
Sebastian C. Skalberg
An Interactive Proof System for
Map Theory

Ph.D. Dissertation
Faculty of Science
University of Copenhagen

1ST OCTOBER 2002

Preface

This dissertation is submitted in partial fulfillment of the requirements for the Danish Ph.D. degree. It documents work done between August 1999 and September 2002 at the Department of Computer Science at the University of Copenhagen (DIKU).

Acknowledgments

As any other Ph.D. student, I have met plenty of people during the course of my studies, that should be thanked in some way or the other.

First and foremost, my heartfelt thanks to Neil D. Jones for being my strict supervisor and kind boss, during both my master's thesis and my Ph.D. studies.

Also to Klaus Grue, who suggested the topic to me in early 1999, when I was complaining about total functions in HOL, and for helping me with numerous technical questions regarding map theory.

Tobias Nipkow at Technische Universität München having me as a guest in the first half of 2000. Thanks also to the rest of the Isabelle group: Gertrud Bauer, Stefan Berghofer, Gerwin Klein, David von Oheimb, Leonor Prensa Nieto, and Markus Wenzel, as well as my co-guests Giampaolo Bella and Ramaswamy Ramanujam.

Thomas Marthedal Rasmussen, Jakob Grue Simonsen, Peter Møller Neergaard, and Niels Hellraiser Christensen for being good friends and colleagues.

And finally Anders Bystrup, Søren Debois, Dina Friis, Jarl Friis, Morten Fjord-Larsen, Signe Reuss, Christian Stefansen, and Rolf Dyre Svestrup for at least being good friends.

Contents

1	Introduction	5
1.1	This Thesis	6
1.2	Syntactic Issues	7
1.3	Map Theory	8
1.3.1	The Constant Nil	9
1.3.2	Equality	10
1.3.3	Ordering maps	13
1.3.4	The Deduction Theorem	14
1.3.5	Classical Maps	14
1.3.6	The Choice Operator	15
1.3.7	Relative Consistency of Map Theory	15
2	Preliminary Types	16
2.1	Meaninglessness	17
2.2	Meaningfulness	17
2.3	Strong vs. Weak Representatives	18
2.4	Truth	19
2.5	Definedness	19
2.6	Functions	20
2.6.1	Some Typed Functions	21
2.7	Basic Pairs	22
2.7.1	Elimination	25
2.7.2	fst and snd	26
2.8	The Root Type	26
2.9	Booleans	28
2.9.1	Conditional	28
2.9.2	Case Analysis	29
2.10	Order	31
2.11	Equality	32
2.12	General Proof Techniques	34

3	Propositional and Predicate Logic	38
3.1	Propositional Logic	38
3.1.1	Non-Strict Connectives	41
3.2	Predicate Logic	42
3.2.1	Classical Maps	42
3.2.2	Some Intuition	44
3.2.3	Types and Quantifiers	45
3.2.4	The Missing Function Type Constructor	49
4	Set Theory	51
4.1	Finite Sets	52
4.2	Comprehension	52
4.3	Union	53
4.4	Power Set	53
4.5	Infinity	53
4.6	Choice Operator	54
4.7	Axioms	54
5	Natural Numbers	55
5.1	Defining the Natural Numbers	55
5.2	Arithmetic	66
6	Pairs and Lists	68
6.1	Pairs	68
6.2	Lists	71
7	Real Numbers	75
7.1	Non-Negative Rationals	75
7.2	Rationals	76
7.3	Reals	77
8	Implementation	81
8.1	Isabelle	82
8.1.1	An Example	83
8.2	Initial Development	84
8.3	Tools	87
8.3.1	Formalizing the Types	87
8.3.2	The Simplifier	89
8.3.3	The Classical Reasoner	91
8.4	Assessment	92
9	Conclusion	94
9.1	Future Work	94

Chapter 1

Introduction

While set theory has proved its worth as a foundational logic for mathematics, it has only been a partial success as a foundational logic for computer science. The reason is primarily that recursive functions, some of the main objects of study in computer science, are not easily modeled in set theory, witness the elaborate constructions of domain theory.

Alonzo Church tried in the beginning of the 1930's [Chu32, Chu33] to construct a foundational logic for mathematics, that had functions as the basic building blocks, rather than sets. Unfortunately, he never succeeded [KR35], although a byproduct of his efforts, the lambda calculus, has proved to be a very handy tool in discussing and expressing functions.

After Gödel's death blow to Hilbert's foundational program [Göd31], interest steadily turned away from foundational logics and a plethora of smaller (weaker) special-purpose logics have emerged instead. One of the oldest is Higher Order Logic, the logic that was the immediate successor of Church's earlier efforts [Chu40].

In computer science today, Higher Order Logic is heavily used in formal verification. Systems like PVS, HOL, and Isabelle/HOL, probably the currently three most widely used theorem provers, all have Higher Order Logic as their foundation.

A key reason for the success of Higher Order Logic in verification of software and, especially, hardware verification is that it has functions as the object of study, albeit only total functions. In practice, however, this is not a problem, since the functions implemented in software systems will always be total anyway.

Where Higher Order Logic may fail to deliver, however, is not in modeling the computer systems that we wish to verify, but in representing the specifications that these systems must meet. In a nutshell, while any computer program that we in fact sit down to verify can only approximate any infinite structure, we might want to express the properties that should hold using the infinite, non-constructive structures that the program is approx-

imating.

This, then has been the problem: One could either use set theory, which was poor at modeling functions, but great at modeling specifications, or Higher Order Logic, which is great at modeling programs, but may be too restrictive on the specification side.

Map theory [Gru92] is a logic that has the strength of set theory and has the lambda calculus as a syntactic sub theory. It was first published in 1992 by Klaus Grue, and is, in fact, a fulfillment of Church's goal from then 1930's. In theory, then, it should beat both set theory and Higher Order Logic as a logic for verifying computer systems.

1.1 This Thesis

In 1999, when I started my Ph.D. studies, this was the thesis:

A proof system with map theory as its foundation will allow for shorter and more intuitive proofs than present proof systems.

The task was clearcut: Implement a proof system to aid in proving theorems in map theory, prove a lot of theorems with it, and compare them with proofs in other logics. From work on my master's thesis I was already familiar with the generic theorem prover Isabelle, so the plan was to make a prototype using this system.

In practice, however, the map theory of 1992 was very ill-suited for implementing in a proof system. The reason was that one of the key lemmas (the Totality Lemma), allowing one to prove that a specific map was well-founded, in a sense, had such a horrid formulation that it was effectively unimplementable in higher order abstract syntax, the syntax used in Isabelle. Further, trying to implement it using de Bruijn indexes [dB72] proved equally doomed to fail.

In late 2001, a new version of map theory emerged [Gru01], which was a vast improvement on the original version. This is the version that I have based my dissertation work on. I have succeeded in formalizing the entire article (roughly 150 pages of pure Hilbert-style proofs), including the development of ZFC set theory within map theory, as well as a fair chunk of elementary number theory.

The goal of the following presentation is to give the reader an insight into how this formalization came to be. Since the main article [Gru01] has plenty of very low-level proofs in map theory, I have decided to let the following proofs be comparably high-level. Also, I skip any proofs that are "standard". For example, once I have presented my implementation of the natural numbers and proved that the usual properties hold, I trust that the reader will allow me not to write down, say, yet another development of elementary number theory. Readers interested in the gory details can always look in the source code of my system (see Chapter 8 for a reference).

The rest of dissertation is as follows:

Chapter 1 The next section will briefly tell about the syntactic conventions used in the remaining presentation, and the rest of the chapter is an introduction to map theory. The presentation is quite informal; The intent is to give an intuitive understanding, or feel, for map theory.

Chapter 2 Is about how types are represented in map theory. It also defines some of the basic types, as well as present some general proof techniques.

Chapter 3 Concerns propositional and predicate logic. Primarily, time is spent on how to formalize quantifiers for general types.

Chapter 4 Is a sketch of the ZFC formalization. Just the basic definitions and an intuitive understanding of how ZFC sets are modeled in map theory are given.

Chapter 5 Natural numbers are introduced. Also, basic properties of recursive definitions is discussed.

Chapter 6 This presents the definition of typed pairs as well as typed lists.

Chapter 7 The non-negative rationals, the rationals, and finally the real numbers are introduced.

Chapter 8 Gives an overview for the implementation in Isabelle.

Chapter 9 Concludes.

1.2 Syntactic Issues

Map theory is theory of functions, and we will use notation known from the lambda calculus literature. To wit, our syntactic word is divided into terms and variables. Variables will always be written with small letters, possibly subscripted or primed: x , p_2 , t' . Terms will be denoted by capital letters, again, also possibly subscripted or primed: A , F_3 , Z'' . Further, we shall use higher order abstract syntax [PE88]. For example, the term

$$\lambda x.A$$

denotes the abstraction of x over the term A where x does not occur free in A . On the other hand, the term

$$\lambda x.A x$$

denotes the abstraction of x over a term, Ax that may depend on x . For these higher order terms, we shall use scripted letters \mathcal{A} , \mathcal{F} , \mathcal{L} , as well as include the arguments in brackets. The term above will then be written

$$\lambda x.\mathcal{A}(x)$$

and we reserve the normal, capital letter A , for the first-order term. When we want to talk about the higher order term as a function, without applying it to a term, we shall just write $\mathcal{A}(\cdot)$.

1.3 Map Theory

Map theory is first and foremost the classic, untyped lambda calculus extended so as to be strong enough to be comparable to ZFC set theory. Syntactically, this means that the lambda calculus is a sub theory of map theory, and we shall use the syntax of the lambda calculus, writing $\lambda x.\mathcal{A}(x)$ for the binding of x in $\mathcal{A}(x)$, and $A'B$ for the application of the term A to the term B .

We note that all terms of the lambda calculus are also terms of map theory, and we shall later see that they behave just as they always have. For example, we have the following, prominent maps in map theory:

I	$\stackrel{\text{def}}{=} \lambda x.x$	Identity
K	$\stackrel{\text{def}}{=} \lambda xy.x$	K-combinator
Y	$\stackrel{\text{def}}{=} \lambda f.(\lambda x.f'(x'x))'(\lambda x.f'(x'x))$	Fixed Point Operator
\perp	$\stackrel{\text{def}}{=} (\lambda x.x'x)'(\lambda x.x'x)$	Bottom

In the literature of lambda calculus, one often finds the map we denote by \perp denoted instead by Ω . However, since the map in question best can be thought of as either undefined or divergent, the map has been given a slightly non-standard denotation. When talking about \perp in running text, we shall call it “bottom” or “undefined”.

Like the lambda calculus, map theory is an equational theory. Hence, judgments in map theory are of the form

$$A = B$$

where $=$ is an equivalence relation, i.e., reflexive, symmetric, and transitive. We also have the familiar rule

$$(\lambda x.\mathcal{A}(x))'B = \mathcal{A}(B) \quad (\beta\text{-reduction})$$

With this, we can prove

$$\begin{aligned}
Y^{\cdot}A &= (\lambda f.(\lambda x.f^{\cdot}(x^{\cdot}x))^{\cdot}(\lambda x.f^{\cdot}(x^{\cdot}x)))^{\cdot}A \\
&= (\lambda x.A^{\cdot}(x^{\cdot}x))^{\cdot}(\lambda x.A^{\cdot}(x^{\cdot}x)) \\
&= A^{\cdot}((\lambda x.A^{\cdot}(x^{\cdot}x))^{\cdot}(\lambda x.A^{\cdot}(x^{\cdot}x))) \\
&= A^{\cdot}((\lambda f.(\lambda x.f^{\cdot}(x^{\cdot}x))^{\cdot}(\lambda x.f^{\cdot}(x^{\cdot}x)))^{\cdot}A) \\
&= A^{\cdot}(Y^{\cdot}A)
\end{aligned}$$

as usual. As expected, we also have

$$\begin{aligned}
I^{\cdot}A &= A \\
K^{\cdot}A^{\cdot}B &= A
\end{aligned}$$

for all terms A and B .

In general, whenever two terms are β -equivalent, they are also equal in map theory. However, the converse cannot hold if we are to move beyond the strength of the lambda calculus. And, indeed, letting

$$F_n \stackrel{\text{def}}{=} Y^{\cdot}(\lambda f x_1 \dots x_n.f)$$

we shall later see that we can prove

$$F_2 = F_3$$

though F_2 and F_3 are not β -equivalent.

1.3.1 The Constant Nil

The lambda calculus is, as a logic, very weak. It turns out that by adding new constants to it, one can extend it to a logic as powerful as set theory. In fact, it suffices to introduce just a single, new constant. While this does not change the strength of the computable part of map theory (the lambda calculus is, after all, already Turing complete), this new constant will be convenient, even when implementing computable concepts.

The constant added will be denoted by N , and we shall pronounce it “nil”. To be interesting to us, we need to be able to distinguish it from the lambda terms already in the system. To this end, the conditional

ifnil A then B else C

is introduced to map theory, together with N . It’s semantics are defined by the rules

$$\begin{aligned}
\mathbf{ifnil\ } N \mathbf{\ then\ } B \mathbf{\ else\ } C &= B \\
\mathbf{ifnil\ } (\lambda x.A(x)) \mathbf{\ then\ } B \mathbf{\ else\ } C &= C \\
\mathbf{ifnil\ } \perp \mathbf{\ then\ } B \mathbf{\ else\ } C &= \perp
\end{aligned}$$

Conceptually, then, **ifnil** reduces it's first argument until it can determine whether it is nil or not. It suffices to reduce this first argument to weak head normal form. If it has no such normal form, we consider it equal to \perp , and **ifnil** will not terminate.

While not strictly necessary, it will be convenient to define the result of applying nil to another term as nil itself, and likewise for \perp :

$$\begin{aligned} \mathbf{N}'B &= \mathbf{N} \\ \perp'B &= \perp \end{aligned}$$

(We could define a function, **apply**, by $\mathbf{apply}(A, B) \stackrel{\text{def}}{=} \mathbf{ifnil} A \mathbf{then} \mathbf{N} \mathbf{else} A'B$, and use this in all our definitions.)

1.3.2 Equality

In addition to calling \mathbf{N} nil and \perp bottom, we shall call any term on the form $\lambda x.\mathcal{A}(x)$ a “function”. Note that, though normally the words “map” and “function” are frequently taken as synonymous, we make a slight distinction in the context of map theory: Any function is a map, but not all maps are functions: nil, for example, is a map but not a function.

Since equal terms are indistinguishable in map theory, we shall extend our nomenclature such that any term equal to nil shall be called nil. Likewise, any term equal to bottom shall be called bottom and any term equal to a function shall, itself, be called a function.

We can now state one of the fundamental theorems of map theory:

Theorem 1.3.1 *Given any function A , the maps nil, bottom, and A are pairwise distinct.*

Argument Assume that it was not true. Assume, for example, that the function $\lambda x.\mathcal{A}(x)$ was equal to bottom. Then all terms B would be provably equal to bottom, and hence equal:

$$\begin{aligned} B &= \mathbf{ifnil} (\lambda x.\mathcal{A}(x)) \mathbf{then} \perp \mathbf{else} B \\ &= \mathbf{ifnil} \perp \mathbf{then} \perp \mathbf{else} B \\ &= \perp \end{aligned}$$

Thus, assuming map theory is consistent, the theorem above holds. \square

In fact, we not only have that bottom, nil, and any function are distinct; We have that any closed term must be one of the three:

Theorem 1.3.2 (Quantum Non Datur) *A closed term of map theory is either nil, bottom, or a function.*

In the following, we shall not explicitly distinguish between closed and open terms: We tactically assume all terms to be closed. That is, when

stating and proving theorems we shall assume all terms to be closed. Thus, a theorem about some, possibly open, map A will actually be a theorem about all closed instantiations of A .

We can use the previous theorem to do case analysis on terms:

Theorem 1.3.3 *If*

$$\mathcal{A}(\mathbf{N}) = \mathcal{B}(\mathbf{N})$$

$$\mathcal{A}(\lambda x.C'x) = \mathcal{B}(\lambda x.C'x)$$

and

$$\mathcal{A}(\perp) = \mathcal{B}(\perp)$$

then

$$\mathcal{A}(C) = \mathcal{B}(C)$$

Proof. By Theorem 1.3.2, the map C is either nil, bottom, or a function. If C is a function, then C is equal to $\lambda y.\mathcal{D}(y)$ for some context $\mathcal{D}(\cdot)$. Hence,

$$\begin{aligned} \lambda x.C'x &= \lambda x.(\lambda y.\mathcal{D}(y))'x \\ &= \lambda x.\mathcal{D}(x) \\ &= C \end{aligned}$$

Thus, in any case, $\mathcal{A}(C) = \mathcal{B}(C)$ follows from one of the premises. \square

We can use Theorem 1.3.3 to prove, e.g.,

Lemma 1.3.4 *For all maps A we have **ifnil** A **then** \perp **else** $\perp = \perp$.*

Proof. From the semantics of **ifnil** we have

$$\mathbf{ifnil} \mathbf{N} \mathbf{then} \perp \mathbf{else} \perp = \perp$$

$$\mathbf{ifnil} (\lambda x.A'x) \mathbf{then} \perp \mathbf{else} \perp = \perp$$

and

$$\mathbf{ifnil} \perp \mathbf{then} \perp \mathbf{else} \perp = \perp$$

From these, **ifnil** A **then** \perp **else** $\perp = \perp$ follows directly from Theorem 1.3.3. \square

We are now ready to handle the notion of equality in map theory. Since two terms are equal if they are indistinguishable, we shall first consider when we *can* tell two terms apart.

Using the **ifnil** construction, we can obviously tell nil, bottom, and a function apart. Clearly, however, not all functions are equal. To distinguish two functions, we must find some input on which they differ. Using this reasoning recursively, two maps are different if, when applied to some series

of maps, they return different output, that is: $A \neq B$ exactly when there exists maps C_1, \dots, C_n such that

$$(\dots(A'C_1)\dots)'C_n \neq (\dots(B'C_1)\dots)'C_n$$

To express the converse, we shall introduce the construct

$$\mathbf{root}(A) = \mathbf{ifnil} A \mathbf{then} N \mathbf{else} \lambda x.N$$

Consequently, we have

$$\mathbf{root}(A) = \begin{cases} N & \text{if } A \text{ is nil} \\ \lambda x.N & \text{if } A \text{ is a function} \\ \perp & \text{if } A \text{ is bottom} \end{cases}$$

Also, by Theorem 1.3.1, we have that $\mathbf{root}(A)$ is equal to $\mathbf{root}(B)$ exactly when either A and B are both nil, both bottom, or both functions. Now the equality $A = B$ can be characterized by, for all n and C_1, \dots, C_n :

$$\mathbf{root}(A'C_1'\dots'C_n) = \mathbf{root}(B'C_1'\dots'C_n)$$

We call $\mathbf{root}(A)$ the *root* of A .

When thinking of terms, it is beneficiary to think of the term A as a tree with each branch labeled by a map and the node along the path C_1, \dots, C_n labeled by the root of $A'C_1'\dots'C_n$. For example, the tree of bottom is one where all nodes are labeled \perp .

Two terms are equal exactly when their trees are equal. This is utilized in the following theorem:

Theorem 1.3.5 (Extentionality) *If*

$$\mathbf{root}(\mathcal{A}(C)) = \mathbf{root}(\mathcal{B}(C))$$

for all maps C and further there exists a context $\mathcal{D}(\cdot)$ such that

$$(\mathcal{A}(C))'C' = \mathcal{A}(\mathcal{D}(C, C'))$$

and

$$(\mathcal{B}(C))'C' = \mathcal{B}(\mathcal{D}(C, C'))$$

for all maps C and C' , then

$$\mathcal{A}(C) = \mathcal{B}(C)$$

holds for all D .

Argument When the premises of the extensionality theorem just stated are satisfied, we can prove that

$$\begin{aligned}
\mathbf{root}(\mathcal{A}(C) \dot{C}_1 \dot{\dots} \dot{C}_n) &= \mathbf{root}(\mathcal{A}(\mathcal{D}(C, C_1)) \dot{C}_2 \dot{\dots} \dot{C}_n) \\
&= \mathbf{root}(\mathcal{A}(\mathcal{D}(\mathcal{D}(C, C_1), C_2)) \dot{C}_3 \dot{\dots} \dot{C}_n) \\
&\quad \vdots \\
&= \mathbf{root}(\mathcal{A}(\mathcal{D}(\dots (\mathcal{D}(\mathcal{D}(C, C_1), C_2), \dots, C_n)))) \\
&= \mathbf{root}(\mathcal{B}(\mathcal{D}(\dots (\mathcal{D}(\mathcal{D}(C, C_1), C_2), \dots, C_n)))) \\
&\quad \vdots \\
&= \mathbf{root}(\mathcal{B}(\mathcal{D}(\mathcal{D}(C, C_1), C_2)) \dot{C}_3 \dot{\dots} \dot{C}_n) \\
&= \mathbf{root}(\mathcal{B}(\mathcal{D}(C, C_1)) \dot{C}_2 \dot{\dots} \dot{C}_n) \\
&= \mathbf{root}(\mathcal{B}(C) \dot{C}_1 \dot{\dots} \dot{C}_n)
\end{aligned}$$

That is, the trees of $\mathcal{A}(C)$ and $\mathcal{B}(C)$ are equal. Thus, $\mathcal{A}(C)$ and $\mathcal{B}(C)$ are equal. \square

When using the extensionality theorem, the first part, showing that the roots of $\mathcal{A}(C)$ and $\mathcal{B}(C)$ are equal for all maps C , is normally the easy part. The trouble is finding the context $\mathcal{D}(\cdot)$, expressing how the output of the functions $\lambda x.\mathcal{A}(x)$ and $\lambda x.\mathcal{B}(x)$ depends on the input. Thinking in terms of trees (no pun intended), $\mathcal{D}(\cdot)$ expresses how the trees of $\lambda x.\mathcal{A}(x)$ and $\lambda x.\mathcal{B}(x)$ can be collapsed in a uniform way (this should be evident in the argument for the extensionality theorem).

On a final note on equalities, we shall often write “we have proved A ”, though the only judgment in map theory is equality. What we mean by the statement is “we have proved $A = \mathbf{True}$,” where \mathbf{True} is the term representing truth. This convention will be in use everywhere we just write a map where an equality is expected.

1.3.3 Ordering maps

We can, as seen earlier, define a fixed point operator in map theory that is completely analogous to the fixed point operator found in the lambda calculus. Up until now, the only thing we can say about the result of applying Y is that it returns *a* fixed point, but nothing else. We shall now define an ordering (\preceq) on maps that allows us to reason that the fixed point operator returns the *least* fixed point with respect to this ordering. For all fixed points B of A , we shall have

$$Y \dot{A} \preceq B$$

The ordering that will be defined is one of the information level contained in the terms considered. Recalling that \perp is to be thought of as either infinitely looping or undefined, it should be natural that \perp be the least element in this ordering. Since \mathbf{N} is just as informative as any term $\lambda x.\mathcal{A}(x)$,

it follows that such two terms are unrelated (since they are not equal). A lambda term $\lambda x.\mathcal{A}(x)$ is less than a lambda term $\lambda x.\mathcal{B}(x)$ exactly when $\mathcal{A}(C)$ is less than $\mathcal{B}(C)$ for all maps C . Pictorially, we have

$$\begin{array}{ccc} \mathbf{N} & & \lambda x.\mathcal{A}(x) \\ & \searrow & \nearrow \\ & \perp & \end{array}$$

We also have that all maps are monotone, in the sense that

$$A \preceq B$$

implies

$$\mathcal{C}(A) \preceq \mathcal{C}(B)$$

for all maps A, B , and contexts $\mathcal{C}(\cdot)$.

Since **True** is modeled by **nil**, this implies that equality in map theory is not definable as term, for assume that $\text{eq } A B$ was true exactly when A was equal to B . Then, since $\perp = \perp$, we would have $\text{eq } \perp \perp = \text{True}$, and by monotonicity of eq and the maximality of **True**, we would have $\text{eq } A B = \text{True}$ for all maps A and B .

1.3.4 The Deduction Theorem

By defining

$$A \triangleright B = C \stackrel{\text{def}}{=} \text{ifnil } A \text{ then } B \text{ else } \perp = \text{ifnil } A \text{ then } C \text{ else } \perp$$

we have that $A \triangleright B = C$ is equivalent to $B = C$ when A is true (**nil**) and equivalent to $\perp = \perp$ otherwise. Consequently, \triangleright is a kind of implication: It is valid if A is not true or if $B = C$. We even have

Theorem 1.3.6 (Deduction) *If, assuming A , one can prove $B = C$ then $A \triangleright B = C$.*

1.3.5 Classical Maps

So far, the system we have described has been no stronger than the lambda calculus we originally started with. The strength of map theory comes from its choice operator on a special class \mathcal{C} of maps which we call classical. In a very real sense, the classical maps can be thought of as the well-founded sets of set theory, with **nil** playing the role of the empty set. Indeed, the classical maps are defined with the explicit purpose of giving a model of set theory in map theory.

To get an intuitive understanding of the classical maps, it helps to understand what the properties of the classical maps are, that allow us to use them in constructing this model.

First, nil is a classical map. This is analogous to first stating that the empty set is well-founded. We will also model the empty set by nil .

Second, \mathfrak{C} is closed under application. That is, applying a classical map to another will again yield a classical map. And, even stronger, the image of \mathfrak{C} under a classical map will be sufficiently small as to be considered a set. We then let the classical map A represent the set

$$\{A'B \mid B \in \mathfrak{C}\}$$

Hence, all classical maps represent some set.

It should now be easy to see how the identity function represents the class of all sets, and how the K-combinator represents the class of singleton sets. Consequently, neither the identity function nor the K-combinator are classical.

1.3.6 The Choice Operator

As the primitive operator on the classical maps, we have the choice operator:

$$\varepsilon x.\mathcal{A}(x)$$

The semantics of this term is as follows: If $\mathcal{A}(B)$ is defined, i.e., not \perp , for all classical maps B then $\varepsilon x.\mathcal{A}(x)$ is classical. If further there exists classical B such that $\mathcal{A}(B)$ is true, then $\varepsilon x.\mathcal{A}(x)$ is such a B , otherwise $\varepsilon x.\mathcal{A}(x)$ chooses an arbitrary classical map. If $\mathcal{A}(B)$ is undefined for any classical B then the value of $\varepsilon x.\mathcal{A}(x)$ is unknown. Note that this does not mean undefined, it means $\varepsilon x.\mathcal{A}(x)$ in this case returns some arbitrary term. This may be defined or not, classical or not, we have no way of telling.

Even when there exists a classical B such that $\mathcal{A}(B)$ is true, we can say no more about $\varepsilon x.\mathcal{A}(x)$ than that $\mathcal{A}(\varepsilon x.\mathcal{A}(x))$ holds. For example, the choice operator need not be minimal among the classical maps satisfying A .

Note also, that A need not itself be classical. We have, for example,

$$\varepsilon x.\lambda x.\perp$$

is classical, even though $\lambda x.\perp$ is practically as unclassical as terms get (only surpassed by \perp itself).

1.3.7 Relative Consistency of Map Theory

While out of scope for this thesis, a few words on the relative consistency of map theory are in order here. As we have mentioned earlier, and we shall see later, there exists a model of ZFC in map theory. This proves that ZFC is consistent relative to map theory. Conversely, a model of map theory can be constructed in ZFC extended with assumption of the existence of a strongly inaccessible cardinal (SI). Thus, map theory is consistent, relative to ZFC+SI. Full details of the construction of a model for map theory can be found in [BG97].

Chapter 2

Preliminary Types

Foundational logics always pick out a number of concepts (frequently just a few or maybe even a single one) as primitive, and define all other mathematical concepts by them. Thus, according to set theory everything is a set, in type theories (intuitionistic or otherwise) everything boils down to types, and in map theory everything is a map.

This does not mean, of course, that set theorists deny the existence of functions, that type theorists do not consider sets interesting, or that map theorists (the few that exist) have no use for types. It just means that functions are represented by certain sets in set theory, sets belong to a certain type in type theory, and types are just special maps in map theory.

Maps are the building blocks of map theory, and thus functions are readily expressible in map theory. Almost as primitive as maps are sets in map theory. While they are, of course, ultimately maps, the foundations of map theory are constructed with the explicit content of proving that set theory is part of map theory (and thus that map theory has the strength of set theory). Of the three foundational concepts mentioned above, this leaves us with the question of how to model types in map theory.

Like set theory, map theory is fundamentally an untyped logic (or a mono-typed logic, if one prefers). That is, there is only one syntactic group of terms, and thus, syntactically, one can interchange any sub term of a term with any other and still get a term back. Compare this to typed logics, where terms are, within the foundational system, assigned a (possibly generic) type, and only sub terms of the same type may be interchanged.

In map theory, in other words, if both B and $A(B)$ are valid terms, then $A(C)$ will also be a valid term whenever C is. In typed logics, this will only hold if B and C are of the same type. To be more specific, in HOL, say, the expression

if True then B else C

is a valid term, while the expression

if 1 then B else C

is ill-typed and thus not a valid term. Thus, as a user of HOL, one does not need to explicitly handle a large class of meaningless terms, namely those where the term is ill-typed. In map theory, on the other hand, if we have boolean conditionals and natural numbers, then both the above terms are completely valid, and we need to assign a meaning to them both.

