UNIVERSITY OF GRONINGEN

Inductive Types in Constructive Languages

Peter J. de Bruin

March 24, 1995

Abstract

Logic grammar is used to partly define a formal mathematical language "ADAM", that keeps close to informal mathematics and yet is reducible to a foundation of Constructive Type Theory (or Generalized Typed Lambda Calculus). This language is employed in making a study of inductive types and related subjects, as they appear in languages for constructive mathematics and lambda calculi. The naturality property of objects with type parameters is described and employed.

Cover diagram

Behold the mathematical universe,
developing from original unity
into categorical duality.

The central beam contains
the initial and the final type,
together with the remaining flat finite types.

It is flanked by the dual principles
of generalized sum and product,
and of initial and final fixed point construction.

Printed by: Stichting Drukkerij C. Regenboog, Groningen.

RIJKSUNIVERSITEIT GRONINGEN

Inductive Types in Constructive Languages

Proefschrift

ter verkrijging van het doctoraat in de Wiskunde en Natuurwetenschappen aan de Rijksuniversiteit Groningen op gezag van de Rector Magnificus Dr. F. van der Woude in het openbaar te verdedigen op

vrijdag 24 maart 1995 des namiddags te 4.00 uur

door

Peter Johan de Bruin

geboren op 20 september 1964 te Harderwijk

Promotor: Prof. dr. G. R. Renardel de Lavalette

Preface

The fascination for mathematical truth has been the driving force for my research as it had been for my master's thesis written in Nijmegen. Being amazed at the lack of a general language for the irrefutable expression of mathematical argument, I looked for it in the direction of what is called Type Theory. I started my research in Groningen in 1988 under supervision of Roland Backhouse, and enjoyed his discussion club with Paul Chisholm, joyous Grant Malcolm, Albert Thijs, and broadly interested Ed Voermans. When Backhouse moved to Eindhoven in 1990, I and Thijs remained in Groningen and our roads parted.

The advent in 1991 of Gerard Renardel who accepted to take up my supervision with fresh interest, started a new period of seeking to assemble all available pieces. I owe him much for his constant trust and support in keeping up courage, his help in formulating ideas, his effort to curtail outgrowing branches, and for leaving the choice to carry this work through entirely to me. I also thank Jan Terlouw for his modest cooperation, and my fellow Ph.D. students for their good company and friendship.

Now my heart has turned from abstract truth to living Truth, and since 1993 I'm living in community *Agapè* of the Blessed Sacrament Fathers in Amsterdam. I am glad to thank its members, Aad, Eugène, Gerard, Herman, Jan, Paul, Pieter, and Theo, for their support while finishing this thesis. I thank the members of the Ph.D. committee, Roland C. Backhouse (Eindhoven), Wim H. Hesselink (Groningen), Gerard R. Renardel de Lavalette (Groningen), and Michel Sintzoff (Louvain-la-Neuve), for accepting this duty and providing kind comments on the manuscript, especially Wim who gave detailed comments which helped me to prepare the final text. Though it is tough material, my presentation sometimes being terse and not all ideas given sufficient scientific support, I hope you will get a catch of its beauty.

Peter de Bruin

Contents

Summary

This dissertation deals with constructive languages: languages for the formal expression of mathematical constructions. The concept of *construction* does not only encompass computations, as expressed in programming languages, but also propositions and proofs, as expressed in a mathematical logic, and in particular the construction of structured mathematical objects like sequences and trees. *Types* may be conceived of as classes of such objects, and *inductive types* are types whose objects are generated by production rules.

The purpose of this dissertation is twofold. First, I am searching for languages in which the mathematician can express his inspirations well structured, correct, and yet as freely as possible. Secondly, I want to collect the diverging approaches to inductive types within one framework, so that it becomes apparent how the diverse construction and deduction rules arise from a single basic idea and also how these rules may be generalized, if desired. As basic idea I use the concept of *initial algebra* from category theory.

My research into mathematical languages has not led to a complete proposal. The present treatise is confined to general reflections and the partly formal, partly informal description of a language, *ADAM* (chapter 2). This language serves subsequently as a medium for the study of inductive types, which constitutes the main body of the dissertation.

The set-up of *ADAM* is as follows. To guarantee the validity of arguments expressed in the language, it needs a sound foundation. I develop a constructive type theory (called ATT) for this, a combination of the "Intuitionistic Theory of Types" of P. Martin-Löf and the "Calculus of Constructions" of Th. Coquand. In order to comprise all mathematical principles of deduction, I add the iota or description operator of Frege. It is not necessary to include inductive types as a basic principle; natural numbers suffice to construct these.

On this foundation I build the language *ADAM* by looking at how constructions and proofs that I encountered or drafted could be formulated as naturally as possible while adhering to the rules of type theory. The formal definition of *ADAM*, as far as it is available, and its semantics in the underlying type theory are simultaneously given by means of a two-level grammar. This makes it in principle possible to extend the language, while preserving validity, with notations or sublanguages for special applications, like program correctness. The proposed notations should therefore not be regarded as immutable. Perhaps the only typical language element is the notation for (and the consistent use of) *families* of objects.

As a preparation for inductive types, I start with rendering the classical approaches to inductive definitions (chapter 3), followed by the introduction of the machinery which we require—elementary category theory and algebra (chapter 4).

The central part of the treatise consists of the description and justification of inductive types as initial algebras. First, I consider at an abstract level the various ways of specifying inductive types, and how these specifications designate (via a polynomial functor) an algebra signature, possibly with equations (chapter 5). Next, I analyse and generalize the ways of defining recursive functions on an inductive type (chapter 6). Then I investigate to what extent these construction principles can be dualized to co-inductive types, which are final co-algebras (chapter 7). Finally, I construct, using either elementary set theory or type theory, initial algebras and final co-algebras for an arbitrary polynomial functor, which actually proves the relative consistency of all discussed construction principles in relation to *ADAM*'s Type Theory ATT (chapter 8).

The dissertation is concluded with the treatment of some issues related to inductive types. In chapter 9, I consider recursive datatypes with partial objects, as they occur in programming language in which one should reckon with possibly non-terminating program parts. I summarize the required domain theory, and construct such domains in *ADAM* using final co-algebras. In chapter 10, I briefly discuss inductive types in impredicative languages, types as collections of type-free values, and the principle of bar induction, and I suggest the possibility of inductive definition of new type universes within a type theory. Chapter 11 gives a number of further reflections on mathematical language and proof notation, and summarizes the approaches to inductive types.

The appendices contain the basic principles of set theory and of ATT, the required addition of either the iota operation or proof elimination to type theory, and a study of uniformity properties (*naturality*) of polymorphic objects, which I need on certain occasions.

Chapter 1

Introduction

This thesis circles around two themes that closely intertwine: formalized mathematical language, and inductive types. We speak about the former in section 1.1–1.6, resulting in the language description in chapter 2; inductive types are addressed in 1.7–1.10 and fill the major part of the thesis. The remaining sections of this chapter introduce some basic concepts for further use.

1.1 Mathematical language

Mathematical language is a field of interest that is shared by mathematicians and computing scientists. For mathematicians firstly as a medium to express their abstract thinking, secondly as the subject of formal analysis itself.

The computing scientist stands in between these two approaches. For the theoretical analysis of computing, (s)he needs a medium of expression just as the mathematician. But he is also fascinated by the possibility of rendering mathematical treatises accessible for computer manipulation, in order that computers may assist in creating, verifying, and transforming mathematical texts.

Here a yawning gap appears. For the mathematician excels in creative use of his language, inventing new styles of notation, new modes of reasoning. The formalist on the other hand requires a well-defined system of permitted notations and deduction steps. This may include a scheme for introducing some sort of new notation, yet time and again the language user will find good reason to step outside the provided schemes in order to attain more clarity of expression. Especially when the deduction steps themselves have to be noted down, formal calculus becomes too cumbersome for most application fields.

Our primal impetus was to work on diminishing this gap by developing a kind of universal calculus. On the one hand it would contain a sound definition of correct mathematical construction and deduction principles, complete for all practical purposes. On the other hand it should permit the user a freedom of notational definition that restricts him as little as possible.

Needless to say, this is an ideal, floating in the air, that we can hardly expect to realize on earth. Not letting ourselves be dispirited by this, we have tried to grasp the inspirations we received and to mould them into concrete form. Our second theme, inductive types, has served as a playground to gain experience in using our notational ideas. At the same time, our treatment of inductive types contains in abstract form the various forms in which inductive types appear in other languages.

The formal realization of these ideas has often been unruly, and we apologize for the defects in our present work. Some ideas are not worked out in full detail, but there are also ideas whose formalization would require extensive elaboration and reconsideration of the language set-up.

We call the resulting language *ADAM*, as we hope virtually all of mathematics to appear among its offspring, and also in honor of the city of Amsterdam, which name abbreviates to *A'dam*.

1.2 Our approach to mathematical language

To begin with, let us note that our aim is a language that serves as a universal medium of mathematical expression, not a calculus that is directed at specific purposes such as problem solving through formal manipulation of the language expressions themselves. Rather, it should be possible to embed any specific calculus within the language. This aims in particular at *programming logics*, that describe the semantics of particular programming languages.

When offering the user notational freedom and brevity, one cannot avoid that some standard or user-defined notations may overlap. Thus, the possibility of ambiguity is inherent in our approach. Furthermore, the user should have the option to omit some details when he expects these to be obvious or reconstructible by the reader. In either case, it is the responsibility of the user (writer) to keep his text comprehensible.

Yet, the language should have a sound formal foundation, guaranteeing all results to be correct. We were drawn to use *Constructive Type Theory* (or Generalized Typed Lambda Calculus), being attracted by its elegant unified treatment of proofs and objects, and of finite and infinite products and sums. Besides the construction principles that are directly related to the basic type constructors, one needs some additional axioms to obtain a full foundation. Rather than leaving each user to establish his own foundation, we prefer to establish a fixed foundation for common use. For foundational research, one may of course study alternatives.

In shaping the notations of *ADAM*, we looked at the established notations of mathematics, making some adaptations to give them a more regular type structure. A typical example is the notation for set formation.

Example 1.1 Traditionally, one writes

$$\{\, f(x) \mid P(x)\} \tag{1.1}$$

for the set containing $f(x)$ for all x such that condition $P(x)$ holds. This notation does not indicate which of the variables occurring free in $f(x)$ and $P(x)$ are locally bound. A minor objection is that it may be necessary to look ahead to the condition $P(x)$ before one can fully understand expression $f(x)$. In the 'Eindhoven quantifier notation', introduced by E.W. Dijkstra, one writes

$$\{x : P(x) : f(x)\,\}$$

to overcome both problems. When using generalized types, the variable has to be typed; furthermore, typings and conditions are both assumptions to be treated on a par, so we move the condition to the left of the colons, where an arbitrary declaration (sequence of assumptions) may appear. In this case, we get:

$$\{x{:}\, A;\ P(x) :: f(x)\,\}$$

For the special case when $f(x)$ is just x, we introduce a notation similar to (1.1):

$$\{\, x{:}\, A \mid{:}\ P(x)\} \ := \ \{x{:}\, A;\ P(x) :: x\,\}$$

The interesting thing about these notations for sets is that they are suited to be used for logical quantifiers and generalized constructors too. This will be described in section 2.7.

