathsf{in}_0\ x) \mapsto (\mathsf{in}_2\ (\mathsf{inl}\ (f\ c_0))) \\ (\mathsf{in}_1\ n') \mapsto (\mathsf{in}_2\ (\mathsf{inr}\ \langle [y](f\ (g\ \langle n', y\rangle)), n'\rangle)) \\ (\mathsf{in}_2\ n') \mapsto (\mathsf{in}_2\ (\mathsf{inr}\ \langle f, n'\rangle)). \end{cases}$$

We can use again the operator $\mathcal{R}$ to define the general version of primitive recursion. Let $f\colon (\mathbb{N}^\nu)^k \to \mathbb{N}^\nu$ and $g\colon (\mathbb{N}^\nu)^{k+2} \to \mathbb{N}^\nu$. We want to define the function

$$\begin{array}{l} h\colon (\mathbb{N}^\nu)^{k+1} \to \mathbb{N}^\nu \\ h(\overline{x}, 0) := (f\ \overline{x}) \\ h(\overline{x}, (S\ n)) := (g\ \overline{x}\ n\ (h\ \overline{x}\ n)). \end{array}$$

We can put

$$h := \lambda\overline{x}.\lambda n.\mathcal{R}_{(f\ \overline{x}),(g\ \overline{x})}(\mathsf{id}_{\mathbb{N}^\nu}, n).$$

In the case of primitive recursion, there is no lazy version of h, since, by its nature, primitive recursion requires that the inductive argument is completely evaluated to produce a result. Whether h is strict on the other arguments, the parameters $\overline{x}$, depends on whether the functions f and g are strict on them.

We have shown how to implement every primitive recursive function.

Proposition 7.3.1 *Every primitive recursive function $f\colon \mathbb{N}^k \to \mathbb{N}$ can be represented by a type-theoretic function $f^\nu\colon (\mathbb{N}^\nu)^k \to \mathbb{N}^\nu$ such that, for every pair of natural numbers n, m,*

$$f(n) = m \iff (f^\nu\ \ulcorner n \urcorner^\nu) \simeq \ulcorner m \urcorner^\nu.$$

We could prove the proposition in a different way. Since every primitive recursive function is representable in type theory, we just need to show that every function on $\mathbb{N}$ can be extended on $\mathbb{N}^\nu$.

Lemma 7.3.2 *For every type-theoretic function $f\colon \mathbb{N}^k \to \mathbb{N}$, there exists a type-theoretic function $f^\nu\colon (\mathbb{N}^\nu)^k \to \mathbb{N}^\nu$ such that, for every pair of natural numbers n, m,*

$$f(n) = m \iff (f^\nu\ \ulcorner n \urcorner^\nu) \simeq \ulcorner m \urcorner^\nu.$$

Proof The idea is that, given a k-tuple $\overline{x}\colon (\mathbb{N}^\nu)^k$, we read all its elements and, if they are all finite, we produce a k-tuple $\overline{x}^\mu\colon \mathbb{N}^k$ and we compute f on them.

We consider first the simpler example when $k = 1$. Assume $f\colon \mathbb{N} \to \mathbb{N}$. We define the auxiliary function $\mathcal{F}_f$ with an extra natural number argument m that counts the number of Ss.

$$\begin{aligned}
&\mathcal{F}_f\colon \mathbb{N}^\nu \times \mathbb{N} \to \mathbb{N}^\nu\\
&\mathcal{F}_f(0, m) := \ulcorner (f\ m) \urcorner^\nu\\
&\mathcal{F}_f((\mathsf{S}\ n), m) := _\ \mathcal{F}_f(n, (\mathsf{S}\ m))\\
&\mathcal{F}_f((_\ n), m) := _\ \mathcal{F}_f(n, m).
\end{aligned}$$

The desired function is then

$$\begin{aligned}
&f^\nu\colon \mathbb{N}^\nu \to \mathbb{N}^\nu\\
&f^\nu(n) := \mathcal{F}_f(n, 0).
\end{aligned}$$

The general case is obtained by iterating this procedure. Let $f\colon \mathbb{N}^{k+1} \to \mathbb{N}$. We define the auxiliary function $\mathcal{F}_{k+1,f}$ using $\mathcal{F}_{k,f'}$, with the base case $\mathcal{F}_{1,f^1} := \mathcal{F}_{f^1}$.

$$\begin{aligned}
&\mathcal{F}_{k+1,f}\colon (\mathbb{N}^\nu)^{k+1} \times (\mathbb{N}^{k+1}) \to \mathbb{N}^\nu\\
&\mathcal{F}_{k+1,f}(0, \overline{n}, m_0, \overline{m}) := \mathcal{F}_{k,(f\ m_0)}(\overline{n}, \overline{m})\\
&\mathcal{F}_{k+1,f}((\mathsf{S}\ n_0), \overline{n}, m_0, \overline{m}) := _\ \mathcal{F}_{k+1,f}(n_0, \overline{n}, (\mathsf{S}\ m_0), \overline{m})\\
&\mathcal{F}_{k+1,f}((_\ n_0), \overline{n}, m_0, \overline{m}) := _\ \mathcal{F}_{k+1,f}(n_0, \overline{n}, m_0, \overline{m}).
\end{aligned}$$

Finally, a function $f\colon \mathbb{N}^k \to \mathbb{N}$ is implemented as

$$\begin{aligned}
&f^\nu\colon {\mathbb{N}^\nu}^k \to \mathbb{N}^\nu\\
&f^\nu(\overline{n}) := \mathcal{F}_{k,f}(\overline{n}, \overline{0}).
\end{aligned}$$

□

Minimization

Let $f\colon \mathbb{N}^\nu \to \mathbb{N}^\nu$. We want to define $(\mathsf{min}\ f)\colon \mathbb{N}^\nu$, that is intuitively the least $y\colon \mathbb{N}^\nu$ such that $(f\ y) = 0$. The minimization is computed in the following way.

- If $(f\ 0) = 0$, then $(\mathsf{min}\ f) = 0$;
- If $(f\ 0) \neq 0$, then $(\mathsf{min}\ f) = (\mathsf{S}\ (\mathsf{min}\ \lambda x.(f\ (\mathsf{S}\ x))))$.

Two remarks need to be made here. First, bicounting equality on $\mathbb{N}^\nu$, $\simeq$, is not decidable, so we cannot perform the first step trivially. Second, we can be sure that $(f\ 0) \neq 0$ as soon as we know that $(f\ 0)$ is a successor; that is, we do not need to compute $(f\ 0)$ completely: even if it diverges, we may be able to find out that it cannot be 0. This means that we can define a lazy version of min, that may return a finite value even if f does not terminate for some inputs lower that that value.

A standard way to define the minimization operator uses an auxiliary function

$$\begin{aligned}
&\mathcal{G}\colon \mathbb{N}^\nu \to \mathbb{N}^\nu\\
&\mathcal{G}(n) := \mathsf{if}\ (f\ n) \simeq 0\ \mathsf{then}\ n\ \mathsf{else}\ \mathcal{G}(\mathsf{S}\ n)
\end{aligned}$$

and then puts $(\mathsf{min}\ f) := \mathcal{G}(0)$. As we have noted, $(f\ n) \simeq 0$ is not a decidable property, which makes it impossible to use a standard $\mathsf{if} - \mathsf{then} - \mathsf{else}$ construction. However, a refined version of the if construction manages to give the right result even without deciding the equality.

$$\begin{array}{l}\mathsf{if}_0 \colon \mathbb{N}^\nu \to \mathbb{N}^\nu \to \mathbb{N}^\nu \to \mathbb{N}^\nu \\ \mathsf{if}_0(0, y, z) := \_\ y \\ \mathsf{if}_0((\mathsf{S}\ x), y, z) := \_\ z \\ \mathsf{if}_0((\_\ x), y, z) := \_\ \mathsf{if}_0(x, y, z)\end{array}$$

This version of the function was lazy. A strict (in the first argument) version is

$$\begin{array}{l}\mathsf{if}_0 \colon \mathbb{N}^\nu \to \mathbb{N}^\nu \to \mathbb{N}^\nu \to \mathbb{N}^\nu \\ \mathsf{if}_0(0, y, z) := \_\ y \\ \mathsf{if}_0((\mathsf{S}\ x), y, z) := \_\ \mathsf{if}_0(x, z, z) \\ \mathsf{if}_0((\_\ x), y, z) := \_\ \mathsf{if}_0(x, y, z).\end{array}$$

Unfortunately, the definition

$$\mathcal{G}(n) := \mathsf{if}_0((f\ n), n, \mathcal{G}(\mathsf{S}\ n))$$

is not correct, since the call of $\mathcal{G}$ on the right-hand side occurs under the application of another function, if_0 (it is not guarded by a constructor). We cannot easily formalize this definition of $\mathcal{G}$.

Instead, we use a variant of this function, keeping track of the values of f:

$$\begin{array}{l}\mathcal{G}_f \colon \mathbb{N} \times \mathbb{N}^\nu \to \mathbb{N}^\nu \\ \mathcal{G}_f(i, 0) := 0 \\ \mathcal{G}_f(i, (\mathsf{S}\ n)) := \mathsf{S}\ \mathcal{G}_f((\mathsf{S}\ i), (f\ (\mathsf{S}\ \ulcorner i \urcorner^\nu))) \\ \mathcal{G}_f(i, (\_\ n)) := \_\ \mathcal{G}_f(i, n).\end{array}$$

In the second clause there are four occurrences of S with different meanings. In $(\mathsf{S}\ i)$ the successor constructor of $\mathbb{N}$ is meant, while in the other three cases the successor function on $\mathbb{N}^\nu$ is meant.

Intuitively, $\mathcal{G}(x, (f\ x))$ is the least y such that $(f\ (x + y)) = 0$. We can then define

$$\begin{array}{l}\mathsf{min} \colon (\mathbb{N}^\nu \to \mathbb{N}^\nu) \to \mathbb{N}^\nu \\ \mathsf{min}(f) := \mathcal{G}_f(0, (f\ 0)).\end{array}$$

Formally, $\mathcal{G}_f$ is defined as

$$\mathcal{G}_f := \left(\nu\text{-it}\ [p = \langle i, n \rangle] \mathsf{Cases}\ (\nu\text{-out}\ n)\ \mathsf{of}\ \left\{ \begin{array}{l} (\mathsf{in}_0\ x) \mapsto (\mathsf{in}_0\ x) \\ (\mathsf{in}_1\ n') \mapsto (\mathsf{in}_1\ \langle (\mathsf{S}\ i), (f\ (\mathsf{S}\ i)) \rangle) \\ (\mathsf{in}_2\ n') \mapsto (\mathsf{in}_2\ \langle i, n' \rangle) \end{array} \right. \right).$$

This was the lazy version of the minimization operator: It looks for the first i such that $(f\ i) = 0$ by testing each i; if, for a certain i, the value of $(f\ i)$ is a

successor, then it immediately goes to $i+1$, without completing the evaluation of $(f\ i)$. So it may give a finite solution j even if, for some $i < j$, $(f\ i)$ diverges.

We can define a strict version of the function by reading the whole value of $((f\ i)$ before performing the zero test.

$$\begin{array}{l}\mathcal{H}_f\colon \mathbb{N}\times\mathbb{N}^\nu\times\mathbb{N}\to\mathbb{N}^\nu\\ \mathcal{H}_f(i,0,j) := \_\ \mathsf{if}\ j=0\ \mathsf{then}\ \ulcorner i\urcorner^\nu\ \mathsf{else}\ \mathcal{H}_f((\mathsf{S}\ i),(f\ (\mathsf{S}\ i)),0)\\ \mathcal{H}_f(i,(\mathsf{S}\ n),j) := \_\ \mathcal{H}_f(i,n,(\mathsf{S}\ j))\\ \mathcal{H}_f(i,(\_\ n),j) := \_\ \mathcal{H}_f(i,n,j)\end{array}$$

where the $\mathsf{if}-\mathsf{then}-\mathsf{else}$ function is the one on the standard natural numbers $\mathbb{N}$. Then we can define

$$\begin{array}{l}\mathsf{min}\colon (\mathbb{N}^\nu\to\mathbb{N}^\nu)\to\mathbb{N}^\nu\\ \mathsf{min}(f) := \mathcal{H}_f(0,(f\ 0),0).\end{array}$$

The idea here is that we try to translate $(f\ i)$ into an element of $\mathbb{N}$ before testing it for equality. If $(f\ i)$ is an infinite element, this process of translation does not terminate and the result is an infinite sequence of $\_$.

The function $\mathcal{H}_f$ seems to suffer from the same problem that afflicted the function $\mathcal{G}$: the occurrence of $\mathcal{H}_f$ inside the right-hand side of the first clause is not guarded by a constructor. However, we can exploit the fact that the $\mathsf{if}-\mathsf{then}-\mathsf{else}$ operator this time is applied to standard, that is, decidable, natural numbers. The formal version of $\mathcal{H}$ is

$$\begin{array}{l}\mathcal{H}_f\colon \mathbb{N}\times\mathbb{N}^\nu\times\mathbb{N}\to\mathbb{N}^\nu\\ \mathcal{H}_f := (\nu\text{-}\mathsf{rec}\ [t=\langle i,n,j\rangle\colon \mathbb{N}\times\mathbb{N}^\nu\times\mathbb{N}]w[i,n,j])\end{array}$$

with

$$\begin{array}{l}i\colon\mathbb{N}, n\colon\mathbb{N}^\nu, j\colon\mathbb{N}\vdash w[i,n,j]\colon \mathsf{Unit}+(\mathbb{N}^\nu+\mathbb{N}\times\mathbb{N}^\nu\times\mathbb{N})+(\mathbb{N}^\nu+\mathbb{N}\times\mathbb{N}^\nu\times\mathbb{N})\\ w[i,n,j] :=\\ \mathsf{Cases}\ (\nu\text{-}\mathsf{out}\ n)\ \mathsf{of}\ \left\{\begin{array}{ll}(\mathsf{in}_0\ x)\ \mapsto & \mathsf{Cases}\ j\ \mathsf{of}\\ & \left\{\begin{array}{ll}0 & \mapsto(\mathsf{in}_2\ (\mathsf{inl}\ \ulcorner i\urcorner^\nu))\\ (\mathsf{S}\ j') & \mapsto(\mathsf{in}_2\ (\mathsf{inr}\ \langle(\mathsf{S}\ i),(f\ (\mathsf{S}\ i)),0\rangle))\end{array}\right.\\ (\mathsf{in}_1\ n')\mapsto(\mathsf{in}_2\ (\mathsf{inr}\ \langle i,n,(\mathsf{S}\ j)\rangle))\\ (\mathsf{in}_2\ n')\mapsto(\mathsf{in}_2\ (\mathsf{inr}\ \langle i,n,j\rangle)).\end{array}\right.\end{array}$$

The generic form of minimization is obtained by abstraction. Let $f\colon(\mathbb{N}^\nu)^{k+1}\to\mathbb{N}^\nu$. We want to define $(\mathsf{min}^k\ f)\colon(\mathbb{N}^\nu)^k\to\mathbb{N}^\nu$, such that $(\mathsf{min}^k\ f\ \overline{x})$ is the least y such that $(f\ \overline{x}\ y)=0$. We just put $(\mathsf{min}^k\ f) := \lambda\overline{x}.(\mathsf{min}\ (f\ \overline{x}))$.

We have therefore proved that all the standard constructors for recursive functions are representable as functions on $\mathbb{N}^\nu$.

Theorem 7.3.3 *Every (partial) recursive function $f\colon\mathbb{N}^k\to\mathbb{N}$ can be represented by a type-theoretic function $f^\nu\colon(\mathbb{N}^\nu)^k\to\mathbb{N}^\nu$ such that, for every pair of natural numbers n, m,*

$$f(n)=m \iff (f^\nu\ \ulcorner n\urcorner^\nu)\simeq\ulcorner m\urcorner^\nu.$$

7.4 Strict version of a function

We have seen again and again that every function defined on $\mathbb{N}^\nu$ may have a lazy and a strict version. To follow the standard semantics of recursive functions, we use the strict version. This means that, even when the lazy formalization is more natural, we have to look for a strict version. On the other hand, in some cases, a strict function may be more efficient.

We give a general method to define the strict version of a function. Given a function $f\colon \mathbb{N}^\nu \to \mathbb{N}^\nu$, we want to define a new function $f_\downarrow\colon \mathbb{N}^\nu \to \mathbb{N}^\nu$ that diverges if the argument diverges, and gives the same result as f if the argument converges. We start by defining an auxiliary function

$$\begin{array}{l} f'\colon \mathbb{N}^\nu \times \mathbb{N}^\nu \to \mathbb{N}^\nu \\ f'(n,0) := (f\ n) \\ f'(n,(\mathsf{S}\ m)) := \_\ f'(n,m) \\ f'(n,(\_\ m)) := \_\ f'(n,m) \end{array}$$

and then

$$\begin{array}{l} f_\downarrow\colon \mathbb{N}^\nu \to \mathbb{N}^\nu \\ f_\downarrow(n) := f'(n,n). \end{array}$$

Formally, f' is defined as

$$\begin{array}{l} f'\colon \mathbb{N}^\nu \times \mathbb{N}^\nu \to \mathbb{N}^\nu \\ f' := (\nu\text{-}\mathsf{rec}\ [p = \langle n,m\rangle\colon \mathbb{N}^\nu \times \mathbb{N}^\nu] w[n,m]) \end{array}$$

with

$$n,m\colon \mathbb{N}^\nu \vdash w[n,m]\colon \mathsf{Unit} + (\mathbb{N}^\nu + \mathbb{N}^\nu \times \mathbb{N}^\nu) + (\mathbb{N}^\nu + \mathbb{N}^\nu \times \mathbb{N}^\nu)$$

$$w[n,m] := \mathsf{Cases}\ m\ \mathsf{of} \begin{cases} (\mathsf{in}_0\ x) & \mapsto (\mathsf{in}_2\ (\mathsf{inl}\ (f\ n))) \\ (\mathsf{in}_1\ m') & \mapsto (\mathsf{in}_2\ (\mathsf{inr}\ \langle n,m'\rangle)) \\ (\mathsf{in}_2\ m') & \mapsto (\mathsf{in}_2\ (\mathsf{inr}\ \langle n,m'\rangle)). \end{cases}$$

Theorem 7.4.1 *For every function $f\colon \mathbb{N}^\nu \to \mathbb{N}^\nu$ and for every $n\colon \mathbb{N}^\nu$, we have that*

$$\begin{array}{lcl} n \simeq \ulcorner m \urcorner^\nu & \Rightarrow & (f_\downarrow\ n) \simeq (f\ n) \\ n \simeq \infty & \Rightarrow & (f_\downarrow\ n) \simeq \uparrow_0 \\ n \simeq \uparrow_m & \Rightarrow & (f_\downarrow\ n) \simeq \uparrow_0. \end{array}$$

7.5 The Computation Algorithm

We have encoded all partial recursive functions as functions on $\mathbb{N}^\nu$. However, $\mathbb{N}^\nu$ is a coinductive type, which means that its elements do not reduce to a canonical form. This implies that functions defined on $\mathbb{N}^\nu$ by corecursion do not reduce to their value. Suppose f is a partial recursive function on natural numbers. Then we have a function $f^\nu\colon \mathbb{N}^\nu \to \mathbb{N}^\nu$. We want to compute $(f\ n)$ for some $n\colon \mathbb{N}$ using f^ν. We need to determine the result of $(f^\nu\ \ulcorner n \urcorner^\nu)$. Suppose f terminates on n and the result is $(f\ n) = m$. It is not true that $(f^\nu\ \ulcorner n \urcorner^\nu)$

reduces to $\ulcorner m \urcorner^\nu$. It is true that $(f^\nu \ \ulcorner n \urcorner^\nu) \simeq \ulcorner m \urcorner^\nu$, but this relation is weaker than convertibility.

However, there exists an algorithm that can be implemented to compute recursive functions on $\mathbb{N}^\nu$. It consists in a notion of reduction for sequents, rather than for terms. We start with the sequent

$$x\colon \mathbb{N}, h\colon (f^\nu \ \ulcorner n \urcorner^\nu) \simeq \ulcorner x \urcorner^\nu \vdash h\colon (f^\nu \ \ulcorner n \urcorner^\nu) \simeq \ulcorner x \urcorner^\nu.$$

The algorithm will generate a sequence of sequents of the form

$$\Gamma \vdash p\colon (f^\nu \ \ulcorner n \urcorner^\nu) \simeq \ulcorner m \urcorner^\nu.$$

The algorithm terminates when we reach a sequent with an empty context Γ, that is,

$$\vdash p\colon (f^\nu \ \ulcorner n \urcorner^\nu) \simeq \ulcorner m \urcorner^\nu.$$

Then we have computed the result m of the function.

We use the following three results in the algorithm.