2.1 Meaninglessness

As we have just seen, we have no way of keeping conceptually ill-typed terms out of our system. There are also other meaningless terms, $1/0$, say, or dubious definitions like $P \stackrel{\text{def}}{=} \{x \mid x \in P \leftrightarrow x \notin P\}$. In all cases, we choose to let the terms in question be equal to bottom. In other words, bottom will be the canonical undefined/ill-typed/meaningless term.

This makes sense in several ways. By the minimality of the Y-combinator, many ill-defined terms are equal to \perp already anyway. The absurd definition $\mathcal{A}(B) \stackrel{\text{def}}{=} \mathcal{A}(\mathcal{C}(B))$ (this could be a basically sound recursive definition where the base case was forgotten), for example, is easily satisfied by letting $\mathcal{A}(B)$ be equal to $Y(\lambda f x. f(\mathcal{C}(x)))B$, which is bottom for all B . Also, were we to allow bottom to represent some meaningful element of a type, this would lead to all sorts of problems: Assume that A was another element of the same type, and $\mathcal{S}(\cdot)$ swapped \perp and A , i.e., $\mathcal{S}(A) = \perp$ and $\mathcal{S}(\perp) = A$. Certainly such a function should be definable for any type. But then $\mathcal{S}(\perp) \preceq \mathcal{S}(A) = \perp$, and hence $A = \mathcal{S}(\perp) = \perp$ (remember, bottom is the minimal map). So, if bottom represents a meaningful element of a type then no swap function can be defined for this type.

2.2 Meaningfulness

Having thus argued for the elimination of bottom from all types, do we need to eliminate other maps? The answer is no, we shall later see types where all defined maps are meaningful. However, other types will exclude some defined maps, and we will need to be able to distinguish meaningful terms from meaningless terms within the system. That is, for any type α and map A we want a term,

$$A \text{ is } \alpha$$

to express whether A is a meaningful term of type α (hereafter just: whether A is of type α) or not.

Preferably, $A \text{ is } \alpha$ would be equal to **True** if A was of type α and **False** if not. However, when A was of the desired type, this would imply $\text{False} = \perp \text{ is } \alpha \preceq A \text{ is } \alpha = \text{True}$, which, while possible, would be awkward. Besides the fact that, under our interpretation of \preceq , this would mean that **False** was less defined than **True** (a somewhat strange assertion), modeling **True** by nil

has distinct advantages and would not be possible if $\text{False} \preceq \text{True}$ was to hold.

Another possibility would be to have $A \text{ is } \alpha$ be true if A was of type α and either bottom or false if not. Since there are types where the type membership is strictly semi-decidable anyway (it conceptually takes forever to decide that a term is not of the type), and for general simplicity, we chose $A \text{ is } \alpha$ to be true if A is of type α and to be undefined otherwise.

2.3 Strong vs. Weak Representatives

Ideally, when modeling types, the representation of the elements would be unique. After all, there is only *one* number four. Unfortunately, this will hardly ever be possible in map theory. The intuitive reason is that when deciding which member of a certain type a term represents, only a finite part of the term tree is examined, and so any two terms that are equal in that part of the tree will be considered equal under the type.

In general, therefore, an element of a type will have several representatives. Rather than choosing a random representative each time we need one, we shall designate one of the them as the *strong* representative, and shall call all other representatives *weak*. It will be convenient to be able to pick out the strong representative of an element within map theory. To this end, for each type α we define normalization function, $\approx_\alpha(\cdot)$ such that $\approx_\alpha(A)$ strongly represents the element that A represents, and is undefined if A represents no element of α . If A already is a strong representative, it follows that $\approx_\alpha(A)$ must be equal to A , and thus that $\approx_\alpha(\cdot)$ must be idempotent. In fact, we shall use the properties just described as the basis for defining what we shall call simple types in map theory:

Definition 2.3.1 (Simple Type) α is a simple type if two functions

$$\cdot \text{ is } \alpha$$

and

$$\approx_\alpha(\cdot)$$

are given with the following properties:

- $\approx_\alpha(\cdot)$ is idempotent, i.e., $\approx_\alpha(\approx_\alpha(A)) = \approx_\alpha(A)$ for all terms A .
- $\approx_\alpha(\cdot)$ is strict, i.e., $\approx_\alpha(\perp) = \perp$.
- $A \text{ is } \alpha$ is true if $\approx_\alpha(A)$ is defined, and bottom otherwise.

(Slightly abusing the language, we shall call any map in the range of $\approx_\alpha(\cdot)$ a strong representative, that is, we shall also call \perp a strong representative. If we need to talk about the strong representatives of α besides bottom, we shall call them “strong α ”, e.g., strong booleans or strong natural numbers.)

With this definition in hand, we can start formalizing some of the simple types that we have already talked about.

2.4 Truth

Truth is the simplest type we shall have occasion to see here: It only has a single element, and this is even represented by a unique map.

Definition 2.4.1 (*true*)

$$\begin{aligned} \text{True} &\stackrel{\text{def}}{=} N \\ \approx_{\text{true}}(A) &\stackrel{\text{def}}{=} \text{ifnil } A \text{ then } N \text{ else } \perp \\ A \text{ is true} &\stackrel{\text{def}}{=} \text{ifnil } A \text{ then True else } \perp \end{aligned}$$

Note that the definitions of $\approx_{\text{true}}(\cdot)$ and $\cdot \text{ is true}$ are, in fact, identical. This is one of the side effects, mentioned earlier, of the untypedness of map theory. Conceptually, the two functions are different: One returns a map, the other a boolean value. Both may never terminate.

That it really is a simple type, is captured in the following theorem.

Theorem 2.4.2 *true is a simple type.*

Proof. By case analysis. □

Also, note that $\approx_{\text{true}}(\text{True}) = \text{True}$ and True is true . That is, True is the strong representative of truth.

Of course the definition of True will be the same as in the booleans, which we will define in the next chapter. We shall see that true can be considered a subtype of the booleans, but true is a type in its own right, and it will be ubiquitous in rest of the development.

2.5 Definedness

The second type we can define now is the type of definedness. In our formulation, all defined maps strongly represent themselves. We could have chosen any definition of $\approx_{\text{defined}}(\cdot)$, as long it mapped any defined map to some defined map and it was strict, since we shall have no use for normalizing defined maps; Only whether a map is defined or not shall be of interest to us. The present normalization function will be handy, though, when dealing with type of functions, which we shall deal with later.

Definition 2.5.1 (*defined*)

$$\begin{aligned} \approx_{\text{defined}}(A) &\stackrel{\text{def}}{=} A \\ A \text{ is defined} &\stackrel{\text{def}}{=} \text{ifnil } A \text{ then True else True} \end{aligned}$$

And, like truth, it is a simple type:

Theorem 2.5.2 defined *is a simple type*.

Proof. By case analysis. □

With truth and definedness defined, we can express the relationship between \cdot **is** α and $\approx_\alpha(\cdot)$ in a single equation:

$$A \text{ is } \alpha = \approx_\alpha(A) \text{ is defined}$$

By unfolding the definition of \cdot **is** defined, we can express A **is** α in terms of $\approx_\alpha(A)$:

$$A \text{ is } \alpha = \text{if nil } \approx_\alpha(A) \text{ then True else True}$$

One could simplify the definition of simple type given earlier by using the above equation as a definition of \cdot **is** α . However, in the case of recursive definitions, it will be convenient to keep $\approx_\alpha(\cdot)$ and \cdot **is** α separate.

2.6 Functions

Since we have now defined what is meant by a type, the next question is how to use types in practice. To answer this, one must look at typed functions: When told that a function $\mathcal{F}(\cdot)$ is of type $\alpha \Rightarrow \beta$, what does this mean in a basically untyped setting? That $\mathcal{F}(\cdot)$ expects an argument of type α means that the result of applying $\mathcal{F}(\cdot)$ to something of not of type α should be undefined:

$$A \text{ is } \alpha = \perp \implies \mathcal{F}(A) = \perp \tag{2.1}$$

Further, for $\mathcal{F}(\cdot)$ to be well-defined, if A and A' represent the same element of α , then $\mathcal{F}(\cdot)$ should return the same element of β :

$$\approx_\alpha(A) = \approx_\alpha(A') \implies \approx_\beta(\mathcal{F}(A)) = \approx_\beta(\mathcal{F}(A')) \tag{2.2}$$

In for this last implication, we shall demand something slightly stronger, namely that $\mathcal{F}(\cdot)$ always returns strong representatives of β . The properties $\mathcal{F}(\cdot)$ must obey are gathered in the following definition

Definition 2.6.1 (Function Type) *A function $\mathcal{F}(\cdot)$ has type $\alpha \Rightarrow \beta$ if α and β are simple types and if further $\mathcal{F}(\cdot)$*

- *is strict, i.e., $\mathcal{F}(\perp) = \perp$,*
- *returns equal results for equivalent arguments, i.e., if $\approx_\alpha(A) = \approx_\alpha(A')$ then $\mathcal{F}(A) = \mathcal{F}(A')$, and*
- *always returns strong representatives of β , i.e., $\approx_\beta(\mathcal{F}(A)) = \mathcal{F}(A)$ for all maps A .*

From this definition, we immediately see that any function $\mathcal{F}(\cdot)$ of type $\alpha \Rightarrow \beta$ satisfies Equation 2.2. Equation 2.1 also holds, by the strictness of $\mathcal{F}(\cdot)$: If A is α is undefined then $\approx_\alpha(A) = \perp = \approx_\alpha(\perp)$ and so $\mathcal{F}(A) = \mathcal{F}(\perp) = \perp$.

We shall sometimes want a more lax functional type, where we do not demand that the function be strict:

Definition 2.6.2 (Function Quasi-Type) *A function $\mathcal{F}(\cdot)$ has quasi-type $\alpha \Rightarrow \beta$ if α and β are simple types and if further $\mathcal{F}(\cdot)$*

- *returns equal results for equivalent arguments, i.e., if $\approx_\alpha(A) = \approx_\alpha(A')$ then $\mathcal{F}(A) = \mathcal{F}(A')$ and*
- *always returns strong representatives of β , i.e., $\approx_\beta(\mathcal{F}(A)) = \mathcal{F}(A)$ for all maps A .*

Note that, with this definition, *all* functions $A(\cdot)$ are quasi-typeable: They will all have the quasi-type **defined** \Rightarrow **defined**.

For various reasons, we are unable to define a functional (quasi-)type constructor, i.e., a type constructor $\mathcal{F}(\cdot, \cdot)$ such that F is $\mathcal{F}(\alpha, \beta)$ exactly when F has (quasi-)type $\alpha \Rightarrow \beta$. We shall come back to this point in Section 3.2.

2.6.1 Some Typed Functions

We already have seen a couple of typed functions, $\approx_\alpha(\cdot)$ and \cdot is α :

Lemma 2.6.3 *For all simple types α , the function $\approx_\alpha(\cdot)$ has type $\alpha \Rightarrow \alpha$.*

Proof. All needed properties follow directly from the definition of a simple type. \square

Lemma 2.6.4 *For all simple types α , the function \cdot is α has type $\alpha \Rightarrow \text{true}$.*

Before we prove this lemma, it will be convenient to have

Lemma 2.6.5 (Distribution over ifnil) *If $\mathcal{P}(\perp) = \perp$ then*

$$\mathcal{P}(\text{ifnil } A \text{ then } B \text{ else } C) = \text{ifnil } A \text{ then } \mathcal{P}(B) \text{ else } \mathcal{P}(C)$$

Proof. By case analysis (if $\mathcal{P}(\cdot)$ was not strict, then the left hand side would be defined while the right hand side was undefined for $A = \perp$). \square

Proof (of Lemma 2.6.4). Assume α is a simple type. Then \cdot is α is strict by assumption, and if $\approx_\alpha(A) = \approx_\alpha(A')$ then

$$\begin{aligned} A \text{ is } \alpha &= \text{ifnil } \approx_\alpha(A) \text{ then True else True} \\ &= \text{ifnil } \approx_\alpha(A') \text{ then True else True} \\ &= A' \text{ is } \alpha \end{aligned}$$

Finally, using the previous lemma,

$$\begin{aligned}
\approx_{\text{true}}(A \text{ is } \alpha) &= \approx_{\text{true}}(\mathbf{ifnil} \approx_{\alpha}(A) \text{ then True else True}) \\
&= \mathbf{ifnil} \approx_{\alpha}(A) \text{ then } \approx_{\text{true}}(\text{True}) \text{ else } \approx_{\text{true}}(\text{True}) \\
&= \mathbf{ifnil} \approx_{\alpha}(A) \text{ then True else True} \\
&= A \text{ is } \alpha
\end{aligned}$$

as wanted. □

2.7 Basic Pairs

It is frequently useful to be able to bundle several objects together. Probably the simplest implementation of “bundling” is the pair: For any two objects A and B , we want a pair object, conventionally denoted $\langle A, B \rangle$, and two decomposition functions, frequently denoted π_1 and π_2 (in mathematics) or **fst** and **snd** (in computer science). To emphasize that we think of map theory as a (powerful) programming language, we shall use the latter denotation.

Having just discussed typed functions, one might consider modeling typed pairs straight away, i.e., defining a pairing constructor $\mathcal{P}(\cdot, \cdot)$ of type $\alpha \Rightarrow \beta \Rightarrow \alpha * \beta$ for a suitable definition of $\alpha * \beta$. However, even the most relaxed pair type, defined *** defined** would preclude the pairing of bottom with any other map. As we would like a pairing function were *any* two maps could be paired, especially also bottom, we will define two pairing types: A basic, untyped one and a typed one. In this section we will deal only with the former but in Chapter 6 we will define typed pairs.

Moving on, we need to come up with a representation of these basic pairs in map theory. We choose to strongly represent the pair $\langle A, B \rangle$ by the map

$$\lambda x. \mathbf{ifnil} x \text{ then } A \text{ else } B$$

As for weak representatives, we will let any defined map A represent the pair $\langle A'N, A'(\lambda x.N) \rangle$. When defining the normalization function for pairs, we need to be careful. Just defining $\approx_{\text{pair}}(A)$ as $\langle A'N, A'(\lambda x.N) \rangle$ leads to problems: With this definition, we would get

$$\begin{aligned}
\approx_{\text{pair}}(\perp) &= \langle \perp'N, \perp'(\lambda x.N) \rangle \\
&= \langle \perp, \perp \rangle \\
&= \lambda x. \mathbf{ifnil} x \text{ then } \perp \text{ else } \perp \\
&= \lambda x. \perp
\end{aligned}$$

(**ifnil** A **then** \perp **else** $\perp = \perp$ for all A follows, e.g., by case analysis.) Since $\approx_{\text{pair}}(\cdot)$ must be strict, we need to a slightly different definition. Since this problem will occur again and again, we will introduce the guard function:

Definition 2.7.1 (Guard) $A \mapsto B \stackrel{\text{def}}{=} \text{ifnil } A \text{ then } B \text{ else } \perp$

Semantically, then $A \mapsto B$ denotes B , unless A is not true, in which case it is undefined. As direct consequences of the definition, we have $A \mapsto \text{True} = \approx_{\text{true}}(A)$ and $\text{True} \mapsto B = B$. We also have the distributive law:

Lemma 2.7.2 (Distribution over guards) *If $\mathcal{P}(\perp) = \perp$ then*

$$\mathcal{P}(A \mapsto B) = A \mapsto \mathcal{P}(B)$$

Proof. Straightforward, using distribution over **ifnil**. □

We can also prove that $\cdot \mapsto B$ is of the type **true** \Rightarrow **defined**. This isn't saying much, since any strict function trivially has the return type of **defined** and truth has a unique representation. However, it is still a compact notation for the fact that the guard function expects a truth value as its first argument. In general, all of the conditionals (the guard function is just the "conditional" for the truth type) and case-constructs we will define later will have similarly simplistic types. In any case we've found that even these types are useful to think about, so we will mention them as they occur.

Using a guard, we can now define the relevant maps for pairs:

Definition 2.7.3 (pair)

$$\begin{aligned} \langle A, B \rangle &\stackrel{\text{def}}{=} \lambda x. \text{ifnil } x \text{ then } A \text{ else } B \\ A \text{ is pair} &\stackrel{\text{def}}{=} A \text{ is defined} \\ \approx_{\text{pair}}(A) &\stackrel{\text{def}}{=} A \text{ is pair} \mapsto \langle A'N, A'(\lambda x.N) \rangle \end{aligned}$$

These definitions allow us to prove the following useful lemmas.

Lemma 2.7.4 *For all maps A and B , $\langle A, B \rangle$ is pair holds.*

Proof. By definition. □

Lemma 2.7.5 *For all maps A and B we have $\approx_{\text{pair}}(\langle A, B \rangle) = \langle A, B \rangle$.*

Proof. We have

$$\begin{aligned} \approx_{\text{pair}}(\langle A, B \rangle) &= \langle A, B \rangle \text{ is pair} \mapsto \langle \langle A, B \rangle'N, \langle A, B \rangle'(\lambda x.N) \rangle \\ &= \langle \langle A, B \rangle'N, \langle A, B \rangle'(\lambda x.N) \rangle \\ &= \langle \text{ifnil } N \text{ then } A \text{ else } B, \text{ifnil } (\lambda x.N) \text{ then } A \text{ else } B \rangle \\ &= \langle A, B \rangle \end{aligned}$$

verifying the statement. □

We can now prove

Lemma 2.7.6 *pair is a simple type.*

Proof. The strictness of $\approx_{\text{pair}}(\cdot)$ ultimately follows from the strictness of \cdot **is defined**. That $\approx_{\text{pair}}(\cdot)$ is idempotent is easy to verify:

$$\begin{aligned} \approx_{\text{pair}}(\approx_{\text{pair}}(A)) &= \approx_{\text{pair}}(A \text{ is pair} \mapsto \langle A'N, A'(\lambda x.N) \rangle) \\ &= A \text{ is pair} \mapsto \approx_{\text{pair}}(\langle A'N, A'(\lambda x.N) \rangle) \\ &= A \text{ is pair} \mapsto \langle A'N, A'(\lambda x.N) \rangle \\ &= \approx_{\text{pair}}(A) \end{aligned}$$

Finally, we also have

$$\begin{aligned} \approx_{\text{pair}}(A) \text{ is defined} &= (A \text{ is pair} \mapsto \langle A'N, A'(\lambda x.N) \rangle) \text{ is defined} \\ &= A \text{ is pair} \mapsto (\langle A'N, A'(\lambda x.N) \rangle \text{ is defined}) \\ &= A \text{ is pair} \mapsto \text{True} \\ &= \approx_{\text{true}}(A \text{ is pair}) \\ &= A \text{ is pair} \end{aligned}$$

as required. □

Before proceeding we shall prove a small lemma:

Lemma 2.7.7 *If $\langle A, B \rangle = \langle A', B' \rangle$ then $A = A'$ and $B = B'$.*

Proof. Assume $\langle A, B \rangle = \langle A', B' \rangle$. Then

$$A = \langle A, B \rangle'N = \langle A', B' \rangle'N = A'$$

Analogously, one can prove $B = B'$. □

We have proved that **pair** is a simple type. But it is exceedingly uninteresting as long as we are unable to take pairs apart. To this end, we have a choice: We can either define **fst** and **snd** separately, or we can define a single case-construct in the style of, say, ML. The two approaches are equivalent—having defined **fst** and **snd** we can define a case-construct and vice versa—so the choice is primarily an aesthetic one. As it is, we prefer the case-construct to be primitive and **fst** and **snd** to be derived.

What we want is a function **case** \cdot **of** $\langle x, y \rangle \Rightarrow \mathcal{C}(x, y)$ of the type **pair** \Rightarrow defined with the property

$$\text{case } \langle A, B \rangle \text{ of } \langle x, y \rangle \Rightarrow \mathcal{C}(x, y) = \mathcal{C}(A, B)$$

The following definition achieves this.

Definition 2.7.8 (Case-construct for basic pairs)

$$\text{case } A \text{ of } \langle x, y \rangle \Rightarrow \mathcal{C}(x, y) \stackrel{\text{def}}{=} A \text{ is pair} \mapsto \mathcal{C}(A'N, A'(\lambda x.N))$$

Lemma 2.7.9 *The function $\mathbf{case} \cdot \mathbf{of} \langle x, y \rangle \Rightarrow \mathcal{C}(x, y)$ has the type $\mathbf{pair} \Rightarrow$ defined.*

Proof. First of all, it is obviously strict.

Now, assume $\approx_{\mathbf{pair}}(A) = \approx_{\mathbf{pair}}(A')$. It follows that $A \mathbf{is pair}$ is equal to $A' \mathbf{is pair}$.

If they are both bottom, then it follows that A and A' are both undefined and hence equal. Hence, $\mathbf{case} A \mathbf{of} \langle x, y \rangle \Rightarrow \mathcal{C}(x, y)$ and $\mathbf{case} A' \mathbf{of} \langle x, y \rangle \Rightarrow \mathcal{C}(x, y)$ are equal.

If they are both true, then we have

$$\begin{aligned} \langle A' \mathbf{N}, A'(\lambda x. \mathbf{N}) \rangle &= A \mathbf{is pair} \mapsto \langle A' \mathbf{N}, A'(\lambda x. \mathbf{N}) \rangle \\ &= \approx_{\mathbf{pair}}(A) \\ &= \approx_{\mathbf{pair}}(A') \\ &= A' \mathbf{is pair} \mapsto \langle A' \mathbf{N}, A'(\lambda x. \mathbf{N}) \rangle \\ &= \langle A' \mathbf{N}, A'(\lambda x. \mathbf{N}) \rangle \end{aligned}$$

Hence $A' \mathbf{N} = A' \mathbf{N}$ and $A'(\lambda x. \mathbf{N}) = A'(\lambda x. \mathbf{N})$. All in all we have

$$\begin{aligned} \mathbf{case} A \mathbf{of} \langle x, y \rangle \Rightarrow \mathcal{C}(x, y) &= A \mathbf{is pair} \mapsto \mathcal{C}(A' \mathbf{N}, A'(\lambda x. \mathbf{N})) \\ &= A' \mathbf{is pair} \mapsto \mathcal{C}(A' \mathbf{N}, A'(\lambda x. \mathbf{N})) \\ &= \mathbf{case} A' \mathbf{of} \langle x, y \rangle \Rightarrow \mathcal{C}(x, y) \end{aligned}$$

as wanted. □

We will also want distribution over the construct:

Lemma 2.7.10 (Distribution over $\mathbf{case}_{\mathbf{pair}}$) *If $\mathcal{P}(\perp) = \perp$ then*

$$\mathcal{P}(\mathbf{case} A \mathbf{of} \langle x, y \rangle \Rightarrow \mathcal{C}(x, y)) = \mathbf{case} A \mathbf{of} \langle x, y \rangle \Rightarrow \mathcal{P}(\mathcal{C}(x, y))$$

Proof. By distribution over guards. □

2.7.1 Elimination

In proofs about basic pairs, we shall often find ourselves in the situation that we know that some map C is a basic pair. From that assumption we would like to deduce that C must be some, specific pair. The lemma we shall use in these cases is

Lemma 2.7.11 (Elimination of Basic Pairs) *If, for all maps A and B , $\approx_{\mathbf{pair}}(C) = \langle A, B \rangle$ implies $D = E$ then $C \mathbf{is pair}$ implies $D = E$.*

Proof. We have $\approx_{\mathbf{pair}}(C) = \langle C' \mathbf{N}, C'(\lambda x. \mathbf{N}) \rangle$ for all pairs C . From the assumption we have the conclusion $D = E$ as wanted. □

The lemma is so called, since we are “eliminating” the assumption $C \mathbf{is pair}$.

2.7.2 fst and snd

Here, at the end of the section, we can quickly define the **fst** and **snd** functions:

Definition 2.7.12

$$\begin{aligned}\mathbf{fst} A &\stackrel{def}{=} \mathbf{case} A \mathbf{of} \langle x, y \rangle \Rightarrow x \\ \mathbf{snd} A &\stackrel{def}{=} \mathbf{case} A \mathbf{of} \langle x, y \rangle \Rightarrow y\end{aligned}$$

It is straightforward to prove that both **fst** and **snd** are of the type $\mathbf{pair} \Rightarrow$ defined and that the usual equalities

$$\begin{aligned}\mathbf{fst} \langle A, B \rangle &= A \\ \mathbf{snd} \langle A, B \rangle &= B\end{aligned}$$

hold.

2.8 The Root Type

The next type we shall touch upon in this chapter is the root type. This is the type of the first argument of the **ifnil** construct, and is defined thus:

Definition 2.8.1 (root)

$$\begin{aligned}\approx_{\mathbf{root}}(A) &\stackrel{def}{=} \mathbf{ifnil} A \mathbf{then} N \mathbf{else} (\lambda x.N) \\ A \mathbf{is root} &\stackrel{def}{=} A \mathbf{is defined}\end{aligned}$$

As one can see, $\approx_{\mathbf{root}}(\cdot)$ is equal to **root**(\cdot).

Theorem 2.8.2 *root is a simple type.*

Proof. Straightforward. □

Lemma 2.8.3 *The function **ifnil** \cdot **then** B **else** C has the type **root** \Rightarrow defined.*

Proof. It is by definition strict. Further, if $\approx_{\mathbf{root}}(A) = \approx_{\mathbf{root}}(A')$ then

$$\begin{aligned}\mathbf{ifnil} A \mathbf{then} B \mathbf{else} C &= \mathbf{ifnil} \approx_{\mathbf{root}}(A) \mathbf{then} B \mathbf{else} C \\ &= \mathbf{ifnil} \approx_{\mathbf{root}}(A') \mathbf{then} B \mathbf{else} C \\ &= \mathbf{ifnil} A' \mathbf{then} B \mathbf{else} C\end{aligned}$$

where the first and last equality follows by case analysis on A . □

A stronger result holds:

Lemma 2.8.4 (Strong Congruence for ifnil) *If $\approx_{\text{root}}(A) = \mathbf{N}$ implies $B = B'$ and $\approx_{\text{root}}(A) = \lambda x.\mathbf{N}$ implies $C = C'$ then*

$$\mathbf{ifnil} A \mathbf{ then } B \mathbf{ else } C = \mathbf{ifnil} A \mathbf{ then } B' \mathbf{ else } C'$$

In other words, when simplifying B in **ifnil** A **then** B **else** C we can assume A to have root nil and, likewise, we can assume A to have root $\lambda x.\mathbf{N}$ when simplifying C . The proof uses the fact that we can represent the equality $\approx_{\text{root}}(A) = \mathbf{N}$ as the term $\mathcal{N}(A)$ where

$$\mathcal{N}(A) = \mathbf{ifnil} A \mathbf{ then } \text{True} \mathbf{ else } \text{False}$$

With this, we have $\approx_{\text{root}}(A) = \mathbf{N} \iff \mathcal{N}(A)$. The “ \implies ” direction follows directly, while the other direction follows by the deduction theorem: Assuming $\mathcal{N}(A)$ is true, we have

$$\approx_{\text{root}}(A) = \mathcal{N}(A) \mapsto \approx_{\text{root}}(A) = \mathcal{N}(A) \mapsto \mathbf{N} = \mathbf{N}$$

where the second equation holds by case distinction on A , using Theorem 1.3.3. A shorter proof is this: $\mathcal{N}(A) \triangleright \approx_{\text{root}}(A) = \mathbf{N}$ follows by case distinction on A . Hence $\mathcal{N}(A)$ implies $\approx_{\text{root}}(A) = \mathbf{N}$ by the Deduction Theorem.

By the analogous reasoning, from

$$\mathcal{L}(A) = \mathbf{ifnil} A \mathbf{ then } \text{False} \mathbf{ else } \text{True}$$

we can prove $\approx_{\text{root}}(A) = \lambda x.\mathbf{N} \iff \mathcal{L}(A)$.