1.2.1 Defining ADAM. To define the basic syntax and simultaneously all correctness requirements of *ADAM*, we use the powerful mechanism of two-level grammar, containing Horn-clause logic, which is described in section 2.1. Ideally, this grammar mechanism should be available within *ADAM* itself, so that the user may introduce new non-conventional notations, special-purpose calculi, or other (programming) languages.

The definition of *ADAM* proceeds in the following stages:

1. Description of the language definition mechanism

2. Definition of the underlying type theory using Horn clauses

3. Definition of the abstract and concrete syntax classes and their production rules, which reduce the meaning of language constructs to type theory

4. Development of a body of useful theory and notations

Actually, our definition is not so systematic. The definition mechanism is not described in full detail, and points 3 and 4 are mingled. Some language features are defined only partially, or described merely by a suggestive example, as their full formalization would go beyond the scope of this thesis.

1.2.2 Our use of ADAM. The main body of this thesis uses both *ADAM* and informal proof notation, but it can be thought of as encoded wholly in *ADAM*. This stands in contrast with appendix D, where typed lambda calculus is used as the object of study itself, rather than as a medium of expression.

We use *ADAM* mainly to formulate principles of inductive types, to justify them and establish relationships between them. As such, *ADAM* both provides a unifying framework in which principles from many different languages can be represented, and gives a sound foundation to these principles.

1.2.3 Semantics. The semantics of *ADAM* is given by its type theory, named ATT. The rules of ATT may be regarded as a foundation for mathematics, yet for better understanding we outline an interpretation in extended set theory in section B.10. We have not studied more “mathematical” models like PER models (based on partial equivalence relations on a simple set) or categorical models [44], but we remark that finding such models may be very difficult, because of the sheer strength of ATT, transcending ZFC set theory.

1.3 Type Theory and Set Theory

The foundation of mathematics is usually sought in axiomatic set theory: all mathematical objects are assumed to exist within a single universe of sets, where each set consists of other sets.

In a formalized language, it is convenient to have all objects classified into types, in order to avoid anomalies. In such a typed language, one cannot speak about an object without specifying its type, and one may only apply operations to objects of appropriate type. For example, it does not make sense to compare two objects of different types.

A *simple* type system consists of a number of primitive types together with a number of finitary type constructors. The class of simple type expressions can be algebraically defined prior to the further language definition. A typical language is Typed Lambda Calculus, with type constructors like function space, finite cartesian product, and disjoint sum, and possibly inductive or user-defined types. It has term rewrite (or reduction) rules operating on lambda terms, not on type expressions.

A *generalized* type system contains infinitary type constructors as well. As any concrete type expression is necessarily finite, type expressions have to be parameterized with object variables. The typical type constructor is the generalized product $\prod_{x:A} B_x$ for a type A and a type B_x for any object x of type A. The product is written as $\Pi(x\colon A :: B_x)$ in this thesis. Its inhabitants are tuples that contain, for any object $x\colon A$, an object $b_x\colon B_x$. Such a tuple is written as $(x :: b_x)$ in this thesis.

Due to the generalization, object expressions may appear within type expressions. Thus, both have to be defined simultaneously, and rewriting may affect type expressions too. Type correctness (validity) and equivalence of expressions are usually inductively defined as meta-predicates (also called *judgements*) on contexts, object and type expressions. The resulting system, consisting of expressions and judgements defined by derivation rules, is called a *type theory*.

We are interested in *constructive* type theories (CTT's), which are generalized type theories based on lambda calculus in such a way that any valid expression of some type provides a mathematical construction for that type. The typical example is a disjoint sum type $B_0 + B_1$; a closed expression of this type must reduce to a canonical form containing an expression either of type B_0 or type B_1. One can even have empty types, for which there are no closed expressions.

An interesting point is that constructive type theory immediately accommodates predicate calculus. When we identify any proposition with a type that contains all proofs of that proposition, all propositional connectives and quantifiers coincide with standard type constructors. Notably, the universal quantifier proposition $\forall x\colon A.P_x$ becomes the generalized product $\Pi(x\colon A :: P_x)$. The proposition is true exactly when its proof type has an inhabitant, and any valid object expression of this type provides a proof of the proposition. This is called the *propositions-as-types* principle.

If one needs higher-order quantification, i.e., propositions that quantify over all subsets of a type, one has to make a formal distinction between propositions and data types. In fact, one assumes propositions to be given *a priori*, before the hierarchy of types has been generated. This is called *impredicativity*.

To justify constructive type theory, one may seek for a set-theoretic model. However, the basic rules of CTT are so fundamental, that one may just as well consider it to constitute an alternative foundation of mathematics, which replaces axiomatic set theory. We outline two models of set theory within extended type theory in section A.5 and A.6, and a model of type theory within an extended set theory in B.10.

1.4 Related efforts

Several mathematical languages based on type theory have been developed. Note that we do not regard a bare logical derivation system, like first or higher order logic, as a mathematical language, because it provides no notation for proofs other than as a sequence of statements.

N.G. de Bruijn developed *Automath* (in many variants) [11] exactly to provide an automatically verifiable notation for definitions and proofs in any logical calculus. It has only one principle of type construction (namely generalized product), which suffices for allowing the user to axiomatically introduce any type, object, or proof constructor he would need as a so-called “primitive notion”. The necessity to write down all parameters of each constructor made Automath rather unwieldy to use. The *Mathematical Vernacular* (MV) [12] was introduced to overcome this: it allowed more syntactic freedom and the possibility to omit parts of a construction. This inhibits automatic verification.

P. Martin-Löf formulated his *Intuitionistic Type Theory* (ITT) [56] in order to explicate the basic principles of intuitionistic reasoning. Its basic type structure is very much like Automath, but it includes a number of construction principles (including inductive types) that provide a sufficient logical foundation for many purposes. It does not have impredicative propositions, as this is contrary to intuitionistic philosophy. The way ITT treats the equality predicate, internalizing it by means of “equality types” (sometimes called “identity types”), generates some anomalies in the type structure. Because of this, and the omission of some type parameters, type correctness (validity) of expressions may itself require a non-trivial proof.

R.L. Constable’s *Nuprl* [18] is an interactive computer implementation of a variant of ITT. It assists the user in finding valid expressions, by following the approach introduced by the automated programming logic *Edinburgh LCF* (Logic of Computable Functions) [35, 70] to employ a functional programming language, the “meta-language” (ML), which is offered to the user who may call on predefined or user-defined “tactics” that perform a goal-directed search for valid expressions. (This special-purpose language ML has developed into the general-purpose language Standard ML, SML.) Nuprl provides a notation for the search process, recursively listing all goals and subgoals.

Th. Coquand's *Calculus of Constructions* (CC) [21] is a type theory based on impredicative quantification and has been implemented in LCF-style, too. Type constructors as used in typed lambda calculus can be defined using impredicative quantification, but no induction rule can be derived inside the calculus. C. Paulin-Mohring extended CC with embedded principles for inductive types [73, 68], which were implemented in the system Coq.

A more direct style of interactive proof editing was designed by Th. Coquand and B. Nordström and implemented in Göteborg as the *ALF* proof editor [50]. Being based on ITT with inductive definitions, it allows direct manipulation through a multiple-window presentation of the current state of the proof object, which may contain "placeholders" for incomplete parts. There are windows for the current theory, the proof under construction, the current list of placeholders (or goals) with their type and context, and equational constraints on the placeholders. It features a nice syntax for recursive definition through pattern matching, described in [23].

M. Sintzoff's calculus *DEVA* [78, 85] comes closest to our goal, because of the much greater attention it pays to the readability of the resulting proof and object expressions. It offers more structuring primitives, such as a kind of labeled records with dependent field types. DEVA distinguishes between explicitly and implicitly valid expressions. The former are verifyably correct proof or object constructions, the latter "amount to developments with missing parts, e.g. incomplete proofs and tactics. They are characterized by the undecidable existence of an explication, which completes the missing parts and yields a valid expression."[1] These explications go further than the missing proofs of De Bruijn's MV, but do not give the full power of ML-tactics.

1.5 Relational calculus

Now and then we use relational notation for easy expression, and sometimes proof, of properties. Such notations are being developed by a group around Backhouse into a calculational method for deriving programs from specifications [1]. There is an essential difference between this use of relations and ours: the relational calculus employs relations to model (possibly nondeterministic) input-output behavior, making much use of relation composition, while we use relations to establish relationships between functions and other objects, using arrow composition but hardly ever relation composition. A study of inductive properties in relational calculus is given in [8].

1.6 ADAM's Type Theory

The language *ADAM* is based on a type theory, ATT, that combines the features of Martin-Löf's ITT and Coquand's CC. Thus, it has a set of basic type constructors combined with impredicative propositions, and almost all typed lambda calculi appear as subsystems. There is, however, one gap that hinders the coding of arbitrary mathematical proofs: given a constructive proof that a predicate has a *unique* solution, one

[1]Weber in [84]

cannot obtain within the calculus a term denoting that solution. To mend this, we add a stronger elimination rule for the existential quantifier, described in appendix C.

We have the following construction principles and axioms:

1. Generalized products (Π). These subsume the function space constructor.
2. Generalized sums (Σ).
3. Finite types ($0, 1, 2, \ldots$). Combined with generalized products and sums they give finite products and sums. Many recursion constructs can already be expressed using only these and function space, but extra equational calculus is needed to express their properties.
4. A hierarchy of universes ($\mathbf{Type}_i$).
5. Impredicative propositions (**Prop**), turning $\mathbf{Type}_i$ into a *topos* [46] and yielding higher order logic.
6. Equality types, i.e., an internal equality predicate ($x =_A y$).
7. Strong existential elimination ($\exists\_\mathsf{elim}$ or the description operator ι). This is our new extension, see appendix C.
8. Infinity (ω). Together with the preceding principles it allows us to construct a representation for all inductive types, as we will see in chapter 8.
9. Axiom of choice, needed because propositions are distinguished from types.
10. Finally, one may also think of adding classical propositions, for which the principle of *reductio ad absurdum* holds.

The exact rules are listed in appendix B.

1.7 Aspects of induction and recursion

In philosophy, induction stands often for the process of discovery of a general statement out of some particular cases: "In all the umpty cases we encountered we found that statement P held, which induces us to suppose that P holds in all cases." This is not what we mean by induction in this thesis.

In mathematics, induction may be understood as the production of an infinite set of things through iterated application of a fixed set of rules. One distinguishes between inductive definition and inductive proof.

Inductive definition signifies the definition of a set as the totality of all objects produced through iterated application of a fixed set of production rules.

Inductive proof signifies proving a general statement by giving proof steps that incrementally produce proofs for all individual instantiations of the statement.

Recursion stands basically for the (re-)occurrence of an object within a description of that very object. Such a description is not necessarily a valid definition of the object: it may be seen as an equation $x = f(x)$ which can have either no, a single, or multiple solutions. Yet under suitable restrictions recursive equations have unique solutions, so that they may be used for definition. Alternatively, one can use a complete partial order so that suitable recursive equations have unique least solutions. In most applications, the object x under definition is itself a function.

The notions of induction and recursion overlap. On the one hand, any inductive type definition can be written as a recursive type equation; this is in fact what happens in most programming languages that allow such types. Inductive proofs can, within a suitable calculus with dependent types, be written as recursive dependent function definitions. On the other hand, a valid recursive function definition can be reduced to an inductive predicate definition (as a kind of set) together with an inductive proof that this predicate constitutes a function.