Lemma 7.5.1

$$\begin{array}{ll}
\mathsf{nat\text{-}val}_0\colon & \forall n\colon \mathbb{N}^\nu.\forall x\colon \mathsf{Unit}. \\
& \lfloor (\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_0\ x) \rfloor \to n \simeq \ulcorner 0 \urcorner^\nu \\
\mathsf{nat\text{-}val}_\mathsf{S}\colon & \forall n, n'\colon \mathbb{N}^\nu.\forall m'\colon \mathbb{N}. \\
& \lfloor (\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_1\ n') \rfloor \to n' \simeq \ulcorner m' \urcorner^\nu \to n \simeq \ulcorner (\mathsf{S}\ m') \urcorner^\nu \\
\mathsf{nat\text{-}val}__\colon & \forall n, n'\colon \mathbb{N}^\nu.\forall m\colon \mathbb{N}. \\
& \lfloor (\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_2\ n') \rfloor \to n' \simeq \ulcorner m \urcorner^\nu \to n \simeq \ulcorner m \urcorner^\nu
\end{array}$$

Proof The statements are easily proved by choosing the appropriate bicounting relations:

$$\begin{array}{l}
(R_0\ n_1\ n_2) := (\exists x\colon \mathsf{Unit}.(\nu\text{-}\mathsf{out}\ n_1) = (\mathsf{in}_0\ x)) \wedge n_2 = \ulcorner 0 \urcorner^\nu, \\
(R_1\ n_1\ n_2) := \exists n'\colon \mathbb{N}^\nu.\exists m'\colon \mathbb{N}.\ (\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_1\ n') \wedge \\
\qquad\qquad\qquad\qquad n' \simeq \ulcorner m' \urcorner^\nu \wedge n_2 \simeq \ulcorner (\mathsf{S}\ m') \urcorner^\nu, \\
(R_2\ n_1\ n_2) := \exists n'\colon \mathbb{N}^\nu.\exists m\colon \mathbb{N}.\ (\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_2\ n') \wedge \\
\qquad\qquad\qquad\qquad n' \simeq \ulcorner m \urcorner^\nu \wedge n_2 \simeq \ulcorner m \urcorner^\nu,
\end{array}$$

for $\mathsf{nat\text{-}val}_0$, $\mathsf{nat\text{-}val}_\mathsf{S}$, and $\mathsf{nat\text{-}val}_\_$, respectively. □

Notice that the properties $(\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_0\ x)$, $(\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_1\ n')$, and $(\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_2\ n')$ are decidable, since they are just conversions (= is Leibniz equality). Therefore, we do not need to give their proof explicitly: If, for example, $(\nu\text{-}\mathsf{out}\ n) = (\mathsf{in}_0\ x)$, then a proof of this fact is always $(\mathsf{refl}\ (\mathsf{in}_0\ x))$. We treat them as implicit arguments: The notation $\lfloor x \rfloor$ states that the argument x is not explicitly written in the applications.

Now we can formally define the algorithm. First, we describe an algorithm that, given a coinductive natural numbers $n\colon \mathbb{N}^\nu$, finds an $m\colon \mathbb{N}$ such that $n \simeq \ulcorner m \urcorner^\nu$. This is a partial algorithm that terminates only if n is finite.

Definition 7.5.2 *Given an element $n\colon \mathbb{N}^\nu$, the following algorithm computes $m\colon \mathbb{N}$ such that $n \simeq \ulcorner m \urcorner^\nu$, if n is finite; diverges if n is infinite. The algorithm gives m and a proof $h\colon n \simeq \ulcorner m \urcorner^\nu$ as output. We denote by* $(\mathsf{Numeval}\ n)$ *the output. The computation uses as states sequents of the form*

$$\mathcal{S} \equiv m'\colon \mathbb{N}, h'\colon n' \simeq \ulcorner m' \urcorner^\nu \vdash h[m', h']\colon n \simeq \ulcorner m[m'] \urcorner^\nu$$

for some $n'\colon \mathbb{N}^\nu$.

- *Start with $\mathcal{S} \equiv m\colon \mathbb{N}, h\colon n \simeq \ulcorner m \urcorner^\nu \vdash h\colon n \simeq \ulcorner m \urcorner^\nu$;*
- *Assume $\mathcal{S} \equiv m'\colon \mathbb{N}, h'\colon n' \simeq \ulcorner m' \urcorner^\nu \vdash h[m', h']\colon n \simeq \ulcorner m[m'] \urcorner^\nu$. By cases on $(\nu\text{-}\mathsf{out}\ n')$:*
  - *If $(\nu\text{-}\mathsf{out}\ n') = (\mathsf{in}_0\ x)$, the algorithm terminates with output*

$$(\mathsf{Numeval}\ n) := m[0], h[0, (\mathsf{nat\text{-}val}_0\ n'\ x)];$$

  - *If $(\nu\text{-}\mathsf{out}\ n') = (\mathsf{in}_1\ n'')$, we put*

$$\begin{aligned}\mathcal{S} \equiv\ & m''\colon \mathbb{N}, h''\colon n'' \simeq \ulcorner m'' \urcorner^\nu \\ & \vdash h[(\mathsf{S}\ m''), (\mathsf{nat\text{-}val}_\mathsf{S}\ n'\ n''\ m''\ h'')]\colon n \simeq \ulcorner (\mathsf{S}\ m'') \urcorner^\nu\end{aligned}$$

    and repeat this step;
  - *If $(\nu\text{-}\mathsf{out}\ n') = (\mathsf{in}_2\ n'')$, we put*

$$\begin{aligned}\mathcal{S} \equiv\ & m'\colon \mathbb{N}, h''\colon n'' \simeq \ulcorner m' \urcorner^\nu \\ & \vdash h[m', (\mathsf{nat\text{-}val}_\_\ n'\ n''\ m'\ h'')]\colon n \simeq \ulcorner m' \urcorner^\nu\end{aligned}$$

    and repeat this step.

Theorem 7.5.3 *The algorithm* Numeval *terminates on input $n\colon \mathbb{N}^\nu$ if and only if there exists $m\colon \mathbb{N}$ such that $n \simeq \ulcorner m \urcorner^\nu$. In that case, if the output is* $(\mathsf{Numeval}\ n) \equiv m, h$ *then*

$$\vdash h\colon n \simeq \ulcorner m \urcorner^\nu$$

is a valid judgment.

Corollary 7.5.4 *Let $f\colon (\mathbb{N}^\nu)^k \to \mathbb{N}^\nu$ and $\overline{x}\colon (\mathbb{N}^\nu)^k$. The algorithm* Numeval *terminates on input $(f\ \overline{x})$ with output m, h if and only if*

$$\vdash h\colon (f\ \overline{x}) \simeq \ulcorner m \urcorner^\nu$$

is a valid judgment.

Part III

Formal Development of Mathematics

Chapter 8

Universal Algebra

In this Chapter we describe an implementation of Universal Algebra in type theory. The contents of the chapter, up to the first homomorphism theorem, were first published in [Cap99].

8.1 Introduction

The development of mathematical theories inside type theory presents some technical problems that make it difficult to translate an informal mathematical proof into a formalized one. In trying to carry out such a translation, one soon realizes that notions that were considered non-problematic and obvious at the informal level need a delicate formal analysis. Additional work is often needed just to define the mathematical structures under study and the basic tools to manipulate them. Besides the difficulty of rendering exactly what is expressed only in intuitive terms, there is the non-trivial task of translating into type theory what was originally intended to be expressed inside some form of set theory (for example in ZF). This chapter presents a development of such tools for generic algebraic reasoning, which has been completely formalized in the Coq proof development system (see [BBC+98]). We want to enable the users of such tools to easily define their own algebraic structures, manipulate objects and reason about them in a way that is not too far from ordinary mathematical practice.

This work stemmed from an original project of formal verification of Computer Algebra algorithms in type theory. I realized then that the definition of common mathematical structures, like those of ring and field, together with tools to manipulate them, was essential to the success of the enterprise. I decided to develop Universal Algebra as a general tool to define algebraic structures.

Previous work on Algebra in type theory was done by Paul Jackson using the proof system Nuprl (see [Jac94]), by Peter Aczel on Galois Theory (see [Acz94]) and by Huet and Saïbi on Category Theory (see [HS00]). A large class of algebraic structures has been developed in Coq by Loïc Pottier.

Another aim is the use of a two level approach to the derivation of propositions about algebraic objects (see [BRB95]). In this approach, statements about objects are lifted to a syntactic level where they can be manipulated by operators. An example is the simplification of expressions and automatic equational reasoning. This method was already used by Douglas Howe to construct a partial syntactic model of the type theory of Nuprl inside Nuprl itself, which can be used to program tactics inside the system (see [How88]). An application of this reflection mechanism to algebra was developed by Samuel Boutin in Coq for the simplification of ring expressions (see [Bou97]). In the present work the need to parameterize the construction of the syntactic level on the type of signatures posed an additional problem. A very general type construction similar to Martin-Löf's Wellorderings was employed for the purpose.

Finally, the study of the computational content of algebras is particularly interesting. We investigate to what extent algebraic objects can be automatically manipulated inside a proof checker. This can be done through the use of certified versions of algorithms borrowed from Computer Algebra, as was done by Théry in [Thé97] and by Coquand and Persson in [CP98] for Buchberger's algorithm.

The files of the implementation are available via the Internet at the site *http://www.cs.kun.nl/~venanzio/universal_algebra.html*.

Type Theory and Coq

The work presented here has been completely formalized inside Coq, but it could have equally easily been formalized in other proof systems based on type theory, like Lego or Alf. Although Coq is based on the Extended Calculus of Constructions (see [Luo94]) everything could be formalized in a weaker system. Any Pure Type System that is at least as expressive as $\lambda P\overline{\omega}$ (see [Bar92]) endowed with inductive types (see [Ste99]), or Martin-Löf's type theory with at least two universes (see [ML82], [ML84] or [NPS90]) is enough.

We recall that our version of type theory has two universes of types Set for sets and Prop for propositions, and that they both belong to the higher universe Type. The product type $\Pi x : A.B$ is written using Coq notation $(x : A)B$. If $B : \mathsf{Prop}$ we also write $(\forall x : A)B$. In Coq it is possible to define record types in which every field can depend on the values of the preceding fields. We presented their implementation in Chapter 2, Section 2.10. We use the following notation for records.

$$\mathsf{Record}\ Name\ : s\ :=\ constructor\ \begin{cases} field_1 : & A_1 \\ \vdots & \\ field_n : & A_n \end{cases}$$

An element of this record type is in the form $(constructor\ a_1 \dots a_n)$ where $a_1 : A_1, a_2 : A_2[field_1 := a_1], \dots, a_n : A_n[field_1 := a_1, \dots, field_{n-1} := a_{n-1}]$.

We have the projections

$$\begin{array}{l} field_1 : Name \to A_1 \\ field_2 : (x : Name)(A_2[field_1 := (field_1\ x)]) \\ \vdots \\ field_n : (x : Name)(A_n[field_1 := (field_1\ x)] \ldots [field_{n-1} := (field_{n-1}\ x)]) \end{array}$$

In Coq a record type is a shorthand notation for an inductive type with only one constructor. In a system without this facility they could be represented by nested Σ-types.

Algebraic structures in Type Theory

Let us start by considering a simple algebraic structure and its implementation in type theory. The standard mathematical definition of a group is the following.

Definition 8.1.1 *A* group *is a quadruple $\langle G, *, e, \_^{-1}\rangle$, where G is a set, $*$ a binary operation on G, e an element of G and $\_^{-1}$ a unary operation on G such that $(x * y) * z = x * (y * z)$, $x * e = x$ and $x * (x^{-1}) = e$ for all $x, y, z \in G$.*

An immediate translation in type theory would employ a record type.

$$\begin{array}{l} \mathsf{Record\ Group\ :\ Type\ :=} \\ \quad \mathsf{group} \left\{ \begin{array}{ll} \mathsf{elements} & : \mathsf{Setoid} \\ \mathsf{operation} & : \mathsf{elements} \to \mathsf{elements} \to \mathsf{elements} \\ \mathsf{unit} & : \mathsf{elements} \\ \mathsf{inverse} & : \mathsf{elements} \to \mathsf{elements} \end{array} \right. \end{array}$$

where a setoid is a set endowed with an equivalence relation. We introduced setoids in Chapter 5, and we summarize the notion in the next section.

But this is not yet enough since we didn't specify that the group axioms must be satisfied. This is usually done by enlarging the record to contain proofs of the axioms.

$$\begin{array}{l} \mathsf{Record\ Group\ :\ Type\ :=} \\ \quad \mathsf{group} \left\{ \begin{array}{ll} \mathsf{elements} & : \mathsf{Setoid} \\ \mathsf{operation} & : \mathsf{elements} \to \mathsf{elements} \to \mathsf{elements} \\ \mathsf{unit} & : \mathsf{elements} \\ \mathsf{inverse} & : \mathsf{elements} \to \mathsf{elements} \\ \mathsf{associativity} & : (\forall x, y, z : \mathsf{elements}) \\ & \quad (\mathsf{operation}\ (\mathsf{operation}\ x\ y)\ z) \\ & \quad = (\mathsf{operation}\ x\ (\mathsf{operation}\ y\ z)) \\ \mathsf{unitax} & : (\forall x : \mathsf{elements})(\mathsf{operation}\ x\ \mathsf{unit}) = x \\ \mathsf{inverseax} & : (\forall x : \mathsf{elements})(\mathsf{operation}\ x\ (\mathsf{inverse}\ x)) = \mathsf{unit} \end{array} \right. \end{array}$$

So to declare a specific group, for example the group of integers with the sum operation, we must specify all the fields:

$$\begin{array}{ll} \mathsf{Integer} & : \mathsf{Group} \\ & := (\mathsf{group}\ \mathbb{Z}\ \mathsf{plus}\ 0\ -\ p1\ p2\ p3). \end{array}$$

where $p1$, $p2$, $p3$ are proofs of the axioms.

Why it is useful to develop Universal Algebra

Once an algebraic structure has been specified in this way, we proceed to give standard definitions like those of subgroup, product of groups, quotient of a group by a congruence relation, homomorphism of groups, and we prove standard results. In this way, many algebraic structures can be specified, and theorems can be proved about them (see the work by Loïc Pottier).

Since most of the definitions and basic properties are the same for every algebraic structure, having an abstract general formulation of them would save us from duplicating the same work many times. This is the main reason why it is interesting to develop Universal Algebra. To this aim, we should internalize the generalization of the previous construction to have a general notion of algebraic structure inside type theory.

8.2 Setoids

Why we need setoids, informal definition of setoid

The first step before the implementation of Universal Algebra in Type Theory is to have a flexible translation of the intuitive notion of set. Interpreting sets as types would rise some problems: the structure of types is rather rigid and does not allow the formation of subtypes or quotient types. Since we need to define subalgebras and quotient algebras, we are led to consider a more suitable solution. In some version of (extensional) type theory notions of subtype and quotient type are implemented (for example in the Nuprl system, see [CAB$^+$86]), but the version of (intensional) type theory implemented in Coq does not. Nevertheless, a model of extensional type theory inside intensional type theory has been constructed by Martin Hofmann (see [Hof93]). We use a variant of this model, which has already been implemented by Huet and Saïbi in [HS00] and used by Pottier.

The elements of a type are build up using some constructors, and elements of a type are said to be equal when they are convertible. Thus a type cannot be defined by a predicate over an other type (subtyping) or by redefining the equality (quotienting). We allow ourselves to be more liberal with equality by defining a setoid to be a pair formed by a set and an equivalence relation over it. Thus we can quotient a setoid by just changing the equivalence relation. Subsetoids are obtained by quotienting Σ-types, that is, if $\mathcal{S} = \langle A, =_{\mathcal{S}} \rangle$ is a setoid and P is a predicate over A (that is closed under $=_{\mathcal{S}}$), we can define the subsetoid determined by P to be $\mathcal{S}^P = \langle (\Sigma x : A.(P\ x)), =_{\mathcal{S}^P} \rangle$ where $\langle a_1, p_1 \rangle =_{\mathcal{S}^P} \langle a_2, p_2 \rangle$ iff $a_1 =_{\mathcal{S}} a_2$.

Since we explicitly work with equivalence relations all the definitions on setoids (predicates over setoids, relations between setoids, setoids functions) must be required to be invariant under the given equality.

We gave an analysis of the notion of setoid in Chapter 5. Here we recall the main definitions and properties of (total) setoids.

Formal definition of setoid

Definition 8.2.1

$$\mathsf{Record\ Setoid\ :\ Type\ :=\ setoid}\ \begin{cases} \mathsf{s\_el} & : \mathsf{Set} \\ \mathsf{s\_eq} & : \mathsf{s\_el} \to \mathsf{s\_el} \to \mathsf{Prop} \\ \mathsf{s\_proof} & : (\mathsf{Equiv\ s\_eq}) \end{cases}$$

where (Equiv s_eq) *is the proposition stating that* s_eq *is an equivalence relation over the set* s_el.

We often identify a setoid $\mathcal{S}$ with its carrier set ($\mathsf{s\_el}\ \mathcal{S}$). In Coq this identification is realized through the use of *implicit coercions* (see [Saï97]). Similar implicit coercions are also used to identify an algebraic structure with its carrier. If $a, b : \mathcal{S}$, that is, as we said, $x, y : (\mathsf{s\_el}\ \mathcal{S})$, we use the simple notation $x = y$ in place of ($\mathsf{s\_eq}\ x\ y$); in Coq an infix operator `[=]` is defined so we can write `x [=] y`. As a general methodology, if op is a set operator, we use the notation `[op]` for the corresponding setoid operator. Whenever we want to stress the setoid in which the equality holds (two setoids may have the same elements but different equalities) we write $x =_{\mathcal{S}} y$.

Properties and constructions on setoids

As we have mentioned above, we have to be careful when dealing with constructions on setoids. For example, predicates, relations, and functions should be invariant under the given equality.

Definition 8.2.2 *A predicate P over the carrier of a setoid $\mathcal{S}$, that is, $P : (\mathsf{s\_el}\ \mathcal{S}) \to \mathsf{Prop}$ is said to be* well defined *(with respect to $=_{\mathcal{S}}$) if $(\forall x, y : \mathcal{S}) x =_{\mathcal{S}} y \to (P\ x) \to (P\ y)$. The type of* setoid predicates *over $\mathcal{S}$ is the record type*

$$\mathsf{Record\ Setoid\_predicate\ :\ Type\ :=}\ \mathsf{setoid\_predicate}\ \begin{cases} \mathsf{sp\_pred} & : \mathcal{S} \to \mathsf{Prop} \\ \mathsf{sp\_proof} & : (\mathsf{Predicate\_well\_defined\ sp\_pred}) \end{cases}$$

where (Predicate_well_defined sp_pred) *is the above property.*

Definition 8.2.3 *A relation R on the carrier of a setoid $\mathcal{S}$, that is, $R : (\mathsf{s\_el}\ \mathcal{S}) \to (\mathsf{s\_el}\ \mathcal{S}) \to \mathsf{Prop}$ is said to be* well defined *(with respect to $=_{\mathcal{S}}$) if*

$$(\forall x_1, x_2, y_1, y_2 : \mathcal{S}) x_1 =_{\mathcal{S}} x_2 \to y_1 =_{\mathcal{S}} y_2 \to (R\ x_1\ y_1) \to (R\ x_2\ y_2).$$

The type of setoid relations *on $\mathcal{S}$ is the record type*

$$\mathsf{Record\ Setoid\_relation\ :\ Type\ :=}\ \mathsf{setoid\_relation}\ \begin{cases} \mathsf{sr\_rel} & : \mathcal{S} \to \mathcal{S} \to \mathsf{Prop} \\ \mathsf{sr\_proof} & : (\mathsf{Relation\_well\_defined\ sr\_rel}). \end{cases}$$

By declaring two implicit coercions we can use setoid predicates and relations as if they were regular predicates and relations, that is, if P : (Setoid_predicate $\mathcal{S}$) and x : $\mathcal{S}$ then $(P\ x)$ is a shorthand notation for $((\mathsf{sp\_pred}\ P)\ x)$: Prop and if R : (Setoid_relation $\mathcal{S}$) and $x, y : \mathcal{S}$ then $(R\ x\ y)$ is a shorthand notation for $((\mathsf{sr\_rel}\ R)\ x\ y)$: Prop.

As we have mentioned in the informal discussion, subsetoids can be defined by a setoid predicate by giving a suitable equivalence relation over a Σ-type.

Definition 8.2.4 *Let $\mathcal{S}$ be a setoid and P a setoid predicate over it. Then* the subsetoid of $\mathcal{S}$ separated by P *is the setoid $\mathcal{S}|P$ that has carrier $\Sigma x : \mathcal{S}.(P\ x)$ and equality relation $\langle a_1, p_1\rangle =_{\mathcal{S}|P} \langle a_1, p_1\rangle \iff a_1 =_{\mathcal{S}} a_2$.*

Even easier is the definition of a quotient of a setoid by an equivalence (setoid) relation. It is enough to substitute such relation in place of the original equality.

Definition 8.2.5 *Let $\mathcal{S}$ be a setoid and Eq :* (Setoid_relation $\mathcal{S}$) *such that* (sr_rel Eq) *is an equivalence relation on* (s_el $\mathcal{S}$)*. Then the* quotient setoid *$\mathcal{S}/Eq$ is the setoid with carrier set* (s_el $\mathcal{S}$) *and equality relation Eq.*

Notice that the notion of quotient setoid is different from the notion of quotient set in set theory: the elements of $\mathcal{S}/Eq$ are not equivalence classes, as in set theory, but they are exactly the same as the elements of $\mathcal{S}$.