Proof (of Lemma 2.8.4). Assume that $\approx_{\text{root}}(A) = \mathbf{N}$ implies $B = B'$. By the Deduction Theorem, we then have $\mathcal{N}(A) \triangleright B = B'$ (since $\mathcal{N}(A)$ is equivalent to $\approx_{\text{root}}(A) = \mathbf{N}$). By definition, this means $\mathcal{N}(A) \mapsto B = \mathcal{N}(A) \mapsto B'$. Analogously, from the assumption that $\approx_{\text{root}}(A) = \lambda x.\mathbf{N}$ implies $C = C'$, we get $\mathcal{L}(A) \mapsto C = \mathcal{L}(A) \mapsto C'$. Using these two equalities, we get

$$\begin{aligned} \mathbf{ifnil} A \mathbf{ then } B \mathbf{ else } C &= \mathbf{ifnil} A \mathbf{ then } \mathcal{N}(A) \mapsto B \mathbf{ else } \mathcal{L}(A) \mapsto C \\ &= \mathbf{ifnil} A \mathbf{ then } \mathcal{N}(A) \mapsto B' \mathbf{ else } \mathcal{L}(A) \mapsto C' \\ &= \mathbf{ifnil} A \mathbf{ then } B' \mathbf{ else } C' \end{aligned}$$

where the first and last equation follow by case distinction on A . □

A seemingly more direct version of Lemma 2.8.4 is the equation

$$\mathbf{ifnil} A \mathbf{ then } \mathcal{B}(A) \mathbf{ else } \mathcal{C}(A) = \mathbf{ifnil} A \mathbf{ then } \mathcal{B}(\mathbf{N}) \mathbf{ else } \mathcal{C}(\lambda x.A'x)$$

However, for this to be usable, A , first of all, must be syntactically present in one or both of the cases of the **ifnil** construct, something that might not be the case. Second, Lemma 2.8.4 allows us, e.g., to assume $\approx_{\text{root}}(A) = \mathbf{N}$ when rewriting B . This assumption might be in contradiction with other assumptions. If we only had the equation above, we might not be able to find such a contradiction.

2.9 Booleans

The final type that we will define for now is that of the booleans. It so happens that the booleans we define are exactly the same as the root type defined in the last section. Once again, this is merely a side-effect of the untypedness of map theory; The root type divides the defined maps of map theory into functions and the constant nil, while the booleans divide the same maps into true and false.

The definition of the boolean type is thus, repeating the definition of True for completeness:

Definition 2.9.1 (bool)

$$\begin{aligned} \text{True} &\stackrel{\text{def}}{=} \text{N} \\ \text{False} &\stackrel{\text{def}}{=} \lambda x. \text{N} \\ \text{A is bool} &\stackrel{\text{def}}{=} \text{A is defined} \\ \approx_{\text{bool}}(A) &\stackrel{\text{def}}{=} \mathbf{ifnil} A \mathbf{ then N else } (\lambda x. \text{N}) \end{aligned}$$

It is easy to verify that True and False are strong representatives of booleans, i.e., $\approx_{\text{bool}}(\text{True}) = \text{True}$ and $\approx_{\text{bool}}(\text{False}) = \text{False}$. We have:

Theorem 2.9.2 bool is a simple type.

Proof. Follows directly from the fact that root is a simple type. \square

2.9.1 Conditional

We already have a conditional on root values that we can use as a conditional on booleans:

Definition 2.9.3 (Conditional for booleans)

$$\mathbf{if} A \mathbf{ then } B \mathbf{ else } C \stackrel{\text{def}}{=} \mathbf{ifnil} A \mathbf{ then } B \mathbf{ else } C$$

And it has the expected type:

Lemma 2.9.4 The function $\mathbf{if} \cdot \mathbf{ then } B \mathbf{ else } C$ has the type $\text{bool} \Rightarrow \text{defined}$.

Proof. Follows directly from the type of the **ifnil** function. \square

From the definition of **if**, True, and False we can prove

$$\begin{aligned} \mathbf{if} \text{True} \mathbf{ then } B \mathbf{ else } C &= B \\ \mathbf{if} \text{False} \mathbf{ then } B \mathbf{ else } C &= C \end{aligned}$$

as expected.

From Lemma 2.8.4, we also immediately get

Lemma 2.9.5 (Strong Congruence for if) If $\approx_{\text{bool}}(A) = \text{True}$ implies $B = B'$ and $\approx_{\text{bool}}(A) = \text{False}$ implies $C = C'$ then

$$\mathbf{if} A \mathbf{ then } B \mathbf{ else } C = \mathbf{if} A \mathbf{ then } B' \mathbf{ else } C'$$

2.9.2 Case Analysis

We are not done defining the booleans yet. For if we want to prove, for a suitable definition of conjunction,

$$A \vee B = B \vee A \tag{2.3}$$

we might very well want to do it by case analysis. But at the moment, all we have in this regard is Theorem 1.3.3, that allows us to deduce the above equality if we can prove

$$\begin{aligned} \mathbf{N} \vee B &= B \vee \mathbf{N} \\ (\lambda x. A'x) \vee B &= B \vee (\lambda x. A'x) \\ \perp \vee B &= B \vee \perp \end{aligned} \tag{2.4}$$

which is not quite what we would want: the underlying machinery of map theory is showing through.

The logic of we want is this: We want to prove $\mathcal{A}(C) = \mathcal{B}(C)$, where C should be thought of as a boolean. Either C **is bool** is true or not. If it is, then C either represents true or false. So the case distinction we are making is really captured in the following lemma:

Lemma 2.9.6 *If $\approx_{\text{bool}}(C) = \perp$, $\approx_{\text{bool}}(C) = \text{True}$, and $\approx_{\text{bool}}(C) = \text{False}$ all imply $\mathcal{A}(C) = \mathcal{B}(C)$ then $\mathcal{A}(C) = \mathcal{B}(C)$ holds for all maps C .*

However sound the reasoning is, there is a problem: We cannot prove it in map theory.¹

To clarify this, we shall prove a couple of slightly weaker, related lemmas.

Lemma 2.9.7 (Elimination of Booleans) *If C **is bool** and $\approx_{\text{bool}}(C) = \text{True}$ and $\approx_{\text{bool}}(C) = \text{False}$ both imply $A = B$ then $A = B$ holds.*

Proof. From the assumptions, since C **is bool**, we see that it suffices to prove

$$C \text{ **is bool** } \mapsto A = C \text{ **is bool** } \mapsto B$$

But we have

$$\begin{aligned} C \text{ **is bool** } \mapsto A &= \text{if } C \text{ then } A \text{ else } A \\ &= \text{if } C \text{ then } B \text{ else } B \\ &= C \text{ **is bool** } \mapsto B \end{aligned}$$

where the second equation hold by the assumptions and the strong congruence of **if**, and two other by case distinction on C . \square

¹That is, we have been unable to prove it. Actually *proving* that it cannot be proved amounts to proving the consistency of map theory which we (of course) have not done.

In the proof of Lemma 2.9.7, one might say that we were able to get all information on C into a single equation, and then use case distinction to prove this equation. Getting this information into a single equation amounts to translating all assumptions on C into terms, i.e., $\approx_{\text{bool}}(C) = \text{True} \implies A = B$ and $\approx_{\text{bool}}(C) = \text{False} \implies A = B$ were encoded in the equality

$$\text{if } C \text{ then } A \text{ else } A = \text{if } C \text{ then } B \text{ else } B$$

using the strong congruence of **if**, and we could pull the term C **is bool** into the equation indirectly. The problem with Lemma 2.9.6 is, then, that the assumption $\approx_{\text{bool}}(C) = \perp$ can not be represented by a term, i.e., there is no function $\mathcal{U}(\cdot)$ such that $\mathcal{U}(C) \iff \approx_{\text{bool}}(C) = \perp$ (this is again, as mentioned in the introductory chapter, a consequence of the monotonicity of maps, in this case $\mathcal{U}(\cdot)$).

So what do we do? Well, if we already know that C is boolean, then Lemma 2.9.7 is exactly what we want: It says that if C is boolean, then it must be either **True** or **False**. If we can prove $A = B$ in both cases, we are done.

If, on the other hand, C is not known to be boolean, then we shall use the following flavor of Theorem 1.3.3:

Theorem 2.9.8 (Case Analysis on Booleans) *If*

$$\begin{aligned} \mathcal{A}(\perp) &= \mathcal{B}(\perp) \\ \mathcal{A}(\text{True}) &= \mathcal{B}(\text{True}) \\ \mathcal{A}(\text{False}) &= \mathcal{B}(\text{False}) \end{aligned}$$

then

$$\mathcal{A}(\approx_{\text{pair}}(C)) = \mathcal{B}(\approx_{\text{pair}}(C))$$

We are only testing the equality $\mathcal{A}(C) = \mathcal{B}(C)$ when C is a strong representative, and this might seem too weak: What about all the weak booleans? In practice, functions with boolean arguments will be built up from either the primitive or defined boolean functions, and so we will normally have $\mathcal{A}(\approx_{\text{bool}}(D)) = \mathcal{A}(D)$ for all maps D , and thus especially for C . Likewise for $\mathcal{B}(\cdot)$. In Equation 2.3, for example, \wedge will, once defined in the next chapter, be such a boolean function, and we can prove its commutativity by proving

$$\begin{aligned} \text{True} \vee B &= B \vee \text{True} \\ \text{False} \vee B &= B \vee \text{False} \\ \perp \vee B &= B \vee \perp \end{aligned}$$

which is a vast improvement on Equation 2.4.

2.10 Order

Although we shall not use ordering heavily until after we start defining recursive definitions, we will now briefly state the definitions of greatest lower bounds and ordering, together with their main properties.

We first define the greatest lower bound between two maps A and B :

Definition 2.10.1 (Greatest Lower Bound)

$$A \downarrow B \stackrel{def}{=} \mathbf{ifnil} A \left\{ \begin{array}{l} \mathbf{ifnil} B \left\{ \begin{array}{l} \mathbf{N} \\ \perp \end{array} \right. \\ \mathbf{ifnil} B \left\{ \begin{array}{l} \perp \\ \lambda z.(A'z) \downarrow (B'z) \end{array} \right. \end{array} \right.$$

From this, we can show that \downarrow is commutative, associative, and that, for all A , $A \downarrow A = A$. It is also clearly strict in both arguments. We now define \preceq as

Definition 2.10.2 (Order)

$$A \preceq B \stackrel{def}{=} A = A \downarrow B$$

From the properties of \downarrow we can prove that \preceq is order relation on maps, i.e., it is reflexive, transitive, and if $A \preceq B$ and $B \preceq A$ then $A = B$. For details on proving these results, the reader is referred to Grue [Gru01].

We are now ready to prove some minimality results for **ifnil**, **if**, and guards.

Lemma 2.10.3 (Minimality of ifnil) *If $\approx_{\text{root}}(A) = \mathbf{N}$ implies $B \preceq D$ and $\approx_{\text{root}}(A) = \lambda x.N$ implies $C \preceq D$ then*

$$\mathbf{ifnil} A \text{ then } B \text{ else } C \preceq D$$

Proof. Using $\mathcal{N}(\cdot)$ and $\mathcal{L}(\cdot)$ from the previous section, we have

$$\begin{aligned} \mathcal{N}(A) \mapsto B &= \mathcal{N}(A) \mapsto B \downarrow D \\ \mathcal{L}(A) \mapsto C &= \mathcal{L}(A) \mapsto C \downarrow D \end{aligned}$$

from the assumptions. Hence, we have

$$\begin{aligned} \mathbf{ifnil} A \text{ then } B \text{ else } C &= \mathbf{ifnil} A \text{ then } \mathcal{N}(A) \mapsto B \text{ else } \mathcal{N}(A) \mapsto C \\ &= \mathbf{ifnil} A \text{ then } \mathcal{N}(A) \mapsto B \downarrow D \text{ else } \mathcal{N}(A) \mapsto C \downarrow D \\ &= \mathbf{ifnil} A \text{ then } B \downarrow D \text{ else } C \downarrow D \\ &= (\mathbf{ifnil} A \text{ then } B \text{ else } C) \downarrow D \end{aligned}$$

using case distinction, strong congruence, case distinction, and finally distribution. \square

Lemma 2.10.4 (Minimality of if) *If $\approx_{\text{bool}}(A) = \text{True}$ implies $B \preceq D$ and $\approx_{\text{bool}}(A) = \text{False}$ implies $C \preceq D$ then*

$$\text{if } A \text{ then } B \text{ else } C \preceq D$$

Proof. By Lemma 2.10.3. □

Lemma 2.10.5 (Minimality of Guard) *If A implies $B \preceq C$ then*

$$A \mapsto B \preceq C$$

Proof. By the definition of guard, the uniqueness of nil, and Lemma 2.10.3. □

We can now prove the little lemma that will be of much use to us later:

Lemma 2.10.6 (Equivalence of Truth) *If $A \iff B$ then*

$$\approx_{\text{true}}(A) = \approx_{\text{true}}(B)$$

Proof. By Lemma 2.10.5 and $A \implies B$ we have

$$A \mapsto \text{True} \preceq \approx_{\text{true}}(B)$$

Using $A \mapsto \text{True} = \approx_{\text{true}}(A)$ we have proved $\approx_{\text{true}}(A) \preceq \approx_{\text{true}}(B)$. Analogously, we can prove $\approx_{\text{true}}(B) \preceq \approx_{\text{true}}(A)$, and by the anti-symmetry of \preceq we have $\approx_{\text{true}}(A) = \approx_{\text{true}}(B)$ as wanted. □

This means that for all typed functions $\mathcal{F}(\cdot)$ and $\mathcal{G}(\cdot)$ with return type true, $\mathcal{F}(A) \iff \mathcal{G}(A)$ exactly when $\mathcal{F}(A) = \mathcal{G}(A)$. Thus, for such functions, we shall interchangeably use either notation, writing bi-implication at some times, and equality at others.

2.11 Equality

As has been remarked earlier, there is no function $\mathcal{E}(\cdot, \cdot)$ such that

$$\mathcal{E}(A, B) \iff A = B$$

for all maps A and B . But in the case where A and B are strong representatives of some type α that has an equality predicate $=_{\alpha}$, we *almost* can. Since strong representatives are unique, it follows that if $A =_{\alpha} B$, that is, if A and B represent the same element of α and A and B are strong representatives, then A must be equal to B . On the other hand, if A and B are equal, then $A =_{\alpha} B$ must hold by the reflexivity of $=_{\alpha}$, if A and B are α . In other words, for all A and B of type α ,

$$A =_{\alpha} B \iff \approx_{\alpha}(A) = \approx_{\alpha}(B)$$

Simple types α , where a function $=_{\alpha}$ exists with the above property, are called simple equality types. They are defined thus:

Definition 2.11.1 (Equality Type) *The simple type α is called an equality type if there exists a function $=_\alpha$ of type $\alpha \Rightarrow \alpha \Rightarrow \text{bool}$ such that $=_\alpha$ is*

- *total on α , i.e., for all A and B of type α , $A =_\alpha B$ is boolean.*
- *reflexive on α , i.e., if A is α then $A =_\alpha A$.*
- *reflective, i.e., $A =_\alpha B$ implies $\approx_\alpha(A) = \approx_\alpha(B)$.*

From these three requirements, we can show that $=_\alpha$ is indeed an equivalence relation:

Lemma 2.11.2 *If α is an equality type, then $=_\alpha$ is an equivalence relation on α .*

Proof. (Reflexivity) Follows from the definition of equality type. (Antisymmetry) Assume $A =_\alpha B$ where A and B are α (since $=_\alpha$ is strict, this actually follows from $A =_\alpha B$). By reflection, we have $\approx_\alpha(A) = \approx_\alpha(B)$. Hence $B =_\alpha A$ follows by reflexivity. (Transitivity) Assume $A =_\alpha B$ and $B =_\alpha C$ (and hence that A , B , and C are α). By reflection we have $\approx_\alpha(A) = \approx_\alpha(B) = \approx_\alpha(C)$ so, by reflexivity we have $A =_\alpha C$, as wanted. \square

With

Definition 2.11.3 (Equality on Truths)

$$A =_{\text{true}} B \stackrel{\text{def}}{=} A \mapsto B \mapsto \text{True}$$

we can prove

Lemma 2.11.4 *true is an equality type.*

Proof. (Correct Type) The function $=_{\text{true}}$ clearly has type $[\text{true}, \text{true}] \Rightarrow \text{bool}$. (Totality) Assume A and B are both true, then $A =_{\text{true}} B$ is also true, and hence a boolean. (Reflexivity) Also obvious. (Reflection) Assume $A =_{\text{true}} B$. By strictness, we have A and B true, hence $\approx_{\text{true}}(A) = \approx_{\text{true}}(B)$. \square

A bit more interesting, and wholly more useful, is equality on booleans. We define

Definition 2.11.5 (Equality on Booleans)

$$A =_{\text{bool}} B \stackrel{\text{def}}{=} \text{if } A \left\{ \begin{array}{l} \text{if } B \left\{ \begin{array}{l} \text{True} \\ \text{False} \end{array} \right. \\ \text{if } B \left\{ \begin{array}{l} \text{False} \\ \text{True} \end{array} \right. \end{array} \right.$$

Lemma 2.11.6 *bool is an equality type.*

Proof. (Correct Type) The function $=_{\text{bool}}$ is seen to be of the desired type, $[\text{bool}, \text{bool}] \Rightarrow \text{bool}$. (Totality) Assume A and B are both boolean. Using Lemma 2.9.7 repeatedly, it is straightforward to prove that $A =_{\text{bool}} B$ is boolean. (Reflexivity) Again, assuming A is boolean, Lemma 2.9.7 allows us to prove the desired: $A =_{\text{bool}} A$. (Reflection) Assume $A =_{\text{bool}} B$ and hence A and B boolean. There are four combinations. If A and B are both true or both false, the desired conclusion, $\approx_{\text{bool}}(A) = \approx_{\text{bool}}(B)$ is clearly true. If A and B are different, we obtain a contradiction by the assumption $A =_{\text{bool}} B$. \square

From this we obviously have the following corollary. As we shall have no use for equality on the root type, this is the last we shall hear about it:

Corollary 2.11.7 *root is an equality type.*

As examples of type that are not equality types, we have `defined` and `pair`. Equality on `pair` would imply equality on all maps, which is not possible, and equality on `defined` is impossible for the same reason as equality on all maps is impossible: monotonicity. The defined map $\lambda.x.\perp$ is equal to itself, but is also the least function under \preceq . Hence, the truth of $\lambda x.\perp =_{\text{defined}} \lambda x.\perp$ would imply the truth of $A =_{\text{defined}} B$ for all functions A and B .

2.12 General Proof Techniques

As one can see from the presentation up until now, it would seem that undefinedness keeps getting into the way. Without bottom, we would not have general recursion, so we definitely want it in the logic. But we would like to work around it as smoothly as possible, when needed.

One of the simplest tricks we have uses the observation that proving the equality $A = B$ can be dealt with in two steps: First proving that for A and B each is as defined (or undefined) as the other, and then proving $A = B$ assuming that they are both defined (since if one is undefined, then the other is too, and hence they are equal). This trick was, for example, used in the proof of Lemma 2.7.9. We shall prove a slightly more general result:

Lemma 2.12.1 *If A is α and B is α both imply C , then from $C \implies A = B$ we can deduce $A = B$.*

Proof. Using case distinction, strong congruence of guard, and the property

$A \mapsto B \mapsto C = B \mapsto A \mapsto C$ of guard we have

$$\begin{aligned}
A &= A \text{ is defined} \mapsto A \\
&= A \text{ is defined} \mapsto (C \mapsto A) \\
&= C \mapsto (A \text{ is defined} \mapsto A) \\
&= C \mapsto (B \text{ is defined} \mapsto B) \\
&= B \text{ is defined} \mapsto (C \mapsto B) \\
&= B \text{ is defined} \mapsto B \\
&= B
\end{aligned}$$

proving the desired. \square

By using this lemma repeatedly, we can assume any property that follows from both sides of the equality being defined.

As an example of its use, we can give another sketch of a proof of Equation 2.3: Since both $A \text{ is bool}$ and $B \text{ is bool}$ follows (will follow) from both $A \vee B \text{ is defined}$ and $B \vee A \text{ is defined}$ (since disjunction is (will be) a typed, and hence strict, function), we can assume A and B to be booleans. Now use Lemma 2.9.7 to prove $A \vee B = B \vee A$.

In the presence of equality types, we can now prove the following central lemma:

Lemma 2.12.2 (General Equivalence) *If α is an equality type and*

$$A \text{ is } \alpha \iff B \text{ is } \alpha$$

and finally we can show

$$A =_{\alpha} B$$

assuming A and B of type α , then we can conclude

$$\approx_{\alpha}(A) = \approx_{\alpha}(B)$$

Proof. We will use Lemma 2.12.1 twice, with C equal to $A \text{ is } \alpha$ and $B \text{ is } \alpha$: We have

$$\approx_{\alpha}(A) \text{ is defined} = A \text{ is } \alpha$$

and, likewise, $\approx_{\alpha}(B) \text{ is defined} = B \text{ is } \alpha$. Obviously, then, $A \text{ is defined}$ implies $A \text{ is } \alpha$, and further, by the equivalence of $A \text{ is } \alpha$ and $B \text{ is } \alpha$, we also have $B \text{ is defined}$ implies $A \text{ is } \alpha$. We may assume $B \text{ is } \alpha$ by a similar reasoning.

We therefore now assume A and B to be α . therefore also $B) \text{ is } \alpha$. From the assumptions we then have $A =_{\alpha} B$ and, using reflection, this implies $\approx_{\alpha}(A) = \approx_{\alpha}(B)$. \square

The explicit normalization will, as always, often disappear in the presence of concrete terms A and B , as they will normally already be strong representatives of α . By virtue of their return type as a typed function, for example.

The great value of this result is that it allows us to “translate” the equality $\approx_\alpha(A) = \approx_\alpha(B)$ to the term $A =_\alpha B$. This will be extremely convenient, especially with respect to quantifiers. More on this in Section 3.2 in the next chapter.

As a final note, in order to use the General Equivalence Lemma, we will, needless to say, have to be able to prove the assumption

$$A \text{ is } \alpha \iff B \text{ is } \alpha$$

for a multitude of equality types α and terms A and B . To aid in this, we shall use results of the following kind, allowing us to simplify what is essentially type constraints:

Lemma 2.12.3 (Guard Type Elimination)

$$(A \mapsto B) \text{ is } \alpha \implies A \text{ is true and } B \text{ is } \alpha$$

Proof. From the assumption, by the strictness of guard, we have A is true. Simplifying the assumption, we also get B is α . \square

Lemma 2.12.4 (Guard Type Introduction)

$$A \text{ is true and } B \text{ is } \alpha \implies (A \mapsto B) \text{ is } \alpha$$

Proof. Straightforward. \square

For conditionals for types of more than one constructor, the formulation of the type introduction/elimination rules is not as clean. The simplest type elimination rule we can state for **if** is

Lemma 2.12.5 (if Type Elimination)

$$(\text{if } A \text{ then } B \text{ else } C) \text{ is } \alpha \implies A \text{ is bool}$$

Proof. By strictness of **if**. \square

After using this lemma, one can then use case distinction (on A), and possibly later simplify **if** A **then** B **else** C **is** α to either B **is** α or C **is** α , depending on the value of A . For type introduction, we have no choice but to incorporate case distinction on A :

Lemma 2.12.6 (if Type Introduction) *If A is boolean and*

$$\approx_{\text{bool}}(A) = \text{True} \implies B \text{ is } \alpha$$

and

$$\approx_{\text{bool}}(A) = \text{False} \implies C \text{ is } \alpha$$

then

$$(\text{if } A \text{ then } B \text{ else } C) \text{ is } \alpha$$

Proof. By elimination of A as a boolean. □

The type introduction and elimination rules for **ifnil** are the expected variants of Lemmas 2.12.5 and 2.12.6.

For total functions, like equality on types, we can capture the type introduction and elimination rules in one single equivalence:

Lemma 2.12.7 *For all equality types α we have*

$$A =_{\alpha} B \text{ is bool} \iff A \text{ is } \alpha \text{ and } B \text{ is } \alpha$$

Proof. “ \implies ” by strictness of equality. “ \impliedby ” by the definition of equality type. □

Chapter 3

Propositional and Predicate Logic

Before we can model set theory in map theory, we need to model first order logic, or predicate logic. This is done in the current chapter. First, we model propositional logic, and then we turn our attention to the quantifiers of predicate logic.

3.1 Propositional Logic

We can, quite straightforwardly, represent the usual connectives of propositional logic using the booleans defined in the last chapter. Negation is completely as expected:

$$\neg A = \text{if } A \text{ then False else True}$$

The definitions for the other connectives, however, need some more care. Arbitrarily singling out disjunction, we will take the equality

$$A \vee B = \text{if } A \text{ then True else } B$$

as the basis for our later definition. Unfortunately, there are several problems with using this equality as the definition of disjunction.

We would like disjunction to have the type $\text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}$. But with the equality above, we can neither prove

$$A \vee B = A \vee B'$$

from $\approx_{\text{bool}}(B) = \approx_{\text{bool}}(B')$, nor can we prove

$$\approx_{\text{bool}}(A \vee B) = A \vee B$$

for all maps A and B . In both cases, problems occur when A is true. Then we would need to prove $B = B'$ from $\approx_{\text{bool}}(B) = \approx_{\text{bool}}(B')$ and

prove $\approx_{\text{bool}}(B) = B$ for all B , which does not hold. The first remedy is to normalize the result of disjunction:

$$A \vee B = \text{if } A \text{ then True else } \approx_{\text{bool}}(B)$$

(True is of course already normalized.) While this allows us to prove

$$\approx_{\text{bool}}(A \vee B) = A \vee B$$

for all A and B , it does not solve the whole problem, though, since this function is not strict in its second argument. As for the normalization function for pairs, the solution will be to use the guard function.

These considerations lead us to the following definition:

Definition 3.1.1 (Propositional Connectives)

$$\begin{aligned} \neg A &\stackrel{\text{def}}{=} \text{if } A \text{ then False else True} \\ A \vee B &\stackrel{\text{def}}{=} B \text{ is bool } \mapsto \text{if } A \text{ then True else } \approx_{\text{bool}}(B) \\ A \wedge B &\stackrel{\text{def}}{=} B \text{ is bool } \mapsto \text{if } A \text{ then } \approx_{\text{bool}}(B) \text{ else False} \\ A \rightarrow B &\stackrel{\text{def}}{=} B \text{ is bool } \mapsto \text{if } A \text{ then } \approx_{\text{bool}}(B) \text{ else True} \\ A \leftrightarrow B &\stackrel{\text{def}}{=} \text{if } A \text{ then } \approx_{\text{bool}}(B) \text{ else } \neg B \end{aligned}$$

We can now prove:

Lemma 3.1.2 *Negation has type $\text{bool} \Rightarrow \text{bool}$. Disjunction, conjunction, implication, and bi-implication all have type $\text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}$.*

Proof. Straightforward. □

Note that bi-implication is, of course, equal to the equality function $=_{\text{bool}}$ from last chapter. We shall therefore use the notation $=_{\text{bool}}$ for bi-implication in the following.

With the definitions above we can show, for example, the following standard equality for all maps A and B :

Lemma 3.1.3 (Commutativity of Disjunction)

$$A \vee B = B \vee A$$

The proof can proceed in a number of different ways. We can use Lemma 2.9.8:

Proof (Variation One). We have $\perp \vee B = B \vee \perp$ by strictness of disjunction. Also

$$\begin{aligned} \text{True} \vee B &= B \text{ is bool } \mapsto \text{if True then True else } \approx_{\text{bool}}(B) \\ &= B \text{ is bool } \mapsto \text{True} \\ &= B \text{ is bool} \\ &= \text{if } B \text{ then True else True} \\ &= \text{True is bool } \mapsto \text{if } B \text{ then True else } \approx_{\text{bool}}(\text{True}) \\ &= B \vee \text{True} \end{aligned}$$

Likewise, we can prove $\text{False} \vee B = B \vee \text{False}$. So Lemma 2.9.8 gives us $(\approx_{\text{bool}}(A)) \vee B = B \vee (\approx_{\text{bool}}(A))$, and thus $A \vee B = B \vee A$, as wanted. \square

Another variation is the one sketched in Section 2.12:

Proof (Variation Two). Since disjunction is typed, we have that $A \vee B$ defined implies A and B boolean. Using Lemma 2.9.7, the proof proceeds as follows: From the assumption $\approx_{\text{bool}}(A) = \text{True}$ we have

$$\begin{aligned}
A \vee B &= B \text{ is bool} \mapsto \mathbf{\text{if } A \text{ then True else } \approx_{\text{bool}}(B)} \\
&= B \text{ is bool} \mapsto \text{True} \\
&= B \text{ is bool} \\
&= \mathbf{\text{if } B \text{ then True else True}} \\
&= A \text{ is bool} \mapsto \mathbf{\text{if } B \text{ then True else } \approx_{\text{bool}}(A)} \\
&= B \vee A
\end{aligned}$$

Likewise, if $\approx_{\text{bool}}(A) = \text{False}$, we have

$$\begin{aligned}
A \vee B &= B \text{ is bool} \mapsto \mathbf{\text{if } A \text{ then True else } \approx_{\text{bool}}(B)} \\
&= B \text{ is bool} \mapsto \approx_{\text{bool}}(B) \\
&= \approx_{\text{bool}}(B) \\
&= \mathbf{\text{if } B \text{ then True else False}} \\
&= A \text{ is bool} \mapsto \mathbf{\text{if } B \text{ then True else } \approx_{\text{bool}}(A)} \\
&= B \vee A
\end{aligned}$$

Hence, by Lemma 2.9.7 we have $A \vee B = B \vee A$. \square

Finally, the third variation we shall state here uses the General Equivalence Lemma of last chapter. To use it properly, we will also need the following type elimination/introduction rules:

Lemma 3.1.4

$$\begin{aligned}
\neg A \text{ is bool} &\iff A \text{ is bool} \\
A \vee B \text{ is bool} &\iff A \text{ is bool and } B \text{ is bool} \\
A \wedge B \text{ is bool} &\iff A \text{ is bool and } B \text{ is bool} \\
A \rightarrow B \text{ is bool} &\iff A \text{ is bool and } B \text{ is bool}
\end{aligned}$$

Proof. “ \implies ” follows from types of the respective functions. “ \impliedby ” expresses totality, and is easily proved in each case, using Lemma 2.9.7. \square

The third variation on the proof of commutativity of disjunction now reads:

Proof (Variation Three). By Lemma 3.1.4 we have

$$A \vee B \text{ is bool} \iff B \vee A \text{ is bool}$$

Also, assuming $A \vee B$ and $B \vee A$ are boolean, we can show $A \vee B =_{\text{bool}} B \vee A$: From $A \vee B$ boolean we have A and B are boolean, and we can use Lemma 2.9.7 to prove the wanted. Hence, by general equivalence, we have $\approx_{\text{bool}}(A \vee B) = \approx_{\text{bool}}(B \vee A)$, and thus, again, $A \vee B = B \vee A$. \square

The proof is, of course, quite like the second variation.