The phrase recursive is often taken to denote *effectively computable*. The branch of mathematics called (classical) *recursion theory* [65] deals with hierarchies of computable functions on natural numbers, and studies their complexity. Computational complexity falls outside the scope of this thesis.

1.8 Frameworks for studying induction

There are several quite different notions of inductive sets; there are many ways inductive types can be described; and there are many settings, languages, or frameworks in which one may introduce inductive types. A brief survey follows.

1.8.1 Set theory. The foundational justification for the use of induction may be found in set theory (using for example the Zermelo-Fraenkel axioms of powersets, infinity, union etc.). The simplest interpretation of an inductive set definition is that it denotes the intersection of all sets that are closed under the rules of the definition. It follows immediately that proof by induction is valid for this set, but the construction is only welldefined if one has already some domain that is closed under the rules. Otherwise, one can iterate application of the rules to get a possibly transfinite series of sets, and take its limit. Under a certain restriction this limitset is closed under the rules indeed.

1.8.2 Type theory. An alternative foundation for mathematics is provided by Constructive Type Theory in the sense of Martin-Löf [56], which is a generalization of typed lambda calculus. In these systems a principle of inductive (data-)type construction can be included as basic. Extensive use is made of dependent types; in particular the type of the result of a recursive function may usually depend on its argument value. As types can represent propositions, the recursion axiom can be used for inductive proofs as well. Thus, while classically recursion is reduced to induction, type theory reduces induction to recursion.

An advantage of dependent types is that one can easily handle constructors that have an infinite number of arguments, as opposed over the ordinary use of types in, for example, purely finitary algebraic theories.

N.P. Mendler introduced [59] an alternative recursion construct that employed a quantification over types and a subset relation on types. It is not obvious how this construct relates to the ordinary ones. We will use the *naturality property* of polymorphic objects to show that Mendler's construct has in fact the same power as the ordinary (dependent) recursion rule, at least when we replace the subset relation by explicit mappings. (See section 6.3.)

1.8.3 Category theory. Category theory captures inductive types in a particularly simple axiom about *initial algebras*. No recursive functions with dependent types are used nor is there an induction axiom, but there is a uniqueness condition from which these may be derived. Categorical notions and the initial algebra axiom can easily be put in a type-theoretical context. In fact, the general formulation of several alternative recursion axioms, too, is most easily expressed when using categorical notions. The categorical notions do also allow a very simple generalization to *mutual* and *parametrized inductive definitions*, for each construct. There are, however, other parametrized recursion constructs possible that are sometimes easier to use.

1.8.4 Impredicative type theories. Ordinary Constructive Type Theory does not allow to form types by quantification over the class of all types, as such a principle is intuitively not well-founded, and indeed inconsistent with some other constructs of type theory. However, such impredicative quantification can be added to either simple typed lambda calculus, resulting in polymorphic lambda calculus, or to a calculus with dependent types. This is done in the *Calculus of Constructions* of Coquand [21]. It allows the construction of inductive types for which there is a recursion construct, without using extra axioms. Unfortunately one cannot derive an induction rule inside the calculus, although it may be possible to prove outside the calculus, using a generalized naturality theorem, that the induction principle does hold. One might just assume a primitive induction axiom. When *subtypes* are available one is indeed able to construct an inductive type that has an induction rule. (See subsection 10.1.1.)

1.9 Our treatment of induction and recursion

In this thesis, we shall describe inductive types as they appear or might appear in various typed languages. Using *ADAM* all the time, we start with the traditional description of inductive sets by means of well-founded relations. Then we move to the categorical framework, which will be our main tool to bring various induction principles under a common denominator.

Our treatment is separated into construction principles for inductive types, and recursion principles over inductive types. The former, discussed in chapter 5, describe for which forms of algebra signature an initial algebra does exist. The latter, discussed in chapter 6, describe the forms which a total function definition using structural recursion over an inductive type may take. These rules suggest possible language rules for including inductive types in other constructive languages. Any such rule may be taken:

- either in its full generality, if the language includes all primitives that we use in the formulation of the rule,
- or in a more restricted form. For example, instead of a generalized product $\Pi(x\colon A ::$

B_x) one might allow only finite products $B_0 \times B_1$, parametrized types B^A, and combinations of these.

In chapter 8 we construct algebras in *ADAM* that satisfy the given induction and recursion rules. This proves the relative consistency with respect to ATT of these rules. It also proves the relative consistency of other calculi with inductive types that are directly embedded in ATT, like ITT and CC.

1.10 Other kinds of inductive types

Apart from inductive sets or types as mentioned in section 1.7, there are other kinds of inductive type definitions to which we shall give some attention.

1.10.1 Co-induction. In chapter 7, we see how the categorical description of inductive types can be dualized to describe *final coalgebras*, also called *co-inductive datatypes*. These model tree structures that may be infinitely deep, while staying in a pure setting that contains only total functions and totally defined objects. All recursion constructs can be dualized too, provided the usage of dependent functions is removed. One has to add a uniqueness condition in order to preserve completeness. The set interpretation of these objects is not evident, as infinitely deep sets conflict with the foundation axiom of standard set theory, but they may interpreted as graphs (section 8.2).

1.10.2 Domain theory. An entirely distinct notion of recursion is used in programming languages. Here it is often allowed to define types and objects in terms of themselves, without significant restrictions. This may result in partial or infinite objects, where it is effectively undecidable whether some part of an object is defined or not. Domain theories, such as the theory of Complete Partial Orders, were developed to give meaning (semantics) to such recursive definitions.

In chapter 9, we will describe several rules for reasoning about partial objects. We show how partial or infinite objects can be modeled by co-inductive datatypes, and how lazy recursive object definitions, with possibly nonterminating parts, can be interpreted in this model.

1.10.3 Inductive universe formation. The intuitive justification for the set-theoretical axioms is in fact an extraordinary kind of inductive set definition itself: one where a big set is generated such that each generated element is associated with a set itself, and where the production rules may use this associated set. In section 10.3, we suggest that existence of such big sets may be presented as a general principle, and name it *inductive universe formation.*

1.11 Original contributions

The main contribution of this thesis consists first of the presentation of an alternative view on the development of formal mathematical language, secondly of the description and generalization of principles of inductive types in constructive languages, within a coherent framework.

In the course of this work, we presented a number of constructions and proofs. We think the following ones are minor contributions of this thesis:

- The introduction of strong existential elimination into constructive type theory (appendix C)
- The equivalence proof between non-dependent Mendler recursion and initiality (theorem 6.4), using naturality
- The generalized formulation of liberal mutual recursion (paragraph 6.4.3)
- The dualization of algebras with equations (section 7.4)
- The proof that Kerkhoff's initial algebra construction can be dualized (theorem 8.3)
- The construction of recursive cpo's by means of co-induction (paragraph 9.2.3 and section 9.3), and the interpretation of recursive object definitions within this representation (section 9.4)
- The inductive model of Zermelo-Fraenkel set theory in type theory (section A.6)
- The derivation of dinaturality from naturality (section D.6)

Furthermore, we have obtained a few not very remarkable results, for which we yet do not know whether they are known in the literature. These are:

- The relation between monads and initial algebras (theorem 4.6)
- The model of set theory within type theory by means of directed graphs (section A.5)

Chapter 2

The language ADAM

In this chapter we introduce our mathematical language named *ADAM*. It serves to provide clear and precise notations for mathematical constructions based on strong typing, including generalized products and sums.

ADAM is based on a formal system, ATT, that is an extension of Martin-Löf's Intuitionistic Type Theory. This type theory is described in appendix B, but it is not necessary to study it separately if one has some acquaintance with generalized type theories. The primitive notions that are included are generalized products, generalized sums, finite types, a cumulative hierarchy of universes, a universe of propositions, impredicative quantification for propositions, the equality predicate, and existential propositions with strong elimination. Furthermore, there are no inductive types except a type of natural numbers. The system is powerful enough to serve as a mathematical foundation of *ADAM*, for set theory (ZFC) can be encoded in it, using the first universe. Conversely, all primitives can be given a set-theoretical interpretation (section B.10) within ZFC extended with a hierarchy of universes, so we have a set-theoretical semantics for *ADAM* as well.

The theory contains many simpler typed lambda calculi as subsystems, and most of our constructions are valid in some of these too.

While being formal, *ADAM* attempts to come much closer to the natural way of expression of informal mathematical language than most other systems based on Type Theory. We achieve this by making a sharp distinction between the abstract expressions that are manipulated by the formal system (sometimes called the "deep structure" or "kernel language"), and the concrete text as it is written down by the mathematician.

In type theory, propositions and proof objects are represented as a kind of types and objects. We have not developed a really natural notation for proof construction, so we describe proofs mainly in ordinary English, sometimes using an equational style.

2.1 Language definition mechanism

Mathematical languages are often defined by first defining a kernel language and subsequently adding "syntactic sugar". We choose for a more sophisticated correspondence between the concrete language and its basic type theory, for the following reasons.

1. There are new syntactic classes, like 'declarations', that do not correspond to expressions in the primitive type theory.
2. Context information for an expression contains more information about the syntactic forms that are allowed for that expression than contexts in the primitive type theory.
3. We wish to have the possibility to introduce new sublanguages within *ADAM* that may have a structure very different from type theory.

We seek to define the language by a form of two-level grammar in the style of Van Wijngaarden [86], but using abstract trees as parameters rather than strings. The grammar formalism is informally described in this section. The description of *ADAM* in the following sections uses this grammar formalism, but some of the more complicated features will be described merely by example.

2.1.1 Two-level grammar. The first level consists of a number of *abstract* syntax classes A, which are defined by context-free production rules. Members of these classes are abstract trees, rather than strings as in Van Wijngaarden grammar [86].

The second level consists of a number of parametrized *concrete* syntax classes $T(\bar{x})$, where $\bar{x}$ is a sequence of parameters or meta-variables x_i, each ranging over an abstract syntax class A_i. Names of syntax classes of both levels are in *Slanted* font and begin with a capital. The concrete classes are defined by production rules which may contain meta-variables. Members of a concrete class (with actual parameters) are pieces of concrete text. The parameters serve to pass information from the context into the derivation of a piece of concrete text and vice versa.

Thus, our notion of two-level grammar is roughly the same as the notion of *Definite Clause Grammar* (or unification grammar) [2, page 79–80] used in the logic programming community. It has also some similarity with the notion of *attribute grammar*, except that we expect parsing to be done simultaneously with parameter instantiation, and do not allow a parse tree to be recomputed with different parameter values. There is no need even to mention parse trees. See Małuszyński [53] for a comparison between Definite Clause Grammars and attribute grammars.

The distinction between the two levels is not really necessary: a first level class A can be seen as a second-level class $A(x)$ on type-free trees, and its context-free production rules as abbreviations for second-level production rules.

2.1.2 Derived notions. We introduce some (syntactic) *predicates* $p(\bar{x})$ over abstract classes. These are inductively defined by Horn clause logic with equality. Clauses are written using '$\Leftarrow$' as main operator. Each such predicate may be identified with a syntax class that contains the empty string ϵ just when the predicate holds. To effect this, replace each Horn clause $p(\bar{x}) \Leftarrow \bar{q}(\bar{x})$ by a production rule $p(\bar{x}) \longrightarrow \bar{q}(\bar{x})$, and any production rule containing a condition $p(\bar{x})$ will be blocked when $p(\bar{x})$ is not rewritable into ϵ.