Definition 8.2.6 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two setoids. Their* product *is the setoid $\mathcal{S}_1[\times]\mathcal{S}_2$ with carrier set* (s_el $\mathcal{S}_1$) $\times$ (s_el $\mathcal{S}_2$) *and equality relation*

$$\langle x_1, x_2\rangle =_{\mathcal{S}_1[\times]\mathcal{S}_2} \langle y_1, y_2\rangle \iff x_1 =_{\mathcal{S}_1} y_1 \wedge x_2 =_{\mathcal{S}_1} y_2.$$

Definition 8.2.7 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two setoids. The setoid of functions from $\mathcal{S}_1$ to $\mathcal{S}_2$ is the setoid $\mathcal{S}_1[\rightarrow]\mathcal{S}_2$ with carrier set the type of those functions between the two carriers that are well-defined with respect to the setoid equalities*

$$\mathsf{Record}\ \mathcal{S}_1[\rightarrow]\mathcal{S}_2 := \mathsf{setoid\_function}\left\{\begin{array}{ll}\mathsf{s\_function} & : \mathcal{S}_1 \rightarrow \mathcal{S}_2\\ \mathsf{s\_fun\_proof} & : (\mathsf{fun\_well\_defined}\ \mathsf{s\_function})\end{array}\right.$$

where (fun_well_defined s_function) *is the proposition $(\forall x_1, x_2 : \mathcal{S}_1) x_1 =_{\mathcal{S}_1} x_2 \rightarrow (\mathsf{s\_function}\ x_1) =_{\mathcal{S}_2} (\mathsf{s\_function}\ x_2)$, and $f =_{\mathcal{S}_1[\rightarrow]\mathcal{S}_2} g$ is the extensional equality relation $(\forall x : \mathcal{S}_1)(f\ x) =_{\mathcal{S}_2} (g\ x)$.*

In a similar way we can define other constructions on setoids and define operators on them (see the source files for a complete list).

8.3 Signatures and Algebras

Using the development of setoids from the previous section as our notion of sets we can now translate Universal Algebra into type theory. We use as a guide the

chapter on Universal Algebra by K. Meinke and J. V. Tucker from the *Handbook of Logic in Computer Science* ([MT92]). We differ from that work only in that we consider just finite signatures (so that they can be implemented by lists) and we do not require that carrier sets are non-empty. This second divergence is justified by the difference between first order predicate logic (which is the logic usually employed to reason about algebraic structures), that always assumes the universe of discourse to be non-empty, and type theory, in which this assumption is not present and, therefore, we can reason about empty structures (about this see also [Bar92], section 5.4).

Definition of signature

We begin by defining the notion of a (many-sorted) signature. A signature is an abstract specification of the carrier sets (called *sorts*) and operations of an algebra, and it is given by the number of sorts n and a list of operation symbols $[f_1, \ldots, f_m]$ where each of the functions f_i must be specified by giving its type, that is, by saying how many arguments the function has, to which one of the sorts each argument belongs and to which sort the result of the application of the operation belongs. Each sort is identified by an element of the finite set $\mathbb{N}_n = \{0, \ldots, n-1\}$ (in the Coq implementation, $\mathbb{N}_n$ is represented by `(Finite n)` and its elements are represented by `n}-(0)`, `n}-(1)`, ..., `n}-(n-1)`). It is possible to use a finite type of labels in place of $\mathbb{N}_n$, so that the labels have meaningful names that help the intuition. Here, we use $\mathbb{N}_n$ instead, to avoid the extra complication of managing the type of labels.

As an example suppose we want to define an algebraic structure with two sorts $\langle nat, bool; O, S, true, false, eq\rangle$ to model the natural numbers and booleans together with a test function for equality with boolean values. So we want that

$nat, bool$	: Setoid		
O	: nat	$true, false$	: $bool$
S	: $nat \to nat$	eq	: $nat \times nat \to bool$

So in this case $n = 2$, the index of the sort nat is 0, the index of the sort $bool$ is 1, and the types of constants and functions are

O	$\Rightarrow \langle [], 0\rangle$	(no arguments and result in nat)
S	$\Rightarrow \langle [0], 0\rangle$	(one argument from nat, result in nat)
$true$	$\Rightarrow \langle [], 1\rangle$	(no arguments, result in $bool$)
$false$	$\Rightarrow \langle [], 1\rangle$	(no arguments, result in $bool$)
eq	$\Rightarrow \langle [0, 0], 1\rangle$	(two arguments from nat, result in $bool$)

Definition 8.3.1 *Let $n : \mathbb{N}$ be a fixed natural number. Let $\mathsf{Sort}_n \equiv \mathbb{N}_n$. A* function type *is a pair $\langle args, res\rangle$, where $args$ is a list of elements of Sort_n (indicating the type of the arguments of the function) and res is an element of Sort_n (indicating the type of the result). So in type theory we define the type of function types as* $(\mathsf{Function\_type}\ n) := (\mathsf{List}\ \mathsf{Sort}_n) \times \mathsf{Sort}_n$.

Definition 8.3.2 *A* signature *is a pair* $\langle n, fs\rangle$ *where* $n : \mathbb{N}$ *and* $fs \equiv [f_1, \ldots, f_m]$ *is a list of function types. We represent it in type theory by a record type:*

$$\mathsf{Record\ Signature\ : Set\ :=}\ \mathsf{signature}\left\{\begin{array}{ll}\mathsf{sorts\_num} & : \mathbb{N}\\ \mathsf{function\_types} & : (\mathsf{List\ (Function\_type\ sorts\_num)})\end{array}\right.$$

The signature of natural numbers and booleans is then defined as $\sigma =$ (signature 2 $[\langle[],0\rangle, \langle[0],0\rangle, \langle[],1\rangle, \langle[],1\rangle, \langle[0,0],1\rangle]$).

Definition of algebra

Let σ : Signature, we want to define the notion of a σ-algebra. To define such a structure we need to interpret the sorts as setoids, and the function types as setoid functions. Suppose $\sigma = \langle n, [f_1, \ldots, f_s]\rangle$. The interpretation of the sorts is a family of n setoids: Sorts_interpretation $:= \mathbb{N}_n \to$ Setoid. So let us assume that $sorts$: Sorts_interpretation, and define the interpretation of $f_1, \ldots, f_n$. There are several ways of defining the type of a function, depending on how the arguments are given. Suppose $f = \langle[a_1, \ldots, a_k], r\rangle$ is a function type. If $x_j : (sorts\ a_j)$ for $j = 1, \ldots, k$, then we would like the interpretation of f, $\|f\|$, to be applicable directly to its arguments, $(\|f\|\ x_1\ \ldots x_k) : (sorts\ r)$. This means that $\|f\|$ should have the curried type $(sorts\ a_1)[\to]\ldots[\to](sorts\ a_k)[\to](sorts\ r)$ This type may be defined by using a general construction to define types of curried functions with arity and types of the arguments as parameters. This is done by the function

$$\mathsf{Curry\_type\_setoid} : (n : nat)(\mathbb{N}_n \to \mathsf{Setoid}) \to \mathsf{Setoid} \to \mathsf{Setoid}$$

such that if n is a natural number, $A : \mathbb{N}_n \to$ Setoid is a family of setoids defining the type of the arguments, and B : Setoid is the type of the result, then

$$(\mathsf{Curry\_type\_setoid}\ n\ A\ B) = (A\ 0)[\to]\ldots[\to](A\ n-1)[\to]B$$

So in the previous example the type of $\|f\|$ may be defined as

$$(\mathsf{Curry\_type\_setoid}\ k\ [i : \mathbb{N}_k](sorts\ a_i)\ (sorts\ r))$$

But this representation is difficult to use when reasoning abstractly about functions, for example, if we want to prove general properties of the functions which do not depend on the arity. In this situation it is better to see the function having just one argument containing all the x_j's. We can do that by giving the arguments as k-tuples or as functions indexed on a finite type. We choose this second option. So we represent the arguments as an object of type $(j : \mathbb{N}_k)(sorts\ a_j)$. Then the interpretation of the function f could have the type $((j : \mathbb{N}_k)(sorts\ a_j)) \to (sorts\ r)$. This is still not completely correct. Since the sorts are setoids, the interpretation of the functions must preserve the setoid equality. With the aim of formulating this condition, we first make the type of arguments $(j : \mathbb{N}_k)(sorts\ a_j)$ into a setoid by stating that two elements $args_1, args_2$ are equal if they are extensionally equal.

Definition 8.3.3 *Let* $k : \mathbb{N}$, $A : \mathbb{N}_k \to$ Setoid. *Then* (FF_setoid k A) *is the setoid that has carrier* $(j : \mathbb{N}_k)(A\ j)$ *and equality relation*

$$(args_1 =_{(\mathsf{FF\_setoid}\ k\ A)} args_2) \iff (\forall j : \mathbb{N}_k)((args_1\ j) =_{(A\ j)} (args_2\ j))$$

We can now interpret a function type and a list of function types.

Definition 8.3.4 *Let* $f = \langle [a_1, \ldots, a_k], r\rangle$ *be a function type. Then*

$$\begin{array}{l}(\mathsf{Function\_type\_interpretation}\ n\ sorts\ f) :=\\ (\mathsf{FF\_setoid}\ k\ [i : \mathbb{N}_k](sorts\ a_k))[\to](sorts\ r)\end{array}$$

A list of function types is interpreted by the operator

$$\begin{array}{l}(\mathsf{Function\_list\_interpretation}\ n\ sorts) :\\ (\mathsf{List}\ (\mathsf{Function\_type}\ n)) \to \mathsf{Setoid}\end{array}$$

where the carrier of (Function_list_interpretation n *sorts* $[f_1, \ldots, f_s]$) *is*

$$[i : \mathbb{N}_s](\mathsf{Function\_type\_interpretation}\ n\ sorts\ f_i)$$

(we do not need to take into consideration how the equality relation is defined).

This is the way in which functions are represented in the algebra. Whenever we want to have them in the curried form we can apply a conversion operator

$$\mathsf{fun\_arg\_to\_curry} : ((\mathsf{FF\_setoid}\ k\ A)[\to]B) \to (\mathsf{Curry\_type\_setoid}\ k\ A\ B).$$

The inverse conversion is performed by the operator curry_to_fun_arg.

Eventually, the type of σ-algebras can be defined as

Definition 8.3.5 *The type of* algebras over the signature σ *is the record type*

$$\begin{array}{l}\mathsf{Record}\ (\mathsf{Algebra}\ \sigma)\ : \mathsf{Type}\ :=\\ \qquad \mathsf{algebra}\left\{\begin{array}{ll}\mathsf{sorts} & : (\mathsf{Sorts\_interpretation}\ (\mathsf{sorts\_num}\ \sigma))\\ \mathsf{functions} & : (\mathsf{Function\_list\_interpretation}\\ & \quad (\mathsf{sorts\_num}\ \sigma)\ \mathsf{sorts}\ (\mathsf{function\_types}\ \sigma))\end{array}\right.\end{array}$$

The type of arguments corresponding to the i*-th function of the signature* σ *in an algebra* $\mathcal{A}$ *are also indicated by* (Fun_arg_arguments $\mathcal{A}$ i).

If $\sigma \equiv \langle n, [f_0, \ldots, f_{m-1}]\rangle$ and $\mathcal{A} : (\mathsf{Algebra}\ \sigma)$, we indicate by $f_{i\mathcal{A}}$ the interpretation of the ith function symbol $f_{i\mathcal{A}} \equiv (\mathsf{functions}\ \sigma\ \mathcal{A}\ i)$ for every $i : \mathbb{N}_m$. As an example let us define a σ-algebra for the signature considered before, interpreting the two sorts as the setoids of natural numbers and booleans (in this case the equivalence relation is trivially Leibniz equality). Suppose we have already defined

$\mathcal{N}, \mathcal{B}$	: Setoid		
0	: $\mathcal{N}$	true, false	: $\mathcal{B}$
S	: $\mathcal{N}[\to]\mathcal{N}$	Eq	: $\mathcal{N}[\to]\mathcal{N}[\to]\mathcal{B}$

Then we can give the interpretation of the sorts

```
Srt : (Sorts_interpretation 2)
(Srt 0) := 𝒩
(Srt 1) := ℬ
```

and of the functions

```
Fun : (Function_list_interpretation Srt (function_types sigma))
 (Fun 0) := (curry_to_fun_arg 0)
 (Fun 1) := (curry_to_fun_arg S)
 (Fun 2) := (curry_to_fun_arg true)
 (Fun 3) := (curry_to_fun_arg false)
 (Fun 4) := (curry_to_fun_arg Eq).
```

Then we can define the σ-algebra

```
nat_bool_alg := (algebra σ Srt Fun) : (Algebra σ).
```

8.4 Term algebras

Informal definition of term algebras

A class of algebras of special interest is that of *term algebras*. The sorts of a term algebra are the terms freely generated by the function symbols of the signature. For example, in the signature defined above we would have that the expressions O, $S(O)$, $S(S(O))$ are terms of the first sort, while $true$, $false$, $eq(O, S(O))$ are terms of the second. In general, given a signature $\sigma = \langle n, [f_1, \ldots, f_m]\rangle$, the algebra of terms has carriers T_i, for $i : \mathbb{N}_n$, whose elements have the form $f_j(t_1, \ldots, t_k)$ where $j : \mathbb{N}_m$, the type of f_j is $\langle [a_1, \ldots, a_k], r\rangle$, $t_1, \ldots, t_k$ belong to the term sorts $T_{a_1}, \ldots, T_{a_k}$, respectively, and the resulting term is in the sort T_r.

Similarly we can define an algebra of open terms or expressions, that is, terms in which variables can appear. We start by a family of sets of variables X_i for $i : \mathbb{N}_n$, and we construct terms by application of the function symbols as before.

Problem: the uniform definition

In type theory this can be easily modeled by inductively defined types whose constructors correspond to the functions of the signature. For example, the

sorts of terms of the previous signature are the (mutually) inductive types

```
Inductive nat_term : Set :=
   o_symb : nat_term
   s_symb : nat_term → nat_term
end,

Inductive bool_term : Set :=
    t_symb : bool_term
    f_symb : bool_term
   eq_symb : nat_term → nat_term → bool_term
end.
```

If the signature is single-sorted, a simple inductive definition gives the type of terms; if it is many-sorted, we have to use mutually inductive definitions. In this way we can define the types of sorts for any specific signature, but it is not possible to define it parametrically. We would like to define term algebras as a higher type function

$$\mathsf{Term\_algebra} : (\sigma : \mathsf{Signature})(\mathsf{Algebra}\ \sigma)$$

that associates the corresponding term algebra to each signature. In order to do this, we would need mutually inductive definitions in which the number of sorts and constructors and the type of the constructors are parametric. Such a general form of inductive definition is not available in current implementations of type theory (like Coq), so we have to look for a different solution. The general form of induction could be added to future implementations of Coq.

Discussion on possible solutions

The problem is more general and regards the definition of families of inductive types in which every element of the family is a correct inductive type, but the family itself cannot be defined. In the general case we have a family of set operators indexed on a set A, $\Phi : A \to (\mathsf{Set} \to \mathsf{Set})$ and we want to define a family of inductive types each of which is the minimal fixed point of the corresponding operator, that is, we want a family $I : A \to \mathsf{Set}$ such that for every $a : A$, $(I\ a)$ is the minimal fixed point of $(\Phi\ a)$. In type theory it is possible to define the minimal fixed point of a set operator $\Psi : \mathsf{Set} \to \mathsf{Set}$ if and only if the set operator is strictly positive, that is, in the expression $(\Psi\ X)$, where $X : \mathsf{Set}$, X occurs only to the right of arrows. But it may happen that even if for every concrete element (closed term) a of the set A, the operator $(\Phi\ a)$ is strictly positive or reduces to a strictly positive operator, this does not hold for open terms, that is, if $x : A$ is a variable $(\Phi\ x)$ does not satisfy the strict positivity condition. There are several possibilities to overcome this difficulty. A thorough analysis of this subject was given in Chapter 4. Here we adopt a solution that represents every inductive type by a type of trees.

Solution using Wellorderings

$\mathbb{W}$ types are a type theoretic implementation or the notion of well orderings as well-founded trees. They were introduced by Per Martin-Löf in [ML82] (see also [ML84] and [NPS90], chapter 15). We introduced wellorderings in Chapter 2, Section 2.11, and used them in Chapter 4, Section 4.4. Let us recall the notion. Suppose that we want to define a type of trees such that the nodes of the trees are labeled by elements of the type B, and, for each node labeled by an element $b : B$, the branches stemming from the node are labeled by the elements of a set $(C\ b)$, that is, the b-node has as many branches as the elements of $(C\ b)$. Then the $\mathbb{W}$ type constructor has two parameters: a type $B : \mathsf{Set}$ and a family of types $C : B \to \mathsf{Set}$. To define a new element of the type $(\mathbb{W}\ B\ C)$ we have to specify the label of the root by an element $b : B$ and, for each branch, that is, for every element $c : (C\ b)$, the corresponding subtree; this is done by giving a function $h : (C\ b) \to (\mathbb{W}\ B\ C)$.

Formally we can define $(\mathbb{W}\ B\ C)$ in the Calculus of Inductive Constructions (see [CP90] and [Gim98]) as the inductive type $(\mathbb{W}\ B\ C)$ with one constructor $\mathsf{wtree} : (b : B)((C\ b) \to (\mathbb{W}\ B\ C)) \to (\mathbb{W}\ B\ C)$. As for any inductive definition, we automatically get principles of recursion and induction associated with the definition. If $(C\ b)$ is infinite for some $b : B$, we get transfinite induction.

We can use this construction to define term algebras for single-sorted signatures, representing a term by its syntax tree. We choose B to be the set of function symbols of the signature (or just $\mathbb{N}_m$ where m is the number of the functions), and $(C\ f) = \mathbb{N}_{k_f}$ where k_f is the arity (number of arguments) of the function symbol f.

For example, let us take the signature $\langle 1, [\langle [], 0\rangle, \langle [0], 0\rangle, \langle [0,0], 0\rangle]\rangle$ describing a structure with one sort, one constant, one unary operation and one binary operation. Let us indicate the three functions by f_0 (the constant), f_1 (the unary operation) and f_2 (the binary operation). The type of terms is represented by the type $(\mathbb{W}\ \mathbb{N}_3\ C)$ where

$$C := [i : \mathbb{N}_3]\mathsf{Cases}\ i\ \mathsf{of}\ \begin{cases} 0 \mapsto \mathbb{N}_0 \\ 1 \mapsto \mathbb{N}_1 \\ 2 \mapsto \mathbb{N}_2. \end{cases}$$

Then the term $f_2(f_1(f_0), f_2(f_0, (f_1(f_0))))$ is represented by the tree

or, formally, by the element of $(\mathbb{W}\ \mathbb{N}_3\ C)$

$$
\begin{array}{l}
\mathsf{wtree}\ 2\ \ [i : \mathbb{N}_2] \\
\qquad \mathsf{Cases}\ i\ \mathsf{of} \\
\qquad \left\{ \begin{array}{l}
0 \mapsto (\mathsf{wtree}\ 1\ \ [j : \mathbb{N}_1] \\
\qquad\qquad\qquad \mathsf{Cases}\ j\ \mathsf{of} \\
\qquad\qquad\qquad 0 \mapsto (\mathsf{wtree}\ 0\ [k : \mathbb{N}_0]\mathsf{Cases}\ k\ \mathsf{of})\ ) \\
1 \mapsto (\mathsf{wtree}\ 2\ \ [l : \mathbb{N}_2] \\
\qquad\qquad\qquad \mathsf{Cases}\ l\ \mathsf{of} \\
\qquad\qquad\qquad \left\{ \begin{array}{l}
0 \mapsto (\mathsf{wtree}\ 0\ [k : \mathbb{N}_0]\mathsf{Cases}\ k\ \mathsf{of}) \\
1 \mapsto (\mathsf{wtree}\ 1\ \ [j : \mathbb{N}_1] \\
\qquad\qquad\qquad\qquad \mathsf{Cases}\ j\ \mathsf{of} \\
\qquad\qquad\qquad\qquad 0 \mapsto (\mathsf{wtree}\ 0\ [k : \mathbb{N}_0]\mathsf{Cases}\ k\ \mathsf{of})\ ).
\end{array} \right.
\end{array} \right.
\end{array}
$$

(Of course, for practical uses we have to define some syntactic tools, to spare the user the pain of writing such terms.)

General Tree Types

To deal with multi-sorted signatures we need to generalize the construction. The General Trees type constructor that we use is very similar to that introduced by Kent Petersson and Dan Synek in [PS89] (see also [NPS90], chapter 16). We already introduced general tree types in Chapter 2, Section 2.11.