Having given three variations on a proof of commutativity of disjunction, the question of which proof is “best” presents itself. Working with booleans, a non-recursive type, the choice really doesn’t matter; Any of the variations will do, though the first variation is probably the most simple of the three, since it does not introduce any new assumptions. In the next chapters, however, the third variation will win hands down. The reason is primarily that it avoids dealing with bottom explicitly.

For equalities between maps of no equality type, when general equivalence can not be used, the situation is either extremely simple (in the case of non-recursive types) or extremely bad (in the case of recursive types). More on this in Chapter 6.

However we wish to prove them, the following standard equalities hold for all maps A and B :

$$\begin{aligned} (A \vee B) \vee C &= A \vee (B \vee C) \\ A \wedge B &= B \wedge A \\ (A \wedge B) \wedge C &= A \wedge (B \wedge C) \\ \neg(A \wedge B) &= \neg A \vee \neg B \\ \neg(A \vee B) &= \neg A \wedge \neg B \end{aligned}$$

Ignoring explicit normalization, many more of the standard equalities, e.g., $A \wedge \text{True} = \approx_{\text{bool}}(A)$ or $\neg(\neg A) = \approx_{\text{bool}}(A)$, also hold for all maps. Others do not. For example, $A \vee \neg A$ is only true for boolean A , i.e., if $A \text{ is bool}$. The closest we can get to the usual form is $A \text{ is bool} \implies A \vee \neg A$ or $A \vee \neg A = A \text{ is bool} \mapsto \text{True}$.¹

In the following we shall use tautologies of propositional logic without proof.

3.1.1 Non-Strict Connectives

Once in a while it is inconvenient that the boolean connectives are strict. To this end, we shall define non-strict versions of disjunction, conjunction,

¹Since $A \text{ is } \alpha \mapsto \text{True} = A \text{ is } \alpha$, we could just have written $A \vee \neg A = A \text{ is bool}$. However, conceptually, we would then have had a boolean expression on the left hand side of the equality, and a truth expression on the right, which would be bending the types a bit. Likewise, we shall write $A \text{ is } \alpha \mapsto \text{False}$ rather than the shorter $\neg(A \text{ is } \alpha)$.

and implication (there are, of course, only strict versions of negation and bi-implication).

Definition 3.1.5 (Non-Strict Propositional Connectives)

$$\begin{aligned} A \tilde{\vee} B &\stackrel{def}{=} \text{if } A \text{ then True else } \approx_{\text{bool}}(B) \\ A \tilde{\wedge} B &\stackrel{def}{=} \text{if } A \text{ then } \approx_{\text{bool}}(B) \text{ else False} \\ A \tilde{\rightarrow} B &\stackrel{def}{=} \text{if } A \text{ then } \approx_{\text{bool}}(B) \text{ else True} \end{aligned}$$

Part of the price for non-strictness is that we cannot prove equalities like $A \tilde{\vee} B = B \tilde{\vee} A$ for arbitrary A and B , since, for example, $\text{True} \tilde{\vee} \perp = \text{True} \neq \perp = \perp \tilde{\vee} \text{True}$.

It should be clear that non-strict disjunction, conjunction, and implication all have quasi-type $\text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}$.

3.2 Predicate Logic

Map theory has no truly universal quantifier, that is, no construct $Ax.\mathcal{B}(x)$ that is true exactly when $\mathcal{B}(C)$ is true for all maps C , and false otherwise. (Nor one that is true if $\mathcal{B}(C)$ is true for all maps C and either false or undefined otherwise, for that matter.) As we have also mentioned several times now, we also have no construct for equality, e.g., no function that represents equality. What we *do* have, though, is a very large class of maps, the classical, on which we have equality as well as universal and existential quantifiers.

3.2.1 Classical Maps

As mentioned in the introductory chapter, the classical maps have been introduced into map theory with a purpose: to allow a model of set theory to be constructed. For this reason, their definition is indeed very technical, and we shall only need some of their simpler properties. We will therefore, without proof, state only the lemmas we will need for later developments. As always, the details can be found in Grue [Gru01].

Lemma 3.2.1 *classical is an equality type.*

Apart from the constructs and properties that this entails, we have the choice operator

$$\varepsilon x.\mathcal{A}(x)$$

also mentioned in the introductory chapter. From this we define the existential and universal quantifiers:

Definition 3.2.2 (Existential and Universal Quantifiers)

$$\begin{aligned}\exists x.\mathcal{A}(x) &\stackrel{def}{=} \approx_{\text{bool}}(\mathcal{A}(\varepsilon x.\mathcal{A}(x))) \\ \forall x.\mathcal{A}(x) &\stackrel{def}{=} \neg(\exists x.\neg(\mathcal{A}(x)))\end{aligned}$$

We have the following results about classical maps, the choice operator and the quantifiers:

Lemma 3.2.3 (Universal Introduction) *If $\mathcal{A}(B)$ holds for all classical B then $\forall x.\mathcal{A}(x)$.*

Lemma 3.2.4 (Universal Elimination) *If $\forall x.\mathcal{A}(x)$ holds and B is classical, then $\mathcal{A}(B)$.*

Lemma 3.2.5 (Existential Introduction) *If $\mathcal{A}(B)$ is boolean for all classical B and there exists classical C such that $\mathcal{A}(C)$ then $\exists x.\mathcal{A}(x)$*

Lemma 3.2.6 (Existential Elimination) *If $\exists x.\mathcal{A}(x)$ then $\mathcal{A}(\varepsilon x.\mathcal{A}(x))$*

The following lemma describes the type rules for the classical operators

Lemma 3.2.7 (Classical Type Rules)

$$\begin{aligned}\varepsilon x.\mathcal{A}(x) \text{ is classical} &\iff \forall x.(\mathcal{A}(x) \text{ is bool}) \\ \exists x.\mathcal{A}(x) \text{ is bool} &\iff \forall x.(\mathcal{A}(x) \text{ is bool}) \\ \forall x.\mathcal{A}(x) \text{ is bool} &\iff \forall x.(\mathcal{A}(x) \text{ is bool})\end{aligned}$$

In the following developments, we shall only need the following lemmas for proving certain maps classical.

Lemma 3.2.8 \mathbb{N} is classical.

Lemma 3.2.9 For all classical B , $\mathbb{K}'B$ is classical.

Lemma 3.2.10 (Induction on Classical Maps) *If one can prove*

- $\mathcal{A}(\mathbb{N})$ and
- for all classical B , $\approx_{\text{root}}(B) = \lambda x.\mathbb{N}$ and $\forall x.\mathcal{A}(B'x)$ implies $\mathcal{A}(B)$.

then $\forall x.\mathcal{A}(x)$.

Drawing on the connection with sets given in the introductory chapter, these lemmas can be read as: \emptyset is a set. For all sets B , $\{B\}$ is a set. If we can prove that $\mathcal{A}(\emptyset)$ and that, for all non-empty sets B , if $\mathcal{A}(x)$ holds for all $x \in B$ then $\mathcal{A}(B)$ then $\mathcal{A}(\cdot)$ holds for all sets.

Finally, we have the following congruence rules for the classical operators:

Lemma 3.2.11 (Congruence of Choice) *If $\mathcal{A}(C) = \mathcal{B}(C)$ for all classical C then*

$$\varepsilon x.\mathcal{A}(x) = \varepsilon x.\mathcal{B}(x)$$

From this lemma, we can see that we also have congruence rules for the quantifiers:

Lemma 3.2.12 (Congruence of the Quantifiers) *If $\mathcal{A}(C) = \mathcal{B}(C)$ for all classical C then*

$$\exists x.\mathcal{A}(x) = \exists x.\mathcal{B}(x)$$

and

$$\forall x.\mathcal{A}(x) = \forall x.\mathcal{B}(x)$$

Proof. By the definitions of the existential and universal quantifiers, with the congruence of the choice operator. \square

3.2.2 Some Intuition

We can now try to sum up the properties of the classical operators: Intuitively, they all evaluate $\mathcal{A}(\cdot)$ for all classical input. If $\mathcal{A}(\cdot)$ loops, i.e., is undefined, for any one of these, then the result of the whole construct is undefined. If $\mathcal{A}(\cdot)$ is boolean for all classical input, then

- the choice operator returns some classical map making $\mathcal{A}(\cdot)$ true, or just any classical map if $\mathcal{A}(\cdot)$ is false for all classical maps,
- the existential quantifier returns true if any one of the classical inputs make $\mathcal{A}(\cdot)$ true and false otherwise, and
- the universal quantifier returns true if $\mathcal{A}(\cdot)$ is true for all classical input, and false otherwise.

For the quantifiers, their return value is uniquely described by the rules and intuition above. The result of the choice operator, however, is not. The term $\varepsilon x.\text{True}$ is classical is by Lemma 3.2.7, for example, but since any classical map will make True true, $\varepsilon x.\text{True}$ may be any of these maps. By the congruence rules, though, we can prove

$$\varepsilon x.\text{True} = \varepsilon x.\text{if } x \text{ then True else True}$$

So although both terms denote arbitrary classical terms, we can prove that they at least denote the same, arbitrary term.

The situation is similar for $\varepsilon x.\mathcal{A}(x)$ when $\mathcal{A}(\cdot)$ is not boolean for all classical input. This is because, while the intuition is that $\varepsilon x.\mathcal{A}(x)$ is undefined, the axiomatization of map theory does not capture this. Hence, $\varepsilon x.\mathcal{A}(x)$ is effectively unspecified: It may assume any value whatsoever, although it will be provably equal to $\varepsilon x.\mathcal{A}'(x)$ for any function $\mathcal{A}'(\cdot)$ equivalent, on

the classical maps, to $\mathcal{A}(\cdot)$. In particular, although $\varepsilon x.\perp$ “is” bottom, we cannot prove it. Although mildly inconvenient at times, there is an easy workaround. The term

$$(\forall x.\mathcal{A}(x) \text{ is bool}) \mapsto \varepsilon x.\mathcal{A}(x)$$

has the behavior described above: It returns the same value as $\varepsilon x.\mathcal{A}(x)$ when $\mathcal{A}(\cdot)$ is boolean for all classical input, and is bottom if not.

3.2.3 Types and Quantifiers

The cautious reader may have noticed a potential type problem with the term

$$\varepsilon x.\text{if } x \text{ then True else True}$$

shown above: the function **if** \cdot **then True else True** expects a boolean, while the choice operator expects a function on classical maps. In general, we would like the choice operator, as well as the existential and universal quantifiers, to be polymorphic, or at least parametrized, such that we had a quantifier $\forall_\alpha x.\mathcal{A}(x)$ for each type α , expressing that $\mathcal{A}(\cdot)$ was true for all input of type α . Obviously we would also want constructs for typed choice and existential quantifiers. The term above could then rightly be written

$$\varepsilon_{\text{bool}} x.\text{if } x \text{ then True else True}$$

As it is, any classical map is defined, and hence also a boolean. So $\varepsilon x.\mathcal{A}(x)$ can actually be thought of as testing $\mathcal{A}(\cdot)$ for boolean input. But does it also test $\mathcal{A}(\cdot)$ for *all* boolean input? In one word: No. The map $\lambda x.\perp$ represents false, but is not classical. Hence, if $\mathcal{A}(\cdot)$ is false for all classical booleans (boolean representatives that are also classical), but true for $\lambda x.\perp$ (or other non-classical booleans)², then $\exists x.\mathcal{A}(x)$ would incorrectly return false, although there in fact *was* boolean input satisfying $\mathcal{A}(\cdot)$. If $\mathcal{A}(\cdot)$ has quasi-type $\text{bool} \Rightarrow \alpha$, on the other hand, there is a solution: If all strong booleans are classical (**True** and **False**, in other words) then if any non-classical boolean B satisfies $\mathcal{A}(\cdot)$, then $\approx_{\text{bool}}(B)$, which is classical, will also satisfy $\mathcal{A}(\cdot)$. And, indeed, since **True** = **N** and **False** = **K****N**, all strong booleans are classical.

Quantifiers are of course vital for any serious development, and we will construct our data types, as often as we can, to be able to use them. Needless to say, while quantifiers can be removed for data types with a finite number of elements, any recursively defined data type will be seriously hampered without quantifiers. The simple types that allow use of quantifiers will be called classical types, and are defined thus

²One needs to go out of one’s way to construct such a $\mathcal{A}(\cdot)$, but since they exist we of course need to handle them. (As an example, using the notation of the next chapter, consider $\forall x.\exists y.x =_{\text{set}} A’y$. This is false for all classical A but true for, e.g., $A = \lambda x.x$.)

Definition 3.2.13 (Classical Type) *A simple type α is called a classical type if all strong representatives of α are classical, i.e.,*

$$A \text{ is } \alpha \implies \approx_\alpha(A) \text{ is classical}$$

and if there exists a function $\mathbf{CL}_\alpha(\cdot)$ such that

- $\mathbf{CL}_\alpha(\cdot)$ is a boolean predicate on the classical maps, i.e.,

$$A \text{ is classical} \implies \mathbf{CL}_\alpha(A) \text{ is bool}$$

- of the classical maps, only those of type α satisfy $\mathbf{CL}_\alpha(\cdot)$, i.e.,

$$A \text{ is classical and } \mathbf{CL}_\alpha(A) \implies A \text{ is } \alpha$$

- $\mathbf{CL}_\alpha(\cdot)$ holds for at least all strong α , i.e.,

$$A \text{ is } \alpha \implies \mathbf{CL}_\alpha(\approx_\alpha(A))$$

In short, $\mathbf{CL}_\alpha(\cdot)$ is a predicate on the classical maps that is true for a subset of α including the strong representatives.

As can be seen, the property of all strong representatives of a type be classical has been adopted directly from the corresponding property of the booleans. But the property that all classical maps represent some element of α will sometimes be inconvenient; An example is the non-negative rational numbers defined in Chapter 7. For such a type α , the expression $\exists x.\mathcal{A}(\cdot)$ will be undefined even when $\mathcal{A}(\cdot)$ is a total function of type $\alpha \Rightarrow \mathbf{bool}$, since there will be classical maps that make $\mathcal{A}(\cdot)$ loop. The solution we have chosen is to have a predicate, the function $\mathbf{CL}_\alpha(\cdot)$ in the definition above, to step around the classical maps that are not of type α . Of course, if all classical maps actually do represent α , then we can just let $\mathbf{CL}_\alpha(A)$ be \mathbf{True} .

As has already been remarked:

Lemma 3.2.14 *bool is a classical type.*

Proof. Both \mathbf{True} and \mathbf{False} are, as mentioned above, classical. Hence $A \text{ is bool}$ implies $\approx_{\mathbf{bool}}(A) \text{ is classical}$ follows using case distinction. Since all classical maps are defined and hence booleans, setting $\mathbf{CL}_{\mathbf{bool}}(A) = \mathbf{True}$ allows us to prove the rest of the properties. \square

Further, we also have

Lemma 3.2.15 *true is a classical type.*

Proof. Setting $\mathbf{CL}_{\mathbf{true}}(A) = \mathbf{ifnil } A \text{ then } \mathbf{True} \text{ else } \mathbf{False}$ allows us to prove the desired properties. \square

and, as the name suggests,

Lemma 3.2.16 *classical is a classical type.*

Proof. Let $\mathbf{CL}_{\text{classical}}(A) = \text{True}$. □

As examples of simple types that are not classical, we have `pair` and `defined`; They both have non-classical strong representatives, e.g., $\langle \perp, \perp \rangle$ and $\lambda x. \perp$.

We can now define typed quantifiers as well as a typed choice operator.

Definition 3.2.17 (Typed Choice and Quantification)

$$\begin{aligned} \varepsilon_{\alpha} x. \mathcal{A}(x) &\stackrel{\text{def}}{=} \varepsilon x. \mathbf{CL}_{\alpha}(x) \tilde{\wedge} \mathcal{A}(x) \\ \exists_{\alpha} x. \mathcal{A}(x) &\stackrel{\text{def}}{=} \exists x. \mathbf{CL}_{\alpha}(x) \tilde{\wedge} \mathcal{A}(x) \\ \forall_{\alpha} x. \mathcal{A}(x) &\stackrel{\text{def}}{=} \forall x. \mathbf{CL}_{\alpha}(x) \tilde{\rightarrow} \mathcal{A}(x) \end{aligned}$$

Before we prove the equivalents of Lemmas 3.2.3 through 3.2.7, note that when $\mathbf{CL}_{\alpha}(\cdot) = \text{True}$ then the typed versions of choice and the quantifiers is just the normal, classical constructs.³ Also notice that we have used the non-strict versions of conjunction and implication in the definition above: If we did not, the $\mathcal{A}(\cdot)$ would be evaluated regardless of the value of $\mathbf{CL}_{\alpha}(\cdot)$, and any undefinedness of $\mathcal{A}(\cdot)$ would “shine through”.

In the following proofs, we shall use

Lemma 3.2.18

$$\begin{aligned} A \tilde{\wedge} B \text{ is bool} &\iff A \tilde{\rightarrow} B \text{ is bool} \\ &\iff A \tilde{\rightarrow} (B \text{ is bool}) \end{aligned}$$

Proof. By distribution. □

Lemma 3.2.19 (Typed Universal Introduction) *If α is a classical type and $\mathcal{A}(B)$ holds for all B of type α then $\forall_{\alpha} x. \mathcal{A}(x)$.*

Proof. Assume α is a classical type. We are to prove $\forall x. \mathbf{CL}_{\alpha}(x) \tilde{\rightarrow} \mathcal{A}(x)$, so assume B is classical. Then either $\mathbf{CL}_{\alpha}(B)$ is true or false. If it is false, then $\mathbf{CL}_{\alpha}(B) \tilde{\rightarrow} \mathcal{A}(B)$ is true, regardless of the value of $\mathcal{A}(B)$. If $\mathbf{CL}_{\alpha}(B)$ is true then B must be α (since B was classical), and by assumption we have $\mathcal{A}(B)$. Again, $\mathbf{CL}_{\alpha}(B) \tilde{\rightarrow} \mathcal{A}(B)$ is true, and so $\mathbf{CL}_{\alpha}(B) \tilde{\rightarrow} \mathcal{A}(B)$ is true for all classical B , as claimed. □

³Except for a normalization, since, e.g., $\text{True} \tilde{\wedge} B = \approx_{\text{bool}}(B)$. Should complete equality be needed, using non-strict versions of implication and disjunction with the normalization of the second argument removed will allow this, while all the proofs in this section go through unmodified.

Lemma 3.2.20 (Typed Universal Elimination) *If α is a classical type, $\forall_\alpha x.\mathcal{A}(x)$ holds and B is α , then $\mathcal{A}(\approx_\alpha(B))$.*

Proof. Assume α is a classical type and $\forall_\alpha x.\mathcal{A}(x)$, that is, $\forall x.\mathbf{CL}_\alpha(x) \dot{\rightarrow} \mathcal{A}(x)$. Assuming B is α , $\approx_\alpha(B)$ must be classical. Hence $\mathbf{CL}_\alpha(\approx_\alpha(B)) \dot{\rightarrow} \mathcal{A}(\approx_\alpha(B))$. But from B is α we have $\mathbf{CL}_\alpha(\approx_\alpha(B))$, and thus we have $\mathcal{A}(\approx_\alpha(B))$, as claimed. \square

Lemma 3.2.21 (Typed Existential Introduction) *For all classical types α , if $\forall_\alpha x.(\mathcal{A}(x) \text{ is bool})$ and there exists C of type α such that $\mathcal{A}(\approx_\alpha(C))$, then $\exists_\alpha x.\mathcal{A}(x)$*

Proof. Assume α is a classical type. Assume further that $\forall_\alpha x.(\mathcal{A}(x) \text{ is bool})$, that is, $\forall x.\mathbf{CL}_\alpha(x) \dot{\rightarrow} (\mathcal{A}(x) \text{ is bool})$. Finally, assume $\mathcal{A}(\approx_\alpha(C))$ with C of type α . Since C is of type α we have $\mathbf{CL}_\alpha(\approx_\alpha(C))$. Hence

$$\mathbf{CL}_\alpha(\approx_\alpha(C)) \tilde{\wedge} \mathcal{A}(\approx_\alpha(C))$$

To prove $\exists x.\mathbf{CL}_\alpha(x) \tilde{\wedge} \mathcal{A}(x)$, we now only need to prove $\mathbf{CL}_\alpha(B) \tilde{\wedge} \mathcal{A}(B)$ is boolean for all classical B . So assume B is classical. Hence

$$\mathbf{CL}_\alpha(B) \dot{\rightarrow} (\mathcal{A}(B) \text{ is bool})$$

by the assumptions, and so $\mathbf{CL}_\alpha(B) \tilde{\wedge} \mathcal{A}(B) \text{ is bool}$ by Lemma 3.2.18. \square

Lemma 3.2.22 (Typed Existential Elimination) *If α is a classical type and $\exists_\alpha x.\mathcal{A}(x)$ then $\mathcal{A}(\varepsilon_\alpha x.\mathcal{A}(x))$*

Proof. Assume α is a classical type and $\exists_\alpha x.\mathcal{A}(x)$, that is, $\exists x.\mathbf{CL}_\alpha(x) \tilde{\wedge} \mathcal{A}(x)$. Hence

$$\mathbf{CL}_\alpha(\varepsilon x.\mathbf{CL}_\alpha(x) \tilde{\wedge} \mathcal{A}(x)) \tilde{\wedge} \mathcal{A}(\varepsilon x.\mathbf{CL}_\alpha(x) \tilde{\wedge} \mathcal{A}(x))$$

and thus $\mathcal{A}(\varepsilon x.\mathbf{CL}_\alpha(x) \tilde{\wedge} \mathcal{A}(x))$, which is equal $\mathcal{A}(\varepsilon_\alpha x.\mathcal{A}(x))$. \square

The type rules for the typed constructs are completely analogous to the classical versions:

Lemma 3.2.23 (Typed Classical Type Rules) *For all classical types α we have*

$$\begin{aligned} \varepsilon_\alpha x.\mathcal{A}(x) \text{ is classical} &\iff \forall_\alpha x.(\mathcal{A}(x) \text{ is bool}) \\ \exists_\alpha x.\mathcal{A}(x) \text{ is bool} &\iff \forall_\alpha x.(\mathcal{A}(x) \text{ is bool}) \\ \forall_\alpha x.\mathcal{A}(x) \text{ is bool} &\iff \forall_\alpha x.(\mathcal{A}(x) \text{ is bool}) \end{aligned}$$

Proof. Using Lemma 3.2.18, all rules are easily proved using the classical type rules. The first two follow by

$$\forall x. \mathbf{CL}_\alpha(x) \tilde{\wedge} \mathcal{A}(x) \text{ is bool} \iff \forall x. \mathbf{CL}_\alpha(x) \tilde{\rightarrow} (\mathcal{A}(x) \text{ is bool})$$

the last by

$$\forall x. \mathbf{CL}_\alpha(x) \tilde{\rightarrow} \mathcal{A}(x) \text{ is bool} \iff \forall x. \mathbf{CL}_\alpha(x) \tilde{\rightarrow} (\mathcal{A}(x) \text{ is bool})$$

□

Concluding our introduction of typed choice and quantifiers, we note that the central lemmas are completely analogous to their classical counterparts. Only typed existential introduction and typed universal elimination differ a bit, with an explicit normalization creeping in. In practice this will, as always, seldom be a problem. (In the next section, unfortunately, it will.)

3.2.4 The Missing Function Type Constructor

We will conclude this chapter with a discussion on the missing function type constructor of the last chapter. The problem was to find a function $\mathcal{F}(\cdot, \cdot)$ such that for all simple types α and β , $\mathcal{F}(\alpha, \beta)$ would be the type of functions of type $\alpha \Rightarrow \beta$.

The normalization function is easy to write, if we assume $\cdot \text{ is } \alpha \Rightarrow \beta$ given:

$$\approx_{\alpha \Rightarrow \beta}(F) = F \text{ is } \alpha \Rightarrow \beta \mapsto \lambda x. \approx_\beta(F'(\approx_\alpha(x)))$$

Assuming $\cdot \text{ is } \alpha \Rightarrow \beta$ is strict, then the function $\approx_{\alpha \Rightarrow \beta}(\cdot)$ above is clearly strict and idempotent, as we would want.

The problem is, of course, to define $\cdot \text{ is } \alpha \Rightarrow \beta$, i.e., capturing the properties of a function of type $\alpha \Rightarrow \beta$. Recalling the definition, we want $F \text{ is } \alpha \Rightarrow \beta$ to be true if and only if

1. $F' \perp = \perp$,
2. for all A and A' , $\approx_\alpha(A) = \approx_\alpha(A')$ implies $F'A = F'A'$, and
3. for all A , $\approx_\beta(F'A) = F'A$.

We shall have to abandon the third requirement: In general we cannot capture that a map A is a strong representative of a simple type α , i.e., $\approx_\alpha(A) = A$. The third requirement is, as should be expected by now, also a problem. Lowering our expectations somewhat, we shall be satisfied if $F \text{ is } \alpha \Rightarrow \beta$ it true exactly when, for all A and A' of type α ,

$$\approx_\alpha(A) = \approx_\alpha(A') \implies \approx_\beta(F'A) = \approx_\beta(F'A')$$

If α or β is not an equality type, then we have no chance getting this into a single term, so we will assume that both α and β are both equality types. If α is also a classical type, we can try

$$F \text{ is } \alpha \Rightarrow \beta = \approx_{\text{true}}(\forall_{\alpha} xy. x =_{\alpha} y \rightarrow F'x =_{\beta} F'y)$$

As when we defined the typed quantifiers, there will be problems if F is not total on α . So let's assume F is total for the moment: Using the typed universal elimination, we can from $F \text{ is } \alpha \Rightarrow \beta$ and A and A' of type α conclude

$$\approx_{\alpha}(A) =_{\alpha} \approx_{\alpha}(A') \rightarrow F' \approx_{\alpha}(A) =_{\beta} F' \approx_{\alpha}(A')$$

Simplifying, this amounts to $A =_{\alpha} A' \rightarrow F' \approx_{\alpha}(A) =_{\beta} F' \approx_{\alpha}(A')$. So assuming $A =_{\alpha} A'$, we can conclude $F' \approx_{\alpha}(A) =_{\beta} F' \approx_{\alpha}(A')$, which is trivial: It follows from the reflexivity of $=_{\beta}$ on β . The problem is that our version of typed universal elimination was formulated as to keep classicality out. One could prove a version like

Lemma 3.2.24 *If α is a classical type, $\forall_{\alpha} x. \mathcal{A}(x)$ holds and B is a classical α , then $\mathcal{A}(B)$.*

with which we could, from the assumption that $F \text{ is } \alpha \Rightarrow \beta$ was true and $A =_{\alpha} A'$ for A and A' classical of type α , conclude $F'A =_{\beta} F'A'$.

We have been assuming that F was total on α . If it isn't, one could use a trick similar to the one used for typed choice and quantification: using a term, perhaps called $\mathbf{CL}_F(\cdot)$, that was a predicate on α such that $\mathbf{CL}_F(A)$ was true exactly when $F'A$ was defined. (Then, of course, our type of functions of type $\alpha \Rightarrow \beta$ would actually have pairs $\langle F, \lambda x. \mathbf{CL}_F(x) \rangle$ as elements.)

As is hopefully clear, no neat solution to the definition of a functional type constructor has presented itself. In a nutshell, the problem is that types might have, and often do have, non-classical representatives, which are hard to capture using either the typed or classical quantifiers. One could of course demand that types only ever had classical representatives, effectively adding a guard $A \text{ is classical} \mapsto \cdot$ to all type membership functions $A \text{ is } \alpha$, but this would overly complicate many of the simpler types, and seems unnecessary: The lack of a function type constructor has not, it seems, impeded the formalization in the following chapters.

Finally, one can add that if such function type be classical, then, since all classical maps are strongly terminating (for classical input), this would mean that all the strong representatives of the function type would be total, and hence that all typeable functions would be total on the classical maps. Thus, a function type admitting partial functions could only hope to be simple anyway, and the value of such a type diminishes further.