We also introduce some (syntactic) *operations* $f(\bar{x})$ on abstract classes. Such an operation may be replaced by a predicate $f'(\bar{x}, y)$ that holds just when $f(\bar{x})$ equals y.

The parameters of a class can often be separated into input (or *inherited*) and output (or *synthesized*) parameters. The actual value of an input parameter is assumed to be fixed by the production rules that invoke the syntax class. The actual value of an output parameter is to be determined by the production rules for the class itself.

We use a *generic* abstract class, A^*, for any class A. This class contains all finite (possibly empty) sequences of members of A.

2.1.3 Grammar notation. Syntactic predicate and operation names are in lower case.

The production rules for abstract classes are given in Backus-Naur Form, using meta-symbols '::=', '|', and '.', and braces '{', '}' for grouping. In particular, '{}' stands for an empty production. All other symbols are terminal symbols.

The production rules for concrete classes are written using '$\longrightarrow$' as main operator, a comma for string concatenation, terminal symbols between double quotes, and using italic or greek symbols as variables.

Rules for predicates are written using $\Leftarrow$ as main operator. Note that string concatenation for classes becomes conjunction for predicates. For any abstract class, we assume syntactic equality and inequality predicates, $(a = a')$ and $(a \neq a')$.

2.2 Basic grammar of ADAM

This section introduces the abstract and concrete classes that we will use. The next section (2.3) gives production rules for general use. The other sections describe specific language features; most of these are formalized by means of definitions to be collected in a *standard environment*. Several features require additional syntax classes and production rules; these are only informally described, either by means of example or by means of definitions that treat only some special cases.

The main context-free classes are (abstract) *terms* and *contexts*, just as in B.1. The rules in appendix B define a predicate '$\Gamma \vdash t\colon T$' for contexts Γ and terms t and T, meaning that under the assumptions in Γ, abstract term t has type T.

The main concrete syntax class is $\mathit{Term}_{\Gamma,\gamma}(t, T)$, where context Γ records all assumptions $v\colon A$ currently made, γ is an *environment* recording all name bindings and other definitions. Now, if T represents a type (i.e. $\Gamma \vdash T\colon \mathbf{Type}_i$, where i is called the level of T), then a concrete text produced by $\mathit{Term}_{\Gamma,\gamma}(t, T)$ denotes the abstract term t, and the production rules shall be such that $\Gamma \vdash t\colon T$ holds. (We conceive that this might be checked by an appropriate grammar manipulation tool.)

2.2.1 Abstract classes. We have the following abstract classes, listed together with the meta-variables that range over them.

α	A^*	: For any syntax class A: finite sequences of expressions of class A
v	*Var*	: Abstract variables
c, Q	*Const*	: Abstract primitive constants

i	*Nat*	: Natural numbers at the syntactic level, e.g. to index the hierarchy of universes
t, a, b	*Term*	: Terms, which denote objects, including types
T, A, B	*Type*	: Types, being just terms
Γ, Δ	*Context*	: Contexts, being sequences of assumptions like $v{:}\,T$
x	*Name*	: Names (identifiers) used in concrete text
γ, δ	*Env*	: Environments, being sequences of environment items
	EnvItem	: Environment items, being either name bindings or coercions (This may be extended.)
ϕ	*Subst*	: Substitutions for abstract variables

We distinguish between variables used in abstract terms and names written in concrete text for the following reasons:

- Concrete names may be ambiguous; abstract variables must be unambiguous.
- The concrete name x used in a concrete abstraction, $(x :: b)$, may be different from the name y used in the type of the abstraction, $\Pi(y{:}\, A :: By)$.
- When using pattern matching, for example $((x, y) :: b)$, multiple concrete names refer to components of the value of the same unnamed abstract variable.

An environment γ contains, for any visible name x, an item $x{:}\,T := t$ that gives the type and (abstract) value of x. This value may be either the abstract variable named by x, or the value of x as given by some definition. The environment may furthermore specify other information that influences the parsing of concrete terms, such as coercions (described in 2.3.5), and infix operator symbols (not formally described here).

For *Var* and *Const*, see 2.2.3. The other classes are given by:

$$\begin{array}{rcl}
\mathit{Nat} & ::= & \mathit{0} \mid \mathit{Nat}' \;. \\
A^* & ::= & \{\} \mid A\, A^* \;. \\
\mathit{Term} & ::= & \mathit{Var} \\
 & \mid & \mathit{Const}(\{\mathit{Term},\}^*) \\
 & \mid & (\mathit{Var} :: \mathit{Term}) \;. \\
\mathit{Type} & ::= & \mathit{Term} \;. \\
\mathit{Context} & ::= & \{\mathit{Var}{:}\, \mathit{Type};\}^* \;. \\
\mathit{EnvItem} & ::= & \mathit{Name}{:}\, \mathit{Type} := \mathit{Term} \\
 & \mid & \mathit{Const}(\mathit{Context}){:}\, \mathit{Type} := \mathit{Term} \\
 & \mid & \mathit{Context} \vdash \mathit{Term} : \mathit{Type} \subseteq_{\mathsf{t}} \mathit{Type} \;. \\
\mathit{Env} & ::= & \{\mathit{EnvItem};\}^* \;. \\
\mathit{Subst} & ::= & \{\mathit{Var} := \mathit{Term},\}^* \;.
\end{array}$$

(*EnvItem* may be extended with other kinds of environment information.)

2.2.2 Concrete classes.

We did already introduce the class $\mathit{Term}_{\Gamma,\gamma}(t,T)$ of terms of inherited type T. Besides we have a class $\mathit{TTerm}_{\Gamma,\gamma}(t,T)$ of terms of synthesized type T. Thus, expressions of this class define both the abstract term t and its type T, and we will have both $\Gamma \vdash T\colon \mathbf{Type}_i$ for some i, and $\Gamma \vdash t\colon T$.

$\mathit{Term}_{\Gamma,\gamma}(t,T)$	: A term t of inherited type T
$\mathit{TTerm}_{\Gamma,\gamma}(t,T)$	: A term t of synthesized type T, guaranteed to be correct (if Γ,γ are correct)
$\mathit{Def}_{\Gamma,\gamma}(\delta)$	: A definition yielding the new environment δ
$\mathit{Defs}_{\Gamma,\gamma}(\delta)$	: A (nonempty) sequence of definitions
$\mathit{Decl}_{\Gamma,\gamma}(\Delta,\delta,i)$	: A declaration yielding the new assumptions Δ and environment δ, containing only types up to level i
$\mathit{Pat}_{\Gamma,\gamma}(T,t,\delta)$	: An exhaustive pattern for type T that, when matched against term t, yields name bindings δ

Predicates are the following:

$(a = a')$	: a is structurally equal to a'
$(v \neq v')$	: variable v is different from v'
$(t => t')$	: term t reduces to head normal form t'
$(t == t')$	: term t is convertible to t'
$\Gamma \vdash t\colon T$	: in context Γ, t is a term of type T
$\mathit{in}(a,\alpha)$	: a occurs in list α
$\mathit{fresh}(v,\Gamma)$	: Abstract variable v is fresh with respect to Γ
$\mathit{valu}_\Gamma(\phi,\Delta)$	: ϕ is a valid valuation for Δ in context Γ

Predicates $=>$, $==$, and $\vdash$ are given in appendix B. Predicates *in*, *fresh*, and *valu* are given by:

$$\begin{array}{rcl}
\mathit{in}(a, a\$\alpha) & & . \\
\mathit{in}(a, b\$\alpha) & \Leftarrow & \mathit{in}(a,\alpha)\ . \\
\mathit{fresh}(v, \{\}) & & . \\
\mathit{fresh}(v, \{v'\colon T;\ \Gamma\}) & \Leftarrow & (v' \neq v), \mathit{fresh}(v,\Gamma)\ . \\
\mathit{valu}_\Gamma(\{\},\{\}) & & . \\
\mathit{valu}_\Gamma(\{\phi, x := t\}, \{\Delta;\ x\colon T\} & \Leftarrow & \mathit{valu}_\Gamma(\phi,\Delta),\ \Gamma \vdash t[\phi]\colon T[\phi]\ .
\end{array}$$

Syntactic operations:

$\alpha\alpha'$	: List concatenation
$t[\phi]$	: Term t under substitution ϕ
$\mathit{repq}(Q,\Delta,t)$	: Repeatedly apply quantifier Q for all typings in Δ to term t

List concatenation and term substitution are defined as usual, taking care for variable renaming. Substitution of a single variable is needed for term reduction in appendix B; substitution of multiple variables is currently only used in our description of coercion, subsection 2.3.5. Repeated quantifier application is defined in 2.5.5.

2.2.3 Identifiers. We consider availability of several styles of identifiers indispensable for writing serious mathematical texts (and hope that automatic proof checkers will soon provide them).

We use single-character names in italic, greek, or calligraphic font, and multiple-character names in sans-serif and boldface font. So the expression 'fax' is the juxtaposition of three variables, while 'fax' is a single constant. In patterns, the underscore '_' will act as an anonymous variable.

Adding a prime '$'$' builds a new name. Other decorations such as subscripts usually denote some operation applied to the undecorated symbol, but may also be used to build new identifiers if this is explicitly stated.

Furthermore, we sometimes use other mathematical symbols, sometimes written in infix (or postfix, outfix, mixfix, ...) position.

Capitalized identifiers are normally used for types, boldface identifiers for types of types, sets of sets, categories etc. When introducing new notation, characters from the end of the alphabet usually stand for arbitrary identifiers, while other symbols stand for expressions.

2.3 Production rules

2.3.1 Terms

As said, we distinguish between terms with inherited and with synthesized type. A typical example of a term with synthesized type is a variable occurence. A term with synthesized type may be used where an inherited type is already given, provided both types are convertible. Terms of both classes may be surrounded by parentheses, and may make use of local definitions.

$$\begin{array}{rcl}
\mathit{TTerm}_{\Gamma,\gamma}(t,T) & \longrightarrow & \mathit{Name}\,x,\ \mathit{in}(\{x\!:T := t\},\gamma). \\
\mathit{Term}_{\Gamma,\gamma}(t,T) & \longrightarrow & \mathit{TTerm}_{\Gamma,\gamma}(t,T'),(T == T'). \\
\mathit{TTerm}_{\Gamma,\gamma}(t,T) & \longrightarrow & \text{“(”}, \mathit{TTerm}_{\Gamma,\gamma}(t,T), \text{“)”}. \\
\mathit{Term}_{\Gamma,\gamma}(t,T) & \longrightarrow & \text{“(”}, \mathit{Term}_{\Gamma,\gamma}(t,T), \text{“)”}. \\
\mathit{TTerm}_{\Gamma,\gamma}(t,T) & \longrightarrow & \text{“}\underline{\mathsf{let}}\text{ ”}, \mathit{Defs}_{\Gamma,\gamma}(\delta), \text{“ }\underline{\mathsf{in}}\text{ ”}, \mathit{TTerm}_{\Gamma,\gamma\delta}(t,T). \\
\mathit{Term}_{\Gamma,\gamma}(t,T) & \longrightarrow & \text{“}\underline{\mathsf{let}}\text{ ”}, \mathit{Defs}_{\Gamma,\gamma}(\delta), \text{“ }\underline{\mathsf{in}}\text{ ”}, \mathit{Term}_{\Gamma,\gamma\delta}(t,T). \\
\mathit{Term}_{\Gamma,\gamma}((v :: b),\Pi(A;B)) & \longrightarrow & \mathit{fresh}(v,\Gamma), \text{“(”}, \mathit{Pat}_{\Gamma,\gamma}(A,v,\delta), \\
 & & \text{“::”}, \mathit{Term}_{\{\Gamma;v:A\},\gamma\delta}(b,B(v)), \text{“)”}. \\
\mathit{TTerm}_{\Gamma,\gamma}((fa),(Ba)) & \longrightarrow & \mathit{TTerm}_{\Gamma,\gamma}(f,\Pi(A;B)), \mathit{Term}_{\Gamma,\gamma}(a,A).
\end{array}$$

As one sees, the context and environment parameters Γ, γ are always passed on to subexpressions. Production rules would be quite easier to read if we could omit them from our rule notation. For now, we will accept the load.