In the multi-sorted case we have to define not just one type of terms, but n types, if n is the number of sorts. These types are mutually inductive. So we define a family $\mathbb{N}_n \to \mathsf{Set}$. In general, we consider the case in which we want to define a family of tree types indexed on a given type A, so the elements of A are though of as indexes for the sorts. For what regards the functions, besides their arity, we have to take into account from which sort each argument comes and to which sort the result belongs. As before, we have a type B of indexes for the functions. To each $b : B$ we have to associate, as before, a set $(C\ b)$ indexing its arguments. But now we must also specify the type of the arguments: to each $c : (C\ b)$ we must associate a sort index (the sort of the corresponding argument) $(g\ b\ c) : A$. Therefore, we need a function $g : (b : B)(C\ b) \to A$. Furthermore, we must specify to which sort the result of the application of the function b belongs, so we need an other function $f : B \to A$. Then in the context

$$
\begin{array}{llll}
A, B & : \mathsf{Set} & f & : B \to A \\
C & : B \to \mathsf{Set} & g & : (b : B)(C\ b) \to A
\end{array}
$$

we define the inductive family of types $(\mathbb{T}\ A\ B\ C\ f\ g) : A \to \mathsf{Set}$ with the constructor (we write $\mathbb{G}$ for $(\mathbb{T}\ A\ B\ C\ f\ g)$)

$$\mathsf{gtree} : (b : B)(\Pi c : (C\ b).(\mathbb{G}\ (g\ b\ c))) \to (\mathbb{G}\ (f\ b)))$$

In the case of a signature $\sigma = \langle n, [f_1, \ldots, f_m]\rangle$ such that for every $i : \mathbb{N}_m$,

$f_i = \langle [a_{i,0}, \ldots, a_{i,k_i-1}], r_i \rangle$ (k_i is the arity of f_i), we have

$$\begin{array}{lll} A & := \mathbb{N}_n & f := [b : B]r_b \\ B & := \mathbb{N}_m & g := [b : B][c : (C\ b)]a_{b,c} \\ (C\ b) & := \mathbb{N}_{k_b} \quad \text{for every } b : B & \end{array}$$

The family of types of terms is $(\mathsf{Term}\ \sigma) := (\mathbb{T}\ A\ B\ C\ f\ g)$.

The problem of intensionality

One problem that arises when using the General Trees constructor to define term algebras is the intensionality of equality. A term (tree) is defined by giving a constructor $b : B$ and a function $h : (c : (C\ b))(\mathsf{Term}\ \sigma\ (g\ b\ c))$. It is possible that two functions $h_1, h_2 : (c : (C\ b))(\mathsf{Term}\ \sigma\ (g\ b\ c))$ are extensionally equal, that is, $(h_1\ c) = (h_2\ c)$ for all $c : (C\ b)$, but not intensionally equal, that is, not convertible. In this case the two trees $(\mathsf{gtree}\ b\ h_1)$ and $(\mathsf{gtree}\ b\ h_2)$ are intensionally distinct. But we want two terms obtained by applying the same function to equal arguments to be equal. Since algebras are required to be setoids, and not just sets, we can solve this problem by defining an inductive equivalence relation on the types of terms that captures this extensionality,

$$\begin{array}{l} \mathsf{Inductive\ tree\_eq} : (a : A)(\mathsf{Term}\ \sigma\ a) \to (\mathsf{Term}\ \sigma\ a) \to \mathsf{Prop} := \\ \quad \mathsf{tree\_eq\_intro} : \ (b : B)(h_1, h_2 : (c : (C\ b))(\mathsf{Term}\ \sigma\ (g\ b\ c))) \\ \qquad\qquad\qquad\qquad (\forall c : (C\ b).(\mathsf{tree\_eq}\ (g\ b\ c)\ (h_1\ c)\ (h_2\ c))) \\ \qquad\qquad\qquad\qquad \to (\mathsf{tree\_eq}\ (f\ b)\ (\mathsf{gtree}\ b\ h_1)\ (\mathsf{gtree}\ b\ h_2)) \\ \mathsf{end.} \end{array}$$

Now we would like to prove that $(\mathsf{tree\_eq}\ a)$ is an equivalence relation on $(\mathsf{Tree}\ \sigma\ a)$. Unfortunately, no proof of transitivity could be found. The problem can be formulated and generalized in the following way. If we have an inductive family of types and a generic element of one of the types in the family, it is not generally possible to prove an inversion result stating that the form of the element is an application of one of the constructors corresponding to that type. This was proved for the first time by Hofmann and Streicher in the case of the equality types (see [HS94]). Therefore, we just take $=_{\mathsf{Term}}$ to be the transitive closure of $\mathsf{tree\_eq}$.

Functions

We have constructed an interpretation of the sorts of a signature in setoids of terms as syntax trees. We still have to interpret the functions. This is not difficult, given the way we defined the function interpretation. The functions are associated to the elements of the type $B = \mathbb{N}_m$. Given an element $b : B$, we have a function

$$(\mathsf{gtree}\ b) : (\Pi c : (C\ b).(\mathsf{Term}\ \sigma\ (g\ b\ c))) \to (\mathsf{Term}\ \sigma\ (f\ b))$$

It is straightforward to prove that it preserves the setoid equality, so it has the right type to be the interpretation of the function symbol b. Let us call (functions_interpretation σ) the family of setoid functions with underlying type functions (gtree b). We then obtain the algebra of terms

Term_algebra : $(\sigma : \mathsf{Signature})(\mathsf{Algebra}\ \sigma)$
(Term_algebra σ) := (algebra σ (Term σ) (functions_interpretation σ)).

Expressions Algebras

We defined algebras whose elements are closed terms constructed from the function symbols of the signature σ. It is also very important to have algebras of open terms, or expressions, where free variables can appear. To do this, we modify the definition of term algebras allowing a set of variables alongside that of functions. So in the construction of syntax trees some leaves may consist of variable occurrences. We assume that every sort has a countably infinite number of variables, so the set of variables is $\mathsf{Var} := \mathbb{N}_n \times \mathbb{N}$. Then a variable is a pair $\langle s, n\rangle$ where s determines the sorts to which it belongs and n says that it is the n-th variable of that sort. Variables are treated as constants, that is, as function symbols of zero arity. In the definition of the term algebra we modify the set B of constructors: $B := \mathbb{N}_m + \mathsf{Var}$ and also the family C giving the types of the subtrees in such a way that $(C\ (\mathsf{inr}\ v))$ is the empty set for every variable v, while $(C\ (\mathsf{inl}\ j)) := \mathbb{N}_{k_j}$ as before. The rest of the definition remains the same. We may also abstract from the actual set of variables and use any family $X : \mathbb{N}_n \to \mathsf{Set}$ as the family of sets of variables. The closed terms are then a particular case obtained by taking $(X\ i) := \emptyset$ for every $i : \mathbb{N}_n$, and the previous case is obtained by taking $(X\ i) := \mathbb{N}$. We obtain the algebra of expressions

$$\mathsf{Expression\_algebra} : (\sigma : \mathsf{Signature})(\mathsf{Algebra}\ \sigma).$$

8.5 Quotients, subalgebras, homomorphisms

Congruences and quotients

If σ is a signature and $\mathcal{A}$ a σ-algebra, we call congruence a family of equivalence relations over the sorts of $\mathcal{A}$ that is consistent with the operations of the algebra, that is, when we apply one of the operations to arguments that are in the relation, we obtain results that are in the relation. Such condition is rendered in type theory by the following

Definition 8.5.1 *Let* $\sigma \equiv \langle n, [f_0, \ldots, f_{m-1}]\rangle$: Signature *with function types* $f_i := \langle [a_{i,0}, \ldots, a_{i,h_i}], r_i\rangle$ *for* $i : \mathbb{N}_m$, *and* $\mathcal{A}$: (Algebra σ). *A family of relations on the sorts* $A :=$ (sorts $\mathcal{A}$), $(\equiv_s)$: (Setoid_relation $(A\ s)$) *for* $s : \mathbb{N}_n$, *satisfies the* substitutivity condition (Substitutivity $(\equiv)$) *if and only if*

$(\forall i : \mathbb{N}_m)(\forall args_1, args_2 : (\mathsf{Fun\_arg\_arguments}\ \mathcal{A}\ i))$
$((\forall j : \mathbb{N}_{h_i})(args_1\ j) \equiv_{a_{i,j}} (args_2\ j)) \to (f_{i\mathcal{A}}\ args_1) \equiv_{r_i} (f_{i\mathcal{A}}\ args_2)$.

The type of congruences *over a σ-algebra $\mathcal{A}$ is the record type*

$$\begin{array}{l}\mathsf{Record\ Congruence\ :\ Type\ :=} \\ \qquad \mathsf{congruence} \left\{ \begin{array}{ll} \mathsf{cong\_relation} & : (s : \mathbb{N}_n)(\mathsf{Setoid\_relation}\ (A\ s)) \\ \mathsf{cong\_equiv} & : (s : \mathbb{N}_n)(\mathsf{Equiv}\ (\mathsf{cong\_relation}\ s)) \\ \mathsf{cong\_subst} & : (\mathsf{Substitutivity\ cong\_relation}) \end{array} \right.\end{array}$$

So a congruence on $\mathcal{A}$ has the form (congruence *rel eqv sbs*) where *rel* is a family of setoid relations on the sorts of $\mathcal{A}$, *eqv* is a proof that every element of the family is an equivalence relation and *sbs* is a proof that the family satisfies the substitutivity condition.

Given a congruence over an algebra we can construct the quotient algebra. In classic Universal Algebra this is done by taking as sorts the sets of equivalence classes with respect to the congruence. In type theory, as we have already said about quotients of setoids, the quotient has exactly the same carriers, but we replace the equality relation. The substitutivity condition guarantees that what we obtain is still an algebra.

Lemma 8.5.2 *Let σ : Signature, $\mathcal{A}$: (Algebra σ) and $(\equiv)$: (Congruence σ $\mathcal{A}$). If we consider the family of setoids obtained by replacing each $=_{(\mathsf{sorts}\ \mathcal{A}\ s)}$ by $\equiv_s$ the functions of $\mathcal{A}$ are still well defined. We can therefore define the* quotient algebra *$\mathcal{A}/_\sigma \equiv$ as the algebra whose carrier sorts are the setoid quotients of the carriers of $\mathcal{A}$ by $\equiv$, and whose operations are the same as those of $\mathcal{A}$.*

Subalgebras

The definition of subalgebra can be given in the same spirit of the definition of quotient algebras.

Definition 8.5.3 *Let $\mathcal{A}$: (Algebra σ) and $\mathcal{P}_s$: (Setoid_predicate (sorts $\mathcal{A}$ s)) a family of predicates on the sorts of $\mathcal{A}$. We say that $\mathcal{P}$ is* closed under the functions *of $\mathcal{A}$ if*

$$\begin{array}{l}(\forall i : \mathbb{N}_m)(\forall args : (\mathsf{Fun\_arg\_arguments}\ \mathcal{A}\ i)) \\ ((\forall j : \mathbb{N}_{h_i})(\mathcal{P}_{a_j}\ (args\ j))) \to (\mathcal{P}_{r_j}\ (f_{i\mathcal{A}}\ args)).\end{array}$$

Definition 8.5.4 *The* subalgebra *$\mathcal{A}|_\sigma\mathcal{P}$ is the σ-algebra with sorts* (sorts $\mathcal{A}$ s)$|\mathcal{P}_s$ *and functions the restrictions of the functions of $\mathcal{A}$.*

Notice that the restrictions of the functions of $\mathcal{A}$ to the subsetoids (sorts $\mathcal{A}$ s)$|\mathcal{P}_s$ are well-defined because $\mathcal{P}$ is closed under function application. The proof of this fact gives the proof of $(\mathcal{P}_{r_j}\ (f_{i\mathcal{A}}\ args))$ and therefore allows the construction of a well-typed element of the Σ-type which is the carrier of the subsetoid.

Homomorphisms

Given a signature σ : Signature and two σ-algebras $\mathcal{A}$ and $\mathcal{B}$, we want to define the notion of homomorphism between $\mathcal{A}$ and $\mathcal{B}$. Informally, a homomorphism

is a family of functions $\phi_s : (A\ s) \to (B\ s)$, where $s : \mathbb{N}_n$ and A and B are the families of sorts of $\mathcal{A}$ and $\mathcal{B}$ respectively, that commutes with the interpretation of the functions of σ. That means that if f is one of the function types of σ and $a_1, \ldots, a_k$ are elements of the algebra A, belonging to the sorts prescribed by the types of the arguments of f, then, suppressing the index i in ϕ_i, $(\phi\ (\|f\|_{\mathcal{A}}\ a_1\ \ldots\ a_k)) = (\|f\|_{\mathcal{B}}\ (\phi\ a_1)\ \ldots\ (\phi\ a_k))$ where $\|f\|_{\mathcal{A}}$ indicates the curried version of the interpretation of the function type f in the algebra $\mathcal{A}$.

Formally we have to require, first of all, that ϕ is a family of setoid functions $\phi : (i : \mathbb{N}_n)(A\ i)[\to](B\ i)$. Then, the requirement that ϕ must commute with the functions of the signature must take into account the way we interpreted the function symbols. Let $i : \mathbb{N}_m$ be a function index, and $f_i = \langle [a_{i,0}, \ldots, a_{i,k_i-1}], r_i \rangle$ be the corresponding function type of σ. Assume we have an argument function for $f_{i\mathcal{A}}$, $args_{\mathcal{A}}$: (Fun_arg_arguments $\mathcal{A}$ i). Remember that this is a function that to every $j : \mathbb{N}_{k_i}$ assign an element $(args_{\mathcal{A}}\ j)$: (sorts $\mathcal{A}$ $a_{i,j}$). By applying ϕ to each argument, we obtain an argument function for $f_{i\mathcal{B}}$, $args_{\mathcal{B}} := [j : \mathbb{N}_{k_i}](\phi_{a_{i,j}}\ (args_{\mathcal{A}}\ j)))$: (Fun_arg_arguments $\mathcal{B}$ i). For ϕ to be an homomorphism we must then require that for every function index i the equality $(\phi_{r_i}\ (f_{i\mathcal{A}}\ args_{\mathcal{A}})) =_{(B\ r_i)} (f_{i\mathcal{B}}\ args_{\mathcal{B}})$ holds. Let us call this property (Homomorphic ϕ). Then we can define the type of homomorphisms as the record

$$\begin{array}{l}\mathsf{Record\ Homomorphism\ :\ Set\ :=} \\ \quad \mathsf{homomorphism} \left\{ \begin{array}{ll} \mathsf{hom\_function} & : (i : \mathbb{N}_n)(A\ i)[\to](B\ i) \\ \mathsf{hom\_proof} & : (\mathsf{Homomorphic}\ \phi) \end{array} \right.\end{array}$$

By requiring that the setoid functions ϕ_i are injective, surjective or bijective we get the notions of monomorphism, epimorphism, and isomorphism. We also call endomorphisms (automorphisms) the homomorphisms (isomorphisms) from an algebra A to itself.

Term evaluation

One important homomorphism is evaluation function from the term algebra $\mathcal{T} :=$ (Term_algebra σ) to any σ-algebra $\mathcal{A}$. This homomorphism is unique since the interpretation of all terms is determined by the interpretation of functions. The function term_evaluation can be defined by induction on the tree structure of terms in such a way that $(\mathsf{term\_evaluation}\ (f_{i\mathcal{T}}\ args)) = (f_{i\mathcal{A}}\ args')$ where $args' := [j : \mathbb{N}_{k_i}](\mathsf{term\_evaluation}\ (args\ j))$ and we have suppressed the sort indexes. After proving that term_evaluation is a setoid function (preserves the equality of terms) and that it commutes with the operations of σ, we obtain an homomorphism term_ev : (Homomorphism σ $\mathcal{T}$ $\mathcal{A}$).

Similarly, we can define the evaluation of expressions containing free variables. In this case the function expression_evaluation takes an additional argument α : (Assignment σ $\mathcal{A}$) assigning a value in the right sort of $\mathcal{A}$ to every variable: (Assignment σ $\mathcal{A}$) := $(v : (\mathsf{Var}\ \sigma))(A\ (\pi_1\ v))$. Using this extra argument to evaluate the variables, we can construct as before an homomorphism $\mathsf{expression\_ev}_\alpha$: (Homomorphism σ $\mathcal{E}$ $\mathcal{A}$) where $\mathcal{E} :=$ (Expression_algebra σ).

Kernel of a homomorphism

Associated to every homomorphism of σ-algebras ϕ : (Homomorphism σ $\mathcal{A}$ $\mathcal{B}$) there is a congruence on $\mathcal{A}$ called the *kernel* of ϕ.

Definition 8.5.5 *The* kernel *of the homomorphism ϕ is the family of relations*

$$\begin{array}{ll}(\mathsf{ker\_rel}\ \phi) & : (s : \mathbb{N})(A\ s) \to (A\ s) \to \mathsf{Prop} \\ (\mathsf{ker\_rel}\ \phi\ s\ a_1\ a_2) & \iff (\phi_s\ a_1) =_{(B\ s)} (\phi_s\ a_2)\end{array}$$

Lemma 8.5.6 ker_rel *is a congruence on $\mathcal{A}$.*

The kernel of ϕ is indicated by the standard notation $\equiv^{\phi}$.

We can, therefore, take the quotient $\mathcal{A}/_{\sigma}\ \equiv^{\phi}$ and consider the natural homomorphism between $\mathcal{A}$ and $\mathcal{A}/_{\sigma}\ \equiv^{\phi}$. In classic Universal Algebra this homomorphism associates to every element a in $\mathcal{A}$ the equivalence class $[a]_{\equiv^{\phi}}$. But in our implementation the carriers of $\mathcal{A}$ and $\mathcal{A}/_{\sigma}\ \equiv^{\phi}$ are the same and so the natural homomorphism is just the identity. We only have to verify that it is actually an homomorphism (it preserves the setoid equality and it commutes with the operation of the signature).

Lemma 8.5.7 *For any σ-algebra $\mathcal{A}$ and any congruence $\equiv$ on $\mathcal{A}$, the family of identity functions $[s : \mathbb{N}_n][x : (\mathsf{sorts}\ \sigma\ \mathcal{A}\ s)]x$ is a homomorphism from $\mathcal{A}$ to $\mathcal{A}/_{\sigma}\ \equiv$.*

In the case of $\equiv^{\phi}$ such homomorphism is indicated by nat^{ϕ}.

First homomorphism theorem

Once we have developed the fundamental notions of Universal Algebra in type theory and we have constructed operators to manipulate them, we can prove some standard basic results like the following.

Theorem 8.5.8 (First Homomorphism Theorem) *Let $\mathcal{A}$ and $\mathcal{B}$ be two σ-algebras and ϕ : (*Epimorphism *σ $\mathcal{A}$ $\mathcal{B}$). Then there exists an isomorphism*

$$(\mathsf{ker\_quot\_iso}\ \phi) : (\mathsf{Isomorphism}\ \sigma\ \mathcal{A}/_{\sigma}\ \equiv^{\phi}\ \mathcal{B})$$

such that $(\mathsf{ker\_quot\_iso}\ \phi) \circ \mathsf{nat}^{\phi} = \phi$*, where the equality is the extensional functional equality.*

8.6 Term Algebras (Continuation)

In Section 8.4 and in [Cap99] we defined term algebras as families of tree structures of the kind presented by Petersson and Synek in [PS89] (see also Chapter 16 of [NPS90]). That definition provides all the properties needed from term algebras, but has an heavy overhead, due to the functional nature of constructor arguments, that makes it difficult to work with them. Therefore, now we use

a different definition, based on a less general but lighter construction, that is a generalization to many-sorted algebras of the one used by Ruys in chapter 3 of his thesis [Ruy99] for the single-sorted case. This solution exploits the extension of the strict positivity condition implemented in Coq. See Chapter 4 for an exposition of this notion of positivity.

Let $\sigma = \langle n; [f_1, \ldots, f_m]\rangle$ be a signature, where each of the function specifications f_i has the form

$$f_i = \langle [a_{i,1}, \ldots, a_{i,k_i}], r_i\rangle$$

with $a_{i,1}, \ldots, a_{i,k_i}, r_i : \mathbb{N}_n$. ($\mathbb{N}_n$ is the type with n elements.)

A *σ-algebra* is a specific instantiation of the signature σ, that is, an object of the form

$$\mathcal{A} = \langle A_1, \ldots A_n; F_1, \ldots, F_m\rangle$$

where $A_1, \ldots, A_n$ are setoids, called the *sorts* of $\mathcal{A}$, and $F_1, \ldots, F_m$ are setoid functions of the type corresponding to the specifications $f_1, \ldots, f_m$:

$$F_i : (A_{a_{i,1}}[\times]\cdots[\times]A_{a_{i,k_i}})[\rightarrow]A_{r_i}.$$

We use the notation $\mathcal{A}^{[a_{i,1},\ldots,a_{i,k_i}]}$ for $A_{a_{i,1}}[\times]\cdots[\times]A_{a_{i,k_i}}$, so the type of F_i is written $\mathcal{A}^{[a_{i,1},\ldots,a_{i,k_i}]}[\rightarrow]A_{r_i}$.