Chapter 4

Set Theory

In this chapter we will sketch the formalization of set theory in map theory. The formalization is, needless to say, very technical, and since the entire development is painstakingly spelled out in Grue [Gru01], we will just try to give an impression of how sets are modelled and how the definitions of the usual set theoretic constructs look.

As briefly described in the introduction, any classical map will represent a ZFC set. To begin with, nil represents the empty set

$$\emptyset \stackrel{\text{def}}{=} \mathbf{N}$$

Since the classical maps are closed over application, we let any non-nil classical map A represent the set consisting of the sets represented by the classical maps in the image of the classical maps under A . Regarding set membership, we have $A \in B$ roughly when A is in the image of B , that is, when there exists a classical C such that A is equal to $B'C$. Perhaps after seeing the definition of equality between sets, the representation will become clearer:

$$A =_{\text{set}} B \stackrel{\text{def}}{=} \mathbf{ifnil} A \left\{ \begin{array}{l} \mathbf{ifnil} B \left\{ \begin{array}{l} \text{True} \\ \text{False} \end{array} \right. \\ \mathbf{ifnil} B \left\{ \begin{array}{l} \text{False} \\ (\forall x. \exists y. A'x =_{\text{set}} B'y) \wedge (\forall x. \exists y. A'y =_{\text{set}} B'x) \end{array} \right. \end{array} \right.$$

That is, the empty set is equal to only itself, and any non-empty set A is equal to the non-empty set B if for every C in A , C is also in B and vice versa.

With equality defined, membership has a straightforward definition:

$$A \in B \stackrel{\text{def}}{=} \mathbf{ifnil} A \mathbf{ then False else } \exists x. A =_{\text{set}} B'x$$

The subset relation is defined the usual way:

$$A \subseteq B \stackrel{\text{def}}{=} \forall x. x \in A \rightarrow x \in B$$

4.1 Finite Sets

Rather than defining the doubleton set $\{A, B\}$ directly, we shall define the operation of adding, or inserting, a set into another. Defining

$$\mathbf{insert\ } B \mathbf{ into\ } A \stackrel{\text{def}}{=} \mathbf{ifnil\ } A \left\{ \begin{array}{l} K'B \\ \lambda z.\mathbf{ifnil\ } z \mathbf{ then\ } B \mathbf{ else\ } A'(z'N) \end{array} \right.$$

we automatically have a uniform notation for all finite sets.

Noting that $K'B$ is equal to the singleton set consisting of B , the definition is easily read: If A is N , that is, if A is \emptyset , then $A \cup \{B\}$ is just $\{B\}$. Otherwise we must “insert” B into the range of A . To this end, we use the fact that for all classical maps A , there exists a classical map B ($K'A$, for example) such that A equals $B'N$.

Using the insertion function just defined, the syntax of finite sets can easily be handled. Just define

$$\{A_n, \dots, A_1\} \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \emptyset & \text{if } n = 0 \\ \mathbf{insert\ } A_n \mathbf{ into\ } \{A_{n-1}, \dots, A_1\} & \text{otherwise} \end{array} \right.$$

With this, we can make two quick sanity checks: First, we have $\{A\} = K'A$, as wanted. Second, we have

$$\begin{aligned} \{A, B\} &= \mathbf{insert\ } A \mathbf{ into\ } \{B\} \\ &= \mathbf{insert\ } A \mathbf{ into\ } K'B \\ &= \lambda z.\mathbf{ifnil\ } z \mathbf{ then\ } A \mathbf{ else\ } (K'B)'(z'N) \\ &= \lambda z.\mathbf{ifnil\ } z \mathbf{ then\ } A \mathbf{ else\ } B \\ &= \langle A, B \rangle \end{aligned}$$

Hence, our definition of doubleton sets coincides with that of Grue [Gru01].

4.2 Comprehension

Set comprehension is defined thus

$$\{x \in A | \mathcal{B}(x)\} \stackrel{\text{def}}{=} \mathbf{if\ } \exists x.x \in A \wedge \mathcal{B}(x) \left\{ \begin{array}{l} \lambda z.\mathbf{if\ } \mathcal{B}(A'z) \mathbf{ then\ } A'z \mathbf{ else\ } \varepsilon x.x \in A \wedge \mathcal{B}(x) \\ \emptyset \end{array} \right.$$

and reads very intuitively: Either there exists an x such that $x \in A$ and $\mathcal{B}(x)$ or not. If not, then obviously $\{x \in A | \mathcal{B}(x)\}$ is the empty set. If there does, we consider all the elements of A . If $\mathcal{B}(\cdot)$ holds for them, we pass them right through. Otherwise, we just pass on some element of A satisfying $\mathcal{B}(\cdot)$.

4.3 Union

Using set comprehension just defined, we have the skeleton for defining the union set constructions:

$$\bigcup A \stackrel{\text{def}}{=} \{x \in \mathcal{U}(A) \mid \exists y. x \in y \wedge y \in A\}$$

where $\mathcal{U}(A)$ is a specially constructed superset of $\bigcup A$:

$$\mathcal{U}(A) \stackrel{\text{def}}{=} \lambda z. A'(\mathbf{fst}z)'(\mathbf{snd}z)$$

As z goes through all classical maps, it also goes through all pairs of classical maps, and thus $\mathcal{U}(A)$, as a set, includes the members of the members of A . However, should A include the empty set, then, since $\emptyset' B = \emptyset$ for all B , the empty set will also be in $\mathcal{U}(A)$. In short, $\mathcal{U}(A)$ might well be a true superset of $\bigcup A$, which is why we must explicitly pick out the relevant sets by comprehension.

4.4 Power Set

We define the power set operator by

$$\wp A \stackrel{\text{def}}{=} \{x \in \mathcal{P}(A) \mid x \subseteq A\}$$

where $\mathcal{P}(A)$ is the superset of $\wp A$ defined by

$$\mathcal{P}(A) \stackrel{\text{def}}{=} \lambda x. \mathbf{ifnil} \ x \ \mathbf{then} \ \emptyset \ \mathbf{else} \ \lambda y. A'(x'(A'y))$$

Proving that $\mathcal{P}(A)$ includes all subsets of A is somewhat more involved, and we can unfortunately offer no intuitive argument for why it looks like it does.

4.5 Infinity

We construct the “usual” infinite set:

$$\begin{aligned} A^+ &\stackrel{\text{def}}{=} A \cup \{A\} \\ \omega &\stackrel{\text{def}}{=} \lambda x. \mathbf{ifnil} \ x \ \mathbf{then} \ \emptyset \ \mathbf{else} \ (\omega'(x'\mathbf{N}))^+ \end{aligned}$$

By succesively applying \mathbf{N} , $\lambda x_1. \mathbf{N}$, $\lambda x_1 x_2. \mathbf{N}$, etc., we see that ω comprises the sets \emptyset , $(\emptyset)^+$, $((\emptyset)^+)^+$, and so on.

4.6 Choice Operator

As the choice operator, we can use the usual choice operator:

$$\iota A \stackrel{\text{def}}{=} \varepsilon x. x \in A$$

4.7 Axioms

In the axioms of ZFC, we use the following predicate on contexts

$$\mathbf{setpred} \mathcal{P}(\cdot) \stackrel{\text{def}}{=} \forall xy. x =_{\text{set}} y \rightarrow \mathcal{P}(x) \leftrightarrow \mathcal{P}(y)$$

With the definitions given in the previous sections, we are able to prove that the axioms of ZFC given in Grue [Gru01] are provable in the form stated below.

$$\forall xy. x =_{\text{set}} y \leftrightarrow (\forall z. z \in x \leftrightarrow z \in y) \quad (\text{z-q})$$

$$\forall xyz. x =_{\text{set}} y \rightarrow (x =_{\text{set}} z \rightarrow y =_{\text{set}} z) \quad (\text{z-e})$$

$$\forall xy. x =_{\text{set}} y \rightarrow \iota x =_{\text{set}} \iota y \quad (\text{z-a})$$

$$\mathbf{setpred} \mathcal{A}(\cdot) \implies \forall y. \exists z. \forall x. x \in z \leftrightarrow (x \in y \wedge \mathcal{A}(x)) \quad (\text{z-s})$$

$$\forall xyz. x \in \{y, z\} \leftrightarrow (x =_{\text{set}} y \vee x =_{\text{set}} z) \quad (\text{z-p})$$

$$\forall x. x \notin \emptyset \quad (\text{z-n})$$

$$\forall xy. x \in \bigcup y \leftrightarrow (\exists z. x \in z \wedge z \in y) \quad (\text{z-u})$$

$$\forall xy. x \in \wp y \leftrightarrow (\forall z. z \in x \rightarrow z \in y) \quad (\text{z-w})$$

$$\emptyset \in \omega \wedge (\forall x. x \in \omega \rightarrow x^+ \in \omega) \quad (\text{z-i})$$

$$\forall u. \mathbf{setpred} \mathcal{A}(u, \cdot) \implies \forall z. \exists y. \forall x. x \in z \wedge (\exists u. \mathcal{A}(u, x) \rightarrow (\exists u. u \in y \wedge \mathcal{A}(u, x))) \quad (\text{z-r})$$

$$\forall x. x \neq_{\text{set}} \emptyset \rightarrow \iota x \in x \quad (\text{z-c})$$

$$\forall x. x \neq_{\text{set}} \emptyset \rightarrow (\exists z. z \in x \wedge \neg(\exists u. u \in x \wedge u \in z)) \quad (\text{z-d})$$

The lemmas z-mp, z-gen, and z-a1 through z-a5 from [Gru01] are tautologies of first order logic.

Chapter 5

Natural Numbers

We now turn to formalize the natural numbers and parts of elementary number theory. This will be needed to formalize the rationals, and then later the reals. The techniques and considerations used in Chapters 2 and 3 will be used here, and we will brush over many of the, hopefully, now familiar details. There is one thing, though, that makes the natural numbers, and hence many of the constructs of elementary number theory, stand out: They are recursively defined. This will give rise to new problems, which will be the main focus of the present chapter.

5.1 Defining the Natural Numbers

As the booleans are the foundation for propositional logic, the natural numbers are the foundation for elementary number theory. We are looking to define a classical equality type `nat` with constants `0` and `Suc(·)`, satisfying the properties captured in the famous Peano axioms. The version of the Peano axioms we shall use are these:

1. `0` is a natural number.
2. For all natural numbers A , `Suc(A)` is a natural number.
3. For all natural numbers A and B , if `Suc(A) =nat Suc(B)` then `A =nat B`.
4. `0` is not the successor of any natural number, i.e., if A is natural number then `0 ≠nat Suc(A)`.
5. If $\mathcal{P}(0)$ and for all natural numbers B , $\mathcal{P}(B)$ implies $\mathcal{P}(\text{Suc}(B))$ then $\mathcal{P}(A)$ holds for all natural numbers A .

(Incidentally, the Peano axioms started life much like the lambda calculus: As part of a larger system intended to serve as a foundation for mathematics. After the larger system was found to be irreparably inconsistent, the

Peano axioms and lambda calculus were salvaged from their respective systems as useful sub theories. Peano's original paper can be found in Van Heijenoort [vH67].)

The model of the natural numbers we shall use in the following is one where the natural number n will be strongly represented by the term

$$\lambda x_1 \dots x_n. \mathbf{N}$$

. The following definition will serve us:

Definition 5.1.1 (nat)

$$\begin{aligned} A \text{ is nat} &\stackrel{\text{def}}{=} \mathbf{ifnil } A \text{ then True else } (A \cdot \mathbf{N} \text{ is nat}) \\ \approx_{\text{nat}}(A) &\stackrel{\text{def}}{=} A \text{ is nat} \mapsto \mathbf{ifnil } A \text{ then } \mathbf{N} \text{ else } K'(\approx_{\text{nat}}(A \cdot \mathbf{N})) \\ 0 &\stackrel{\text{def}}{=} \mathbf{N} \\ \text{Suc}(A) &\stackrel{\text{def}}{=} A \text{ is nat} \mapsto K'(\approx_{\text{nat}}(A)) \end{aligned}$$

As can be seen, our definition of $\cdot \text{ is nat}$ accepts as natural numbers any term that, after applying it to nil a finite number of times, returns nil. Obviously, our intended strong naturals will be accepted. Also, any classical map will be a natural number: Thinking in terms of sets, assume A to be classical. Applying it to nil amounts to taking an element out of A . Since A is a set, it is of course well-founded, and so repeatedly picking out an element of A , then an element of this set, and so on, must eventually reach the empty set. Since all the strong representatives can be written on the form $K'(\dots(K' \mathbf{N}) \dots)$, they are clearly also classical, so we see that the requirements for a classical type are satisfied.

But we're getting ahead of ourselves: We first must prove that the natural numbers constitute a simple type. Since the natural numbers are recursively defined, this is not quite as simple as the case was for, say, the booleans. Proving, for example, that $\approx_{\text{nat}}(\cdot)$ is idempotent will need to use induction on the natural numbers, although these have not yet been defined yet. Here minimality of the Y-combinator comes to our rescue: It allows us to have a rudimentary induction principle for recursively defined functions.

Before proving nat is a simple type, we shall give some general directions on how to formalize recursive definitions, as well as prove some basic properties that will often be useful. For the time being, we will assume that we wish to construct a map F with the property

$$F \cdot A = \mathcal{C}(F, A) \tag{5.1}$$

Using the Y-combinator, such an F is easy to define:

$$F \stackrel{\text{def}}{=} Y'(\lambda f x. \mathcal{C}(f, x)) \tag{5.2}$$

Using the fixed point property of the Y-combinator, that for all B we have $Y'B = B'(Y'B)$, we can easily verify Equation 5.1:

$$\begin{aligned}
F'A &= Y'(\lambda fx.\mathcal{C}(f,x))'A \\
&= (\lambda fx.\mathcal{C}(f,x))'(Y'(\lambda fx.\mathcal{C}(f,x)))'A \\
&= (\lambda fx.\mathcal{C}(f,x))'F'A \\
&= (\lambda x.\mathcal{C}(F,x))'A \\
&= \mathcal{C}(F,A)
\end{aligned}$$

By the same method, we can of course solve the equation

$$F'A_1 \dots A_n = \mathcal{C}(F, A_1, \dots, A_n)$$

by setting F equal

$$Y'(\lambda fx_1 \dots x_n.\mathcal{C}(f, x_1, \dots, x_n))$$

Returning to the F defined in Equation 5.2, we shall often want to prove statements like

$$F'A \preceq G'A$$

If, for example, F and G are recursively defined, we may be able to prove $F'A \preceq G'A$ and $G'A \preceq F'A$, and then use the anti-symmetry of \preceq to conclude $F'A = G'A$.

Using the minimality of the Y-combinator, we have

Lemma 5.1.2 (Minimality of Recursive Definitions) *Assume F is equal to $Y'(\lambda fx.\mathcal{C}(f,x))$ and G is a function. Then if*

$$\mathcal{C}(G, B) \preceq G'B$$

for all maps B , then

$$F'A \preceq G'A$$

holds for all A .

Proof. Assume $F = Y'(\lambda fx.\mathcal{C}(f,x))$ and that $\mathcal{C}(G, B) \preceq G'B$ for all B . Hence $\lambda x.\mathcal{C}(G, x) \preceq \lambda x.G'x$, and so $(\lambda fx.\mathcal{C}(f,x))'G \preceq \lambda x.G'x$. Since G is a function, we have $G = \lambda x.G'x$, and so we've shown that

$$(\lambda fx.\mathcal{C}(f,x))'G \preceq G$$

Thus, using minimality of the Y-combinator, we have

$$F = Y'(\lambda fx.\mathcal{C}(f,x)) \preceq G$$

Given A , using monotonicity, we arrive at $F'A \preceq G'A$, as claimed. \square

Returning to our current main purpose, proving that `nat` is a simple type, we will now state the minimality results corresponding to `· is nat` and $\approx_{\text{nat}}(\cdot)$. First, perhaps, we should define how we exactly define

Lemma 5.1.3 (Minimality of `· is nat`) *If*

$$\mathbf{ifnil} B \mathbf{ then True else } \mathcal{C}(B^{\mathbf{N}}) \preceq \mathcal{C}(B)$$

for all B , then

$$A \mathbf{ is nat } \preceq \mathcal{C}(A)$$

for all A .

Proof. Since the precise definition of `· is nat` is

$$A \mathbf{ is nat } = Y'(\lambda f x. \mathbf{ifnil} x \mathbf{ then True else } f'(x^{\mathbf{N}}))'A$$

we let

$$F = Y'(\lambda f x. \mathbf{ifnil} x \mathbf{ then True else } f'(x^{\mathbf{N}}))$$

and $G = \lambda x. \mathcal{C}(x)$ in Lemma 5.1.2, which proves the wanted. \square

By a similar reasoning we also have

Lemma 5.1.4 (Minimality of $\approx_{\text{nat}}(\cdot)$) *If*

$$B \mathbf{ is nat } \mapsto \mathbf{ifnil} B \mathbf{ then N else } K'(\mathcal{C}(B^{\mathbf{N}})) \preceq \mathcal{C}(B)$$

for all B , then

$$\approx_{\text{nat}}(A) \preceq \mathcal{C}(A)$$

for all A .

Since the statement and proof of these minimality results is so mechanical, we shall, in the following, use the minimality of this or that recursively defined function without having explicitly stated such a lemma, or calling for Lemma 5.1.2.

We are now ready to prove some properties of `· is nat` and $\approx_{\text{nat}}(\cdot)$ that will be useful later.

Lemma 5.1.5 *For all A we have $A^{\mathbf{N}} \mathbf{ is nat } = A \mathbf{ is nat}$.*

Proof. By case distinction on A . \square

Lemma 5.1.6 *For all A we have $(\approx_{\text{nat}}(A)) \mathbf{ is nat } = A \mathbf{ is nat}$.*

Proof. We shall prove the claimed equality by anti-symmetry of \preceq : By minimality of \cdot **is nat** we shall first prove

$$A \text{ is nat} \preceq (\approx_{\text{nat}}(A)) \text{ is nat}$$

Assume B given, we wish to prove

$$\mathbf{ifnil} B \text{ then True else } (\approx_{\text{nat}}(B \cdot \mathbf{N})) \text{ is nat} \preceq (\approx_{\text{nat}}(B)) \text{ is nat}$$

Using minimality of **ifnil**, first assume $\approx_{\text{root}}(B) = \mathbf{N}$. Then both sides of the ordering are equal to **True**, and the ordering follows by reflexivity of \preceq . Now assume $\approx_{\text{root}}(B) = \lambda x. \mathbf{N}$. Using Lemma 5.1.5 we have

$$(\approx_{\text{nat}}(B)) \text{ is nat} = (\approx_{\text{nat}}(B \cdot \mathbf{N})) \text{ is nat}$$

and, again, the ordering follows by reflexivity of \preceq .

From the first half of the proof, we have $A \text{ is nat} \implies (\approx_{\text{nat}}(A)) \text{ is nat}$. Using this, we have

$$\begin{aligned} (\approx_{\text{nat}}(A)) \text{ is nat} &= A \text{ is nat} \mapsto (\approx_{\text{nat}}(A)) \text{ is nat} \\ &\preceq A \text{ is nat} \end{aligned}$$

concluding the proof. □

Lemma 5.1.7 *For all A and B we have $(\approx_{\text{nat}}(A)) \cdot B = \approx_{\text{nat}}(A \cdot \mathbf{N})$.*

Proof. By case distinction on A . □

We are now ready to prove that $\approx_{\text{nat}}(\cdot)$ is idempotent. At first, one might try to prove it using anti-symmetry of \preceq and minimality of $\approx_{\text{nat}}(\cdot)$. However, while minimality will allow us to prove

$$\approx_{\text{nat}}(A) \preceq \approx_{\text{nat}}(\approx_{\text{nat}}(A))$$

it will be of no help in proving the opposite

$$\approx_{\text{nat}}(\approx_{\text{nat}}(A)) \preceq \approx_{\text{nat}}(A)$$

since trying to prove this ordering with minimality gives us the obligation of showing

$$\mathbf{ifnil} B \text{ then } \mathbf{N} \text{ else } K \cdot \approx_{\text{nat}}(B \cdot \mathbf{N}) \preceq B$$

for all B , which is obviously false. In this case, we have found, there is no option but to use the strongest tool available to us: Extensionality.

Lemma 5.1.8 *$\approx_{\text{nat}}(\cdot)$ is idempotent.*

Proof. We are to prove

$$\approx_{\text{nat}}(\approx_{\text{nat}}(A)) = \approx_{\text{nat}}(A)$$

for all A . Using extensionality,

$$\approx_{\text{root}}(\approx_{\text{nat}}(\approx_{\text{nat}}(A))) = \approx_{\text{root}}(\approx_{\text{nat}}(A))$$

follows by case distinction on A , and we are to come up with some function $\mathcal{C}(\cdot, \cdot)$ such that

$$\begin{aligned} (\approx_{\text{nat}}(\approx_{\text{nat}}(A)))'B &= \approx_{\text{nat}}(\approx_{\text{nat}}(\mathcal{C}(A, B))) \\ (\approx_{\text{nat}}(A))'B &= \approx_{\text{nat}}(\mathcal{C}(A, B)) \end{aligned}$$

But using Lemma 5.1.7, we see that $\mathcal{C}(A, B) = A \cdot \mathbb{N}$ will do the trick. \square

Lemma 5.1.9 *nat is a simple type.*

Proof. \cdot is nat, and thus $\approx_{\text{nat}}(\cdot)$, is obviously strict. Idempotency of $\approx_{\text{nat}}(\cdot)$ was proved in Lemma 5.1.8. Finally, we must prove

$$A \text{ is nat} \iff \approx_{\text{nat}}(A) \text{ is defined}$$

“ \iff ” follows directly by the definition of $\approx_{\text{nat}}(\cdot)$. “ \implies ” is proved by showing

$$A \text{ is nat} \preceq \approx_{\text{nat}}(A) \text{ is defined}$$

by minimality of \cdot is nat. Finally, $\approx_{\text{true}}(A \text{ is nat}) = A \text{ is nat}$ for all A is easily proved using minimality of $A \text{ is nat}$. \square

We will now prove that the constructors 0 and $\text{Suc}(\cdot)$ are of the correct types:

Lemma 5.1.10 *0 is a natural number.*

Proof. Straightforward. \square

Lemma 5.1.11 *Suc(·) has type nat \Rightarrow nat and is total on the natural numbers.*

Proof. $\text{Suc}(\cdot)$ is clearly strict. Assume $\approx_{\text{nat}}(A) = \approx_{\text{nat}}(A')$. Then by definition of the successor function, we will have $\text{Suc}(A) = \text{Suc}(A')$. Next,

$$\begin{aligned} \approx_{\text{nat}}(\text{Suc}(A)) &= \approx_{\text{nat}}(A \text{ is nat} \mapsto \mathsf{K}'(\approx_{\text{nat}}(A))) \\ &= A \text{ is nat} \mapsto \approx_{\text{nat}}(\mathsf{K}'(\approx_{\text{nat}}(A))) \\ &= A \text{ is nat} \mapsto \mathsf{K}'(\approx_{\text{nat}}(\approx_{\text{nat}}(A))) \\ &= A \text{ is nat} \mapsto \mathsf{K}'(\approx_{\text{nat}}(A)) \\ &= \text{Suc}(A) \end{aligned}$$

So $\text{Suc}(\cdot)$ has the claimed type. Finally, assume A is natural number. Then

$$\begin{aligned} \text{Suc}(A) \text{ is nat} &= (A \text{ is nat} \mapsto \mathsf{K}'(\approx_{\text{nat}}(A))) \text{ is nat} \\ &= (\mathsf{K}'(\approx_{\text{nat}}(A))) \text{ is nat} \\ &= (\approx_{\text{nat}}(A)) \text{ is nat} \\ &= A \text{ is nat} \end{aligned}$$

and we conclude that $\text{Suc}(\cdot)$ is total on the natural numbers. \square

We already argued that nat was a classical type. To actually prove it, we shall use induction on natural numbers.

Lemma 5.1.12 (Induction on Natural Numbers) *If*

1. $A(0)$ and
2. for all natural numbers C , $A(C)$ implies $A(\text{Suc}(C))$

then if B is a natural number we have $A(\approx_{\text{nat}}(B))$.

Proof. We first prove

$$B \text{ is nat} \preceq \approx_{\text{true}}((B \text{ is nat} \mapsto A(\approx_{\text{nat}}(B))))$$

by minimality of $\cdot \text{ is nat}$ and the first two assumptions. Using the assumption that B was a natural number, we get $A(\approx_{\text{nat}}(B))$, as wanted. \square

Note that this lemma does not allow us to use induction to prove an equality $A = B$; Only terms can be proved true. The problem is, as always, that we cannot capture the equality $A = B$ in a term, and so the proof given above breaks. We have not been able to come up with an induction principle for general equations.

Now, finally,

Lemma 5.1.13 *nat is a classical type.*

Proof. We have just proved that nat is a simple type.

Further, any classical map is a natural number: Using induction on classical maps (Lemma 3.2.10), we are to prove N is a natural number, which it clearly is. Further, for arbitrary A , assuming $\forall x. A'x \text{ is nat}$, we are to prove $A \text{ is nat}$. But N is classical, so from the assumption we have $A'\mathsf{N} \text{ is nat}$, which by Lemma 5.1.5 is equivalent to $A \text{ is nat}$, as wanted.

We will prove all strong natural numbers are classical by induction on the natural numbers: $\approx_{\text{nat}}(0) = 0$ is classical. Assuming A is natural number and $\approx_{\text{nat}}(A)$ is classical, we have

$$\begin{aligned} \approx_{\text{nat}}(\text{Suc}(A)) &= \text{Suc}(A) \\ &= A \text{ is nat} \mapsto \mathsf{K}'(\approx_{\text{nat}}(A)) \\ &= \mathsf{K}'(\approx_{\text{nat}}(A)) \end{aligned}$$

and hence $\approx_{\text{nat}}(\text{Suc}(A))$ is also classical. This concludes the proof. \square

We have at this point proved Peano's first, second, and fifth axiom. To prove the last two, we will have to define equality between natural numbers. As was the case with the other equality types, we shall first define a case-construct:

Definition 5.1.14 (Case-construct for Natural Numbers)

$$\mathbf{case} \ A \ \mathbf{of} \ 0 \Rightarrow B \mid \mathbf{Suc} \ x \Rightarrow \mathcal{C}(x) \stackrel{\text{def}}{=} A \ \mathbf{is} \ \mathbf{nat} \mapsto \mathbf{ifnil} \ A \left\{ \begin{array}{l} B \\ \mathcal{C}(\approx_{\mathbf{nat}}(A'\mathbf{N})) \end{array} \right.$$

Lemma 5.1.15 *The function $\mathbf{case} \cdot \mathbf{of} \ 0 \Rightarrow B \mid \mathbf{Suc} \ x \Rightarrow \mathcal{C}(x)$ has the type $\mathbf{nat} \Rightarrow$ defined.*

Proof. It is clearly strict. Now assume $\approx_{\mathbf{nat}}(A) = \approx_{\mathbf{nat}}(A')$. Hence $A \ \mathbf{is} \ \mathbf{nat} = A' \ \mathbf{is} \ \mathbf{nat}$, and thus the definedness of both $\mathbf{case} \ A \ \mathbf{of} \ 0 \Rightarrow B \mid \mathbf{Suc} \ x \Rightarrow \mathcal{C}(x)$ and $\mathbf{case} \ A' \ \mathbf{of} \ 0 \Rightarrow B \mid \mathbf{Suc} \ x \Rightarrow \mathcal{C}(x)$ implies $A \ \mathbf{is} \ \mathbf{nat}$ and $A' \ \mathbf{is} \ \mathbf{nat}$. Using Lemma 2.12.1, we can therefore assume that A and A' are both natural numbers. Using this and distribution over \mathbf{ifnil} we can now prove

$$\begin{aligned} \approx_{\mathbf{root}}(A) &= \approx_{\mathbf{root}}(\approx_{\mathbf{nat}}(A)) \\ &= \approx_{\mathbf{root}}(\approx_{\mathbf{nat}}(A')) \\ &= \approx_{\mathbf{root}}(A') \end{aligned}$$

The last fact we shall use is

$$\begin{aligned} \approx_{\mathbf{nat}}(A'\mathbf{N}) &= \approx_{\mathbf{nat}}(A)\mathbf{N} \\ &= \approx_{\mathbf{nat}}(A')\mathbf{N} \\ &= \approx_{\mathbf{nat}}(A'\mathbf{N}) \end{aligned}$$

which holds by Lemma 5.1.7. Piecing the facts together we have

$$\begin{aligned} \mathbf{case} \ A \ \mathbf{of} \ 0 \Rightarrow B \mid \mathbf{Suc} \ x \Rightarrow \mathcal{C}(x) &= A \ \mathbf{is} \ \mathbf{nat} \mapsto \mathbf{ifnil} \ A \left\{ \begin{array}{l} B \\ \mathcal{C}(\approx_{\mathbf{nat}}(A'\mathbf{N})) \end{array} \right. \\ &= A' \ \mathbf{is} \ \mathbf{nat} \mapsto \mathbf{ifnil} \ A' \left\{ \begin{array}{l} B \\ \mathcal{C}(\approx_{\mathbf{nat}}(A'\mathbf{N})) \end{array} \right. \\ &= \mathbf{case} \ A' \ \mathbf{of} \ 0 \Rightarrow B \mid \mathbf{Suc} \ x \Rightarrow \mathcal{C}(x) \end{aligned}$$

as wanted. □

We also have

Lemma 5.1.16 $\mathbf{case} \ 0 \ \mathbf{of} \ 0 \Rightarrow B \mid \mathbf{Suc} \ x \Rightarrow \mathcal{C}(x) = B$

Proof. Straightforward. □

Lemma 5.1.17 *If A is a natural number, $\mathbf{case} \ \mathbf{Suc}(A) \ \mathbf{of} \ 0 \Rightarrow B \mid \mathbf{Suc} \ x \Rightarrow \mathcal{C}(x) = \mathcal{C}(\approx_{\mathbf{nat}}(A))$*

Proof. Assume A is natural number. Hence $\text{Suc}(A)$ is also a natural number. Also,

$$\begin{aligned}\text{Suc}(A) &= A \text{ is nat} \mapsto K'(\approx_{\text{nat}}(A)) \\ &= K'(\approx_{\text{nat}}(A))\end{aligned}$$

Therefore, we have

$$\begin{aligned}\text{case Suc}(A) \text{ of } 0 \Rightarrow B \mid \text{Suc } x \Rightarrow \mathcal{C}(x) &= \text{Suc}(A) \text{ is nat} \mapsto \text{ifnil Suc}(A) \left\{ \begin{array}{l} B \\ \mathcal{C}(\approx_{\text{nat}}(\text{Suc}(A)'\mathbf{N})) \end{array} \right. \\ &= \text{ifnil Suc}(A) \left\{ \begin{array}{l} B \\ \mathcal{C}(\approx_{\text{nat}}(\text{Suc}(A)'\mathbf{N})) \end{array} \right. \\ &= \mathcal{C}(\approx_{\text{nat}}(\text{Suc}(A)'\mathbf{N})) \\ &= \mathcal{C}(\approx_{\text{nat}}(K'(\approx_{\text{nat}}(A))'\mathbf{N})) \\ &= \mathcal{C}(\approx_{\text{nat}}(\approx_{\text{nat}}(A))) \\ &= \mathcal{C}(\approx_{\text{nat}}(A))\end{aligned}$$

as claimed. \square

Lemma 5.1.18 (Distribution over case_{nat}) *If $\mathcal{P}(\perp) = \perp$ then*

$$\mathcal{P}(\text{case } A \text{ of } 0 \Rightarrow B \mid \text{Suc } x \Rightarrow \mathcal{C}(x)) = \text{case } A \text{ of } 0 \Rightarrow \mathcal{P}(B) \mid \text{Suc } x \Rightarrow \mathcal{P}(\mathcal{C}(x))$$

Proof. By distribution over guards and **ifnil**. \square

Before continuing, we shall prove an elimination lemma similar to the one given for the booleans.