2.3.2 Patterns

The class $\mathit{Pat}_{\Gamma,\gamma}(T,t,\delta)$ produces exhaustive patterns for type T that, when matched against term t, yield name bindings δ. The simplest form of patterns are single named variables, resulting in a single binding, and the anonymous variable '_', resulting in no binding at all.

For composite patterns, we give only a rule for patterns that match dependent pairs (section 2.6). See subsection 2.12.6 for a more general idea of patterns.

$$\begin{aligned}
Pat_{\Gamma,\gamma}(T, t, \{x{:}\,T := t\,,\}) &\longrightarrow Name\,x.\\
Pat_{\Gamma,\gamma}(T, t, \{\}) &\longrightarrow \text{“\_”}.\\
Pat_{\Gamma,\gamma}(\Sigma(A;B), t, \alpha\beta) &\longrightarrow \text{“(”}, Pat_{\Gamma,\gamma}(A, \mathsf{fst}\,t, \alpha),\\
&\qquad \text{“;”}, Pat_{\Gamma,\gamma}(B(\mathsf{snd}\,t), \mathsf{snd}\,t, \beta), \text{“)”}.
\end{aligned}$$

2.3.3 Definitions

A definition introduces some typed identifiers, and assigns a value to them. We have several forms of definition; not all of these formally described. The class $Def_{\Gamma,\gamma}(\delta)$ produces a single definition, the class $Defs_{\Gamma,\gamma}(\delta)$ a sequence of them.

A (concrete) *simple definition* may have the form $\xi{:}\,T := t$, where ξ is a pattern and T and t are expressions denoting a type and a term of that type. A simplified form of definition is $\xi := t$, which is possible only if t has a synthesized type T. As the rules indicate, a definition yields the bindings obtained by handing the value of the term t over to the pattern ξ.

$$\begin{aligned}
Def_{\Gamma,\gamma}(\delta) &\longrightarrow Pat_{\Gamma,\gamma}(T, t, \delta), \text{“:”},\\
&\qquad Term_{\Gamma,\gamma}(T, \mathbf{Type}_i), \text{“:=”}, Term_{\Gamma,\gamma}(t, T).\\
Def_{\Gamma,\gamma}(\delta) &\longrightarrow Pat_{\Gamma,\gamma}(T, t, \delta), \text{“:=”}, TTerm_{\Gamma,\gamma}(t, T).\\
Defs_{\Gamma,\gamma}(\delta) &\longrightarrow Def_{\Gamma,\gamma}(\delta).\\
Defs_{\Gamma,\gamma}(\delta\delta') &\longrightarrow Def_{\Gamma,\gamma}(\delta), \text{“;”}, Defs_{\Gamma,\gamma\delta}(\delta').
\end{aligned}$$

Secondly we have *parametrized definitions* in various forms, which we do not formally define. Some of these are:

1. Primarily, a parametrized definition consists of some new constant c followed by a type declaration Δ of its parameters, its result type T and defining expression t :

   $$\mathsf{c}(\Delta){:}\,T := t$$

2. Instead of declaring the parameters after the constant, they may be declared in front of the definition, separated by the symbol '$\vdash$' in which case only the sequence of variable names $\bar{x}$ declared in Δ appear after the constant:

   $$\Delta_{\bar{x}} \vdash \quad \mathsf{c}(\bar{x}){:}\,T := t$$

   (Do not confuse this use of '$\vdash$' *inside* the language with the syntactic (meta-) predicate $\vdash$!)

3. Parameters can be made implicit by omitting them from the sequence $\bar{x}$. This is often done for type parameters, e.g.

   $$A, B{:}\,\mathbf{Type};\ (x, y){:}\,A \times B \vdash \quad \mathsf{swap}(x, y){:}\,B \times A := (y, x)$$

4. A parameter typing itself can be made implicit by preceding the definition with a separate variable declaration '<u>Variables</u> $x\colon T; \ldots$', like:

$$\begin{array}{l}\underline{\mathsf{Variables}}\ x\colon \mathbb{R}_+;\ n\colon \mathbb{N}_+;\\ \sqrt[n]{x} := x^{1/n};\\ \sqrt{x} := \sqrt[2]{x}\end{array}$$

   Thus, whenever a variable, that is a declared in a <u>Variables</u> declaration, appears untyped in a subsequent definition, it is implicitly typed by the declaration. We have as yet no clear scope rules for variable declarations.

Later on we will allow several other forms of definitions, e.g.:

- Coercions ($\ldots \subseteq_{\mathsf{t}} \ldots$) in 2.3.5
- Several forms of definition by giving a characteristic property (<u>Define</u> ... <u>by</u> ...) in 2.6.3 and 2.8.5
- Sum types given in BNF form ($S ::= \ldots \mid \ldots$) in 2.8.4

We have yet no formal scheme to define (or declare) new constructs that bind local variables. But we will use a *pseudo definition* that is suggestive of the intended notation, like:

$$(x\colon A \vdash b_x\colon B) \vdash \quad (x \mapsto b_x) \ := \ \ldots$$

2.3.4 Declarations

A declaration in $Decl_{\Gamma,\gamma}(\Delta, \delta, i)$ is a sequence that may contain typings like $\xi\colon T$, where ξ is a pattern, as well as definitions as above. A special form of typing consists of only a type T, and represents an anonymous assumption $\_\colon T$. This is intended to be used only when T is a proposition.

For each typing $\xi\colon T$, the synthesized parameter Δ contains an abstract assumption $x\colon T$, and δ contains the name bindings that are yielded by matching pattern ξ against x.

$$\begin{array}{rcl}
Decl_{\Gamma,\gamma}(\{\},\{\},i) & \longrightarrow & .\\
Decl_{\Gamma,\gamma}(\{v\colon T;\ \Delta\}, \delta\delta', i) & \longrightarrow & fresh(v,\Gamma), Pat_{\Gamma,\gamma}(T,v,\delta), \text{“:”}, Term_{\Gamma,\gamma}(T, \mathbf{Type}_i),\\
 & & \text{“;”}, Decl_{\{\Gamma;\, v:T\},\gamma\delta}(\Delta, \delta', i).\\
Decl_{\Gamma,\gamma}(\{v\colon T;\ \Delta\}, \delta', i) & \longrightarrow & fresh(v,\Gamma), Term_{\Gamma,\gamma}(T, \mathbf{Type}_i),\\
 & & \text{“;”}, Decl_{\{\Gamma;\, v:T\},\gamma}(\Delta, \delta', i).\\
Decl_{\Gamma,\gamma}(\Delta, \delta\delta', i) & \longrightarrow & Def_{\Gamma,\gamma}(\delta), \text{“;”}, Decl_{\Gamma,\gamma\delta}(\Delta, \delta', i).
\end{array}$$

2.3.5 Coercion

We sometimes wish to allow objects of one type to be “coerced” into objects of another type, by implicitly applying some conversion function. These coercions may be user-defined, and are recorded in the environment by an item of the form $\Delta \vdash f : T \subseteq_{\mathsf{t}}$

T'. Here, f is the coercion function from type T to T', and Δ declares substitution parameters for f, T, and T'. The rules below describe how coercions are defined and implicitly applied; functions are described in 2.5.3.

$$\begin{array}{rcl}
\mathit{Def}_{\Gamma,\gamma}(\{\Delta \vdash f : T \subseteq_{\mathsf{t}} T'\,,\}) & \longrightarrow & \mathit{Decl}_{\Gamma,\gamma}(\Delta,\delta),\ \text{“}\vdash\text{”},\ (\Gamma' = \Gamma\Delta),\ (\gamma' = \gamma\delta),\\
 & & \mathit{Term}_{\Gamma',\gamma'}(f, (T \to T')),\ \text{“:”},\\
 & & \mathit{Term}_{\Gamma',\gamma'}(T, \mathbf{Type}_i),\ \text{“}\subseteq_{\mathsf{t}}\text{”},\ \mathit{Term}_{\Gamma',\gamma'}(T', \mathbf{Type}_i).\\
\mathit{Term}_{\Gamma,\gamma}(f[\phi].t, U) & \longrightarrow & \mathit{in}(\{\Delta \vdash f : T \subseteq_{\mathsf{t}} T'\}, \gamma),\\
 & & \mathit{valu}_{\Gamma}(\phi, \Delta),\ (U == T'[\phi]),\ \mathit{Term}_{\Gamma,\gamma}(t, T[\phi]).
\end{array}$$

Note that there are as yet no restrictions on the coercion function, so that one can define very misleading coercions. It would be desirable to require, but difficult to enforce, that if the transitive closure of the set of all coercions in effect contains any circularity $f\colon T \subseteq_{\mathsf{t}} T$, then the coercion function f should be the identity.

This formalization of $\subseteq_{\mathsf{t}}$ is not wholly adequate to cover the use we make of it. For example, many type constructors preserve coercion, like the following. If

$$\begin{array}{c}
\varphi : A' \subseteq_{\mathsf{t}} A\ ;\\
\psi(x\colon A') : Bx \subseteq_{\mathsf{t}} B'x
\end{array}$$

then we would like to have a coercion $\Pi(\varphi;\psi) : \Pi(A;B) \subseteq_{\mathsf{t}} \Pi(A';B')$, where $\Pi(\varphi;\psi)$ is the appropriate function $(\ p \mapsto (x :: \psi x.p(\varphi.x))\ )$.

We will speak a bit loosely about coercions, like stating $T \subseteq_{\mathsf{t}} T'$ without giving the coercion function. We write $T =_{\mathsf{t}} T'$ if we have a bijective pair of coercions $T \subseteq_{\mathsf{t}} T'$ and $T' \subseteq_{\mathsf{t}} T$.

2.4 Types and universes

Types in context Γ are those terms T for which $\Gamma \vdash T\colon \mathbf{Type}_i$ is derivable, for some $\mathit{Nat}\, i$. This i is called the *level* of T. The constants $\mathbf{Type}_i$ are called *universes*. These are types themselves and form a cumulative hierarchy, for we have:

$$\begin{array}{c}
\mathbf{Type}_i\colon \mathbf{Type}_{i+1}\ ,\\
\mathbf{Type}_i \subseteq_{\mathsf{t}} \mathbf{Type}_{i+1}\ .
\end{array}$$

The latter rule means that any type in $\mathbf{Type}_i$ is in $\mathbf{Type}_{i+1}$ too; see paragraph 2.3.5 for $\subseteq_{\mathsf{t}}$.

These universes are called *predicative*, because the rules for introducing types in $\mathbf{Type}_i$ do not assume that $\mathbf{Type}_i$ itself is completely given. There is also an impredicative universe of propositions **Prop**, which we introduce in section 2.11, that is itself an element of $\mathbf{Type}_0$, and all propositions $P\colon \mathbf{Prop}$ are types in $\mathbf{Type}_0$ as well.