Given a family of types of variable names $X : \mathbb{N}_n \rightarrow \mathsf{Set}$ (X_i is the type of variables of the ith sort), the terms are recursively constructed by applying the operations $f_1, \ldots, f_n$. To determine a term we must first of all specify whether it is a variable or a function application. The following type of constructors indexes the ways of constructing terms:

$$C := \mathbb{N}_m + \sum_{i:\mathbb{N}_n} X_i.$$

The first component of C, $\mathbb{N}_m$ represents applications of the operations, and the second, $\sum_{i:\mathbb{N}_n} X_i$ represents occurrences of the variables. For each constructor the number and sort of arguments is determined: no arguments for the variables, the arguments specified by the signature for the functions. This is done by the function

$$\begin{array}{lcl}
\mathsf{argument_sort_list} & : & C \rightarrow (\mathsf{List}\ \mathbb{N}_n)\\
(\mathsf{argument_sort_list}\ \mathsf{inr}(\langle j, x\rangle)) & := & \mathsf{nil}\\
(\mathsf{argument_sort_list}\ \mathsf{inl}(f_i)) & := & [a_{i,1}, \ldots, a_{i,k_i}].
\end{array}$$

Similarly for the sort of the resulting term:

$$\begin{array}{lcl}
\mathsf{result_sort} & : & C \rightarrow \mathbb{N}_n\\
(\mathsf{result_sort}\ \mathsf{inr}(\langle j, x\rangle)) & := & j\\
(\mathsf{result_sort}\ \mathsf{inl}(f_i)) & := & r_i.
\end{array}$$

We define the family of types of terms over the variables X, $\mathcal{T}^{\sigma,X} : \mathbb{N}_n \rightarrow \mathsf{Set}$, inductively by the unique constructor

$$\mathsf{term\_intro} : (c : C)(\mathcal{T}^{\sigma,X})^l \rightarrow \mathcal{T}^{\sigma,X}_r$$

where $l := (\mathsf{argument\_sort\_list}\ c)$ and $r := (\mathsf{result\_sort}\ c)$.

The family of types $\mathcal{T}^{\sigma,X}$ with Leibniz equality is a σ-algebra.

If $\mathcal{A}$ is a σ-algebra and $\alpha : (i : \mathbb{N}_n)X_i \to \mathcal{A}_i$ is an assignment of values in $\mathcal{A}$ to the variables, we define the evaluation of terms into elements of the sorts of $\mathcal{A}$ in the following way. The assignment α can be extended to a homomorphism from the term algebra $\mathcal{T}^{\sigma,X}$ to $\mathcal{A}$,

$$[\![\_]\!]_\alpha : (i : \mathbb{N}_n)\mathcal{T}_i^{\sigma,X} \to \mathcal{A}_i$$

defined by recursion on the terms by interpreting a leading function symbol f_i in a term as an application of the function F_i of $\mathcal{A}$.

Substitution of terms for variables is defined as a special case of evaluation when the target algebra is $\mathcal{T}^{\sigma,X}$ itself. The substitution of the variables $\overline{x} = x_1, \ldots, x_k$ by the terms $\overline{s} = s_1, \ldots, s_k$ (that must be of the same sort as the corresponding variables), $t[\overline{x} := \overline{s}]$, is defined to be equal to $[\![t]\!]_\alpha$, where α is the assignment such that $\alpha(x_i) = s_i$ for every i and $\alpha(y) = y$ for all other variables.

8.7 Equational Theories

An equation is formally represented as a pair of terms of the same sort:

$$\mathsf{Equation}_\sigma := \sum_{s:\mathbb{N}_n} \mathcal{T}_s^{\sigma,X} \times \mathcal{T}_s^{\sigma,X}$$

We use the notation $t_1\ (=)\ t_2$ for $\langle s, \langle t_1, t_2 \rangle\rangle$.

If $\mathcal{A}$ is a σ-algebra and $\alpha : (i : \mathbb{N}_n)X_i \to \mathcal{A}_i$ is an assignment, we say that the equation $t_1\ (=)\ t_2$ is *true in* $\mathcal{A}$ *under* α if the evaluations of the two terms are equal (in the sense of the setoid equality):

$$\langle \mathcal{A}, \alpha \rangle \models t_1\ (=)\ t_2 \qquad \text{iff} \qquad [\![t_1]\!]_\alpha [=] [\![t_2]\!]_\alpha$$

An equation is said to be *valid* in an algebra $\mathcal{A}$ if it is true under every assignment. In this case we write $\mathcal{A} \models t_1\ (=)\ t_2$.

A list of equations $th = [eq_1, \ldots, eq_h]$ is called an *equational theory*. We say that th is *valid* in $\mathcal{A}$ if every element of th is, in symbols $\mathcal{A} \models th$ if and only if $\mathcal{A} \models eq_i$ for all i.

8.8 Equational Logic

Equational logic is the deductive system by which we derive new equations from an equational theory. In Type Theory it is defined as an inductive relation between theories and equations, whose constructors correspond to the usual derivation rules. We will use the notation

$$[e_1, \ldots, e_k] \vdash e$$

to express that the equation e can be derived from the equations $e_1, \ldots e_k$. The type of the infix operator $\vdash$ is thus $(\mathsf{List\ Equation}_\sigma) \to \mathsf{Equation}_\sigma \to \mathsf{Prop}$. It is inductively defined by the constructors

$$\begin{array}{ll}
\mathsf{axiom} & : [e : th] \vdash e \\
\mathsf{weak} & : th \vdash e \longrightarrow [e' : th] \vdash e \\
\mathsf{refl} & : th \vdash t\ (=)\ t \\
\mathsf{symm} & : th \vdash t_1\ (=)\ t_2 \longrightarrow th \vdash t_2\ (=)\ t_1 \\
\mathsf{trans} & : th \vdash t_1\ (=)\ t_2 \longrightarrow th \vdash t_2\ (=)\ t_3 \longrightarrow th \vdash t_1\ (=)\ t_3 \\
\mathsf{subst} & : th \vdash t_1\ (=)\ t_2 \longrightarrow th \vdash t_1[\overline{x} := \overline{s}]\ (=)\ t_2[\overline{x} := \overline{s}]
\end{array}$$

For a fixed equational theory th, $th \vdash t_1\ (=)\ t_2$ defines a congruence on the term algebra, that we indicate by $\equiv_{th}$. Therefore, we can take the quotient algebra $\mathcal{T}^{\sigma,X}/\equiv_{th}$, that is still a σ-algebra and has an important role in the proof of the completeness theorem. Moreover, it satisfies the equations in th:

Lemma 8.8.1 $(\mathcal{T}^{\sigma,X}/\equiv_{th}) \models th$.

8.9 Validity and Completeness

We have now all the elements necessary to prove Birkhoff's validity and completeness theorem.

Theorem 8.9.1 *For every equational theory th and every equation e*

$$th \vdash e \iff th \models e$$

Proof Validity, that is, $th \vdash e \Rightarrow th \models e$, is proved by induction on the proof of $th \vdash e$.

Completeness, that is, $th \models e \Rightarrow th \vdash e$, is proved using the σ-algebra $\mathcal{T}^{\sigma,X}/\equiv_{th}$. This algebra satisfies th by construction. Hence if $th \models e$, then $(\mathcal{T}^{\sigma,X}/\equiv_{th}) \models e$, which is equivalent to $th \vdash e$ by definition of $\equiv_{th}$. □

8.10 Conclusion

We have implemented in type theory (using the proof development system Coq [BBC+98] for the formalization) the fundamental notions and results of Universal Algebras. This implementation allows us to specify any first order algebraic structure and has operators to construct free algebras over a signature. We defined the constructions of subalgebras, product algebras, and quotient algebras and proved their basic properties. There were two main points in which we had to employ special type theoretic constructions: We used setoids as carriers for algebras in order to be able to define quotient algebras and we used wellorderings to represent free algebras.

This implementation is intended to serve two purposes. From the practical point of view it provides a set of tools that make the use of type theory in the

development of mathematical structures easier. From the theoretical point of view it investigates the use of type theory as a foundation for mathematics.

We have formalized in Type Theory the notions of term algebras, equation, equational theory, validity and equational logic and we have proved the validity and completeness theorem. All this was verified by computer using the proof tool Coq. This work is relevant to the field of interactive theorem proving because it provides a theoretical basis to the two levels approach. This approach consists in lifting a goal to a syntactic level and then using syntactic tools such as normalizing functions for the terms, to produce a proof of the syntactic goal that can then be translated to a proof of the original one. In our development the syntactic level is the level of terms and equations, while the semantic one is the level of σ-algebras and their elements. The validity and completeness theorem provides the translation from proofs at one level to proofs at the other level.

Chapter 9

The Fast Fourier Transform

9.1 Introduction

An important field of research in formalized mathematics tackles the verification of classical algorithms widely used in computer science. It is important for a theorem prover to have a good library of proof-checked functions, that can be used both to extract a formally certified program and to quickly verify software that uses the algorithm. The Fast Fourier Transform [CT65] is one of the most widely used algorithms, so I chose it as a case study in formalization using the type-theory proof tool Coq [BBC+99]. Here I present the formalization and discuss in detail the parts of it that are more interesting in the general topic of formal verification in type theory.

Previous work on the computer formalization of FFT was done by Ruben Gamboa [Gam01] in ACL2 using the data structure of powerlists introduced by Jayadev Misra [Mis94], which is similar to the structure of polynomial trees that we use here.

The Discrete Fourier Transform is a function commuting between two representations of polynomials over a commutative ring, usually the algebraic field $\mathbb{C}$. One representation is in the *coefficient domain*, where a polynomial is given by the list of its coefficients. The second representation is in the *value domain*, where a polynomial of degree $n-1$ is given by its values on n distinct points. The function from the coefficient domain to the value domain is called *evaluation*, the inverse function is called *interpolation*. The Discrete Fourier Transform (DFT) and the Inverse Discrete Fourier Transform (iDFT) are the evaluation and interpolation functions in the case in which the points of evaluation are distinct n-roots of the unit element of the ring. The reason to consider such particular evaluation points is that, in this case, an efficient recursive algorithm exists to perform evaluation, the Fast Fourier Transform (FFT), and interpolation, the Inverse Fourier Transform (iFT). Let

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} = \sum_{i=0}^{n-1} a_i x^i$$

be a polynomial of degree n and ω be a primitive n-root of unity, that is, $\omega^n = 1$ but $\omega^j \neq 1$ for $0 < j < n$. We must compute $f(\omega^j)$ for $j = 0, 1, \ldots, n-1$. First of all we write $f(x)$ as the sum of two components, the first comprising the monomials of even power and the second the monomials of odd power, for which we can collect a factor x:

$$f(x) = f_e(x^2) + x f_o(x^2). \tag{9.1}$$

The polynomials f_e and f_o have degree $n/2-1$ (assuming that n is even; if not, we can add a extra term with zero coefficient). We could apply our algorithm recursively to them and to ω^2, which is an $n/2$-root of unity. We obtain the values

$$\begin{array}{llll} f_e((\omega^2)^0), & f_e((\omega^2)^1), & \ldots, & f_e((\omega^2)^{n/2-1}); \\ f_o((\omega^2)^0), & f_o((\omega^2)^1), & \ldots, & f_o((\omega^2)^{n/2-1}). \end{array}$$

We have, therefore, $f_e((\omega^2)^i) = f_e((\omega^i)^2)$ for $i = 0, \ldots, n/2-1$ which we can feed into Formula 9.1. The only problem is that Formula 9.1 must be evaluated for $x = \omega^i$ when $i = 0, \ldots, n-1$. We are still missing the values for $i = n/2, \ldots, n-1$. Here is where the fact that ω is a primitive n-root of unity comes useful: $\omega^{n/2} = -1$, so for $j = 0, \ldots, n/2-1$ we have that

$$\omega^{n/2+j} = \omega^{n/2}\omega^j = -\omega^j$$

and therefore $f_e((\omega^{n/2+j})^2) = f_e((\omega^j)^2)$. So the values of the first term of Formula 9.1 for $i = n/2, \ldots, n-1$ are equal to the values for $i = 0, \ldots, n/2-1$ and we don't need to compute them. A similar argument holds for f_o. If we measure the algorithmic complexity by the number of multiplications of scalars that need to be performed, we see that the algorithm calls itself twice on inputs of half size and then must still perform n multiplications (multiply x by $f_o(x)$ in Formula 9.1). This gives an algorithm of complexity $O(n \log n)$, much better than the naive quadratic algorithm.

Vice versa if we are given the values of the polynomial f on the n-roots of unity, $y_0 = f(\omega^0), \ldots, y_{n-1} = f(\omega^{n-1})$, we can compute the vector of coefficients of f by applying DFT to the vector $\langle y_0, \ldots, y_{n-1}\rangle$ with ω^{-1} in place of ω and then divide by n. The proof of this fact is well-known. We will give a type-theoretic version of it in section 9.5.

One useful application of FFT is the computation the product of two polynomials. If $f(x) = \sum_{i=0}^{n-1} a_i x^i$ and $g(x) = \sum_{i=0}^{n-1} b_i x^i$, we want to compute their product $f(x)g(x) = \sum_{i=0}^{n-1} c_i x^i$. A direct computation of the coefficients would require the evaluation the formula $c_i = \sum_{j+k=i} a_j b_k$ for $i = 0, \ldots, 2n-2$, for a total of n^2 scalar products. A faster way to compute it is, first, to find the representations of the polynomials in the value domain with FFT: $f(\omega^i)$ and $g(\omega^i)$ for $i = 0, \ldots, 2n-1$; second, to multiply the corresponding values to obtain the representation of the product in the value domain: $f(\omega^i)g(\omega^i)$; and third, to return to the coefficient domain iFT. In this way we need to perform only $2n$ multiplications plus the $O(n \log n)$ multiplications needed for the conversions. The overall complexity of the multiplication algorithm is then also $O(n \log n)$, instead of quadratic.

The rest of the chapter describes the formalization of these ideas in type theory, using the proof tool Coq. In Section 9.2 we discuss the different representations for the data types involved. We choose a tree representation that is specifically designed for FFT. We prove that it is equivalent to a more natural one. In Section 9.3 we apply the two-level approach (see [BRB95] and [Bou97]) to the tree representation to prove some basic facts about it. Section 9.4 contains the definition and proof of correctness of FFT. Section 9.5 introduces iFFT and proves its correctness. Finally, Section 9.6 discusses the tools used in the formalization and some implementation issues.

9.2 Data representation

Let us fix the domain of coefficients and values. We need to work in a commutative ring with unity, and in the case of the inverse transform we will need a field. Usually it is required that the domain is an algebraically closed field, because the transform is applied to a polynomial and a root of unity. We will not do that, but just require that, when the algorithm is applied, a root of unity is supplied. This covers the useful case in which we want to apply the algorithm to finite fields (that can never be algebraically closed) or finite rings. In type theory an algebraic structure like that of ring is represented by an underlying *setoid*, which is a type with an equivalence relation, plus some operations on the setoid and proofs of the defining properties of the structure. In our case we will have the following data:

K : Set
K_eq : $K \to K \to$ Prop
K_eq_refl : (reflexive K_eq)
K_eq_symm : (symmetric K_eq)
K_eq_trans : (transitive K_eq).

K_eq is thus an equivalence relation that expresses the equality of the objects represented by terms of the type K. We will write $x \equiv y$ for (K_eq a b). The basic ring constants and operations are

$0, 1 : K$
sm, ml : $K \to K \to K$
sm_inv : $K \to K$.

We will use the notations $x+y$, $x \cdot y$ and $-x$ for (sm x y), (ml x y) and (sm_inv x), respectively. We need to require that they are well behaved with respect to $\equiv$, or, equivalently, that $\equiv$ is a congruence relation with respect to these operations:

sm_congr : $\forall x_1, x_2, y_1, y_2 : K. x_1 \equiv x_2 \to y_1 \equiv y_2 \to x_1 + y_1 \equiv x_2 + y_2$
ml_congr : $\forall x_1, x_2, y_1, y_2 : K. x_1 \equiv x_2 \to y_1 \equiv y_2 \to x_1 \cdot y_1 \equiv x_2 \cdot y_2$
sm_congr : $\forall x_1, x_2 : K. x_1 \equiv x_2 \to (-x_1) \equiv (-x_2)$

The axioms of commutative rings can now be formulated for these operations and for the equality $\equiv$.

We will often require that a certain element $\omega\colon K$ is a primitive n-root of unity. This means that $\omega^n \equiv 1$ (ω is a root) and n is the smallest non-zero element for which this happens (primitivity). At some point we will also need that, if n is even, $\omega^{n/2} \equiv -1$. Note that this fact does not generally follow from ω being a primitive n-root of unity. Indeed, if K is a field, we can prove this from the fact that $\omega^{n/2}$ is a solution of the equation $x^2 \equiv 1$, but cannot be 1 by primitivity of ω. In a field 1 and -1 are the only solutions of the equation. But in a commutative ring this is not always true. For example, take the finite commutative ring $\mathbb{Z}_{15}$ and the value $\omega = 4$, which is a primitive 2-root of unity ($4^2 \equiv 16 \equiv 1$). Nevertheless $4^1 \equiv 4 \not\equiv -1$. So the requirement $\omega^{n/2} \equiv -1$ will have to be stated explicitly when needed.

Polynomials in one variable are usually represented as the vectors of their coefficients. So the polynomial $a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ is represented by the vector $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. We can choose to implement such a vector in three standard ways: as a list (in which case there is no restriction on the length), as an element of a type with fixed length, or as a function from the finite type with n elements, $\mathbb{N}_n$, to K. This last representation is the most elastic for a certain number of purposes, specifically for reasoning about summations. It will be useful when prove the correctness of iFT. So we adopt it as our basic implementation of polynomials. The type of polynomials of degree $n-1$ will then be $\mathbb{N}_n \to K$.

However, in the proof of correctness of FFT, a different representation results more useful. A fundamental step of the computation consists in breaking the polynomial f in two parts f_e and f_o, consisting of the even and odd terms, respectively. We apply the algorithm recursively on these two polynomials. The recursion is wellfounded because f_e and f_o have smaller degree than f. This requires the use of course-of-value recursion on the degree of the polynomial, which can be realized in type theory using a general method for wellfounded recursion (see [BB00]). The formalization of the algorithm that we obtain is more complex than necessary, because it contains the proofs of the fact that the degrees decrease. Simpler algorithms are obtained by using structural recursion in place of wellfounded recursion. To use structural recursion we need that the algorithm calls itself recursively only on structurally smaller arguments.

So we are led to look for a different implementation of polynomials whose structure reflects the steps of the algorithm. A general method to obtain a data type whose structure is derived from the recursive definition of a function is presented in [BC01]. In our case we obtain the result by representing polynomials as tree structures in which the left subtree contains the even coefficients and the right subtree contains the odd coefficients. This results in the recursive definition

$$\begin{aligned}&\mathsf{Tree}\colon \mathbb{N} \to \mathsf{Set} :=\\ &\mathsf{Tree}(0) := K\\ &\mathsf{Tree}(k+1) := \mathsf{Tree}(k) \times \mathsf{Tree}(k).\end{aligned}$$

In short, $\mathsf{Tree}(k) = K^{2^k}$, but notice that the products are arranged in a binary tree structure. An element of $\mathsf{Tree}(k)$ is a binary tree of depth k whose leaves are

elements of the coefficient domain K. We will use the notation $\mathsf{leaf}(a)$ to denote an element $a\colon K$ when it is intended as a tree of depth 0, that is, an element of $\mathsf{Tree}(0) = K$. We will use the notation $\mathsf{node}(t_1, t_2)$ to denote the element $\langle t_1, t_2\rangle\colon \mathsf{Tree}(k+1) = \mathsf{Tree}(k) \times \mathsf{Tree}(k)$, if t_1 and t_2 are elements of $\mathsf{Tree}(k)$. The number of leaves of such a tree is 2^k. This is not a problem since, for the application of FFT, we always assume that the degree of the input polynomial is one less than a power of 2. Otherwise we adjust it to the closest power of 2 by adding terms with zero coefficients.

The equality $\equiv$ on K is extended to the equality $\cong$ on trees. We say that two elements $t_1, t_2\colon \mathsf{Tree}(k)$ are equal, and write $t_1 \cong t_2$, when the relation $\equiv$ holds between all corresponding leaves. The relation $\cong$ can be formally defined by recursion on k.