Lemma 5.1.19 (Elimination of Natural Numbers) *If*

1. $\approx_{\text{nat}}(C) = 0$ implies $A = B$
2. for all natural numbers D , $\approx_{\text{nat}}(C) = \text{Suc}(D)$ implies $A = B$

then if C is a natural number, $A = B$

Proof. Assume C is a natural number. Hence C is defined, and the root of C is either nil or $\lambda x.\mathbf{N}$. If it is nil , then $\approx_{\text{nat}}(C) = 0$ and we have $A = B$ by the first assumption. If $\approx_{\text{root}}(C) = \lambda x.\mathbf{N}$ then

$$\begin{aligned}\text{Suc}(C'\mathbf{N}) &= (C'\mathbf{N}) \text{ is nat} \mapsto K'(\approx_{\text{nat}}(C'\mathbf{N})) \\ &= C \text{ is nat} \mapsto K'(\approx_{\text{nat}}(C'\mathbf{N})) \\ &= \approx_{\text{nat}}(C)\end{aligned}$$

Since $C'\mathbf{N}$ is natural number by Lemma 5.1.5, we can use the second assumption to get $A = B$, as wanted. \square

We then have the following minimality result:

Lemma 5.1.20 (Minimality of case_{nat}) *If*

1. $\approx_{\text{nat}}(A) = 0$ *implies* $B \preceq D$ *and*
2. *for all natural numbers* E , $\approx_{\text{nat}}(A) = \text{Suc}(E)$ *implies* $\mathcal{C}(E) \preceq D$

then **case** A **of** $0 \Rightarrow B \mid \text{Suc } x \Rightarrow \mathcal{C}(x) \preceq D$.

Proof. If A is not a natural number, the left hand side is undefined and hence trivially less than D . So assume A is natural number. Using elimination on A the result is easily obtained. \square

We now proceed to define equality on the natural numbers. A natural definition is

$$A =_{\text{nat}} B = \begin{array}{l} \text{case } A \text{ of } 0 \Rightarrow \text{case } B \text{ of } 0 \Rightarrow \text{True} \mid \text{Suc } y \Rightarrow \text{False} \\ \mid \text{Suc } x \Rightarrow \text{case } B \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc } y \Rightarrow x =_{\text{nat}} y \end{array}$$

There is, however a slight problem with this definition: We will want to prove that $=_{\text{nat}}$ has type $[\text{nat}, \text{nat}] \Rightarrow \text{bool}$, and hence we want to prove

$$\approx_{\text{bool}}(A =_{\text{nat}} B) = A =_{\text{nat}} B$$

for all A and B . We would like to use induction for this, but in order to do so, we must translate the equation above into a term. However, general equivalence will not do the trick, since after proving $\approx_{\text{bool}}(A =_{\text{nat}} B) =_{\text{nat}} A =_{\text{nat}} B$ for all natural numbers A and B , we can only conclude

$$\approx_{\text{bool}}(\approx_{\text{bool}}(A =_{\text{nat}} B)) = \approx_{\text{bool}}(A =_{\text{nat}} B) \quad (5.3)$$

which is trivial. Minimality will also fail us, and extensionality ought to be unnecessary.

It so happens that we can, in fact, capture the strong booleans in a term, i.e., we have a function $\mathcal{S}(\cdot)$ with the property that

$$\mathcal{S}(A) \iff \approx_{\text{bool}}(A) = A \quad (5.4)$$

The trick is to see that **True** and **False** are the only terms with property that they return nil when applied to bottom. Nil obviously has that property, and if a function F satisfies $F' \perp = \mathbf{N}$ then for all B we have

$$\mathbf{N} = F' \perp \preceq F' B$$

implying $F' B = \mathbf{N}$ for all B . We can then use extensionality to prove $F = \mathbf{K}' \mathbf{N} = \text{False}$. In any case, any map B with $B' \perp = \mathbf{N}$ must be either **True** or **False**. Thus, defining the function $\mathcal{S}(\cdot)$ by

$$\mathcal{S}(A) = A' \perp$$

we can prove Equation 5.4. We can use $\mathcal{S}(\cdot)$ in the proof of Equation 5.3, by first proving

$$\forall_{\text{nat}} y. \mathcal{S}(A =_{\text{nat}} y)$$

by induction on A (since $=_{\text{nat}}$ is clearly strict in both arguments, we can assume A and B to be natural numbers by Lemma 2.12.1). Using typed universal elimination, we can have $\mathcal{S}(A =_{\text{nat}} (\approx_{\text{nat}}(B)))$, and thus

$$\approx_{\text{bool}}(A =_{\text{nat}} (\approx_{\text{nat}}(B))) = A =_{\text{nat}} (\approx_{\text{nat}}(B))$$

by Equation 5.4. Using the type of case_{nat} this can then be simplified to Equation 5.3.

So the definition of $=_{\text{nat}}$ above allows us to prove that $=_{\text{nat}}$ has the correct type by induction. But this is only due to the existence of $\mathcal{S}(\cdot)$, capturing the strong booleans. This is, however, a very rare property in a simple type: Besides the booleans (and truth, of course), we have only been able to prove a similar result for the natural numbers.¹ Since we can not, in general, capture the strong representatives of a simple type α , we will adopt the same technique for all recursive definitions: Adding explicit normalization to all recursive calls. Thus, the definition of equality on natural numbers we will actually use is this:

Definition 5.1.21 (Equality on Natural Numbers)

$$A =_{\text{nat}} B = \begin{array}{l} \text{case } A \text{ of } 0 \Rightarrow \text{case } B \text{ of } 0 \Rightarrow \text{True} \mid \text{Suc } y \Rightarrow \text{False} \\ \mid \text{Suc } x \Rightarrow \text{case } B \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc } y \Rightarrow \approx_{\text{bool}}(x =_{\text{nat}} y) \end{array}$$

Now the proof of the following lemma is straightforward.

Lemma 5.1.22 $=_{\text{nat}}$ has type $[\text{nat}, \text{nat}] \Rightarrow \text{bool}$.

Proof. $=_{\text{nat}}$ is clearly strict in both arguments. Assume $\approx_{\text{nat}}(A) = \approx_{\text{nat}}(A')$ and $\approx_{\text{nat}}(B) = \approx_{\text{nat}}(B')$, then $A =_{\text{nat}} B = A' =_{\text{nat}} B'$ follows from the type of case_{nat} . Finally, $\approx_{\text{bool}}(A =_{\text{nat}} B) = A =_{\text{nat}} B$ follows by distribution over case_{nat} . \square

We are ready to prove

Lemma 5.1.23 nat is an equality type.

Proof. (Correct Type) $=_{\text{nat}}$ has the required type by the previous lemma. (Totality) is proved by proving $\forall_{\text{nat}} y. ((A =_{\text{nat}} y) \text{ is bool})$ by induction on A . Hence $A =_{\text{nat}} (\approx_{\text{nat}}(B))$ is bool by typed universal elimination and thus $A =_{\text{nat}} B$ is boolean by the type of $=_{\text{nat}}$. (Reflexivity) is also proved by induction. (Reflection) is the biggest hurdle. It is proved using extensionality on

$$(A =_{\text{nat}} B) \mapsto \approx_{\text{nat}}(A) = (A =_{\text{nat}} B) \mapsto \approx_{\text{nat}}(B)$$

¹The function $\mathcal{S}(\cdot)$ given by $\mathcal{S}(A) = \text{ifnil } A \text{ then True else } \mathcal{S}(A \cdot \perp)$ captures the strong natural numbers. The proof is tedious and uses extensionality.

on both arguments simultaneously (this is done by technically coding up the functions above as single-argument functions taken the pair $\langle A, B \rangle$ as argument). \square

With the definition of equality given, Peano's second and third axioms follow immediately. This concludes our formalization of the natural numbers.

5.2 Arithmetic

We now have all the tools to make a quite standard implementation of Peano arithmetic. The definitions of addition and multiplication read

Definition 5.2.1

$$A +_{\text{nat}} B \stackrel{\text{def}}{=} B \text{ is nat} \mapsto \text{case } A \text{ of } 0 \Rightarrow \approx_{\text{nat}}(B) \mid \text{Suc } x \Rightarrow \text{Suc}(x +_{\text{nat}} \approx_{\text{nat}}(B))$$

$$A \times_{\text{nat}} B \stackrel{\text{def}}{=} B \text{ is nat} \mapsto \text{case } A \text{ of } 0 \Rightarrow 0 \mid \text{Suc } x \Rightarrow B +_{\text{nat}} x \times_{\text{nat}} \approx_{\text{nat}}(B)$$

As can be seen, these are straightforward definitions; We have only added a guard and explicit normalization to each definition. These additions serve to ease the proof of the next lemma.

Lemma 5.2.2 $\cdot +_{\text{nat}} \cdot$ and $\cdot \times_{\text{nat}} \cdot$ are both of type $[\text{nat}, \text{nat}] \Rightarrow \text{nat}$.

Proof. By the type of and distribution over case_{nat} . \square

We can now easily prove the standard equalities

$$\begin{aligned} 0 +_{\text{nat}} B &= \approx_{\text{nat}}(B) \\ \text{Suc}(A) +_{\text{nat}} B &= \text{Suc}(A +_{\text{nat}} B) \\ 0 \times_{\text{nat}} B &= 0 \\ \text{Suc}(A) \times_{\text{nat}} B &= B +_{\text{nat}} A \times_{\text{nat}} B \end{aligned}$$

and on the whole, there are no surprises in our formalization of arithmetic. Further, with

Definition 5.2.3

$$A \leq_{\text{nat}} B \stackrel{\text{def}}{=} \exists_{\text{nat}} x. A +_{\text{nat}} x =_{\text{nat}} B$$

$$A <_{\text{nat}} B \stackrel{\text{def}}{=} A \leq_{\text{nat}} B \wedge A \neq_{\text{nat}} B$$

we can prove that we have a linear ordered ring.

The only slight deviance from “normal” practice is our definition of subtraction, which we use for showing the cancellation law of addition. Instead of defining an explicit function, we have chosen to specify subtraction abstractly:

Definition 5.2.4

$$A -_{\text{nat}} B \stackrel{\text{def}}{=} B \leq_{\text{nat}} A \mapsto \approx_{\text{nat}}(\varepsilon_{\text{nat}} x.B +_{\text{nat}} x =_{\text{nat}} A)$$

Remembering choice—also typed choice—is not provably strict, we have added a guard, to make sure that subtracting B from A is undefined if B is greater than A . If not, we choose some natural number C such that $B +_{\text{nat}} C =_{\text{nat}} A$. We then later prove that the choice is unique, that is, if C is any number with the property just stated, then $\approx_{\text{nat}}(C) = A -_{\text{nat}} B$.

The type rules for subtraction are

$$A -_{\text{nat}} B \text{ is nat} \implies B \leq_{\text{nat}} A$$

and

$$B \leq_{\text{nat}} A \implies A -_{\text{nat}} B \text{ is nat}$$

and using these, the fact that subtraction is a partial function is, in itself, not a problem. Obviously, the equation

$$A -_{\text{nat}} B =_{\text{nat}} C = A =_{\text{nat}} C +_{\text{nat}} B$$

has to be qualified; If B is greater than A then the left hand side is undefined, while the right hand side is always defined. But if we had made subtraction total, for example, by defining $A -_{\text{nat}} B$ to be zero when B is greater than A , or perhaps just an arbitrary value (such as $\approx_{\text{nat}}(\varepsilon x.\text{True})$), we would still have to qualify the statement.

The definitions we have used to formalize the the results of elementary number theory which we shall need in the later chapters, specifically the chapter on the real numbers, are as follows:

Definition 5.2.5

$$A \bmod B \stackrel{\text{def}}{=} \text{if } B =_{\text{nat}} 0 \vee A <_{\text{nat}} B \begin{cases} \approx_{\text{nat}}(A) \\ \approx_{\text{nat}}((A -_{\text{nat}} B) \bmod (\approx_{\text{nat}}(B))) \end{cases}$$

$$A \text{ div } B \stackrel{\text{def}}{=} \text{if } B =_{\text{nat}} 0 \vee A <_{\text{nat}} B \begin{cases} 0 \\ \text{Suc}((A -_{\text{nat}} B) \text{ div } (\approx_{\text{nat}}(B))) \end{cases}$$

$$\text{gcd}(A, B) \stackrel{\text{def}}{=} A \text{ is nat} \mapsto \begin{cases} \text{case } B \text{ of } 0 & \Rightarrow \approx_{\text{nat}}(A) \\ | \text{Suc } x & \Rightarrow \approx_{\text{nat}}(\text{gcd}(B, A \bmod B)) \end{cases}$$

$$A \text{ divides } B \stackrel{\text{def}}{=} \exists_{\text{nat}} x.A \times_{\text{nat}} x =_{\text{nat}} B$$

$$A \div_{\text{nat}} B \stackrel{\text{def}}{=} B \neq_{\text{nat}} 0 \mapsto (B \text{ divides } A) \mapsto \approx_{\text{nat}}(\varepsilon_{\text{nat}} x.B \times_{\text{nat}} x =_{\text{nat}} A)$$

From these we can prove, for example, that for all natural numbers B ,

$$(A \text{ div } B) \times_{\text{nat}} B +_{\text{nat}} (A \bmod B) = \approx_{\text{nat}}(A)$$

and for all natural numbers A and B with $0 <_{\text{nat}} B$ we have

$$A \bmod B <_{\text{nat}} B$$

The proofs and results we shall use later are standard, and there are no surprises in the implementation of them in map theory.

Chapter 6

Pairs and Lists

In this chapter, we will define typed pairs, which we shall use for defining both the non-negative rationals as well the rationals. Also in this chapter is the development of lists.

6.1 Pairs

The typed pairs will be constructed by simply adding types to the basic pairs. That is, a typed pair $\langle A, B \rangle_{\alpha * \beta}$ will just be the pair $\langle \approx_{\alpha}(A), \approx_{\beta}(B) \rangle$, or undefined if either A is not α or B is not β . The exact definition is

Definition 6.1.1 (Typed Pairs)

$$\begin{aligned} P \text{ is } \alpha * \beta &\stackrel{\text{def}}{=} \text{case } P \text{ of } \langle a, b \rangle \Rightarrow a \text{ is } \alpha \mapsto b \text{ is } \beta \\ \approx_{\alpha * \beta}(P) &\stackrel{\text{def}}{=} P \text{ is } \alpha * \beta \mapsto \text{case } P \text{ of } \langle a, b \rangle \Rightarrow \langle \approx_{\alpha}(a), \approx_{\beta}(b) \rangle \end{aligned}$$

We have the following lemmas

Lemma 6.1.2

$$\langle A, B \rangle \text{ is } \alpha * \beta \iff A \text{ is } \alpha \text{ and } B \text{ is } \beta$$

Lemma 6.1.3 *If A is α and B is β then*

$$\approx_{\alpha * \beta}(\langle A, B \rangle) = \langle \approx_{\alpha}(A), \approx_{\beta}(B) \rangle$$

Lemma 6.1.4 *If P is $\alpha * \beta$ then P is pair.*

We will now prove

Lemma 6.1.5 *If α and β are simple types, then so is $\alpha * \beta$.*

Proof. Assume α and β are simple types. $\approx_{\alpha*\beta}(\cdot)$ is clearly strict. Further, in proving idempotency, i.e.,

$$\approx_{\alpha*\beta}(\approx_{\alpha*\beta}(P)) = \approx_{\alpha*\beta}(P)$$

we can assume P is $\alpha * \beta$ by Lemma 2.12.1. Hence P is also a basic pair. So assume $\approx_{\text{pair}}(P) = \langle A, B \rangle$. We now have

$$\begin{aligned} \approx_{\alpha*\beta}(\approx_{\alpha*\beta}(P)) &= \approx_{\alpha*\beta}(P \text{ is } \alpha * \beta \mapsto \mathbf{case } P \text{ of } \langle a, b \rangle \Rightarrow \langle \approx_{\alpha}(a), \approx_{\beta}(b) \rangle) \\ &= P \text{ is } \alpha * \beta \mapsto \mathbf{case } P \text{ of } \langle a, b \rangle \Rightarrow \approx_{\alpha*\beta}(\langle \approx_{\alpha}(a), \approx_{\beta}(b) \rangle) \\ &= P \text{ is } \alpha * \beta \mapsto \mathbf{case } P \text{ of } \langle a, b \rangle \Rightarrow \langle \approx_{\alpha}(\approx_{\alpha}(a)), \approx_{\beta}(\approx_{\beta}(b)) \rangle \\ &= P \text{ is } \alpha * \beta \mapsto \mathbf{case } P \text{ of } \langle a, b \rangle \Rightarrow \langle \approx_{\alpha}(a), \approx_{\beta}(b) \rangle \\ &= \approx_{\alpha*\beta}(P) \end{aligned}$$

Finally, we need to prove

$$P \text{ is } \alpha * \beta = \approx_{\alpha*\beta}(P) \text{ is defined}$$

but

$$\begin{aligned} \approx_{\alpha*\beta}(P) \text{ is defined} &= (P \text{ is } \alpha * \beta \mapsto \mathbf{case } P \text{ of } \langle a, b \rangle \Rightarrow \langle \approx_{\alpha}(a), \approx_{\beta}(b) \rangle) \text{ is defined} \\ &= P \text{ is } \alpha * \beta \mapsto \mathbf{case } P \text{ of } \langle a, b \rangle \Rightarrow \text{True} \\ &= P \text{ is } \alpha * \beta \mapsto P \text{ is pair} \\ &= P \text{ is } \alpha * \beta \mapsto \text{True} \\ &= P \text{ is } \alpha * \beta \end{aligned}$$

as claimed. □

Lemma 6.1.6 (Elimination of Typed Pairs) *If P is $\alpha * \beta$ and for all A of type α and B of type β , $\approx_{\alpha*\beta}(P) = \langle \approx_{\alpha}(A), \approx_{\beta}(B) \rangle$ implies $D = E$, then $D = E$.*

Proof. Assume P is $\alpha * \beta$. Then P is pair, and so assume $\approx_{\text{pair}}(P) = \langle A, B \rangle$. Hence

$$\begin{aligned} P \text{ is } \alpha * \beta &= \mathbf{case } P \text{ of } \langle a, b \rangle \Rightarrow a \text{ is } \alpha \mapsto b \text{ is } \beta \\ &= a \text{ is } \alpha \mapsto b \text{ is } \beta \end{aligned}$$

And so we have A is α and B is β . Further,

$$\begin{aligned} \approx_{\alpha*\beta}(P) &= P \text{ is } \alpha * \beta \mapsto \mathbf{case } P \text{ of } \langle a, b \rangle \Rightarrow \langle \approx_{\alpha}(a), \approx_{\beta}(b) \rangle \\ &= \mathbf{case } P \text{ of } \langle a, b \rangle \Rightarrow \langle \approx_{\alpha}(a), \approx_{\beta}(b) \rangle \\ &= \langle \approx_{\alpha}(A), \approx_{\beta}(B) \rangle \end{aligned}$$

The conclusion $D = E$ now follows from the assumptions. □

Lemma 6.1.7 *If α and β are classical types, then $\alpha * \beta$ is a classical type.*

Proof. Assume P is $\alpha * \beta$. We can assume A and B given such that A is α , B is β and $\approx_{\alpha*\beta}(P) = \langle \approx_\alpha(A), \approx_\beta(B) \rangle$. In other words, any strong typed pair will be of the form

$$\langle \approx_\alpha(A), \approx_\beta(B) \rangle$$

Since α and β are classical types by assumption, it follows that $\approx_\alpha(A)$ and $\approx_\beta(B)$ are classical, and so $\approx_{\alpha*\beta}(P)$ is classical (the pairing construct in set theory is just our basic pair construct).

Setting $\mathbf{CL}_{\alpha*\beta}(P) = \mathbf{case } P \mathbf{ of } \langle a, b \rangle \Rightarrow \mathbf{CL}_\alpha(a) \wedge \mathbf{CL}_\beta(b)$ allows us to prove the rest of the requirements. \square

The definition of the case construct for typed pairs is straightforward:

Definition 6.1.8 (Case-construct for Typed Pairs)

$$\mathbf{case } P \mathbf{ of } \langle a, b \rangle_{\alpha*\beta} \Rightarrow \mathcal{C}(a, b) \stackrel{\text{def}}{=} P \text{ is } \alpha * \beta \mapsto \mathbf{case } P \mathbf{ of } \langle a, b \rangle \Rightarrow \mathcal{C}(\approx_\alpha(a), \approx_\beta(b))$$

Lemma 6.1.9 $\mathbf{case } \cdot \mathbf{ of } \langle a, b \rangle_{\alpha*\beta} \Rightarrow \mathcal{C}(a, b)$ has type $\alpha * \beta \Rightarrow \text{defined}$.

Proof. It is clearly strict. Assume $\approx_{\alpha*\beta}(P) = \approx_{\alpha*\beta}(P')$. We can assume both P and P' to be $\alpha * \beta$, and thus both are pairs. We therefore assume A, A', B and B' given such that $\approx_{\text{pair}}(P) = \langle A, B \rangle$ and $\approx_{\text{pair}}(P') = \langle A', B' \rangle$. We can also prove A and A' are of type α and B and B' are of type β and that, also, $\approx_{\alpha*\beta}(P) = \langle \approx_\alpha(A), \approx_\beta(B) \rangle$ and $\approx_{\alpha*\beta}(P') = \langle \approx_\alpha(A'), \approx_\beta(B') \rangle$. Thus

$$\langle \approx_\alpha(A), \approx_\beta(B) \rangle = \langle \approx_\alpha(A'), \approx_\beta(B') \rangle$$

and hence $\approx_\alpha(A) = \approx_\alpha(A')$ and $\approx_\beta(B) = \approx_\beta(B')$. We finally prove

$$\begin{aligned} \mathbf{case } P \mathbf{ of } \langle a, b \rangle_{\alpha*\beta} \Rightarrow \mathcal{C}(a, b) &= P \text{ is } \alpha * \beta \mapsto \mathbf{case } P \mathbf{ of } \langle a, b \rangle \Rightarrow \mathcal{C}(\approx_\alpha(a), \approx_\beta(b)) \\ &= \mathbf{case } P \mathbf{ of } \langle a, b \rangle \Rightarrow \mathcal{C}(\approx_\alpha(a), \approx_\beta(b)) \\ &= \mathcal{C}(\approx_\alpha(A), \approx_\beta(B)) \\ &= \mathcal{C}(\approx_\alpha(A'), \approx_\beta(B')) \\ &= \mathbf{case } P' \mathbf{ of } \langle a, b \rangle \Rightarrow \mathcal{C}(\approx_\alpha(a), \approx_\beta(b)) \\ &= P' \text{ is } \alpha * \beta \mapsto \mathbf{case } P' \mathbf{ of } \langle a, b \rangle \Rightarrow \mathcal{C}(\approx_\alpha(a), \approx_\beta(b)) \\ &= \mathbf{case } P' \mathbf{ of } \langle a, b \rangle_{\alpha*\beta} \Rightarrow \mathcal{C}(a, b) \end{aligned}$$

as wanted. \square

We can also show

Lemma 6.1.10 *If A is α and B is β then*

$$\mathbf{case } \langle A, B \rangle \mathbf{ of } \langle a, b \rangle_{\alpha*\beta} \Rightarrow \mathcal{C}(a, b) = \mathcal{C}(\approx_\alpha(A), \approx_\beta(B))$$

Lemma 6.1.11 (Minimality of case _{$\alpha*\beta$}) *If, for all A of type α and B of type β , $\approx_{\text{pair}}(P) = \langle \approx_\alpha(A), \approx_\beta(B) \rangle$ implies $\mathcal{C}(\approx_\alpha(A), \approx_\beta(B)) \preceq D$ then*

$$\text{case } P \text{ of } \langle a, b \rangle_{\alpha*\beta} \Rightarrow \mathcal{C}(a, b) \preceq D$$

Equality on typed pairs is done as is usual: coordinatewise.

Definition 6.1.12 (Equality on Typed Pairs)

$$P =_{\alpha*\beta} Q \stackrel{\text{def}}{=} \text{case } P \text{ of } \langle p_1, p_2 \rangle \Rightarrow \text{case } Q \text{ of } \langle q_1, q_2 \rangle \Rightarrow p_1 =_\alpha q_1 \wedge p_2 =_\beta q_2$$

Lemma 6.1.13 *If α and β are equality types, then so is $\alpha * \beta$.*

Proof. All properties follow directly from α and β being equality types. \square

We have up till now not actually defined a constructor for typed pairs, and instead borrowed the one from basic pairs. Should one want such, and for completeness, it is defined thus:

Definition 6.1.14

$$\langle A, B \rangle_{\alpha*\beta} \stackrel{\text{def}}{=} \approx_{\alpha*\beta}(\langle A, B \rangle)$$

Lemma 6.1.15 $\langle \cdot, \cdot \rangle_{\alpha*\beta}$ *has type* $[\alpha, \beta] \Rightarrow \alpha * \beta$.

Proof. Straightforward. \square

6.2 Lists

We choose to build lists as we built pairs: First we define the so-called basic lists, that have no type information, and then we use these to define typed lists.

The basic lists we define will be built using the basic pairs. The strong representatives will be defined by

$$[x_1, \dots, x_n] \stackrel{\text{def}}{=} \begin{cases} \mathbf{N} & \text{if } n = 0 \\ \langle x_1, [x_2, \dots, x_n] \rangle & \text{otherwise} \end{cases}$$

A suitable definition of the type of basic lists is now

Definition 6.2.1 (list)

$$\begin{aligned} L \text{ is list} &\stackrel{\text{def}}{=} \text{ifnil } L \text{ then True else case } L \text{ of } \langle h, t \rangle \Rightarrow (t \text{ is list}) \\ \approx_{\text{list}}(L) &\stackrel{\text{def}}{=} L \text{ is list} \mapsto \text{ifnil } L \text{ then N else case } L \text{ of } \langle h, t \rangle \Rightarrow \langle h, \approx_{\text{list}}(t) \rangle \\ \text{Nil} &\stackrel{\text{def}}{=} \mathbf{N} \\ \text{Cons } A \ L &\stackrel{\text{def}}{=} L \text{ is list} \mapsto \langle A, \approx_{\text{list}}(L) \rangle \end{aligned}$$

This means that, just like a natural number is a term that, when repeatedly applied to nil, eventually returns nil, so a basic list will be a term that, when one repeatedly applies **fst** to it, also returns nil eventually. This similarity is used in the proof of the next lemma.