Many definitions can be given at any level, and the subscript i is often left implicit. Definitions that involve a universe, like Fam in 2.7, do actually define a hierarchy of type-constructors

$$T\colon \mathbf{Type}_i \vdash \mathsf{Fam}_i\, T\colon \mathbf{Type}_{i+1}\ .$$

2.5 Products and function spaces

The first paragraph of this and following sections describes a constant that is a primitive type constructor, constants for introducing and eliminating objects of such types, and some derived notation. In most cases, the types of these constants can be given in the standard declaration, but in some cases (including this section) special derivation rules will be needed.

The next paragraphs introduce notions that are derived from the primitive type constructor.

2.5.1 Products. Informally, if A is a type, and for any $x\colon A$ is B_x a type, then the *(generalized) product* $\Pi(x\colon A :: B_x)$ is a type containing all (infinitary) tuples $(x :: b_x)$, where $b_x\colon B_x$ for any $x\colon A$. This is a derived notation for $\Pi(A; (x :: B_x))$.

Selecting a component from a tuple is denoted by juxtaposition, so a tuple can act as a prefix *operator*. Alternative notations are subscripting and reverse application, using an infix '$^\backslash$' which we took from DEVA [85].

$$\begin{array}{rcl}
A\colon \mathbf{Type};\ B\colon \Pi(A; (x :: \mathbf{Type})) & \vdash & \Pi(A; B)\colon \mathbf{Type}\ ; \\
p\colon \Pi(A; B);\ a\colon A & \vdash & pa\colon Ba\ ; \\
p_a & := & pa\ , \\
a^\backslash p & := & pa \\
(x\colon A \vdash b_x\colon Bx) & \vdash & (x :: b_x)\colon \Pi(A; B) \\
(x\colon A :: B_x) & := & (A;\ (x :: B_x)) \\
p, p'\colon \Pi(A; B) \vdash \quad p =_{\Pi(A;B)} p' & \Leftrightarrow & \forall x\colon A :: px =_{Bx} p'x
\end{array}$$

The last line, which uses the equality predicate of section 2.10, is the *extensionality* rule stating that tuples are equal (but not necessarily convertible) just when their components are equal.

Finite products, like $B_0 \times B_1$, are defined as a generalized product over a finite type in paragraph 2.8.2.

2.5.2 Exponential types. Given types A and B, we write B^A for the type $\Pi(\_\colon A :: B)$ of (possibly infinitary) tuples.

$$A, B\colon \mathbf{Type} \vdash \quad B^A \ := \ \Pi(\_\colon A :: B)$$

Single objects are identified with one-tuples via the obvious coercions: $B =_{\mathsf{t}} B^1$.

2.5.3 Function spaces. The type $A \to B$ of (total) functions from A to B is isomorphic to the type B^A of tuples, but we prefer to make the conceptual distinguish between these two types explicit. This allows us to define different coercions and other notations, like:

- Composition of tuples of functions, say $f, g\colon (A \to A)^N$, will be defined componentwise, $(f \bar{\circ} g)_i = f_i \bar{\circ} g_i$ (as arrows in the category $\mathbf{TYPE}^N$), which would be difficult if the typing were $f, g\colon N \to (A \to A)$.

- Single objects are identified with one-tuples: $B =_{\mathsf{t}} B^1$.
- Functions will occasionally be regarded as (single-valued) binary relations between A and B, by defining a coercion $(A \to B) \subseteq_{\mathsf{t}} \mathcal{P}(A \times B)$. In particular, we have $(A \to A) \subseteq_{\mathsf{t}} \mathcal{P}(A^2)$. We would not do this for tuples.

We define function space $A \to B$ by means of a Backus-Naur notation, that is described in paragraph 2.8.4, as the type containing an object λp for any tuple $p\colon B^A$. Thus, function space is like an Abstract Data Type, that is implemented by the tuple type.

The normal notation for function abstraction will be '$x \mapsto b_x$' where variable x is locally bound. Function application is denoted by an infix low dot, and defined by a case analysis (paragraph 2.8.5) on the only case for $f\colon A \to B$.

$$\begin{array}{rrcl}
A, B\colon \mathbf{Type} \vdash & A \to B & ::= & \lambda(p\colon B^A)\ . \\
(x\colon A \vdash b_x\colon B) \vdash & x \mapsto b_x & := & \lambda(x :: b_x)\colon\ A \to B \\
f\colon A \to B;\ a\colon A \vdash & \multicolumn{3}{l}{\underline{\mathsf{Define}}\ f.a\colon B\ \underline{\mathsf{by}}} \\
& (\lambda p).a & := & pa
\end{array}$$

Note that the definition of '$\mapsto$' is a pseudo definition because new forms of variable binding cannot be formally defined from within *ADAM*.

The coercion to binary relations is given by

$$A, B\colon \mathbf{Type} \vdash \quad f \mapsto \{x\colon A :: (x, f.x)\}\colon\ (A \to B) \subseteq_{\mathsf{t}} \mathcal{P}(A \times B)\ .$$

Identity functions, constant functions, backward and forward composition of functions are defined by:

$$\begin{array}{rrcl}
& \mathsf{I}\colon A \to A & := & x \mapsto x \\
& \mathsf{K}(c\colon C)\colon A \to C & := & x \mapsto c \\
f\colon A \to B;\ g\colon B \to C \vdash & g \circ f\colon A \to C & := & x \mapsto g.(f.x)\ ; \\
& f \mathbin{\bar{\circ}} g\colon A \to C & := & g \circ f
\end{array}$$

(Forward composition '$f \mathbin{\bar{\circ}} g$' is Hoare's composition operator '$f\,;g$'.)

Tuples and functions are combined in the following definitions:

$$\begin{array}{rrcl}
& \pi_{a:A}\colon \Pi(A;B) \to Ba & := & p \mapsto pa \\
f_{x:A}\colon C \to Bx \vdash & \langle f \rangle\colon C \to \Pi(A;B) & := & z \mapsto (x :: f_x.z)
\end{array}$$

2.5.4 Infix operators. We may introduce infix operators $\otimes$, which form a new syntactic class, so that

$$x\colon A;\ y\colon B \vdash x \otimes y\colon C$$

for some types A, B and C. Note that infix abstractors like $\mapsto$ do not follow this pattern. For such an operator $\otimes$ we introduce the following notations. The first two show how to turn an infix operator into either a prefix operator or a function on binary tuples. The other two notations are called *sections*[1].

$$\begin{array}{rrcl} & (\otimes)\colon (A \times B \rhd C) & := & ((x,y) :: x \otimes y) \\ & (\otimes)\colon A \times B \to C & := & \lambda(\otimes) \\ x\colon A \vdash & (x\otimes)\colon (B \rhd C) & := & (y :: x \otimes y) \\ y\colon B \vdash & (\otimes y)\colon (A \rhd C) & := & (x :: x \otimes y) \end{array}$$

2.5.5 Repeated abstraction. To introduce a sequence of assumptions and definitions using a single declaration, we introduce a triangle '$\rhd$' as an infix notation. Informally, if Δ is a (concrete) declaration, and T a type in which the identifiers introduced by Δ may occur, then $(\Delta \rhd T)$ stands for taking the product of T over all assumptions in Δ. For example, $(x\colon A;\ y\colon B \rhd C) = \Pi(x\colon A :: \Pi(y\colon B :: C))$. The formal rule is:

$$\begin{array}{rcl} \mathit{Term}_{\Gamma,\gamma}(\mathit{repq}(\Pi,\Delta,T), \mathbf{Type}_i & \longrightarrow & \text{"(", } \mathit{Decl}_{\Gamma,\gamma}(\Delta,\delta,i), \text{"}\rhd\text{"}, \\ & & \mathit{Term}_{\Gamma\Delta,\gamma\delta}(T, \mathbf{Type}_i), \text{")"}. \end{array}$$

The syntactic operation *repq* (repeated quantifier application) is given by the syntactic equations

$$\begin{array}{rcl} \mathit{repq}(Q,\{\},t) & = & t \\ \mathit{repq}(Q,\{v\colon A;\ \Delta\},t) & = & Q(A;\ (v :: \mathit{repq}(Q,\Delta,t))) \end{array}$$

When the declaration consists only of a type, we have $(A \rhd B) = B^A$.

2.6 Sums and declaration types

2.6.1 Sum types. If A is a type and, for any $x\colon A$, Bx is a type, then the *(generalized) sum* $\Sigma(A;B)$ or $\Sigma(x\colon A :: Bx)$ is a type consisting of all pairs $(a;b)$ where $a\colon A$ and $b\colon Ba$. We introduce the typing rules, and define some auxiliary operations.

$$\begin{array}{rcl} A\colon \mathbf{Type};\ B\colon \mathbf{Type}^A & \vdash & \Sigma(A;B)\colon \mathbf{Type} \\ a\colon A;\ b\colon Ba & \vdash & (a;b)\colon \Sigma(A;B) \\ T\colon \mathbf{Type}^{\Sigma(A;B)};\ t\colon (x\colon A;\ y\colon B \rhd T(x;y)) & \vdash & \Sigma\_\mathsf{elim}\, t\colon \Pi(\Sigma(A;B);T) \\ (x\colon A;\ y\colon Bx \vdash t_{xy}\colon T(x;y)\,) \vdash \quad ((x;y) :: t_{xy}) & := & \Sigma\_\mathsf{elim}(x;y :: t_{xy}) \\ \mathsf{fst}\colon (z\colon \Sigma(A;B) \rhd A) & := & ((x;y) :: x) \\ \mathsf{snd}\colon (z\colon \Sigma(A;B) \rhd B(\mathsf{fst}\, z)) & := & ((x;y) :: y) \\ \sigma_{a:A}\colon Ba \to \Sigma(A;B) & := & y \mapsto (a;y) \\ f_{x:A}\colon Bx \to C \vdash \quad [f]\colon \Sigma(A;B) \to C & := & (x;y) \mapsto f_x.y \end{array}$$

Note that we have $\lambda\mathsf{fst}\colon \Sigma(A;B) \to A$. For finite sums, like $B_0 + B_1$, see paragraph 2.8.2.

[1]after an idea of Richard Bird

Taking sums preserves coercion, for if

$$\begin{gathered}\varphi : A \subseteq_{\mathsf{t}} A' \ ; \\ \psi(x{:}\, A) : Bx \subseteq_{\mathsf{t}} B'x\end{gathered}$$

then we have a coercion $\Sigma(\varphi;\psi) : \Sigma(A;B) \subseteq_{\mathsf{t}} \Sigma(A';B')$, where:

$$\Sigma(\varphi;\psi) \ := \ (x;y) \mapsto (\varphi.x;\psi x.y) \ .$$

2.6.2 Declaration types. From a declaration Δ we can obtain the type of its instances by means of Σ. An instance of a declaration is a sequence of (correctly typed) values for the typed identifiers in the list. For example, declaration

$$(x{:}\, A;\ y{:}\, B_x;\ z{:}\, C_{xy})$$

has triples $(a;b;c)$ as instances, where $a{:}\, A$, $b{:}\, B_a$ and $c{:}\, C_{ab}$. So the class of all instances forms a type, denoted by writing parentheses and braces around the declaration, thus: $\{\,(\Delta)\,\}$. In this case:

$$\{\,(x{:}\, A;\ y{:}\, B_x;\ z{:}\, C_{xy})\,\} \ = \ \Sigma(x{:}\, A :: \Sigma(y{:}\, B_x :: C_{xy}))$$

The following grammar rule describes how a declaration type is to be translated into repeated use of quantifier Σ:

$$\mathit{Term}_{\Gamma,\gamma}(\mathit{repq}(\Sigma,\Delta,1), \mathbf{Type}_i) \quad\longrightarrow\quad \text{“}\{\,(\text{”}, \mathit{Decl}_{\Gamma,\gamma}(\Delta,\delta,i), \text{“})\,\}\text{”}.$$

A more elaborate notation for the instances $(a;b;c)$ of a declaration is to write them like definitions, $(x := 1;\ y := b;\ z := c)$. This is very much like Pebble [16] or DEVA [85]. Unfortunately, we cannot formalize this, because our abstract encoding of declaration types (via Σ) disregards the concrete identifier names.