A polynomial is represented by putting the coefficients of the even powers of x in the left subtree and the coefficients of the odd powers of x in the right one, and this procedure is repeated recursively on the two subtrees. If we have, for example, a polynomial of degree 7 $(= 2^3 - 1)$,

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7,$$

we break it into two parts

$$f_e(y) := a_0 + a_2y + a_4y^2 + a_6y^3, \quad f_o(y) := a_1 + a_3y + a_5y^2 + a_7y^3$$

so that $f(x) = f_e(x^2) + xf_o(x^2)$. Recursively $f_e(y)$ and $f_o(y)$ can be broken into even and odd terms

$$\begin{array}{ll} f_{ee}(z) := a_0 + a_4z, & f_{eo}(z) := a_2 + a_6z, \\ f_{oe}(z) := a_1 + a_5z, & f_{oo}(z) := a_3 + a_7z \end{array}$$

so that $f_e(y) = f_{ee}(y^2) + yf_{eo}(y^2)$ and $f_o(y) = f_{oe}(y^2) + yf_{oo}(y^2)$. With another step we reach single coefficients:

$$\begin{array}{llll} f_{eee}(u) := a_0, & f_{eeo}(u) := a_4, & f_{eoe}(u) := a_2, & f_{eoo}(u) := a_6, \\ f_{oee}(u) := a_1, & f_{oeo}(u) := a_5, & f_{ooe}(u) := a_3, & f_{ooo}(u) := a_7 \end{array}$$

with $f_{ee}(z) = f_{eee}(z^2) + zf_{eeo}(z^2)$, $f_{eo}(z) = f_{eoe}(z^2) + zf_{eoo}(z^2)$, $f_{oe}(z) = f_{oee}(z^2) + zf_{oeo}(z^2)$, $f_{oo}(z) = f_{ooe}(z^2) + zf_{ooo}(z^2)$. Now we transform each of these polynomials in trees, starting with the single-coefficient ones. We simply obtain

$$\begin{array}{ll} t_{eee} := \mathsf{leaf}(a_0) = a_0, & t_{eeo} := \mathsf{leaf}(a_4) = a_4, \\ t_{eoe} := \mathsf{leaf}(a_2) = a_2, & t_{eoo} := \mathsf{leaf}(a_6) = a_6, \\ t_{oee} := \mathsf{leaf}(a_1) = a_1, & t_{oeo} := \mathsf{leaf}(a_5) = a_5, \\ t_{ooe} := \mathsf{leaf}(a_3) = a_3, & t_{ooo} := \mathsf{leaf}(a_7) = a_7. \end{array}$$

The polynomials of degree one are then represented by the trees

$$\begin{array}{ll} t_{ee} := \mathsf{node}(t_{eee}, t_{eeo}) = \langle a_0, a_4\rangle, & t_{eo} := \mathsf{node}(t_{eoe}, t_{eoo}) = \langle a_2, a_6\rangle, \\ t_{oe} := \mathsf{node}(t_{oee}, t_{oeo}) = \langle a_1, a_5\rangle, & t_{oo} := \mathsf{node}(t_{ooe}, t_{ooo}) = \langle a_3, a_7\rangle. \end{array}$$

The polynomials of degree three are represented by

$$t_e := \mathsf{node}(t_{ee}, t_{eo}) = \langle\langle a_0, a_4\rangle, \langle a_2, a_6\rangle\rangle,$$
$$t_o := \mathsf{node}(t_{oe}, t_{oo}) = \langle\langle a_1, a_5\rangle, \langle a_3, a_7\rangle\rangle.$$

Finally, the original polynomial is represented by

$$t := \mathsf{node}(t_e, t_o) = \langle\langle\langle a_0, a_4\rangle, \langle a_2, a_6\rangle\rangle, \langle\langle a_1, a_5\rangle, \langle a_3, a_7\rangle\rangle\rangle.$$

It is clear that the two representations are equivalent, in the sense that the types $\mathsf{Tree}(k)$ and $\mathbb{N}_{2^k} \to K$ are isomorphic, with the isomorphism outlined above.

The type $\mathsf{Tree}(k)$ is similar to the structure of powerlists by Misra [Mis94], used by Gamboa in the verification of FFT in ACL2 [Gam01]. The difference consists in the fact that powerlists are presented as an abstract data type that can be constructed and read in two different ways: by concatenation or by interleaving. It is not specified how powerlists are actually represented. One could implement them as simple lists or as a structure like $\mathsf{Tree}(k)$. The important fact is that there are functions doing and undoing the two different construction methods, and that we have corresponding recursion and induction principles. Here we made the choice of committing to the particular representation $\mathsf{Tree}(k)$ and keep it both for polynomials and for argument lists, avoiding costly representation transformations. Instead of using the normal list operations we have then to define equivalent ones for the tree types.

We go on to the definition of DFT. It is defined simply as the vector of evaluations of a polynomial on the powers of an argument, that is, if f is a polynomial of degree $2^k - 1$ and w is an element of K, we want that $\mathsf{DFT}(f, w) = \langle f(w^0), f(w^1), f(w^2), \ldots, f(\omega^{2^k-1})\rangle$. For consistency we also want that this result vector is represented in the same form as the polynomials, that is, as an element of $\mathbb{N}_{2^k} \to K$ in the first representation and as an element of $\mathsf{Tree}(k)$ in the second representation with the values interleaved in the same way as the coefficient. For example if $k = 3$ we want that

$$\mathsf{DFT}(f, w) = \langle\langle\langle f(w^0), f(w^4)\rangle, \langle f(w^2), f(w^6)\rangle\rangle, \langle\langle f(w^1), f(w^5)\rangle, \langle f(w^3), f(w^7)\rangle\rangle\rangle.$$

The proof that the two definitions of DFT for the two representations are equivalent via the isomorphism of the types $\mathsf{Tree}(k)$ and $\mathbb{N}_{2^k} \to K$ is straightforward.

9.3 The two-level approach for trees

We need some operations on the type of trees and tools that facilitate reasoning about them. First of all, we define the mapping of the operations of the domain K on the trees: When we write $t_1 \circ t_2$ with $t_1, t_2 \colon \mathsf{Tree}(k)$ and $\circ$ one of the binary operations of K $(\cdot, +, -)$, we mean that the operation must be applied to pairs of corresponding leaves on the two trees, to obtain a new tree of the same type. Similarly, the unary operation of additive inversion $(-)$ will be applied to each leaf of the argument tree. Multiplication by a scalar, also indicated by $\cdot$, is the

operation that takes an element a of K and a tree t and multiplies a for each of the leaves of t.

We denote the evaluation of a polynomial with $(t \xrightarrow{e} w)$, meaning "the value given by the evaluation of the polynomial represented by the tree t in the point w":

$$\begin{aligned}
&(_ \xrightarrow{e} _) \colon \mathsf{Tree}(k) \times K \to K \\
&(\mathsf{leaf}(a) \xrightarrow{e} w) := a \\
&(\mathsf{node}(t_1, t_2) \xrightarrow{e} w) := (t_1 \xrightarrow{e} w^2) + w \cdot (t_2 \xrightarrow{e} w^2).
\end{aligned}$$

The evaluation operation can be extended to evaluate a polynomial in all the leaves of a tree. This extension is achieved by mapping $\xrightarrow{e}$ to the trees in a way similar to the basic ring operations,

$$(\_ \xrightarrow{e}\!\!\!\!\rightarrow \_) \colon \mathsf{Tree}(k) \to \mathsf{Tree}(h) \to \mathsf{Tree}(h).$$

Another operation that we need is the duplication of the leaves of a tree, in which every leaf is replaced by a tree containing two copies of it:

$$\begin{aligned}
&\Downarrow \colon \mathsf{Tree}(k) \to \mathsf{Tree}(k+1) \\
&\Downarrow \mathsf{leaf}(a) := \mathsf{node}(\mathsf{leaf}(a), \mathsf{leaf}(a)) \\
&\Downarrow \mathsf{node}(t_1, t_2) := \mathsf{node}(\Downarrow t_1, \Downarrow t_2).
\end{aligned}$$

Note that, even if we wrote $\mathsf{node}(\mathsf{leaf}(a), \mathsf{leaf}(a))$ for clarity, the term on the left-hand side of the first clause could be written simply $\mathsf{node}(a, a)$, since $\mathsf{leaf}(a) = a$ in our implementation. We also want to duplicate a tree changing the sign of the second copy:

$$\begin{aligned}
&\pm \colon \mathsf{Tree}(k) \to \mathsf{Tree}(k+1) \\
&\pm \mathsf{leaf}(a) := \mathsf{node}(a, -a) \\
&\pm \mathsf{node}(t_1, t_2) := \mathsf{node}(\pm t_1, \pm t_2).
\end{aligned}$$

We write $t_1 \pm t_2$ for $t_1 + (\pm t_2)$.

Given any scalar $x \colon K$, we want to generate the tree containing the powers of x in the interleaved order. So, for example for $k = 3$, we want to obtain

$$\langle\langle\langle x^0, x^4\rangle, \langle x^2, x^6\rangle\rangle, \langle\langle x^1, x^5\rangle, \langle x^3, x^7\rangle\rangle\rangle.$$

This is achieved by the function

$$\begin{aligned}
&(_ \Uparrow \text{-}) \colon K \times \mathbb{N} \to \mathsf{Tree}(k) \\
&(x \Uparrow^0) := \mathsf{leaf}(1) \\
&(x \Uparrow^{k+1}) := \mathsf{node}(t, x \cdot t) \text{ with } t := (x^2 \Uparrow^k).
\end{aligned}$$

To facilitate reasoning about trees, we use the method known as *the two-level approach* (see, for example, [BRB95] and [Bou97]). It is a general technique to automate the proof of a class of goals by internalizing it as an inductive type. Suppose we want to implement a decision procedure for a class of goals $\mathcal{G} \subseteq \mathsf{Prop}$. First of all we define a type of codes for the goals, $\mathsf{goal} \colon \mathsf{Set}$, and an interpretation function $[\![\_]\!] \colon \mathsf{goal} \to \mathsf{Prop}$, such that the image of the whole type

goal is $\mathcal{G}$. Then we define a decision procedure as a function $\mathsf{dec}\colon \mathsf{goal} \to \mathsf{bool}$ and we prove that it is correct, that is, $\mathsf{correctness}\colon \forall g\colon \mathsf{goal}.\mathsf{dec}(g) = \mathsf{true} \to [\![g]\!]$. As a consequence, whenever we want to prove a goal $P \in \mathcal{G}$, we can do it with just one command, $\mathsf{correctness}(g)(\mathsf{eq\_refl}(\mathsf{true}))$, where g is the code corresponding to P. In the case of equational reasoning the method can be further refined. We have a certain type of objects $T\colon \mathsf{Set}$, and a certain number of operations over it. We want to be able to prove the equality of two expressions of type T build from constants and variables using the operations. We define a type $\mathsf{code}^T\colon \mathsf{Set}$ of codes for such expressions. It is defined as an inductive type having as basis names for the constants and variables, and as constructors names for the operations. Then we define an interpretation function that associate an object of T to a code under a certain assignment of values to the variables, that is, if α is an assignment that associates an element of T to every variable, and $c\colon \mathsf{code}^T$ is a code, we obtain an element $[\![c]\!]_\alpha\colon T$. We can now use the syntactic level code^T to implement different tactics to decide equality of terms. For example, we may have a simplifying function $\mathsf{simplify}\colon \mathsf{code}^T \to \mathsf{code}^T$. If we prove that the simplification does not change the interpretation of the term, $\forall c\colon \mathsf{code}^T.\forall\alpha.[\![c]\!]_\alpha = [\![\mathsf{simplify}(c)]\!]_\alpha$, then we can easily prove the equality of two terms by simply checking if the simplifications of their codes are equal.

A very helpful use of the two-level approach consists in proving equalities obtained by substitution of equal elements in a context. Suppose that we need to prove $C[\cdots a\cdots] = C[\cdots b\cdots]$ for a certain context $C[\cdots]$ and two objects a and b of which we know that they are equal. If the equality considered is Leibniz equality, then the goal can be proved by rewriting. But if we are using a book equality, this method will not work and we will have to decompose the context $C[\cdots]$ and apply various times the proofs that the operations preserve equality. If the context is complex, this may result in very long and tedious proofs. We want to be able to solve such goals in one step. This can be done by simply encoding the context as an element of code^T containing a variable corresponding to the position of the objects a and b. If we have a proof that the interpretation of a code does not change when we assign equal values to a variable, we are done.

We apply this general method to trees. These objects, trees, do not form a single type but a family of types indexed on the natural numbers according to their depth. This means that also the type of tree codes needs to be parameterized on the depth. Also the variables appearing inside a tree expression can have different depths, not necessarily equal to the depth of the whole expression. Having expressions in which several variables of different depth may appear would create complications that we do not need or desire. Instead we implement expressions in which only one variable appears, in other words, we formalize the notion of a context with a hole that can be filled with different trees. Be careful not to confuse this kind of variable, a hole in the context, from a regular variable of type $\mathsf{Tree}(k)$, that at the syntactic level is treated as a constant. To avoid confusion we could call the hole variables *metavariables*. Here is the definition of tree expressions, it has two natural-number parameters, the first for the depth of the metavariable, the second for the depth of the whole

tree:

$$
\begin{array}{ll}
\textsf{Inductive Tree\_exp}\colon \mathbb{N} \to \mathbb{N} \to \textsf{Set} := & \\
\quad \textsf{trex\_var}(k)\colon (\textsf{Tree\_exp}\ k\ k) & \text{for } k\colon \mathbb{N} \\
\quad \textsf{trex\_const}(h,k,t)\colon (\textsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; t\colon \textsf{Tree}(k) \\
\quad \textsf{trex\_leaf}(h,x)\colon (\textsf{Tree\_exp}\ h\ 0) & \text{for } h\colon \mathbb{N}; x\colon K \\
\quad \textsf{trex\_node}(h,k,e_1,e_2)\colon (\textsf{Tree\_exp}\ h\ k+1) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\textsf{Tree\_exp}\ h\ k) \\
\quad \textsf{trex\_sm}(h,k,e_1,e_2)\colon (\textsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\textsf{Tree\_exp}\ h\ k) \\
\quad \textsf{trex\_ml}(h,k,e_1,e_2)\colon (\textsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\textsf{Tree\_exp}\ h\ k) \\
\quad \textsf{trex\_mn}(h,k,e_1,e_2)\colon (\textsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\textsf{Tree\_exp}\ h\ k) \\
\quad \textsf{trex\_sc\_ml}(h,k,x,e)\colon (\textsf{Tree\_exp}\ h\ k) & \text{for } h,k\colon \mathbb{N}; x\colon K; e\colon (\textsf{Tree\_exp}\ k\ h) \\
\quad \textsf{trex\_pow}(h,k,x,e)\colon (\textsf{Tree\_exp}\ h\ k) & \text{for } h\colon \mathbb{N}; x\colon K; k\colon \mathbb{N} \\
\quad \textsf{trex\_dupl}(h,k,e)\colon (\textsf{Tree\_exp}\ h\ k+1) & \text{for } h,k\colon \mathbb{N}; e\colon (\textsf{Tree\_exp}\ h\ k) \\
\quad \textsf{trex\_id\_inv}(h,k,e)\colon (\textsf{Tree\_exp}\ h\ k+1) & \text{for } h,k\colon \mathbb{N}; e\colon (\textsf{Tree\_exp}\ h\ k) \\
\quad \textsf{trex\_sm\_mn}(h,k,e_1,e_2)\colon (\textsf{Tree\_exp}\ h\ k+1) & \text{for } h,k\colon \mathbb{N}; e_1,e_2\colon (\textsf{Tree\_exp}\ h\ k) \\
\quad \textsf{trex\_eval}(h,k_1,k_2,e_1,e_2)\colon (\textsf{Tree\_exp}\ h\ k_2) & \text{for } h,k_1,k_2\colon \mathbb{N};\ e_1\colon (\textsf{Tree\_exp}\ h\ k_1); \\
 & \qquad\qquad e_2\colon (\textsf{Tree\_exp}\ h\ k_2).
\end{array}
$$

Each of the constructors of Tree_exp corresponds to an operation on trees, except trex_var, that introduces a metavariable, and trex_const that *quotes* an actual tree inside an expression. We now define the interpretation function that takes a tree expression, a tree in which the metavariable must be interpreted, of depth equal to the first argument of the tree expression type, and gives a tree as a result. We omit the first two arguments of the function, the natural-number parameters h and k, because they can be inferred from the types of the other arguments.

$$
\begin{array}{l}
[\![\_]\!]_{\_}\colon (h,k\colon \mathbb{N})(\textsf{Tree\_exp}\ h\ k) \to \textsf{Tree}(h) \to \textsf{Tree}(k) \\
[\![\textsf{trex\_var}(k)]\!]_s := s \\
[\![\textsf{trex\_const}(h,k,t)]\!]_s := t \\
[\![\textsf{trex\_leaf}(h,x)]\!]_s := \textsf{leaf}(x) = x \\
[\![\textsf{trex\_node}(h,k,e_1,e_2)]\!]_s := \textsf{node}([\![e_1]\!]_s, [\![e_2]\!]_s) = \langle [\![e_1]\!]_s, [\![e_2]\!]_s \rangle \\
[\![\textsf{trex\_sm}(h,k,e_1,e_2)]\!]_s := [\![e_1]\!]_s + [\![e_2]\!]_s \\
[\![\textsf{trex\_ml}(h,k,e_1,e_2)]\!]_s := [\![e_1]\!]_s \cdot [\![e_2]\!]_s \\
[\![\textsf{trex\_mn}(h,k,e_1,e_2)]\!]_s := [\![e_1]\!]_s - [\![e_2]\!]_s \\
[\![\textsf{trex\_sc\_ml}(h,k,x,e_2)]\!]_s := x \cdot [\![e_2]\!]_s \\
[\![\textsf{trex\_pow}(h,x,k)]\!]_s := x{\Uparrow}^k \\
[\![\textsf{trex\_dupl}(h,k,e)]\!]_s := {\downdownarrows}[\![e]\!]_s \\
[\![\textsf{trex\_id\_inv}(h,k,e)]\!]_s := \pm[\![e]\!]_s \\
[\![\textsf{trex\_sm\_mn}(h,k,e_1,e_2)]\!]_s := [\![e_1]\!]_s \pm [\![e_2]\!]_s \\
[\![\textsf{trex\_eval}(h,k_1,k_2,e_1,e_2)]\!]_s := ([\![e_1]\!]_s \overset{e}{\twoheadrightarrow} [\![e_2]\!]_s)
\end{array}
$$

The most important use of this setup is in proving equality of substitutions inside a context.

Theorem 9.3.1 (Tree reflection) *Let $h,k\colon \mathbb{N}$; for every context, given as a tree expression $e\colon (\textsf{Tree\_exp}\ h\ k)$, and for every pair of trees $t_1,t_2\colon \textsf{Tree}(h)$;*

if $t_1 \cong t_2$, then the interpretations of the context under t_1 and t_2 are equal, $[\![e]\!]_{t_1} \cong [\![e]\!]_{t_2}$.

Proof A routine induction on the structure of e, using the proofs that the various operations considered preserve tree equality. □

This theorem has been formalized in Coq with the name tree_reflection and is the most powerful tool in the development and proof of correctness of FFT. Whenever we need to prove a goal in the form $C[\cdots a \cdots] \cong C[\cdots a \cdots]$ with trees a and b for which we have a proof $p\colon a \cong b$, we can do it with the single command tree_reflection(h, k, e, a, b, p), where e is an encoding of C as an element of (Tree_exp h k).

This method has been repeatedly used in the formal proofs of the following lemmas, important steps towards the proof of correctness of FFT.

Lemma 9.3.2 (scalar_ml_tree_ml) *For every $x_1, x_2\colon K$ and for every pair of trees $t_1, t_2\colon \mathsf{Tree}(k)$,*

$$(x_1 \cdot x_2) \cdot (t_1 \cdot t_2) \cong (x_1 \cdot t_1) \cdot (x_2 \cdot t_2).$$

(The operator $\cdot$ is overloaded and has three different meanings: product of scalars, product of a scalar by a tree, and product of trees.)

Lemma 9.3.3 (tree_ml_tree_ml) *For every quadruple of trees $t_1, t_2, s_1, s_2\colon \mathsf{Tree}(k)$,*

$$(t_1 \cdot t_2) \cdot (s_1 \cdot s_2) \cong (t_1 \cdot s_1) \cdot (t_2 \cdot s_2).$$

Lemma 9.3.4 (pow_tree_square) *For every $k\colon \mathbb{N}$ and $x\colon K$,*

$$(x \cdot x)\Uparrow^k \cong (x\Uparrow^k \cdot x\Uparrow^k).$$

(Here also $\cdot$ has a different meaning on the left-hand and right-hand side.)

Lemma 9.3.5 (eval_duplicate) *For every polynomial represented as a tree $t\colon \mathsf{Tree}(k)$ and for every vector of arguments represented as a tree $u\colon \mathsf{Tree}(h)$, the evaluation of t on the tree obtained by duplicating the leaves of u is equal to the tree obtained by duplicating the leaves of the evaluation of t on u,*

$$(t \overset{e}{\twoheadrightarrow} \downarrow\downarrow u) \cong \downarrow\downarrow(t \overset{e}{\twoheadrightarrow} u).$$

Lemma 9.3.6 (tree_eval_step) *A polynomial represented by the tree composed of its even and odd halves, $t = \mathsf{node}(t_e, t_o)\colon \mathsf{Tree}(k+1)$ is evaluated on a tree of arguments $u\colon \mathsf{Tree}(h)$ by the equality*

$$(\mathsf{node}(t_e, t_o) \overset{e}{\twoheadrightarrow} u) \cong (t_e \overset{e}{\twoheadrightarrow} u \cdot u) + u \cdot (t_o \overset{e}{\twoheadrightarrow} u \cdot u).$$

(Note that this rule is simply the recursive definition of evaluation when we have a single element of K as argument, but it needs to be proved when the argument is a tree.)