Lemma 6.2.2 *For all maps A we have $(\approx_{\text{list}}(A)) \text{ is list} = A \text{ is list}$.*

Proof. Completely analogous to the proof of Lemma 5.1.6. \square

Lemma 6.2.3 *$\approx_{\text{list}}(\cdot)$ is idempotent.*

Proof. In showing

$$\approx_{\text{list}}(\approx_{\text{list}}(L)) = \approx_{\text{list}}(L)$$

we might first try to use extentionality directly, as we did in the corresponding lemma for the natural numbers, Lemma 5.1.8. This would require as to come up with some function $\mathcal{C}(\cdot, \cdot)$ such that

$$\begin{aligned} (\approx_{\text{list}}(\approx_{\text{list}}(A)))'B &= \approx_{\text{list}}(\approx_{\text{list}}(\mathcal{C}(A, B))) \\ (\approx_{\text{list}}(A))'B &= \approx_{\text{list}}(\mathcal{C}(A, B)) \end{aligned}$$

But such a function does not exist: No matter how we define it, the right hand sides above will be strong basic lists, because of the explicit normalizations. The left hand sides, however, when L is not nil and B is, can be anything.

To use extentionality, and the other proof techniques seem to weak, we therefore need to be able to “tweak” the return type of the functions in question, depending on the value of $\mathcal{C}(\cdot, \cdot)$. In the present case, we will use extentionality to prove

$$\mathbf{case } A \text{ of } \langle x, y \rangle \Rightarrow \mathbf{if } x \begin{cases} y \\ \approx_{\text{list}}(\approx_{\text{list}}(y)) \end{cases} = \mathbf{case } A \text{ of } \langle x, y \rangle \Rightarrow \mathbf{if } x \begin{cases} y \\ \approx_{\text{list}}(y) \end{cases} \quad (6.1)$$

for arbitrary A . The first component is a boolean, dictating whether the second component should be passed through as the result (allowing us to bypass the explicit normalizations), or if the “real” function should be applied. We now define $\mathcal{C}(A, B)$ by

$$\mathbf{case } A \text{ of } \langle x, y \rangle \Rightarrow \mathbf{if } x \begin{cases} \langle \text{True}, y'B \rangle \\ y \text{ is list} \mapsto \mathbf{ifnil } y \begin{cases} \langle \text{True}, \mathbf{N} \rangle \\ \langle \approx_{\text{bool}}(B), y'(\approx_{\text{root}}(B)) \rangle \end{cases} \end{cases}$$

While this may look horrid, it really isn't *that* bad: Assume $A = \langle A', B' \rangle$. If A' is true, then the functions from Equation 6.1 just pass B' through, and the result of applying B to the result will of course be $B'B$. Now, if A' is false, the functions return the “real” results, i.e., $\approx_{\text{list}}(\approx_{\text{list}}(B'))$ and

$\approx_{\text{list}}(B')$. So if B' is not a list, the results of both will be undefined anyway, and we can utilize the strictness of the functions in Equation 6.1. OK, so now that B' can be assumed to be a list, we need to make the distinction mentioned above: If B' is nil, then the result of the normalizations will in any case be nil, so we can just pass it right through. If not, then B' must be a pair $\langle H, T \rangle$, and the result of the original functions will be $\langle H, \approx_{\text{list}}(\approx_{\text{list}}(T)) \rangle$ and $\langle H, \approx_{\text{list}}(T) \rangle$. Hence, if applied to B , then if B is nil, the result is just $H = B' \cdot B$ and if B is a function then the result is, in the second case, $\approx_{\text{list}}(T) = \approx_{\text{list}}(B' \lambda x. \mathbf{N})$.

Once we have set extentionality up with $\mathcal{C}(\cdot, \cdot)$ defined as above, the rest of the proof of Equation 6.1 is tedious but quite straightforward. Having proved it for all A , we have that it holds especially for $A = \langle \text{False}, L \rangle$, proving the desired result. \square

Lemma 6.2.4 *list is a simple type.*

Proof. $\approx_{\text{list}}(\cdot)$ is clearly strict, and it is idempotent by the previous lemma. $L \text{ is list} = (\approx_{\text{list}}(L)) \text{ is defined}$ follows, as earlier, immediately in one direction and by minimality (this time of $\cdot \text{ is list}$) in the other. \square

We can now proceed to prove that Nil is a strong list, and that $\text{Cons } A \cdot$ is total on lists and of the type $\text{list} \Rightarrow \text{list}$. Defining the case construct as

Definition 6.2.5

$$\text{case } L \text{ of Nil} \Rightarrow A \mid \text{Cons } h \ t \Rightarrow \mathcal{C}(h, t) = L \text{ is list} \mapsto \mathbf{ifnil} \ L \left\{ \begin{array}{l} A \\ \text{case } L \text{ of } \langle h, t \rangle \Rightarrow \mathcal{C}(h, \approx_{\text{list}}(t)) \end{array} \right.$$

We get the usual properties of the case constructs: the correct type, distribution, minimality, etc. In the present case, we naturally have

$$\begin{aligned} \text{case Nil of Nil} \Rightarrow A \mid \text{Cons } h \ t \Rightarrow \mathcal{C}(h, t) &= A \\ \text{case (Cons } A \ L) \text{ of Nil} \Rightarrow A \mid \text{Cons } h \ t \Rightarrow \mathcal{C}(h, t) &= \mathcal{C}(A, \approx_{\text{list}}(L)) \end{aligned}$$

We can also prove the following elimination and induction principles. As with the natural numbers, the induction principle is only defined on terms, not equalities. The proofs are straightforward.

Lemma 6.2.6 (Elimination of Basic Lists) *If*

1. $\approx_{\text{list}}(C)$ implies $A = B$ and
2. for all maps H and basic lists T , $\approx_{\text{list}}(C) = \text{Cons } H \ T$ implies $A = B$

then if C is a basic list, $A = B$.

Lemma 6.2.7 (Induction on Basic Lists) *If*

1. $\mathcal{A}(\text{Nil})$ and
2. for all maps H and lists T , $\mathcal{A}(T)$ implies $\mathcal{A}(\text{Cons } H T)$

then if B is a basic list we have $\mathcal{A}(\approx_{\text{list}}(B))$.

When defining normalization of typed lists, we shall use the map function, defined as usual:

Definition 6.2.8

$$\mathbf{map } F L \stackrel{\text{def}}{=} \mathbf{case } L \mathbf{ of Nil} \Rightarrow \text{Nil} \mid \mathbf{Cons } x l \Rightarrow \mathbf{Cons } (F'x) (\mathbf{map } F l)$$

From this definition, we readily have that $\mathbf{map } F \cdot$ has type $\text{list} \Rightarrow \text{list}$. Further, totality on list is proven easily by induction. However, proving the identity

$$\mathbf{map } F (\mathbf{map } G L) = \mathbf{map } (F \circ G) L$$

where $F \circ G = \lambda x.F'(G'x)$, is a problem. It would be an easy task for induction, but we have no way of translating the equality into a term: list is not an equality type. Minimality will not be of use showing the \preceq part, and so we are once more left with extentionality. Using the same technique as when proving idempotency of $\approx_{\text{list}}(\cdot)$, we prove

$$\mathbf{case } A \mathbf{ of } \langle x, y \rangle \Rightarrow \mathbf{if } x \left\{ \begin{array}{l} y \\ \mathbf{map } F (\mathbf{map } G y) \end{array} \right. = \mathbf{case } A \mathbf{ of } \langle x, y \rangle \Rightarrow \mathbf{if } x \left\{ \begin{array}{l} y \\ \mathbf{map } (F \circ G) y \end{array} \right.$$

by extentionality, where we this time define $\mathcal{C}(A, B)$ as

$$\mathbf{case } A \mathbf{ of } \langle x, y \rangle \Rightarrow \mathbf{if } x \left\{ \begin{array}{l} \langle \text{True}, y'B \rangle \\ y \text{ is list} \mapsto \mathbf{ifnil } y \left\{ \begin{array}{l} \langle \text{True}, \mathbf{N} \rangle \\ \mathbf{ifnil } B \left\{ \begin{array}{l} \langle \text{True}, F'(G'(y'\mathbf{N})) \rangle \\ \langle \text{False}, y'(\approx_{\text{root}}(B)) \rangle \end{array} \right. \end{array} \right. \end{array} \right.$$

The proof is, as before, tedious and boring.

Finally, we are able to define the type of typed lists. We do the same as we did when defining typed pairs: We just normalize the components of the untyped lists.

Definition 6.2.9 (Typed Lists)

$$\begin{aligned} L \text{ is } \alpha \text{ list} &\stackrel{\text{def}}{=} \mathbf{case } L \mathbf{ of Nil} \Rightarrow \text{True} \mid \mathbf{Cons } a l \Rightarrow a \text{ is } \alpha \mapsto l \text{ is } \alpha \text{ list} \\ \approx_{(\alpha \text{ list})}(L) &\stackrel{\text{def}}{=} L \text{ is } \alpha \text{ list} \mapsto \mathbf{map } (\lambda x.\approx_{\alpha}(x)) L \end{aligned}$$

We shall not continue the development of typed lists here, but move on to define the reals in the next chapter.

Chapter 7

Real Numbers

In this chapter we turn our attention to defining the real numbers, building on top of the natural numbers. There are several ways of accomplishing this, and which ever way one chooses, the development up to and including the rationals, is very straightforward, although quite a bit of work is needed.

The concrete path of formalization that we have chosen is

$$\mathbb{N} \implies \mathbb{Q}^+ \implies \mathbb{Q} \implies \mathbb{R}$$

That is, we shall from the natural numbers define the non-negative rationals, then the rationals, and finally the reals. As the development contains no surprises, we shall brush past the non-negative rationals and the rationals.

7.1 Non-Negative Rationals

The non-negative rational $\frac{A}{B}$ is represented by the pair $\langle A, B \rangle$. And any pair $\langle A, B \rangle$ of natural numbers A and B , where B is not zero, will represent the non-negative rational $\frac{A}{B}$. As strong representatives, we choose the canceled fractions, i.e., pairs $\langle A, B \rangle$ where A and B are co-prime.

Definition 7.1.1 (Non-Negative Rationals)

$$\begin{aligned} A \text{ is nrat} &\stackrel{def}{=} \text{case } A \text{ of } \langle n, m \rangle_{\text{nat} * \text{nat}} \Rightarrow m \neq_{\text{nat}} 0 \mapsto \text{True} \\ \approx_{\text{nrat}}(A) &\stackrel{def}{=} A \text{ is nrat} \mapsto \text{case } A \text{ of } \langle n, m \rangle_{\text{nat} * \text{nat}} \Rightarrow \langle n \div_{\text{nat}} \text{gcd}(n, m), m \div_{\text{nat}} \text{gcd}(n, m) \rangle \end{aligned}$$

We can prove that nrat is a simple type, and setting

$$\mathbf{CL}_{\text{nrat}}(A) = \text{case } A \text{ of } \langle n, m \rangle_{\text{nat} * \text{nat}} \Rightarrow m \neq_{\text{nat}} 0$$

we can also prove that it is a classical type.

We will for the non-negative rationals, and later also for the rationals, opt for explicit deconstructors, rather than a case constructor:

Definition 7.1.2

$$\begin{aligned} \mathbf{num}(A) &= A \text{ is nrat} \mapsto \mathbf{case } A \text{ of } \langle n, m \rangle_{\mathbf{nat} * \mathbf{nat}} \Rightarrow n \\ \mathbf{den}(A) &= A \text{ is nrat} \mapsto \mathbf{case } A \text{ of } \langle n, m \rangle_{\mathbf{nat} * \mathbf{nat}} \Rightarrow m \end{aligned}$$

We can then prove that $\mathbf{num}(\cdot)$ and $\mathbf{den}(\cdot)$ are both of type $\mathbf{nrat} \Rightarrow \mathbf{nat}$ and $\mathbf{den}(\cdot)$ is never zero.

The non-negative rational constants zero and one are different from their natural counterparts. We will therefore subscript the former, and define them as

Definition 7.1.3

$$\begin{aligned} 0_{\mathbf{nrat}} &\stackrel{\text{def}}{=} \langle 0, 1 \rangle \\ 1_{\mathbf{nrat}} &\stackrel{\text{def}}{=} \langle 1, 1 \rangle \end{aligned}$$

We can now define

Definition 7.1.4 (Operations on Non-Negative Rationals)

$$\begin{aligned} A =_{\mathbf{nrat}} B &\stackrel{\text{def}}{=} \mathbf{num}(A) \times_{\mathbf{nat}} \mathbf{den}(B) =_{\mathbf{nat}} \mathbf{den}(A) \times_{\mathbf{nat}} \mathbf{num}(B) \\ A \times_{\mathbf{nrat}} B &\stackrel{\text{def}}{=} \approx_{\mathbf{nrat}} (\langle \mathbf{num}(A) \times_{\mathbf{nat}} \mathbf{num}(B), \mathbf{den}(A) \times_{\mathbf{nat}} \mathbf{den}(B) \rangle) \\ A +_{\mathbf{nrat}} B &\stackrel{\text{def}}{=} \approx_{\mathbf{nrat}} (\langle \mathbf{num}(A) \times_{\mathbf{nat}} \mathbf{den}(B) +_{\mathbf{nat}} \mathbf{den}(A) \times_{\mathbf{nat}} \mathbf{num}(B), \mathbf{den}(A) \times_{\mathbf{nat}} \mathbf{den}(B) \rangle) \\ A \leq_{\mathbf{nrat}} B &\stackrel{\text{def}}{=} \mathbf{num}(A) \times_{\mathbf{nat}} \mathbf{den}(B) \leq_{\mathbf{nat}} \mathbf{den}(A) \times_{\mathbf{nat}} \mathbf{num}(B) \\ A -_{\mathbf{nrat}} B &\stackrel{\text{def}}{=} A \leq_{\mathbf{nrat}} B \mapsto \approx_{\mathbf{nrat}} (\varepsilon_{\mathbf{nrat}} x.B +_{\mathbf{nrat}} x =_{\mathbf{nrat}} A) \\ (A)_{\mathbf{nrat}}^{-1} &\stackrel{\text{def}}{=} A \neq_{\mathbf{nrat}} 0_{\mathbf{nrat}} \mapsto \approx_{\mathbf{nrat}} (\langle \mathbf{den}(A), \mathbf{num}(A) \rangle) \\ A <_{\mathbf{nrat}} B &\stackrel{\text{def}}{=} A \leq_{\mathbf{nrat}} B \wedge A \neq_{\mathbf{nrat}} B \end{aligned}$$

From these, we can prove that \mathbf{nrat} is an equality type, and a host of lemmas about the non-negative rationals, that will eventually help us prove the needed properties of the rationals.

7.2 Rationals

We represent rationals by pairs of non-negative rationals: The positive rational A is represented by $\langle A, 0_{\mathbf{nrat}} \rangle$ and the negative rational A by $\langle 0_{\mathbf{nrat}}, A \rangle$. (Zero is represented by $\langle 0_{\mathbf{nrat}}, 0_{\mathbf{nrat}} \rangle$.) Conversely, any pair of non-negative rationals $\langle A, B \rangle$ will represent the rational $A - B$.

Definition 7.2.1 (Rationals)

$$\begin{aligned} A \text{ is rat} &\stackrel{\text{def}}{=} A \text{ is nrat} * \mathbf{nrat} \\ \approx_{\mathbf{nrat}}(A) &\stackrel{\text{def}}{=} \mathbf{case } A \text{ of } \langle p, n \rangle_{\mathbf{nrat} * \mathbf{nrat}} \Rightarrow \mathbf{if } n \leq_{\mathbf{nrat}} p \left\{ \begin{array}{l} \langle p -_{\mathbf{nrat}} n, 0_{\mathbf{nrat}} \rangle \\ \langle 0_{\mathbf{nrat}}, n -_{\mathbf{nrat}} p \rangle \end{array} \right. \end{aligned}$$

It is classical type since nnrat is.

The deconstructors are easily defined.

Definition 7.2.2

$$\begin{aligned} \mathbf{pos}(A) &\stackrel{\text{def}}{=} \text{case } A \text{ of } \langle p, n \rangle_{\text{nnrat} * \text{nnrat}} \Rightarrow p \\ \mathbf{neg}(A) &\stackrel{\text{def}}{=} \text{case } A \text{ of } \langle p, n \rangle_{\text{nnrat} * \text{nnrat}} \Rightarrow n \end{aligned}$$

and both have the type $\text{rat} \Rightarrow \text{nnrat}$. With them, we can define

Definition 7.2.3 (Operations on Rationals)

$$\begin{aligned} A \times_{\text{rat}} B &\stackrel{\text{def}}{=} \approx_{\text{rat}} (\langle \mathbf{pos}(A) \times_{\text{nnrat}} \mathbf{pos}(B) +_{\text{nnrat}} \mathbf{neg}(A) \times_{\text{nnrat}} \mathbf{neg}(B), \\ &\quad \mathbf{pos}(A) \times_{\text{nnrat}} \mathbf{neg}(B) +_{\text{nnrat}} \mathbf{neg}(A) \times_{\text{nnrat}} \mathbf{pos}(B) \rangle) \\ A +_{\text{rat}} B &\stackrel{\text{def}}{=} \approx_{\text{rat}} (\langle \mathbf{pos}(A) +_{\text{nnrat}} \mathbf{pos}(B), \mathbf{neg}(A) +_{\text{nnrat}} \mathbf{neg}(B) \rangle) \\ (A)_{\text{rat}}^{-1} &\stackrel{\text{def}}{=} \approx_{\text{rat}} \left(\text{if } \mathbf{neg}(A) \leq_{\text{nnrat}} \mathbf{pos}(A) \left\{ \begin{array}{l} \langle (\mathbf{pos}(A) -_{\text{nnrat}} \mathbf{neg}(A))_{\text{nnrat}}^{-1}, 0_{\text{nnrat}} \rangle \\ \langle 0_{\text{nnrat}}, (\mathbf{neg}(A) -_{\text{nnrat}} \mathbf{pos}(A))_{\text{nnrat}}^{-1} \rangle \end{array} \right. \right) \\ -_{\text{rat}} A &\stackrel{\text{def}}{=} \approx_{\text{nnrat}} (\langle \mathbf{neg}(A), \mathbf{pos}(A) \rangle) \\ A \leq_{\text{rat}} B &\stackrel{\text{def}}{=} \mathbf{pos}(A) +_{\text{nnrat}} \mathbf{neg}(B) =_{\text{nnrat}} \mathbf{neg}(A) +_{\text{nnrat}} \mathbf{pos}(B) \\ A <_{\text{rat}} B &\stackrel{\text{def}}{=} A \leq_{\text{rat}} B \wedge A \neq_{\text{rat}} B \end{aligned}$$

Again, like the non-negative rationals, we can prove now prove that rat is an equality type, as well as the fact that the rational numbers constitute a linearly ordered field.

7.3 Reals

With the rationals formalized, we are ready to model the real numbers. The choice is traditionally between

1. Dedekind cuts, where real numbers are represented as a pair of sets: the set of all rationals less than the real number, and the set of rationals greater or equal to it.
2. Cauchy sequences, where real numbers are expressed as a limit of a converging sequence of rationals.

While Dedekind cuts are arguably more elegant, they are also inherently static: There is no direct way of identifying a Dedekind cut with a recursive procedure. Cauchy sequences, on the other hand, are readily seen as functions approximating a certain real number.

We therefore choose to represent a real numbers by rational Cauchy sequences. These will, in turn, be represented by functions from naturals to

rational numbers, that is, a real number will be a map that given a natural number n returns the n th rational in the corresponding Cauchy sequence. Defining a predicate expressing that a map A represents a Cauchy sequence is standard. Of several equivalent formulations, we choose

Definition 7.3.1 (Cauchy Sequence)

$$\begin{aligned} \mathbf{Cauchy}(R) &\stackrel{def}{=} \forall_{\text{rat}} e. 0_{\text{rat}} <_{\text{rat}} e \rightarrow \\ &\quad \exists_{\text{nat}} n. \forall_{\text{nat}} pq. n <_{\text{nat}} p \wedge n <_{\text{nat}} q \rightarrow \\ &\quad |R'(\approx_{\text{nat}}(p)) -_{\text{rat}} R'(\approx_{\text{nat}}(q))| <_{\text{rat}} e \end{aligned}$$

(The normalizations of the arguments of R are to ease later proofs.)

We have $\mathbf{Cauchy}(A)$ is undefined if there is a natural number B such that $A'B$ is not a rational. If $A'B$ is rational for natural B then $\mathbf{Cauchy}(A)$ will be true if, relaxing the notation quite a bit, the sequence $(A'B)_{B \in \mathbb{N}}$ is a Cauchy sequence, and false otherwise. We therefore define

Definition 7.3.2

$$R \text{ is real} \stackrel{def}{=} \approx_{\text{true}}(\mathbf{Cauchy}(R))$$

to get a type that includes exactly the rational Cauchy sequences as representatives.

As for normalization of reals, it is not enough to define $\approx_{\text{real}}(R)$ as

$$R \text{ is real} \mapsto \lambda x. \approx_{\text{rat}}(R'(\approx_{\text{nat}}(x)))$$

Though while this would make $\approx_{\text{real}}(\cdot)$ strict and idempotent, it does not identify the sequences

$$x_n = \frac{1}{n}$$

and

$$y_n = \frac{2}{n}$$

though they both tend to 0 as n tends to infinity. In general, choosing a strong representative for the Cauchy sequence $(x_n)_{n \in \mathbb{N}}$ is not easy. If the limit is rational, then one could argue for the constant Cauchy sequence with this limit being the “best” representative. But if the limit is irrational, no objective criterion for choosing one sequence over the other presents itself—at least not one that would allow for singling out one specific sequence as *the* sequence.

We shall instead have the choice operator make the choice for us. First, though, we will define equality between reals:

Definition 7.3.3 (Equality of Real Numbers)

$$\begin{aligned}
A =_{\text{real}} B &\stackrel{\text{def}}{=} A \text{ is real} \mapsto B \text{ is real} \mapsto \\
&\forall_{\text{rat}} e. 0_{\text{rat}} <_{\text{rat}} e \rightarrow \\
&\exists_{\text{nat}} n. \forall_{\text{nat}} p. n <_{\text{nat}} p \rightarrow \\
&|A^{\iota}(\approx_{\text{nat}}(p)) -_{\text{rat}} B^{\iota}(\approx_{\text{nat}}(p))| <_{\text{rat}} e
\end{aligned}$$

The definition should be readily recognizable (bar the normalizations), and from it we can prove $=_{\text{real}}$ is an equivalence relation on the reals.

We will now define $\approx_{\text{real}}(A)$ as essentially

$$\varepsilon x. A =_{\text{real}} x$$

However, since not all classical maps represent rationals, not all classical maps represent reals, and so the choice above will be unspecified. We want to leave out the classical maps that have non-rational maps in their image of the classical maps. Using $\mathbf{CL}_{\text{rat}}(\cdot)$ we can define $\mathbf{CL}_{\text{real}}(\cdot)$ by

$$\mathbf{CL}_{\text{real}}(R) \stackrel{\text{def}}{=} (\forall_{\text{nat}} x. \mathbf{CL}_{\text{rat}}(R^{\iota}(\approx_{\text{nat}}(x)))) \tilde{\wedge} \mathbf{Cauchy}(R)$$

Since $\mathbf{CL}_{\text{rat}}(\cdot)$ is defined for all classical maps, then if R is classical, $\mathbf{CL}_{\text{real}}(R)$ will be boolean. It is true exactly when R is a rational Cauchy sequence. We can now define normalization of reals by

Definition 7.3.4 (Normalization of Reals)

$$\approx_{\text{real}}(R) \stackrel{\text{def}}{=} R \text{ is real} \mapsto \varepsilon x. \mathbf{CL}_{\text{real}}(x) \tilde{\wedge} R =_{\text{real}} x$$

(Of course, once we have proved real is a classical type, we can prove $\approx_{\text{real}}(R) = R \text{ is real} \mapsto \varepsilon_{\text{real}} x. A =_{\text{real}} x.$)

Lemma 7.3.5 *If R is real, then $\exists x. R =_{\text{real}} x$.*

Proof. From R being real, we can prove $\lambda x. \approx_{\text{rat}}(R^{\iota}(\approx_{\text{nat}}(x)))$ is classical. Since $R =_{\text{real}}(\lambda x. \approx_{\text{rat}}(R^{\iota}(\approx_{\text{nat}}(x))))$ is clear, the conclusion follows. \square

As a corollary, we get that if R is real, then $R =_{\text{real}}(\approx_{\text{real}}(R))$. By strictness of $=_{\text{real}}$ this implies $\approx_{\text{real}}(R)$ is real.

We can now prove

Lemma 7.3.6 *real is a simple type.*

Proof. $\approx_{\text{real}}(\cdot)$ is clearly strict. When proving

$$\approx_{\text{real}}(\approx_{\text{real}}(R)) = \approx_{\text{real}}(R)$$

we can assume R , and hence also $\approx_{\text{real}}(R)$, is real. Expanding the definitions, we are to prove

$$\varepsilon x. \mathbf{CL}_{\text{real}}(x) \tilde{\wedge} (\approx_{\text{real}}(R)) =_{\text{real}} x = \varepsilon x. \mathbf{CL}_{\text{real}}(x) \tilde{\wedge} R =_{\text{real}} x$$

It will suffice to prove $(\approx_{\text{real}}(R)) =_{\text{real}} A = R =_{\text{real}} A$ assuming A is classical and $\mathbf{CL}_{\text{real}}(A)$, where $\mathbf{CL}_{\text{real}}(A)$ further implies A is real. The wanted now follows by transitivity of $=_{\text{real}}$.

Finally, we need to prove R is real $= (\approx_{\text{real}}(R))$ is defined. “ \implies ”: From R real we have $\approx_{\text{real}}(R)$ real, and therefore also defined. “ \impliedby ”: Immediate. \square

We can prove

Lemma 7.3.7 $=_{\text{real}}$ has type $[\text{real}, \text{real}] \Rightarrow \text{bool}$.

and

Lemma 7.3.8 $\mathbf{Cauchy}(\cdot)$ has quasi-type $\text{real} \Rightarrow \text{bool}$.

We also have

Lemma 7.3.9 real is a classical type.

Proof. We have only to prove $\mathbf{CL}_{\text{real}}(\approx_{\text{real}}(R))$ for all real R ; The rest of the properties are either immediate or have been argued. The proof relies on the previous lemma and that $\mathbf{Cauchy}(R)$ implies $\forall_{\text{nat}} x. (R \approx_{\text{nat}}(x) \text{ is rat})$ for all R . \square

Lemma 7.3.10 real is an equality type.

Proof. $=_{\text{real}}$ has the correct type. Totality and reflexivity have already been proved, so we just need to prove reflection: Assuming $A =_{\text{real}} B$, we need to prove $\approx_{\text{real}}(A) = \approx_{\text{real}}(B)$. The proof is similar to the proof of idempotency in Lemma 7.3.6. \square

As examples of how we define real numbers and operations on them, here are the definitions of the constant zero and the addition and multiplication operations:

$$\begin{aligned} 0_{\text{real}} &\stackrel{\text{def}}{=} \approx_{\text{real}}(\lambda n. 0_{\text{rat}}) \\ A +_{\text{real}} B &\stackrel{\text{def}}{=} A \text{ is real} \mapsto B \text{ is real} \mapsto \approx_{\text{real}}(\lambda n. (A' n +_{\text{rat}} B' n)) \\ A \times_{\text{real}} B &\stackrel{\text{def}}{=} A \text{ is real} \mapsto B \text{ is real} \mapsto \approx_{\text{real}}(\lambda n. (A' n \times_{\text{rat}} B' n)) \end{aligned}$$

To make sure that 0_{real} denotes the strong representative of zero, we need to explicitly normalize $\lambda n. 0_{\text{rat}}$, which is clearly a representative of zero.

This concludes our treatment of the real numbers. The proof that the reals just defined are actually “the” real numbers, e.g., that they have the supremum property, should be standard. But we have not proved it, for lack of time.

Chapter 8

Implementation

In this chapter I will describe the implementation work I have done. All results and lemmas mentioned in the previous six chapters have been formalized with the developed system, *except*:

Classical types These were added after the implementation work was frozen.

Real numbers For it to be feasible to implement the real numbers, much more proof assistance is needed than what I have at the moment.

Typed lists Like the classical types, these were added after the implementation work was done.