2.6.3 Structure definitions. Elimination on a sum type can be done by pattern matching, as in the definition of fst and snd above. But definitions of classes of structures in mathematics are often given in combination with special elimination constructs. Consider for example the common definition of posets:

> A *poset* X is a structure $(X;\leq_X)$ where X is a set and $\leq_X$ a partial order on X.

This definition indicates (1) that there is a type of posets, (2) that for any set X and partial order $\leq$ on X, $(X;\leq)$ is a poset, and (3) that the underlying set of a poset X is noted as X as well, and that its corresponding partial order is noted as $\leq_X$.

We write such a combined structure definition as follows. (Posets appear again in section 9.1.)

$$\begin{array}{l}\underline{\mathsf{Define}}\ \mathbf{Poset}{:}\, \mathbf{Type}_1\ \underline{\mathsf{by}} \\ \qquad\qquad X{:}\, \mathbf{Poset} \quad :=: \quad (X{:}\, \mathbf{Set};\ (\leq_X){:}\, \mathsf{PO}\, X)\ .\end{array}$$

For another example, see the definition of ‘Fam’ below.

2.7 Families and quantifiers

For T a type, we define a *family of* T to be a tuple $(D; t)$ where D is a type, called the *domain* of the family, and t associates any $d\colon D$ with an element t_d of T. Formally:

$$\underline{\text{Define}}\ \mathsf{Fam}_i(T\colon \mathbf{Type}_{i+1})\colon \mathbf{Type}_{i+1}\ \underline{\text{by}}$$
$$t\colon \mathsf{Fam}_i\, T \quad :=: \quad (\mathsf{Dom}\, t\colon \mathbf{Type}_i;\ t\colon T^{\mathsf{Dom}\, t})$$

The arguments of Π or Σ now turn out to be families of types. I.e., we may write:

$$\Pi, \Sigma\colon (\mathsf{Fam}_i\, \mathbf{Type}_i \triangleright \mathbf{Type}_i)$$

Such constants Q, which are typed by $Q\colon (\mathsf{Fam}_i\, T \triangleright T)$ for some type T and $Nat\, i$, are called *quantifiers.*

In 2.5.1, we introduced a double-colon notation $(x\colon A :: B_x)$ for families. We shall now generalize this to allow an arbitrary declaration Δ instead of the single typing $x\colon A$. Informally, a quantification

$$Q(\Delta :: t)$$

shall stand for the repeated quantifier application $repq(Q, \Delta, t)$. For example, we can write, using the universal quantifier for propositions, and finite and infinite types:

$$\forall(n\colon \mathbb{N};\ k := n^n;\ p\colon \mathbb{N}^k;\ i\colon k :: P(n, p, i)\,)$$

The formal rule looks like

$$\begin{aligned} TTerm_{\Gamma,\gamma}(repq(Q, \Delta, t), T) \quad \longrightarrow \quad & TTerm_{\Gamma,\gamma}(Q, T^{\mathsf{Fam}_i\, T}),\\ & \text{“(”}, Decl_{\Gamma,\gamma}(\Delta, \delta, i), \text{“::”}, Term_{\Gamma\Delta,\gamma\delta}(t, T), \text{“)”}. \end{aligned}$$

except that Q must be a single identifier, and that the concrete type expression $T^{\mathsf{Fam}_i\, T}$ should be replaced by the abstract tree expression that it denotes.

The notation $(\Delta :: t)$ may be used too for families that are not argument of a quantifier Q, using the pseudo-definition

$$(\Delta_{\bar{x}} :: t_{\bar{x}}\,) \ := \ (\,\{\,(\Delta_{\bar{x}})\,\};\ (\bar{x} :: t_{\bar{x}})\,) : \ \mathsf{Fam}\, T$$

A derived notation for finite families is given in paragraph 2.8.2.

2.8 Finite types

2.8.1 Naturals as types. For $Nat\, n$, a finite type with n elements is noted just 'n' (in decimal notation), and its elements are named 0 till $n-1$. And if T is an n-tuple of types, $T\colon \mathbf{Type}^n$, so $\Pi(n; T)$ is the finite product of all Ti, then we note elements of this product as '$(t_0, \ldots, t_{n-1})$', where $t_i\colon Ti$, and the parentheses are optional.

$$\begin{aligned} &\vdash \ n\colon \mathbf{Type} \quad \text{for } Nat\, n\\ &\vdash \ k\colon n \quad \text{for } Nat\, k,\ k < n\\ T\colon \mathbf{Type}^n;\ t_0\colon T(0);\ \ldots;\ t_{n-1}\colon T(n-1) \ &\vdash \ (t_0, \ldots, t_{n-1})\colon \Pi(n; T) \end{aligned}$$

2.8.2 Finite families, products, and sums. We note a family whose domain is a finite type n by using (overloaded) angle brackets. This gives an elegant notation for finite products and sums as special cases of generalized products and sums:

$$\begin{aligned} \langle t_0, \ldots, t_{n-1}\rangle &:= (n; (t_0, \ldots, t_{n-1})) \\ B_0 \times \cdots \times B_{n-1} &:= \Pi\langle B_0, \ldots, B_{n-1}\rangle \\ B_0 + \cdots + B_{n-1} &:= \Sigma\langle B_0, \ldots, B_{n-1}\rangle \end{aligned}$$

This 'overloaded' use of '$\langle\ldots\rangle$' bears no relationship to the notation '$\langle f\rangle$' in paragraph 2.5.3. The latter notation says that if $f_i\colon A \to B_i$ $(i = 0, 1)$ then $\langle f_0, f_1\rangle\colon A \to B_0 \times B_1$. We have furthermore $\pi_i\colon B_0 \times B_1 \to B_i$ and $\sigma_i\colon B_i \to B_0 + B_1$.

We find our definition $B_0 \times B_1 := \Pi\langle B_0, B_1\rangle$ far more elegant than the more common one in type theory, $B_0 \times' B_1 := \Sigma(x\colon B_0 :: B_1)$. Ours has the advantage that notations and operations on generalized products apply directly to finite products, so that, e.g., B^2 is synonymous with $B \times B$.

2.8.3 Enumerations. For any enumerated list of distinct *labels* ℓ_i, we introduce a finite type $\{\ell_0, \ldots, \ell_{n-1}\}$ whose elements are ℓ_i. Let L abbreviate this type, then L is isomorphic with type n. We define a finite type of booleans as an example.

$$\begin{aligned} &\vdash \ell_i\colon L\colon \mathbf{Type} \\ T\colon \mathbf{Type}^L;\ t_0\colon T(\ell_0);\ \ldots;\ t_{n-1}\colon T(\ell_{n-1}) &\vdash (\ell_0 :: t_0 \mid \cdots \mid \ell_{n-1} :: t_{n-1})\colon \Pi(L; T) \\ \mathsf{Bool} &:= \{\mathsf{true}, \mathsf{false}\} \\ \underline{\mathsf{if}}\ b\ \underline{\mathsf{then}}\ p\ \underline{\mathsf{else}}\ q &:= b^\backslash(\mathsf{true} :: p \mid \mathsf{false} :: q) \end{aligned}$$

(We generally use the symbol '|' to join alternatives. Its shape gives a good separation.)

2.8.4 Labeled sums. Sum types whose domain is an enumeration may be defined by grammar rules in Backus-Naur form (BNF), using '::=', which is not to be confused with the BNF definitions used to define the abstract classes of *ADAM*. So the following two definitions are equivalent, where L is $\{\ell_0, \ldots, \ell_{n-1}\}$ and the Δ_i are declarations:

$$\begin{aligned} S &::= \ell_0(\Delta_0) \mid \cdots \mid \ell_{n-1}(\Delta_{n-1}) \\ S &:= \Sigma(L;\ (\ell_0 :: \{\,(\Delta_0)\,\} \mid \cdots \mid \ell_{n-1} :: \{\,(\Delta_{n-1})\,\})\,) \end{aligned}$$

The elements $(\ell_i; \delta)$ of S, where $\delta\colon\{\,(\Delta_i)\,\}$, may be noted $\ell_i(\delta)$, and we can use a case analysis notation like

$$(\ell_0(x_0) :: t_{0x_0} \mid \cdots \mid \ell_{n-1}(x_{n-1}) :: t_{n-1x_{n-1}})\colon \Pi(S; T)$$

2.8.5 Case analysis in definitions. The notation for finite tuples in 2.8.1 and 2.8.3 is a form of case analysis. We allow a more general form of mapping that uses arbitrary (exhaustive and exclusive) patterns, as yet informally. Definitions may use case analysis using a notation like

$$\underline{\mathsf{Define}}\ \mathsf{c}(z\colon A + B)\colon C_z\ \underline{\mathsf{by}}\ \mathsf{c}(0; x) := c_x \mid \mathsf{c}(1; y) := c'_y\ .$$

This corresponds to $\mathsf{c}\colon (z\colon A + B \triangleright C_z) := \Sigma\_\mathsf{elim}((x :: c_x), (y :: c'_y))$.

2.9 Infinite types

Infinite types are to be characterized by induction principles, and these form the subject matter of this thesis. To prove the existence of inductive types, we need to assume the existence of one infinite type:

2.9.1 Naturals. The syntax class *Nat* of naturals is not itself a type. We assume a type named $\mathbb{N}$, with a dependent recursion rule as is usual in type theory. Its elements may be given in decimal notation, so for $\mathit{Nat}\,n$ we have $n{:}\,\mathbb{N}$. Conversely, we identify any $n{:}\,\mathbb{N}$ with a finite type, so $\mathbb{N} \subseteq_{\mathsf{t}} \mathbf{Type}$, where we skip details. We use ω as a synonym for $\mathbb{N}$, especially to index infinite tuples. So $A^\omega := (\mathbb{N} \triangleright A)$.

$$\begin{array}{rcll}
 & \vdash & \mathbb{N}{:}\,\mathbf{Type} & (2.1)\\
 & \vdash & 0{:}\,\mathbb{N};\ \ \mathsf{s}{:}\,(\mathbb{N} \triangleright \mathbb{N}) & (2.2)\\
T{:}\,\mathbf{Type}^{\mathbb{N}};\ b{:}\,T(0);\ t(n{:}\,\mathbb{N};\ h{:}\,T n){:}\,T(\mathsf{s}\,n) & \vdash & \mathbb{N}\_\mathsf{rec}(b,t){:}\,\Pi(\mathbb{N};T) & (2.3)\\
 & \vdash & \mathbb{N} \subseteq_{\mathsf{t}} \mathbf{Type} & (2.4)\\
\omega & := & \mathbb{N} & (2.5)
\end{array}$$

2.9.2 Finite sequences. Now we can form, for any type A, the type $A^* = \Sigma(n{:}\,\mathbb{N} :: A^n)$ of finite sequences of A. The length of a sequence s is noted $\#s$. Our definition is similar to the one for families in 2.7:

$$A{:}\,\mathbf{Type} \vdash \quad \underline{\mathsf{Define}}\ A^*{:}\,\mathbf{Type}\ \underline{\mathsf{by}}\ s{:}\,A^* :=: (\#s{:}\,\mathbb{N};\ s{:}\,A^{\#s})$$

The angle-bracket notation for finite families of paragraph 2.8.2 can be used for finite sequences in A^* as well, i.e. $\langle t_0, \ldots, t_{n-1}\rangle{:}\,A^*$, and we have $A^* \subseteq_{\mathsf{t}} \mathsf{Fam}\,A$.