Lemma 9.3.7 *Let* $k\colon \mathbb{N}$, $t, t_1, t_2, s_1, s_2\colon \mathsf{Tree}(k)$, *and* $x\colon K$. *The following equalities hold:*

sm_mn_duplicate_id_inv:	$t_1 \pm t_2 \cong (\Downarrow t_1) + (\pm t_2)$;
ml_id_inv_duplicate:	$\pm t_1 \cdot \Downarrow t_2 \cong \pm(t_1 \cdot t_2)$;
scalar_ml_in_inv:	$x \cdot (\pm t) \cong \pm(x \cdot t)$;
scalar_ml_duplicate:	$x \cdot (\Downarrow t) \cong \Downarrow(x \cdot t)$;
node_duplicate:	$\mathsf{node}(\Downarrow t_1, \Downarrow t_2) \cong \Downarrow \mathsf{node}(t_1, t_2)$.

The method of tree reflection and the above lemmas are extensively used in the following sections.

9.4 Definition and correctness of FFT

We have build enough theory to obtain a short formulation of FFT and to prove its correctness.

Definition 9.4.1 (FFT) *The algorithm computing the Fast Fourier Transform of a polynomial represented by a tree* $t\colon \mathsf{Tree}(k)$ *(polynomial of degree* $2^k - 1$*) on the roots of unity generated by a primitive* 2^k*-root* $w\colon K$ *is given by the type-theoretic function*

$$\begin{aligned}&\mathsf{FFT}\colon (k\colon \mathbb{N})\mathsf{Tree}(k) \to K \to \mathsf{Tree}(k)\\ &\mathsf{FFT}(0, \mathsf{leaf}(a_0), w) := \mathsf{leaf}(a_0)\\ &\mathsf{FFT}(k+1, \mathsf{node}(t_1, t_2), w) := \mathsf{FFT}(k, t_1, w^2) \pm (w\Uparrow^k \cdot \mathsf{FFT}(k, t_2, w^2))\end{aligned}$$

We actually do not require that w is a root of unity, but we allow it to be any element of K to keep the definition of the algorithm simple. The correctness statement will hold only when w is a primitive root of unity and states that FFT computes the same function as DFT.

Definition 9.4.2 (DFT) *The Discrete Fourier Transform of a polynomial represented by a tree* $t\colon \mathsf{Tree}(k)$ *(polynomial of degree* $2^k - 1$*) on the roots of unity generated by a primitive* 2^k*-root* $w\colon K$ *is given by the evaluation of the polynomial on every root*

$$\begin{aligned}&\mathsf{DFT}\colon (k\colon \mathbb{N})\mathsf{Tree}(k) \to K \to \mathsf{Tree}(k)\\ &\mathsf{DFT}(k, t, w) := t \overset{e}{\twoheadrightarrow} w\Uparrow^k\end{aligned}$$

The fundamental step in the proof of equivalence of the two functions consists in proving that DFT satisfies the equality expressed in the recursion step of FFT. The equivalence follows by induction on the steps of FFT, that is, by induction on the tree structure of the argument.

Lemma 9.4.3 (DFT_step) *Let* $t_1, t_2\colon \mathsf{Tree}(k)$ *and* ω *be a* 2^{k+1} *principal root of unity such that* $\omega^{2^k} = -1$*; then*

$$\mathsf{DFT}(k+1, \mathsf{node}(t_1, t_2), \omega) \cong \mathsf{DFT}(k, t_1, \omega^2) \pm (\omega\Uparrow^k \cdot \mathsf{DFT}(k, t_2, \omega^2)).$$

Proof We prove the equality through the intermediate steps

$$\begin{aligned}
\mathsf{DFT}(k+1, \mathsf{node}(t_1, t_2), \omega) &\cong (t_1 \overset{e}{\twoheadrightarrow} \omega^2 \Uparrow^{k+1}) + \omega \Uparrow^{k+1} \cdot (t_2 \overset{e}{\twoheadrightarrow} \omega^2 \Uparrow^{k+1}) \\
&\cong \Downarrow\Downarrow (t_1 \overset{e}{\twoheadrightarrow} \omega^2 \Uparrow^{k}) + \omega \Uparrow^{k+1} \cdot \Downarrow\Downarrow (t_2 \overset{e}{\twoheadrightarrow} \omega^2 \Uparrow^{k}) \\
&\cong \Downarrow\Downarrow \mathsf{DFT}(k, t_1, \omega^2) + \omega \Uparrow^{k+1} \cdot \Downarrow\Downarrow \mathsf{DFT}(k, t_2, \omega^2) \\
&\cong \mathsf{DFT}(k, t_1, \omega^2) \pm (\omega \Uparrow^{k} \cdot \mathsf{DFT}(k, t_2, \omega^2)).
\end{aligned}$$

The first step follows from the definition of DFT and Lemma tree_eval_step. The other steps are proved using the lemmas from the previous section, the method of tree reflection, and induction on the structure of trees. In the last step the hypothesis $\omega^{2^k} = -1$ must be used. □

Theorem 9.4.4 (FFT_correct) *Let $k\colon \mathbb{N}$, $t\colon \mathsf{Tree}(k)$ representing a polynomial, and ω be a principal 2^k-root of unity with the property that $\omega^{2^{k-1}} = -1$; then*

$$\mathsf{FFT}(k, t, \omega) \cong \mathsf{DFT}(k, t, \omega).$$

Proof By straightforward induction on the structure of t using the previous lemma in the recursive case. □

9.5 The Inverse Fourier Transform

We formulate and prove the correctness of iFT. We need that K is a field, not just a commutative ring. This means that we assume that there is an operation

$$\mathsf{ml\_inv}\colon (x : K)x \not\equiv 0 \to K$$

satisfying the usual properties of multiplicative inverse. We can therefore define a division operation

$$\mathsf{dv}\colon (x, y\colon K)y \not\equiv 0 \to K.$$

Division is a function of three arguments: two elements of K, x and y, and a proof p that y is not 0. We use the notation x/y for $(\mathsf{dv}\ x\ y\ p)$, hiding the proof p.

The tree representation for polynomials is very useful for the verification of FFT, but the proof of correctness of iFT is easier with the function representation: A polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$ is represented as a function $a\colon \mathbb{N}_n \to K$, and we write a_i for the value $a(i)$. We already proved in Section 9.2 that the two representations are equivalent, so we can freely switch between the two to develop our proofs. Let then DFT^f be the version of the Discrete Fourier Transform for the functional representation of polynomials. Here is the definition of the inverse transform.

Definition 9.5.1 *The* Inverse Discrete Fourier Transform *is the function that applies* DFT *to the multiplicative inverse of a root of unity and then divides by the degree:*

$$\mathsf{iDFT}^f\colon (n:\mathbb{N})(\mathbb{N}_n \to K) \to (\omega\colon K)\omega \not\equiv 0 \to (\mathbb{N}_n \to K)$$
$$\mathsf{iDFT}^f(n,a,\omega,p) := \tfrac{1}{n}\mathsf{DFT}(n,a,\omega^{-1}).$$

Our goal is to prove that this function is indeed the inverse of DFT^f, that is, $\mathsf{iDFT}^f(n,\mathsf{DFT}(n,a,\omega),\omega) = a$. The proof of this fact follows closely the standard proof given in the computer algebra literature, so we do not linger over its details. We only point out the passages where extra work must be done to obtain a formalization in type theory. First of all, we need to define the summation of a vector of values of K. The vector is represented by a function $v\colon \mathbb{N}_n \to K$. The summation $\sum v$ is defined by recursion on n. The main reason for choosing the representation $\mathbb{N}_n \to K$, instead of $\mathsf{Tree}(k)$, is that we often need to exchange the order of double summations, that is, we need the equality

$$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} a_{ij} = \sum_{j=0}^{m-1}\sum_{i=0}^{n-1} a_{ij}.$$

With the function notation, this becomes very easy: $a\colon \mathbb{N}_n \to \mathbb{N}_m \to K$ and the equality is written

$$\sum \lambda i\colon \mathbb{N}_n.\sum a(i) \equiv \sum \lambda j\colon \mathbb{N}_m.\sum \lambda i\colon \mathbb{N}_n a(i)(j).$$

We can easily define a summation function for $\mathsf{Tree}(k)$, but in that case it becomes much more complicated to formulate a double summation. A matrix of values like $\{a_{ij}\}$ would have to be represented as a tree having trees for leaves. Then the swap of the indexes in the summation would correspond to lifting the tree structure of the leaves to the main tree and lowering the main tree structure to the leaves. This would require much more work than simply changing representation.

In the proof we make essential use of the formula for the summation of a geometric series, expressed in type theory by

Lemma 9.5.2 (geometric_series) *For every $n\colon \mathbb{N}$, $x\colon K$ such that $x \not\equiv 1$,*

$$\sum \lambda i\colon \mathbb{N}_n.x^i \equiv \frac{x^n - 1}{x - 1}$$

where we simplified notation treating i as an element of $\mathbb{N}$ (we should really write $x^{(\mathsf{fin\_to\_nat}(i))}$ in place of x^i, where $\mathsf{fin\_to\_nat}\colon \mathbb{N}_n \to \mathbb{N}$ is the canonical embedding function).

Proof Standard. □

Lemma 9.5.3 (summation_root_delta) *For $n\colon \mathbb{N}$, $\omega\colon K$ a primitive n-root of unity, and $k, j\colon \mathbb{N}_n$,*

$$\sum \lambda i\colon \mathbb{N}_n.(\omega^i)^j \cdot ((\omega^{-1})^k)^i \equiv n\delta_{jk}$$

where δ_{ij}, the Kronecker symbol, is 1 if $i = j$, 0 otherwise. Once again we abused notation leaving out the application of the conversion function fin_to_nat. *On the right-hand side we used juxtaposition of the natural number n to an element of K to indicate the function giving the n-fold sum in K.*

Proof Standard. □

Finally we can prove the correctness of the inverse transform.

Theorem 9.5.4 (inverse_DFT) *Let $n\colon \mathbb{N}$, $a\colon \mathbb{N}_n \to K$, $\omega\colon K$ a primitive n-root of unity; then for every $k\colon \mathbb{N}_n$,*

$$\mathsf{DFT}^f(n, \mathsf{DFT}^f(n, a, \omega), \omega^{-1})_k \equiv n \cdot a_k.$$

Proof Standard using the previous lemmas. □

Once iDFT has been defined and proved correct for the functional representation of polynomials, we can use the equivalence of the two representations to obtain a proof of correctness of the version of the inverse transform for trees using the fast algorithm:

$$\mathsf{iFT}(n, t, \omega) := \frac{1}{n}\mathsf{FFT}(n, t, \omega^{-1}).$$

9.6 Conclusion

The definition and the proof of correctness for FFT and iFT have been completely formalized using the proof tool Coq. I have used the graphical interface CtCoq [Ber99] to develop the formalization. CtCoq was extremely useful for several reasons. It affords extendible notation which allows the printing of terms in nice mathematical formalism, hiding parts that have no mathematical content (for example the applications of the commutation function fin_to_nat or the presence of a proof that the denominator is not zero in a division). The technique of *proof-by-pointing* [BKT94] allows the user to construct complicated tactics with a few clicks of the mouse. It is easy to search for theorems and lemmas previously proved and apply them by just pressing a key. I found that the use of CtCoq increased the efficiency and speed of work.

The proof uses two main techniques. First, instead of using the most natural data type to represent the objects on which the algorithm operates, we chose an alternative data type that makes it easier to reason about the computations by structural recursion. Second, we used the two-level approach to automate part of the generation of proofs of equality for tree expressions.

Future work will concentrate in controlling more carefully how the algorithm exploits time and space resources. It is known that FFT runs in $O(n \log n)$ time, but we didn't prove it formally in Coq. The problem here is that there is no formal study of algorithmic complexity in type theory. In general it is difficult to reason about the running time of functional programs, since the reduction strategy is not fixed. A solution could be achieved by translating FFT into a development of imperative programming and then reasoning about its complexity in that framework. Another point is the use of memory. One important feature of FFT is that it can be computed *in place*, that means that the memory occupied by the input data can be reused during the computation, without the need of extra memory. Also memory management is a sore point in functional programming. I think it is possible to use the *uniqueness* types of the programming language Clean [dMJB$^+$] to force the algorithm to reuse the space occupied by the data.

Bibliography

[Acz77] Peter Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*. North-Holland, 1977.

[Acz78] Peter Aczel. The type theoretic interpretation of constructive set theory. In A. Macintyre, L. Pacholski, and J. Paris, editors, *Logic Colloquium '77*. North-Holland, 1978.

[Acz82] Peter Aczel. The type theoretic interpretation of constructive set theory: Choice Principles. In A. S. Troelstra and D. van Dalen, editors, *The L. E. J. Brouwer Centenary Symposium*. North-Holland, 1982.

[Acz86] Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive Definitions. In Barcan Marcus et al., editor, *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.

[Acz88] Peter Aczel. *Non-Well-Founded Sets*. Number 14 in CSLI Lecture Notes. Stanford University, 1988.

[Acz93] Peter Aczel. GALOIS: A theory development project. Unpublished Mansucript, 1993.

[Acz94] Peter Aczel. Notes towards a formalisation of constructive galois theory. draft report, 1994.

[Acz99] Peter Aczel. On relating type theories and set theories. In Altenkirch et al. [ANR99], pages 1–18.

[AGNvS94] T. Altenkirch, V. Gaspes, B. Nordström, and B. von Sydow. *A User's Guide to ALF*, 1994.

[Alb82] Donald J. Albers. Paul Halmos: Maverick mathologist. *Two-Year College Mathematics Journal*, 13(14):226–242, September 1982. Reprinted in [Hal83], pp. 292–304.

[Alt99] T. Altenkirch. Extensional equality in intensional type theory. In *Proceedings of LICS'99*, pages 412–420. IEEE Computer Society Press, 1999.

[ANR99] T. Altenkirch, W. Naraschewski, and B. Reus, editors. *Proceedings of TYPES'98*, volume 1657 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[Aud91] P. Audebaud. Partial objects in the calculus of constructions. In *6th Annual IEEE Symposium on Logic in Computer Science, Amsterdam*, pages 86–95. IEEE Computer Society Press, July 1991.

[AZ99] Martin Aigner and Günter M. Ziegler. *Proofs from THE BOOK*. Springer, 1999.

[Bai93] Antony Bailey. Representing algebra in LEGO. Master's thesis, University of Edinburgh, 1993.

[Bai98] Antony Bailey. *The Machine-Checked Literate Formalisation of Algebra in Type Theory*. PhD thesis, University of Manchester, 1998.

[Bar84] H. P. Barendregt. *The Lambda Calculus - its Syntax and Semantics*. North-Holland, 1984.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*, pages 117–309. Oxford University Press, 1992.

[Bar95a] Gilles Barthe. Extensions of pure type systems. In Dezani-Ciancaglini and Plotkin [DCP95], pages 16–31.

[Bar95b] Gilles Barthe. Formalising mathematics in type theory: fundamentals and case studies. Technical Report CSI-R9508, University of Nijmegen, 1995.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BB96] F. Barbanera and S. Berardi. Proof-irrelevance out of excluded-middle and choice in the calculus of constructions. *Journal of Functional Programming*, 6(3):519–525, May 1996.

[BB00] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [HA00], pages 1–16.

[BBC+98] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring,

Amokrane Saïbi, and Benjanin Werner. *The Coq Proof Assistant Reference Manual. Version 6.2*. INRIA, 1998.

[BBC+99] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjanin Werner. *The Coq Proof Assistant Reference Manual. Version 6.3*. INRIA, 1999.

[BC01] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2001.

[BDH+99] Yves Bertot, Gilles Dowek, André Hirschowits, Christine Paulin, and Laurent Théry, editors. *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[Bee85] Michael J. Beeson. *Foundation of Constructive Mathematics*. Springer-Verlag, 1985.

[Ber99] Yves Bertot. The CtCoq system: Design and architecture. *Formal aspects of Computing*, 11:225–243, 1999.

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Symposium on Theoretical Aspects Computer Software (STACS), Sendai (Japan)*, volume 789 of *LNCS*. Springer, April 1994.

[BM96] Jon Barwise and Lawrence Moss. *Vicious Circles*. Number 60 in CSLI Lecture Notes. CSLI, Stanford, California, 1996.

[BN93] Henk Barendregt and Tobias Nipkow, editors. *Types for Proofs and Programs. International Workshop TYPES '93*, volume 806 of *Lecture Notes in Computer Science*. Springer, 1993.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software. Third International Symposium, TACS'97*, volume 1281 of *LNCS*, pages 515–529. Springer, 1997.

[Bov99] Ana Bove. *Programming in Martin-Löf Type Theory: Unification - A non-trivial Example*. PhD thesis, Departmente of Computer Science, Chalmers University of Technology, November 1999. `http://www.cs.chalmers.se/~bove/Papers/lic_thesis.ps.gz`.

[Bov01] A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, Spring 2001.

[BRB95] G. Barthe, M. Ruys, and H. P. Barendregt. A two-level approach towards lean proof-checking. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs (TYPES'95)*, volume 1158 of *LNCS*, pages 16–35. Springer, 1995.

[Bro13] Luitzen Egbertus Jean Brouwer. Intuitionism and formalism. *Bulletin of the American Mathematical Society*, 20:81–96, 1913. Inaugural address at the University of Amsterdam, read October 14, 1912. Translated by Arnold Dresden. Reprinted in Bulletin (New Series) of the American Mathematical Society, Valume 37, Number 1, Pages 55-564.

[BV92] Annalisa Bossi and Silvio Valentini. An intuitionistic theory of types with assumptions of high-arity variables. *Annals of Pure and Applied Logic*, 57:93–149, 1992.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[BW00] Bruno Barras and Benjamin Werner. Coq in Coq. Draft paper, 2000.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[Cap99] Venanzio Capretta. Universal algebra in type theory. In Bertot et al. [BDH+99], pages 131–148.

[Cap00a] Venanzio Capretta. Equational reasoning in type theory. `http://www.cs.kun.nl/~venanzio`, 2000.

[Cap00b] Venanzio Capretta. Recursive families of inductive types. In Harrison and Aagaard [HA00], pages 73–89.

[CC99] C. Coquand and T. Coquand. Structured type theory. In *Proceedings of LFM'99 (held in conjunction with PLI'99)*, 1999.

[ČDS98] D. Čubrić, P. Dybjer, and P. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8(2):153–192, April 1998.

[Ced97] J. Cederquist. *A point-free approach to constructive analysis in type theory*. PhD thesis, Chalmers Tekniska Högskola, 1997.

[CG00] A. Ciaffaglione and P. Di Gianantonio. A coinductive approach to real numbers. In Th. Coquand, P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of Types '99*, volume 1956 of *Lecture Notes in Computer Science*, pages 114–130. Springer-Verlag, 2000.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CH92] J. Chirimar and D. J. Howe. Implementing constructive real analysis (preliminary report). In J.P. Myers and M.J. O'Donnel, editors, *Constructivity in Computer Science*, volume 613 of *Lecture Notes in Computer Science*, pages 165–178. Springer-Verlag, 1992.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Ann. of Math.*, 33:346–366, 1932.

[Chu41] Alonzo Church. *The calculi of λ conversion*. Princeton University Press, 1941.

[CM85] R. L. Constable and N. P. Mendler. Recursive definitions in type theory. In *Logic of Programs, Brooklyn*, volume 193 of *LNCS*, pages 61–78. Springer, June 1985.

[Con83] Robert L. Constable. Partial functions in constructive formal theories. In *Theoretical Computer Science, 6th GI-Conference, Dortmunt*, volume 145 of *LNCS*, pages 1–18, January 1983.

[Coq86] Thierry Coquand. An analysis of Girard's paradox. In *Proceedings, Symposium on Logic in Computer Science*, pages 227–236, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[Coq90] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, 1990.

[Coq93] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

[Coq94] Thierry Coquand. A new paradox in type theory. In D. Prawitz, B. Skyrms, and D. Westerståhl, editors, *Proceedings 9th Int. Congress of Logic, Methodology and Philosophy of Science, Uppsala, Sweden, 7–14 Aug 1991*, volume 134, pages 555–570. North-Holland, Amsterdam, 1994.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In P. Martin-Löf, editor, *Proceedings of Colog '88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[CP98] Thierry Coquand and Henrik Persson. Integrated Development of Algebra in Type Theory. Presented at the Calculemus and Types '98 workshop, 1998.

[CS87] Robert L. Constable and Scott Fraser Smith. Partial object in constructive type theory. In *Logic in Computer Science, Ithaca, New York*, pages 183–193, Washington, D.C., June 1987. IEEE.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, April 1965.

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on automatic demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970.

[dB80] N. G. de Bruijn. A survey of the AUTOMATH project. In Selding and Hindley [SH80].