On the other hand, this means that the following *has* been implemented:

Set theory Using the formalization found in Grue [Gru01] a model of ZFC set theory has been proved correct from first principles.

Elementary number theory At least enough of it to prove non-trivial lemmas on greatest common divisors and co-primality, which is used in the formalization of the rational numbers.

Rational numbers The rational numbers have been implemented, and we have proved them to be a linearly ordered field.

The implementation is rather large (the source code is about 575 kilobytes), and I have therefore not included it in this thesis. At the time of writing, the implementation can, however, be downloaded from my home page, at

<http://www.diku.dk/users/skalberg/phd/>

Details on how to build the system can also be found there.

Since it has been done in Isabelle, I will first give a (very) brief introduction to Isabelle, and then show part of the initial definitions in my implementation. I then describe how the generic Isabelle tools have been instantiated, or in some cases modified, to help in the construction of proofs in map theory. Finally, I will give an assessment of the implemented system.

8.1 Isabelle

Isabelle is a generic theorem prover, that is, it has been designed as a logical framework, in which a host of different logics can be embedded. Since some traits are common to several logics, some proof techniques and principles are usable, with little or no modification, by several logics. Isabelle has a number of such tools already implemented, and new logic implementations can utilize these tools if their logic has the properties needed. For example, Isabelle has a generic rewriter which any logic, with a concept of equality and general observational equivalence, can use.

When defining a new logic in Isabelle, one most first choose a parent logic, to which one adds new types, constants, and axioms. The primitive logic in Isabelle, in Isabelle parlance: the Pure logic, is quite simply the implicative fragment of intuitionistic higher order logic. For ordinary use, this firstly means that all terms are typed. Secondly, terms are basically higher order: they can be functions. And, thirdly, we have universal quantification (\bigwedge), implication (\implies), and equality (\equiv).

As an example, the axiom of symmetry reads

$$\bigwedge A B. A \equiv B \implies B \equiv A$$

where A and B are of some type α , and \equiv is of type $[\alpha, \alpha] \Rightarrow \mathbf{prop}$, where \mathbf{prop} is the type of propositions. Since all terms are implicitly quantified over, the rule above will normally just be written

$$A \equiv B \implies B \equiv A$$

Note that $A \equiv B$, on the outset, does not mean that A and B are equivalent. It means that they are equal, indistinguishable. If we have proved $A \equiv B$ (or defined them so), then we can replace A by B in any theorem, anywhere. Obtaining an equality $A \equiv B$ can only be done in three ways:

Definition This is the means by which we can extend a logic.

Rewriting Repeatedly substituting equals for equals.

Equivalence If A and B are propositions, i.e., if they are of type \mathbf{prop} , then we may infer $A \equiv B$ from $A \implies B$ and $B \implies A$.

In the Pure logic, term application is denoted by juxtaposition, i.e., the term

$$P A B$$

is the term P applied to A , and then to the term B . If A is of type α and B is of type β , then P must be of type $\alpha \Rightarrow \beta \Rightarrow \gamma$, for some (possibly higher order) type γ . Whereas we have, in the rest of the thesis, made a distinction between maps and contexts, in higher order logic a context is just a higher order term.

8.1.1 An Example

The Pure logic essentially only allows one to reason about syntax, and is the underlying logic for all logics implemented in Isabelle. First Order Logic, for example, can be “implemented” in Isabelle by stipulating the existence of types `D` and `o` (for the domain and booleans), and the constants

$$\begin{aligned} \text{Trueprop} &:: \text{o} \Rightarrow \text{prop} \\ \text{false} &:: \text{o} \\ \neg &:: \text{o} \Rightarrow \text{o} \\ \wedge &:: [\text{o}, \text{o}] \Rightarrow \text{o} \\ \forall &:: (\text{D} \Rightarrow \text{o}) \Rightarrow \text{o} \end{aligned}$$

together with (among other) the axioms

$$\begin{aligned} \llbracket A; B \rrbracket &\Longrightarrow A \wedge B \\ A \wedge B &\Longrightarrow A \\ A \wedge B &\Longrightarrow B \\ (\bigwedge x.P x) &\Longrightarrow \forall x.P x \\ \forall x.P x &\Longrightarrow P A \end{aligned}$$

The constant `Trueprop` is a coercion function from `o` to `prop`, and will normally be used implicitly whenever a term of type `o` is found when a proposition was expected. As for example in the first axiom: With explicit mention of the `Trueprop` constant, this reads

$$\llbracket \text{Trueprop } A; \text{Trueprop } B \rrbracket \Longrightarrow \text{Trueprop } (A \wedge B)$$

With full quantifiers, too, we get

$$\bigwedge A B. \llbracket \text{Trueprop } A; \text{Trueprop } B \rrbracket \Longrightarrow \text{Trueprop } (A \wedge B)$$

Instead of a coercion function we could have used `prop` instead of introducing the `o` type. However, this would collapse the meta-level (the Pure logic) and the object-level (First Order Logic), and we could have expressions like $\neg(A \equiv B)$, i.e., we could negate meta-level equalities. Introducing the `o` type keeps the distinction.

The quantifier rules are a nice example of the convenience of higher order syntax:

$$(\bigwedge x.P x) \Longrightarrow \forall x.P x$$

First, we have used the usual notation $\forall x.P x$, but it is in fact the term $\forall(\lambda x.P x)$ we have written, where the λ -abstraction is that of the Pure logic.

The axiom says that if we can prove PA for all terms A of type D , then we can conclude $\forall x.P x$ (rightly, we conclude `Trueprop` $(\forall x.P x)$). And this holds (implicitly) for all terms P of the type $D \Rightarrow o$. There is no need for any side condition, expressing that x is not free in the assumptions.

We hope that this very brief example has given enough intuition to understand the overall concepts of Isabelle. If not, the introduction to Isabelle that is provided with the system [Pau98a] is very readable.

8.2 Initial Development

We have implemented map theory as an extension of the Pure logic of Isabelle. Since map theory is mono-typed, we only need to add a single type for the terms of map theory, which we shall call `map`.

```
typedecl map
```

Next, we need to setup equality. This is done in three steps: First we need a new type, `equality`, since this is a new syntactic group.

```
typedecl equality
```

Then we tell Isabelle that equalities can be thought of as propositions.

```
judgment
```

```
  Trueprop :: "equality => prop" ("(_) " 5)
```

And lastly, we introduce the equality symbol.

```
consts
```

```
  "op ="      :: "[map, map] => equality" ("(_ =/ _)" [10,10] 5)
```

This declares `=` as an operator of type `[map, map] => equality`. The blob at the end of the line, between the parentheses, is printing/parsing information, and not relevant at this point.

We now introduce the core syntax of map theory. These constants will have no definitions: Their behavior will be dictated by the axioms only.

```
consts
```

```
  Nil      :: map                ("N")
  Lambda  :: "(map => map) => map" (binder "LAM " 10)
  Apply   :: "[map, map] => map" (infixl "" 200)
  IfNil   :: "[map, map, map] => map" ("IfNil _ _ _" [10,10,10] 10)
  Eps     :: "(map => map) => map" (binder "SOME " 10)
  PureEx  :: "(map => map) => map" (binder "PEX " 10)
```

These should be self-explanatory, except that for the pure existence operator (`PureEx` above), which is used in the definition of classicality. Isabelle uses the syntax $\lambda x.Ax$ for constructing higher order terms, so we must use `LAM x.Ax` for our purposes. Finally, the syntax for the `ifnil` constructor is `IfNil A B C`.

Next, we define \perp . We declare a constant, `Bot`, and then define it. (`==` is the ASCII version of \equiv , just as `!!` and `==>` are those of \bigwedge and \implies , respectively.)

```
consts
  Bot :: map

defs
  bot_def: "Bot == (LAM x. x ' x) ' (LAM x. x ' x)"
```

Adding the axioms to the system is easy as pie:

```
axioms
  ifnil_nil      : "IfNil N B C = B"
  ifnil_lambda  : "IfNil (Lambda A) B C = C"
  ifnil_bot     : "IfNil Bot B C = Bot"
  trans'        : "A = B ==> A = C ==> B = C"
  apply_nil     : "N ' B = N"
  apply_lambda  : "(Lambda A) ' B = A B"
  apply_bot     : "Bot ' B = Bot"
```

The `ifnil_lambda` axiom might be expected to look like

```
ifnil_lambda: "IfNil (LAM x. A x) B C = C"
```

but this is too restrictive: Syntactically, `LAM x. A x` is short for `Lambda (%x. A x)` (where `%` is the ASCII version of λ), but the term `LAM y. R x y` is equivalent to `Lambda (R x)` by η -contraction.

The axiom expressing Quantum Non Datur is expressed as

```
axioms
  qnd: "[| S N ; S (LAM y. x ' y) ; S Bot |] ==> S x"
```

Again, one might have expected a different version:

```
qnd: "[| A N = B N
        ; A (LAM y. x ' y) = B (LAM y. x ' y)
        ; A Bot = B Bot
        |] ==> A x = B x"
```

Again, the reason is the syntactic nature of terms: If we use the second formulation, only equalities that are syntactically on the form $A = B$ will be matched by the conclusion of the last formulation. But we have equalities

that are not of that form, for example $A \preceq B$ which is short for the equality $A = A \downarrow B$.

The next set of declarations allow us to express the left and right hand side of equalities, even when they do not include an explicit equality sign. The soundness of the introduced axioms follows from the fact that all expressions of type `equality` is, when all definitions are unfolded, on the form $A = B$.

```
consts
  LHS :: "equality => map"
  RHS :: "equality => map"

axioms
  eq_lhs: "LHS (A = B) == A"
  eq_rhs: "RHS (A = B) == B"
  eq_lr : "LHS S = RHS S == S"
```

Besides \preceq , non-monotonic implication is another construct, besides equality itself, that is of type `equality`:

```
consts
  "op |>" :: "map => equality => equality" ("_ |>_" [10,0] 5)

defs
  nmimp_def: "A |> S == if A then LHS S else Bot =
              if A then RHS S else Bot"
```

The deduction and substitution theorems are not provable within `map` theory. Consequently we must include them as axioms, if we are to use them in our system.

```
axioms
  nmimpI: "(A ==> S) ==> (A |> S)"
  eq_reflection: "A = B ==> A == B"
```

Finally, the convention of just writing A for the equality $A = \text{True}$ needs to be introduced. Analogous to `Trueprop` we introduce `Truemap` as a coercion function from `map` to `equality`.

```
consts
  Truemap :: "map => equality"

defs
  truemap_def: "Truemap A == A = True"

syntax (" " output)
  Truemap :: "map => equality" ("(_)" )
```

The introduction of `Truemap` is unfortunately not a painless one: Because of the inner workings of Isabelle, while we can do our own parsing of terms in order to do fancy parsing/printing operations, we do not have access to the types of the terms being parsed. Consequently, we have been forced to add the coercion at type checking time, which is in the kernel of Isabelle.

8.3 Tools

Should some of the following be complete gibberish, it will hopefully be somewhat clearer after reading the relevant parts of the Isabelle documentation [Pau98b].

8.3.1 Formalizing the Types

The perhaps central concept of the preceding chapters has been that of a type: simple, classical, and equality. When looking at, for example, equality on typed pairs, we see that we need to be able to pass types around within the logic. Since the only way of doing this is by having types be represented as terms, this is what I have done. At the time of implementation, the classical type concept was not fully developed, and so I have only formalized the simple and equality types. I have utilized the equality

A is $\alpha = \approx_\alpha(A)$ is defined

for compactness, and hence the type α is represented by the term

$\langle \lambda x. \approx_\alpha(x), \lambda xy. x =_\alpha y \rangle$

In the Pure logic, it is not possible to define a proposition capturing conjunction, and so we cannot write the requirements for even a simple type as a single proposition. Consequently, we must axiomatize the properties. Concretely, we use the following constants when dealing with types:

```
consts
  Norm    :: "[map,map] => map" ("norm[_] _" [50,50] 50)
  Is      :: "[map,map] => map" ("_ is _" [60,60] 60)
  Eq      :: "[map,map,map] => map" ("_ =[_] _" [50,10,50] 50)
  STYPE   :: "map => equality"
  EQTYPE  :: "map => equality"
```

The three first constants are, of course, the normalization, type membership, and equality functions for types. The expression `STYPE A` is intended to be true when A represents a simple type. For technical reasons,¹ we need the

¹To be able to use conditional rewrite rules where expressions like `STYPE A` occur in the premises, these need to be equalities.

return type of `STYPE` to be `equality`. The function `EQTYPE` analogously captures the equality types.

The properties `STYPE` must have are captured in the following axioms

```
axioms stypeI [intro]: "[|
  !!x. norm[t] (norm[t] x) = norm[t] x ;
  norm[t] Bot = Bot
  |] ==> STYPE t"

axioms
  norm_idem [simp]: "STYPE t ==> norm[t] (norm[t] x) = norm[t] x"
  norm_strict [simp]: " STYPE t ==> norm[t] Bot = Bot"
```

The first is for introducing the expression `STYPE A`, and the last two are for taking them apart again.

Equality types are formalized in a similar way:

```
axioms eqtypeI [intro]: "[|
  STYPE t ;
  !!x y. [| x is t ; y is t |] ==> (x =[t] y) is bool ;
  !!x. x is t ==> x =[t] x ;
  !!x y. x =[t] y ==> norm[t] x = norm[t] y |] ==> EQTYPE t"

axioms
  eqtype_imp_stype: "EQTYPE t ==> STYPE t"

  eqtypeTI: "EQTYPE t ==> [| x is t ; y is t |]
    ==> (x =[t] y) is bool"

  eqtype_refl: "[| x is t ; EQTYPE t |] ==> x =[t] x"

  type_eq_to_obj_eq: "[| x =[t] y ; EQTYPE t |]
    ==> norm[t] x = norm[t] y"
```

The normalization function, as well as the type membership function, are defined in a straightforward manner:

```
defs
  norm_def: "norm[t] x == (t ' N) ' x"
  is_def: "x is t == IfNil norm[t] x True True"
```

The equality predicate is a bit more complicated: We have explicitly added normalization of its arguments, as well ensured that it is strict. The explanation will follow in the next section.

```
defs
  eq_def: "x =[t] y == x is t |-> y is t |->
    norm[bool] ((t ' (LAM x y. N)) ' (norm[t] x) ' (norm[t] y))"
```

8.3.2 The Simplifier

The main proof tool of Isabelle is its rewriting system, called “the simplifier” in Isabelle parlance. It works by rewriting term using (possibly conditional) meta-level rewrite rules, i.e., rules with conclusions on the form $A \equiv B$. Since one would normally like to use (object-level) equalities of the form $A = B$ as rewrite rules, the simplifier will, provided with a rule

$$A = B \implies A \equiv B$$

deduce meta-level rewrite rules from object-level rules.

Since this is just the substitution lemma (axiom `eq_reflection` above), we can easily configure the simplifier to get some work done.

We do have one problem: In the context of types, we might well have rewrite rules on the form

$$\approx_\alpha(A) = \approx_\alpha(B)$$

but we want them to rewrite the term $\mathcal{F}(A)$ to $\mathcal{F}(B)$ when F has type $\alpha \Rightarrow \beta$. As a concrete example, we would like to rewrite $\neg A$ to `true` when we have proved $\approx_{\text{bool}}(A) = \text{False}$. However, the simplifier can, a priori, only use it to replace sub term of the form $\approx_{\text{bool}}(A)$ with the term `False`, and so will find no rewriting possibilities in $\neg A$.

A first attempt is to prove

$$\approx_{\text{bool}}(A) = A' \implies \neg A = \neg A' \tag{8.1}$$

But this may, by using reflexivity on the premise, lead to an infinite rewriting series:

$$\neg A = \neg(\approx_{\text{bool}}(A)) = \neg(\approx_{\text{bool}}(\approx_{\text{bool}}(A))) = \dots$$

The next attempt is to use congruence rules: These are special rules that allow the user of the simplifier to guide its rewriting algorithm. A couple of examples might be the best way to illustrate the concept.

Congruence Rules

Normally, unless told otherwise, the simplifier will try to rewrite any sub term of the concrete term it is supplied. It rewrites terms top-down, so first it will try to find a rewrite rule where the left hand side matches the entire term. If this fails, it will try each of the immediate sub terms, then their sub terms, and so on till it has tried everything.

The congruence rule

$$P = P' \implies \text{if } P \text{ then } R \text{ else } S = \text{if } P' \text{ then } R \text{ else } S.$$

tells the simplifier to not rewrite the branches of conditionals.

The congruence rule

$$\llbracket P = P'; P' \Longrightarrow R = R'; \neg P' \Longrightarrow S = S' \rrbracket \Longrightarrow \mathbf{if} P \mathbf{then} R \mathbf{else} S = \mathbf{if} P' \mathbf{then} R' \mathbf{else} S'.$$

tells the simplifier to first rewrite the term P , calling the result P' . Then, assuming P' is true, it may rewrite R , calling the result R' , and finally, assuming $\neg P'$ is true, it may rewrite S to S' . (This rule is, of course, the strong congruence rule of **if**.) The neat thing about adding congruence rules to the simplifier is that the simplifier checks if they actually rewrite anything. This means that using Equation 8.1 as a congruence rule can not lead to infinite rewriting. But it will rewrite $\neg A$ to $\neg(\approx_{\text{bool}}(A))$, in the absence of any rewrite rules matching the A in question.

This is because it “rewrites” $\approx_{\text{bool}}(A)$ to itself, so the A' in the congruence rule is instantiated as $\approx_{\text{bool}}(A)$ and the rewrite rule $\neg A = \neg \approx_{\text{bool}}(A)$ is applied. When rewriting the new term $\neg \approx_{\text{bool}}(A)$, the A in the congruence rule is instantiated to $\approx_{\text{bool}}(A)$, and so the congruence rule tries to rewrite $\approx_{\text{bool}}(\approx_{\text{bool}}(A))$. Using idempotency, the result is $\approx_{\text{bool}}(A)$, and the rewrite rule induced by the congruence rule becomes $\approx_{\text{bool}}(A) = \approx_{\text{bool}}(A)$. The simplifier sees that this will have no effect, and discards the rule.

The final attempt, and the one I’ve used in my implementation, is the congruence rule

$$\approx_{\text{bool}}(A) = \approx_{\text{bool}}(A') \Longrightarrow \neg A = \neg A' \quad (8.2)$$

However, this does not immediately work. To understand why, one must first know a bit more about the way the simplifier actually uses congruence rules: Given a congruence rule like the one in Equation 8.3.2, it starts by instantiating the terms P , Q , and R with the terms actually found in the term it is trying to simplify. It then first simplifies $P = P'$. Since P' is not yet instantiated, no simplification can occur on it, and the result of this simplification will be some equality $P_r = P'$, where P_r is the result of simplifying P . Now the subgoal is called in to solve this equation. The standard one uses reflexivity, and thus P' is instantiated to P_r . Now, the simplifier tries to simplify the premise $P' \Longrightarrow S = S'$ (where only the term S' is uninstantiated), and the process repeats itself.

The problem with our new congruence rule is this: If the A in $\neg A$ is already a strong boolean, the simplifier will rewrite $\approx_{\text{bool}}(A)$ to just A , and the subgoal will now try to solve the equation $A = \approx_{\text{bool}}(A')$ by reflexivity. This fails since the constant \approx_{bool} is only found on the right hand side of the equation.

The subgoal used in the implementation of map theory studies the premises of the congruence rules before they are handed to the simplifier. If the simplifier removes an explicit normalization in front of the left hand side, the subgoal will explicitly add one back, avoiding the problem explained above.

Regarding the simplifier, for each function \mathcal{F} of (quasi-)type $[\alpha_1, \dots, \alpha_n] \Rightarrow \beta$, we add the congruence rule

$$\llbracket \approx_{\alpha_1}(A_1) = \approx_{\alpha_1}(A'_1); \dots; \approx_{\alpha_n}(A_n) = \approx_{\alpha_n}(A'_n) \rrbracket \Longrightarrow \mathcal{F}(A_1, \dots, A_n) = \mathcal{F}(A'_1, \dots, A'_n)$$

as well as the rewrite rule

$$\approx_{\beta}(\mathcal{F}(A_1, \dots, A_n)) = \mathcal{F}(A_1, \dots, A_n)$$

(Strictness is taken care of by the classical reasoner, as explained in the next section.) This works for all (quasi-)typed functions, but the equality functions for types need special care.

If we changed the formalization of equality types, so that one must prove that

$$\llbracket \approx_{\alpha}(A) = \approx_{\alpha}(A'); \approx_{\alpha}(B) = \approx_{\alpha}(B') \rrbracket \Longrightarrow (A =_{\alpha} B) = (A' =_{\alpha} B') \quad (8.3)$$

in order to prove α is an equality type, then we would have the rule

$$\llbracket \text{EQTYPE } \alpha; \approx_{\alpha}(A) = \approx_{\alpha}(A'); \approx_{\alpha}(B) = \approx_{\alpha}(B') \rrbracket \Longrightarrow (A =_{\alpha} B) = (A' =_{\alpha} B')$$

But we can not use this rule as a congruence rule, since the simplifier will only accept rules where the conclusions of all the premises are equalities, as congruence rules. Hence, to have a congruence rule for typed equality, the rule in Equation 8.3 must be valid. By explicitly normalizing the arguments in the definition of `Eq`, this is accomplished.

8.3.3 The Classical Reasoner

Apart from the simplifier, the classical reasoner of Isabelle is a proof tool for searching through the proof space, by encoding the current proof state in the style of the sequent calculus. The sequent

$$A_1 \cdots A_n \vdash B_1 \cdots B_n$$

can be encoded as the proof state

$$\llbracket A_1 \cdots A_n \neg B_1 \cdots \neg B_{n-1} \rrbracket \Longrightarrow B_n$$

and vice versa. To work on a proof state as though it was a sequent, we need to swap a negated premise with the conclusion: The proof state

$$\llbracket A_1 \cdots A_n \neg B_2 \cdots \neg B_n \rrbracket \Longrightarrow B_1$$

is just a valid representation of the sequent above as the other proof state.

The classical reasoner therefore requires that the so-called swap rule be valid:

$$\llbracket \neg P; \neg Q \rrbracket \Longrightarrow P \Longrightarrow Q$$

However, this is not valid in map theory: If Q is undefined, the second premise is vacuously true, and we are left $\neg P \implies \perp$, for all P . The closest we can get is

$$\llbracket \neg P; \neg Q \implies P; Q \text{ is bool} \rrbracket \implies Q$$

However, this suddenly introduces a lot of type constraints into the classical reasoner, and they need to be taken care of. To this end, I have implemented a proof procedure that extracts type information from the premises of a proof state, and tries to use it to prove a type constraint in the conclusion.

For the proof procedure to work, one needs to feed it with type introduction and elimination rules. For example, for conjunction, we have the elimination rule

$$\llbracket A \wedge B \text{ is bool}; \llbracket A \text{ is bool}; B \text{ is bool} \rrbracket \implies R \rrbracket \implies R \quad (8.4)$$

and the introduction rule

$$\llbracket A \text{ is bool}; B \text{ is bool} \rrbracket \implies A \wedge B \text{ is bool} \quad (8.5)$$

Adding these rules, and the equivalents of the rules found in Lemma 3.1.4, allows for the automatic proof of theorems like $A \vee B \text{ is bool} \implies A \wedge B \text{ is bool}$ (and much more complicated cases, I might add).

8.4 Assessment

The implemented system is in need of several improvements. Most notably, it has become clear that it very inconvenient to have the simplifier and the classical reasoner (almost) completely separated: They both need all the type information they can get, so much is at the moment provided twice: Once to the simplifier and once to the classical reasoner. The type rules for conjunction are, for example, added to the classical reasoner in the form Equations 8.4 and 8.5, as well as to the simplifier in the form

$$A \wedge B \text{ is bool} = A \text{ is bool} \wedge B \text{ is bool}$$

Much other information is also supplied twice.

What is needed is a central proof procedure that will prove type constraints for the classical reasoner, and likewise rewrite terms on the form $A \text{ is } \alpha$ for the simplifier. This proof procedure should be part of a larger type package, that allowed for declaration of (at least) primitive recursive definitions. From these, a non-recursive definition should be given, and the usual type rules should be proved automatically.

A somewhat more longterm project is to have the simplifier use type information directly, and not need to be guided by congruence rules. This

is no trivial task, while the workhorse of the simplifier is the so-called meta-simplifier, which is part of the kernel of Isabelle. Thus, a completely new simplifier would have to be written, from scratch, outside the kernel.

On a much minor level, there are several small annoyances with the Isabelle system as it is right now. One is of course the problem of not having type information ready when parsing terms, resulting in the ugly `TrueMap` hack mentioned earlier. Since this would require a complete rewrite of Isabelle, as I'm told, this will probably not happen in the immediate future. And, anyway, there *is* a hack.

A more protrusive problem is that one can not remove syntax once introduced. In the absence of types, this is grossly irritating: It would be very nice to use $+$, \times , and so on, for addition and multiplication on natural numbers in one context, and for reals in another. As it is, we must append the type explicitly, and so write, e.g., `+nat` for addition of natural numbers. As with the type/parsing problem, being able to remove syntax would also require a complete rewrite of Isabelle. Again, it does not seem likely to happen any time soon.

Finally, there are of course a number of minor grievances, as there must be for a single user of a general purpose system.

Chapter 9

Conclusion

The original thesis of this work was, as stated in the introduction

A proof system with map theory as its foundation will allow for shorter and more intuitive proofs than present proof systems.

Whatever we say, we can not say that the thesis has been verified. Apart from the development of set theory, all the concepts that have been formalized have been well within the capabilities of Higher Order Logic and other logics used in formal verification. The implementation in map theory has therefore, once the types have been defined, been almost the same as it would be in any other used in formal verification. Certainly, because our system has nowhere near as much automation as Isabelle/HOL, proofs in our system may seem much more sluggish than the small, neat proofs one often finds when looking through the source code of Isabelle/HOL. However, we can almost syntactically copy the theorems and lemmas of Isabelle/HOL, and with a bit more manual guidance, we get through with essentially the same proof. Our proofs on Euclid's algorithm for finding greatest common divisors, for example, is almost verbatim copied from the Isabelle/HOL implementation.

We can not say that the thesis has been refuted. If anything, it looks like, with the same amount of automation as the other systems, we can have a system that is of at least comparable complexity. But until we have the real numbers, with decision procedures and heuristics to boot, we shall not be able to compete with, say, Higher Order Logic on verification of floating point hardware. An area where map theory might one day really prove its worth.

9.1 Future Work

There are several directions one could take from here. Some are interconnected, others orthogonal to each other.

- First, as mentioned in the last chapter, the actual Isabelle implementation could easily be improved upon: The types of map theory should be integrated into both the simplifier and the classical reasoner. A package for handling recursive definitions would also be nice.
- A better understanding of how to implement recursive, abstract data types in map theory would be valuable. The implementation of basic lists works, as such, but it is unsatisfactory that we need to use extensionality to prove comparably simple equalities. At least a more systematic approach to showing these equalities using extensionality should be attainable.
- The real numbers should be formalized in the system. This would probably demand the porting of Isabelle/HOL's arithmetic decision procedure, or equivalent.
- A bit simpler, perhaps, is defining the type of typed sets, i.e., define a type constructor (α) set, such that, for example, (nat) set is the type of sets of natural numbers.
- Many tricks are used in order to get around non-classical representatives of types. It might be interesting to experiment with defining types completely included in the classical maps.
- A very ambitious project, but all the more rewarding, is to embed Higher Order Logic in map theory. If the type concept of map theory is able, or can be modified in order to be able to model the types of Higher Order Logic, it might—in due time—be possible to port proofs from Higher Order Logic into map theory in a “natural” way. Gaining access to the vast amount of work done in Higher Order Logic would, in any case, be extremely valuable.

Bibliography

- [BG97] Chantal Berline and Klaus Grue. A κ -denotational semantics for map theory in ZFC+SI. *Theoretical Computer Science*, 179(1–2):137–202, June 1997. DIKU Technical Report 96/11.
- [Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346–366, 1932.
- [Chu33] Alonzo Church. A set of postulates for the foundation of logic (second paper). *Annals of Mathematics*, 34:839–864, 1933.
- [Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, June 1940.
- [dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972. Reprinted in [NGdV94], pp. 375–388.
- [Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
- [Gru92] Klaus Grue. Map theory. *Theoretical Computer Science*, 102(1):1–133, August 1992.
- [Gru01] Klaus Grue. Map theory with classical maps. As yet unpublished. Can be found from <http://www.diku.dk/users/grue>, 2001.
- [KR35] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, July 1935.
- [NGdV94] 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.
- [Pau98a] Lawrence C. Paulson. *Introduction to Isabelle*. University of Cambridge, Computer Laboratory, 1998.

- [Pau98b] Lawrence C. Paulson. *The Isabelle Reference Manual*. University of Cambridge, Computer Laboratory, 1998.
- [PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 199–208, 1988.
- [vH67] Jean van Heijenoort. *From Frege to Gödel: A source book in mathematical logic 1879–1931*. Harvard University Press, 1967.