[DCP95] M. Dezani-Ciancaglini and G. Plotkin, editors. *Proceedings of TLCA'95*, volume 902 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[dMJB+] Paul de Mast, Jan-Marten Jansen, Dick Bruin, Jeroen Fokker, Pieter Koopman, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. *Functional Programming in Clean*. Computing Science Institute, University of Nijmegen.

[DR94] D. Duval and J.-C. Reynaud. Sketches and computation—I: basic definitions and static evaluation. *Mathematical Structures in Computer Science*, 4(2):185–238, June 1994.

[DS99] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Proceedings of TLCA 1999*, volume 1581 of *LNCS*, pages 129–146. Springer-Verlag, 1999.

[DS01] Peter Dybjer and Anton Setzer. Indexed induction-recursion. In *International Seminar, PTCS 2001*, Dagstuhl Castle, Germany, October 7-12, 2001.

[Dyb97] Peter Dybjer. Representing Inductively Defined Sets by Wellorderings in Martin-Löf Type Theory. *Theoretical Computer Science*, 176:329–335, 1997.

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[Ewa96] William Ewald, editor. *From Kant to Hilbert: A Source Book in the Foundations of Mathematics*. Clarendon Press, Oxford, 1996.

[FH83] M. Forti and F. Honsell. Set theory with free construction principles. *Annali Scuola Normale Superiore — Pisa Classe di Scienza*, 10:493–522, 1983. serie IV.

[Fre79] Gottlob Frege. Begriffsschrift, eine der aritmetischen nachgebildete formelsprache des reinen denkens. Halle, 1879. English translation in [vH67], pp. 1–82.

[Fre84] Gottlob Frege. Die grundlagen der arithmetik. Breslau, 1884. English translation *The Foundation of Arithmetic* by J. L. Austin, Blackwell, Oxford, 1978.

[Fre03] Gottlob Frege. Grundgesetze der arithmetik. Jena, vol. 1 1893, vol. 2 1903. English translation *The basic laws of arithmetic* by Montgomery Furth, University of California Press, 1964.

[Gam01] Ruben A. Gamboa. The correctness of the Fast Fourier Transform: a structured proof in ACL2. *Formal Methods in System Design, Special Issue on UNITY*, 2001. in print.

[Geu92] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In Bengt Nordström, Kent Pettersson, and Gordon Plotkin, editors, *Proccedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, pages 193–217, 1992. `ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.dvi.Z`.

[Geu93] Herman Geuvers. *Logics and Type systems*. PhD thesis, University of Nijmegen, September 1993.

[Gie97] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.

[Gim94] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs. International Workshop TYPES '94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[Gim95] Eduardo Giménez. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs. International Workshop TYPES '95*, volume 1158 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 1995.

[Gim98] Eduardo Giménez. A Tutorial on Recursive Types in Coq. Technical Report 0221, Unité de recherche INRIA Rocquencourt, May 1998.

[Gir89] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989. Translated and with appendices by Paul Taylor and Yves Lafont.

[Göd30] Kurt Gödel. Die vollständigkeit der axiome des logischen funktionenkalküls. *Monatsh. Math. Phys.*, 37:349–360, 1930. English translation in [vH67], pp. 582–591.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der Principia Mathematica und verwandter systeme I. *Monatsh. Math. Phys.*, 38:173–198, 1931. English translation in [vH67], pp. 596–616.

[GPWZ01] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The algebraic hierarchy of the FTA project. In *Informal Proceedings of Calculemus'01*, 2001.

[HA00] J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Hal81] Paul R. Halmos. Applied mathematics is bad mathematics. In L. A. Steen, editor, *Mathematics Tomorrow*, pages 9–20, New York, 1981. Springer-Verlag. Reprinted in [Hal83], pp.279–290.

[Hal83] Paul R. Halmos. *Selecta. Expository Writing*. Springer-Verlag, 1983. Edited by Donald E. Sarason and Leonard Gillman.

[Har98] John Harrison. *Thorem Proving with the Real Numbers*. Distinguished dissertations. Springer-Verlag, London, 1998.

[Hey56] A. Heyting. *Intuitionism, an Introduction*. North-Holland, 1956.

[HF75] Arend Heyting and Hans Freudenthal, editors. *Collected Works of Luitzen Egbertus Jean Brouwer*. North-Holland, Amsterdam, 1975.

[Hil05] David Hilbert. Über die grundlagen der logik und der arithmetic. In *Verhandlungen des Dritten Internationalen Mathematiker-Kongresses in Heidelberg vom 8. bis 13. August 1904*, pages 174–185, Teubner, Leipzig, 1905. English translation *On the foundations of logic and arithmetic* in [vH67], pp.129–138.

[Hil26] David Hilbert. Über das unendliche. *Mathematische Annalen*, 95:161–170, 1926. English translation *On the foundations of logic and arithmetic* in [vH67], pp.367–392.

[HJ98] Claudio Hermida and Bart Jacobs. Structural induction and coinduction in a fibrational setting. *Information and Computation*, 145:107–152, 1998.

[HJ99] U. Hensel and B. Jacobs. Coalgebraic theories of sequences in pvs. *Journal of Logic and Computation*, 9(4):463–500, 1999.

[Hof93] Martin Hofmann. Elimination of extensionality in Martin-Löf type theory. In Barendregt and Nipkow [BN93], pages 166–190.

[Hof95a] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1995.

[Hof95b] Martin Hofmann. A simple model for quotient types. In Dezani-Ciancaglini and Plotkin [DCP95], pages 216–234.

[How80] W. A. Howard. The formulae-as-types notion of construction. In Selding and Hindley [SH80], pages 479–490.

[How88] Douglas J. Howe. Computational metatheory in Nuprl. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 238–257. Springer-Verlag, 1988.

[HS94] Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 208–212. IEEE Computer Society Press, 1994.

[HS00] Gérard Huet and Amokrane Saïbi. Constructive category theory. In *Proof, Language and Interaction—Essays in honour of Robin Milner*, pages 239–275. MIT Press, 2000.

[Jac94] Paul Jackson. Exploring abstract algebra in constructive type theory. In *12th International Conference on Automated Deduction – CADE-12*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 590–604. Springer-Verlag, 1994.

[Jac99] Bart Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1999.

[Jac02] Bart Jacobs. Exercises in coalgebraic specification. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, pages 237–280. Springer-Verlag, 2002.

[JHA+99] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. *Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language*, 1999. `http://www.haskell.org/onlinereport/`.

[JMRR01] B. Jacobs, L. Moss, H. Reichel, and J. Rutten, editors. *Coalgebraic methods in computer science*, 2001. Theoretical Computer Science 260(1–2). Special issue including selected papers from the First International Workshop on Coalgebraic Methods in Computer Science (CMCS '98).

[Jon93] C. Jones. Completing the rationals and metric spaces in LEGO. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 297–316. Cambridge University Press, 1993.

[Kle36] S. C. Kleene. General recursive functions of natural numbers. *Math. Ann.*, 112:727–742, 1936.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[Lac95] S. Lack. *The algebra of distributive and extensive categories*. PhD thesis, University of Cambridge, 1995.

[LP92] Zhaohui Luo and Robert Pollak. LEGO proof development system: User's manual. Technical report, University of Edimburgh, May 1992. Technical Report ECS-LFCS-92-211, LSCF.

[LPM93] François Leclerc and Christine Paulin-Mohring. Programming with streams in Coq. a case study: the sieve of Eratosthenes. In Barendregt and Nipkow [BN93], pages 191–212.

[LS86] J. Lambek and P. Scott. *Introduction to Higher Order Categorical Logic*. Oxford University Press, 1986.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*, volume 11 of *International Series of Monographs on Computer Science*. Oxford University Press, 1994.

[LW99] S. Lacas and B. Werner. Which choices imply the excluded middle? Manuscript, 1999.

[Mai99] Maria Emilia Maietti. About effective quotients in constructive type theory. In Altenkirch et al. [ANR99], pages 164–178.

[Mat99] Ralph Matthes. Monotone (co)inductive types and positive fixed-point types. *Theoretical Informatics and Applications*, 33:309–328, 1999.

[MB93] Corrado Mangione and Silvio Bozzi. *Storia della Logica. Da Boole ai nostri giorni*. Garzanti, 1993.

[Men87] Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1987.

[Mis94] Jayadev Misra. Powerlist: a structure for parallel recursion. *TOPLAS*, 16(6):1737–1767, November 1994.

[Miz01] The Mizar home page, 2001. `http://mizar.org/`.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[ML98] Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan Smith, editors, *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford Science Publications, 1998. Proceedings of a Congress Held in Venice, October 1995.

[MN93] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Barendregt and Nipkow [BN93], pages 213–237.

[MT92] K. Meinke and J. V. Tucker. Universal Algebra. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 1*. Oxford University Press, 1992.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[NGV94] R. P. Nederpelt, J. H. Geuvers, and R. C. De Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies In Logic and The Foundations of Mathematics*. North-Holland, 1994.

[Nor88] Bengt Nordström. Terminating general recursion. *BIT*, 28(3):605–619, October 1988.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction*, volume 7 of *International Series of Monographs on Computer Scence*. Oxford University Press, 1990.

[NS00] W. H. Newton-Smith, editor. *A Companion to the Philosophy of Science*. Blackwell Pubishers, 2000.

[OSRSC99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[PA93] G. Plotkin and M. Abbadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 361–375. Springer-Verlag, 1993.

[Pau86] Lawrence C. Paulson. Proving termination of normalization functions for conditional expressions. *Journal of Automated Reasoning*, 2:63–74, 1986.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[Pau97] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175–204, 1997.

[Per99] H. Persson. *Type Theory and the Integrated Logic of Programs*. PhD thesis, Chalmers Tekniska Högskola, 1999.

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, 1993. LIP research report 92-49.

[Poi06] Henri Poincaré. *La Science et l'Hypothèse*. Flammarion, Paris, 1906.

[Poi16] Henri Poincaré. *Science et Méthode*. Flammarion, Paris, 1916.

[Poi25] Henri Poincaré. *La Valeur de La Science*. Flammarion, Paris, 1925.

[Pop59] Karl Raimund Popper. *The Logic of Scientific Discovery*. Hutchinson, London, 1959.

[Pop65] Karl Raimund Popper. *Conjectures and Refutations*. Basic Books, New York, 1965.

[PPM90] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*. Springer-Verlag, 1990. technical report CMU-CS-89-209.

[PR99] Holger Pfeifer and Harals Rueß. Polytypic proof construction. In Bertot et al. [BDH+99], pages 54–72.

[PS89] Kent Petersson and Dan Synek. A Set Constructor for Inductive Sets in Martin-Löf's Type Theory. In *Proceedings of the 1989 Conference on Category Theory and Computer Science, Manchester, U.K.*, volume 389 of *LNCS*. Springer-Verlag, 1989.

[Qia00] H. Qiao. Formalising formulas-as-types-as-objects. In T. Coquand, P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of TYPES'99*, volume 1956 of *Lecture Notes in Computer Science*, pages 174–193. Springer-Verlag, 2000.

[Ram31] F. P. Ramsey. *The Foundations of Mathematics and other logical essays*. Routledge & Kegan, 1931.

[RT93] J. Rutten and D. Turi. On the foundation of final semantics: non-standard sets, metric spaces and partial orders. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *LNCS*, pages 477–530, Berlin, 1993. Springer-Verlag.

[RT94] J. Rutten and D. Turi. Initial algebra and final coalgebra semantics for concurrency. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *LNCS*, pages 530–582, Berlin, 1994. Springer-Verlag.

[Rut01] J. Rutten. Elements of stream calculus (an extensive exercise in coinducton). Technical Report SEN-R0120, CWI, Amsterdam, 2001.

[Ruy99] Mark Ruys. *Studies in Mechanical Verification of Mathematical Proofs*. PhD thesis, Computer Science Institute, University of Nijmegen, 1999.

[Saï97] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *POPL'97: The 12th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 292–301. Association for Computing Machinery, 1997.

[Saï98] Amokrane Saïbi. *Algèbre Constructive en Théorie des Types, Outils génériques pour la modélisation et la démonstration. Application à la théorie des Catégories*. PhD thesis, Université Paris VI, 1998.

[Sco70] Dana Scott. Constructive Validity. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, 1970.

[SH80] J. P. Selding and J. R. Hindley, editors. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[Sli00] K. Slind. Another look at nested recursion. In Harrison and Aagaard [HA00], pages 498–518.

[SS88] A. Salvesen and J. Smith. The strength of the subset type in martin-löf's type theory. In *Proceedings of LICS'88*, pages 384–391. IEEE Computer Society Press, 1988.

[Ste99] Milena Stefanova. *Properties of Typing Systems*. PhD thesis, Computing Science Institute, University of Nijmegen, 1999.

[SU98] Morten Heine B. Sørensen and P. Urzyczyn. Lectures on the curry-howard isomorphism. Available as DIKU Rapport 98/14, 1998.

[Thé97] Laurent Théry. Proving and computing: a certified version of the buchberger's algorithm. Technical report, INRIA, 1997.

[TR98] Daniele Turi and Jan Rutten. On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Mathematical Structures in Computer Science*, 8(5):481–540, 1998.

[Tur36] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proc. London Math. Soc.*, 42:230–265, 1936.

[TvD88] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics*. North Holland, 1988.

[vH67] Jean van Heijenoort, editor. *From Frege to Gödel. A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, Massachusetts, 1967.

[Wer94] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

[Wer97] Benjamin Werner. Sets in types, types in sets. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software. Third International Symposium, TACS'97*, volume 1281 of *LNCS*, pages 530–546. Springer, 1997.

[Wey18] Hermann Weyl. Das kontinuum. Veit, Leipzig, 1918.

[Wig67] Eugene P. Wigner. The unreasonable effectiveness of mathematics in the natural sciences. In *Symmetries and Reflections*, pages 222–237. MIT Press, Cambridge, MA, 1967.

[WR27] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, second edition, 1927.

[Wra89] G. C. Wraith. A note on categorical datatypes. In D. H. Pitt, D. E. Rydehead, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 118–127. Springer-Verlag, 1989.

Index

Samenvatting

Abstractie en Berekening

Type Theorie, Algebraïsche Structuren en Recursieve Functies

Dit is een boek over de formalisering van wiskunde en informatica met behulp van een computer systeem, gebruikmakend van type theorie.

De titel van dit proefschrift verwijst naar twee grondelementen die essentieel zijn in de formele ontwikkeling van wiskunde in type theorie.

Abstractie verwijst naar de behoefte om structuren en ontwikkeling zo algemeen mogelijk te hoden, zodat de resultaten bruikbaar zijn in vele toestanden. Wij pakken deze kwestie aan door een algemeen formulering van verzameling theorie binnen type theorie door het begrip *setoid*, de ontwikkeling van Universele Algebra en equationele redenering.

Berekening verwijst naar het doel om een directe verbinding te houden tussen formele bewijzen en algoritmische inhoud. Type theorie is zelf een geïntegreerde grondslag voor logica en berekening. Wij streven er naar gereedschappen te ontwikkelen die het redeneren over algemene algoritmen in type theorie mogelijk maken. Daarom studeren wij de formulering van algemene recursietheorie en wij presenteren als *case study* een formele verificatie van een computer algebra algoritme, de Snelle Fourier Transformatie.

Dit proefschrift bestaat uit drie delen.

Deel I bevat een introductie tot type theorie, in het bijzonder de *Calculus of Constructions* met inductieve en coinductieve typen. In Hoofdstuk 2 geven wij de formele regels van de type theorie. In Hoofdstuk 3 introduceren wij coinductieve typen, eerst op een intuïtief niveau, dan met zijn formele regels.

Deel II verschaft enkele gereedschappen die nodig zijn om wiskunde in type theorie te formaliseren. In Hoofdstuk 4 bediscussieren wij het probleem om oneindige families van inductieve typen te definieren; wij bieden verschillende oplossingen voor het probleem. In Hoofdstuk 5, gezamelijk werk met Gilles Barthe en Olivier Pons, ontwikkelen wij een theorie van verzamelingen binnen type theorie. Verzamelingen worden afgebeld als paren bestaand uit een type en een daarop gedefineerd gelijkheid. *Setoids* komen in verschillende versies in de literatuur. Wij geven een systematisch behandeling en een wiskundige analyse van de verschillende mogelijkheden. Hoofdstukken 6 en 7 behandelen wij

hetzelfde probleem: hoe kunnen wij algemene recursieve functies in type theorie definiëren. Hoofdstuk 6, gezamelijk werk met Ana Bove, lost het probleem op door het domein van een recursieve functie door een inductief predikaat te karakteriseren. Hoofdstuk 7 lost het probleem op door een coinductief type te gebruiken om oneindige berekeningen weer te geven.

Deel III beschrijft de formele ontwikkeling van delen van de wiskunde in type theorie, namelijk door deze te formaliseren met de stellingbewijzer Coq. In Hoofdstuk 8 ontwikkelen wij Universele Algebra en equationele redeneren over algebrïsche structuren. In Hoofdstuk 9 presenteren wij een implementatie van de Snelle Fourier Transformatie in Coq met een formeel bewijs van zijn correctheid.

Curriculum vitae

Date of Birth: October 12, 1969

Married to Terri Stewart

Home Address: villa Les Hauts de Grasse,
50 Boulevard Georges Clemenceau,
06130 GRASSE,
France

Work Address: INRIA project Lemme,
2004 Route des Lucioles,
BP 093
06902 Sophia Antipolis,
CEDEX,
France

Telephone: +34+4+92385051 (office)
+34+4+93362166 (home)

e-mail address: Venanzio.Capretta@sophia.inria.fr

Education:

- *Laurea in Matematica* (University degree in Mathematics)
  from University of Padua (Italy),
  Department of Pure and Applied Mathematics,
  Thesis title: *Un'analisi costruttiva della prova di normalizzazione del system F di Girard*
  Thesis Supervisor: Prof. Silvio Valentini
  Date of Graduation: November 18, 1996
  Final Graduation mark: 110/110 with honours
- Marktoberdorf Summer School 97
  July 29 to August 10, 1997
- EEF Trends School in Logic and Computation
  Heriot-Watt University, Edinburgh, 10-13 April 1999

- 11th Summer School in Logic, Language and Information (ESSLLI)
  Utrecht University, August 9-20, 1999
- PhD education
  University of Nijmegen, Faculty of Natural Sciences, Mathematics and Compute Science
  Supervisor: Prof. Henk Barendregt
  Start Date: November 1, 1997
  End Date: October 31, 2001
  Date of Graduation: April 23, 2002

Teaching:

- Assistant to the course *Logica II* (Logic II)
  Main teacher: Prof. Silvio Valentini
  Subject: ML Programming and Type Theory
  University of Padua, Department of Pure and Applied Mathematics
  Period: Winter-Spring 1997
- Assistant to the course *T2: Analyse van algoritmen* (Theory 2: Analysis of Algorithms)
  Main teacher: Dr. Dick van Leijenhorst
  University of Nijmegen, Computing Science Department
  Period: Fall-Winter 1999
- Assistant to the course *T1: Talen en Machines* (Theory 1: Languages and Machines)
  Main teachers: Prof. Henk Barendregt and Dr. Herman Geuvers
  University of Nijmegen, Computing Science Department
  Period: Winter-Spring 2001

Refereed Publications:

- Venanzio Capretta and Silvio Valentini,
  A general method for proving the normalization theorem for first and second order typed lambda-calculi,
  Mathematical Structures in Computer Science,
  (9)719-739, December 1999
- Venanzio Capretta,
  Universal Algebra in Type Theory,
  Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99,
  Yves Bertot, Gilles Dowek, André Hirschowits, Christine Paulin, and Laurent Théry (editors),
  LNCS 1690, pages 131-148, Springer, 1999
- Venanzio Capretta,
  Recursive Families of Inductive Types,

Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000,
J. Harrison and M. Aagaard (editors),
LNCS 1869, pages 73-89, Springer, 2000

- Pieter A. M. Seuren, Venanzio Capretta and Herman Geuvers,
  The logic and mathematics of occasion sentences,
  Linguistics and Philosophy, 24:531–595,
  Kluwer Academic Publishers, 2001.
- Ana Bove and Venanzio Capretta,
  Nested General Recursion and Partiality in Type Theory,
  Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001,
  Richard J. Boulton and Paul B. Jackson (editors),
  LNCS 2152, pages 121–135, Springer, 2001
- Venanzio Capretta,
  Certifying the Fast Fourier Transform with Coq,
  Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001,
  Richard J. Boulton and Paul B. Jackson (editors),
  LNCS 2152, pages 154–168, Springer, 2001

Unpublished Papers: (available at http://www.cs.kun.nl/ venanzio)
Equational Reasoning in Type Theory
Setoids in type theory (with Gilles Barthe and Olivier Pons)
Type-theoretic functional semantics (with Yves Bertot and Kuntal Das Barman)

Programming Experience with the Languages: Pascal, C, Prolog, ML.

Experience with the proof tools: Alf, Coq

Experience with the Computer Algebra Systems Maple, Cocoa, Mathematica

Languages Spoken: Italian (native), English, Dutch (State Diploma), French.

Present position: Marie Curie post-doc fellowship, Project Lemme, INRIA Sophia Antipolis, France.