2.10 Equality predicate

We use a primitive equality predicate on objects of any type. It yields for any type A and objects $u, v{:}\,A$, a type $(u =_A v)$ that is inhabited just when u equals v. This type is actually a proposition in **Prop**, which we introduce in section 2.11. The inhabitant for trivial equalities, including reflexivity, is denoted by 'eq'.

$$\begin{array}{rcl}
A{:}\,\mathbf{Type};\ u, v{:}\,A & \vdash & (u =_A v){:}\,\mathbf{Prop}\\
a{:}\,A & \vdash & \mathsf{eq}{:}\,(a =_A a)\\
p{:}\,(A =_{\mathbf{Type}} B) & \vdash & A =_{\mathsf{t}} B
\end{array}$$

This last rule implies that, if in some context there is an expression $p{:}\,(A =_{\mathbf{Type}} B)$, and $a{:}\,A$, then one has $a{:}\,B$ as well. This is called *type conversion*. Existence of such an expression p is not effectively decidable, so a correctness checker may have to reject expressions $a{:}\,B$ when it is not evident how to find p.

The equality predicate (or equality type) might also be defined by the Leibniz equality, setting $(u =_T v)$ equal to

$$\forall(P{:}\,\mathbf{Prop}^T :: Pu \Rightarrow Pv)\ ,$$

Replacing '⇒' by '⇔' would be equivalent. One still needs a rule for type conversion, and one for extensionality of (infinite) tuples.

2.10.1 Uniqueness. For any type A, we define $!A$ to be the type containing the unique element of A, if one exists; otherwise $!A$ will be empty:

$$\begin{array}{l} \underline{\mathsf{Define}}\ !(A\colon \mathbf{Type})\ \underline{\mathsf{by}} \\ \qquad a\colon !A \quad :=: \quad (a\colon A;\ \mathsf{uniq}\, a\colon (x\colon A \rhd a = x))\ . \end{array}$$

That is to say, any element a of $!A$ consists of an element of A that is noted a as well, together with a proof, noted $\mathsf{uniq}\, a$, that any $x\colon A$ equals a. We have $!A \subseteq_{\mathsf{t}} A$.

2.11 The type of propositions

ITT fully identifies prositions with types, while CC contains a special type **Prop** to represent propositions. This is not only useful for making a conceptual distinction between types and propositions, but also necessary for constructing types that represent the class of all subsets of some type, while staying in the same universe, by defining $\mathcal{P}(A)\colon \mathbf{Type}_i$ as $\mathbf{Prop}^A$, for any $A\colon \mathbf{Type}_i$.

Thus, **Prop** must be a member of any universe of the hierarchy. It is construed *a priori* as being a type of types whose members have at most one element, and are members of any other universe. As the product of any number of propositions still has at most one element, it is again a proposition, the universal quantification noted '$\forall(A;P)$'. We introduce **Prop** here in *ADAM*.

$$\begin{array}{rcl} & \vdash & \mathbf{Prop}\colon \mathbf{Type}_i \\ & \vdash & \mathbf{Prop} \subseteq_{\mathsf{t}} \mathbf{Type}_i \\ P\colon \mathbf{Prop};\ p,q\colon P & \vdash & \mathsf{eq}\colon p =_P q \\ A\colon \mathbf{Type}_i;\ P\colon \mathbf{Prop}^A & \vdash & \forall(A;P)\colon \mathbf{Prop} \\ p\colon \forall(A;P);\ a\colon A & \vdash & pa\colon Pa \\ (x\colon A \vdash p_x\colon Px) & \vdash & (x :: p_x)\colon \forall(A;P) \end{array}$$

From the universal quantifier we derive all other propositional operators and quantifiers. The existential quantifier is defined here as an operator '$\exists A$', meaning "type A is inhabited". We give it a subscript $_\mathsf{w}$ now, as it is soon to be replaced.

The product $\forall(A;P)$ is usually written as $\forall(x\colon A :: P_x)$. This and other quantifiers and connectives are defined below. Note that $\exists_\mathsf{w}$ is defined as an operator that turns a type into a proposition; existential quantification over a family of propositions is derived from this.

$$\begin{array}{rcl} P \Rightarrow Q & := & \forall \_\colon P :: Q \\ P_0 \wedge \cdots \wedge P_{n-1} & := & \forall\langle P_0, \ldots, P_{n-1}\rangle \\ \mathsf{True} & := & \forall\langle\,\rangle \\ P \Leftrightarrow Q & := & (P \Rightarrow Q) \wedge (Q \Rightarrow P) \end{array}$$

$$\begin{array}{rcl} \exists_{\mathsf{w}}(A\colon \mathbf{Type}) & := & \forall(X\colon \mathbf{Prop};\ \forall(x\colon A :: X) :: X) \\ \exists_{\mathsf{w}}(P\colon \mathsf{Fam}\,\mathbf{Prop}) & := & \exists_{\mathsf{w}}\{\,(x\colon \mathsf{Dom}\,P;\ Px)\,\} \\ P_0 \vee \cdots \vee P_{n-1} & := & \exists_{\mathsf{w}}\langle P_0, \ldots, P_{n-1}\rangle \\ \mathsf{False} & := & \exists_{\mathsf{w}}\langle\,\rangle \\ \neg P & := & P \Rightarrow \mathsf{False} \\ P, Q\colon \mathbf{Prop} & \vdash & \mathsf{eq}\colon (P =_{\mathbf{Prop}} Q) \;=\; (P \Leftrightarrow Q) \end{array}$$

In addition, one may need the *axiom of choice.* This axiom states that, given a family of nonempty types, there exists a tuple picking an inhabitant of each type from the family.

$$B\colon \mathbf{Type}^A;\ p\colon \forall(x\colon A :: \exists_{\mathsf{w}}\, Bx) \;\vdash\; \mathsf{ac}\,p\colon \exists_{\mathsf{w}}\,\Pi(A; B)$$

An existential assumption $p\colon \exists_{\mathsf{w}} A$ may be called weak because it can only be used for proving other propositions. As explained in appendix C, for some purposes we need a stronger notion of existential quantification. On other occasions we want to do classical reasoning. In a constructive calculus, these should not be combined in a single type, for this strong quantifier destroys the constructivity of terms of all types as soon as proofs of propositions need not be constructive. For this reason, we introduce two alternative versions of **Prop**, which we name $\mathbf{Prop}_{\mathsf{c}}$ and $\mathbf{Prop}_{\mathsf{i}}$ for classical and constructive (though not really intuitionistic, for intuitionistic philosophy does not admit impredicative quantification) propositions. Thus, we have two variants of *ADAM*. Combining them in a single language might be desirable, but requires a more careful distinction between classical and constructive propositional operators.

Except when stated otherwise, we use constructive logic and omit the subscript i.

2.11.1 Constructive propositions. As expressions should be equivalent to constructive object definitions, we should have an operator ι (iota) that, given a constructive proof that some type has a unique element, denotes that element. Consequently, the information contained in a proof becomes relevant for computing the object denoted by an expression, and one should not introduce representations in the abstract (kernel) language in which this proof information is removed.

Rather than adding primitive rules for iota, appendix C proposes to introduce a new existential quantifier with a stronger elimination rule. The idea is that the proposition $\exists T$ should be equivalent to the quotient type of T modulo the equivalence relation that identifies everything. Thus $\exists T$ contains at most one equivalence class.

The type $\mathbf{Prop}_{\mathsf{i}}$ of constructive propositions has rules as stated above for **Prop** and this new existential quantifier.

$$\begin{array}{lcl} A\colon \mathbf{Type} & \vdash & \exists A\colon \mathbf{Prop}_{\mathsf{i}} \\ & \vdash & \exists\_\mathsf{in}\colon A \to \exists A \\ T\colon \mathbf{Type}^{\exists A}; & & \\ t\colon \Pi(x\colon A :: T(\exists\_\mathsf{in}\, x)); & & \\ d\colon \forall x, y\colon A :: tx = ty & \vdash & \exists\_\mathsf{elim}\, t\colon \Pi(\exists A;\ T) \end{array}$$

Here we let the types A and T and proof d be hidden in the concrete notation $\exists\_\mathsf{elim}\, t$ because we are not so much concerned with proof objects. (In the appendix, rule C.3, we chose to show d explicitly.) Using $\exists\_\mathsf{elim}$ we can define iota, as follows:

$$\begin{array}{rcl} \iota_T\colon (\exists!T \rhd T) & := & \exists\_\mathsf{elim}((u;p)\colon !T :: u) \\ & & \text{noting } (u;p),(v;q)\colon !T \vdash pv\colon (u=v) \end{array}$$

When **Prop** in the definition of $\exists_{\mathsf{w}}A$ is read as $\mathbf{Prop}_{\mathsf{i}}$, then $\exists_{\mathsf{w}}A$ and $\exists A$ become equivalent. I.e., one has

$$\exists A \;\Leftrightarrow\; \forall(X\colon \mathbf{Prop}_{\mathsf{i}};\ \forall(x\colon A :: X) :: X)$$

2.11.2 Classical propositions. For classical logic, one just adds a *reductio ad absurdum* rule:

$$P\colon \mathbf{Prop}_{\mathsf{c}} \;\vdash\; \mathsf{raa}\colon (\neg\neg P \Rightarrow P)$$

(Adding $\exists\_\mathsf{elim}$ for $\mathbf{Prop}_{\mathsf{c}}$ is possible but destroys the constructive nature of object terms.)

2.12 More derived notions

2.12.1 Predicates

A predicate P on a type T is obviously an operator $P\colon (T \rhd \mathbf{Prop})$, and if $\prec$ is an infix relation symbol, say

$$x\colon A;\ y\colon B \vdash x \prec y\colon \mathbf{Prop}\ ,$$

then we have $(\prec)\colon (A \times B \rhd \mathbf{Prop})$. By "sectioning" (paragraph 2.5.4), $(\prec b)$ stands for the predicate $(x :: x \prec b)$.

In a declaration, we may write $(x\colon \prec b)$ for $(x\colon A;\ x \prec b)$.

2.12.2 Subtypes

If A is a type and P a predicate over A, then $\{\,x\colon A \mid\colon Px\}$ is a *subtype* of A which is isomorphic to $\Sigma(A;P)$, but for which we allow a more convenient notation for its elements. Symbol '$|$:' may be read as 'such that'. We can pseudo define it by:

$$\begin{array}{l} \underline{\mathsf{Define}}\ \{\,x\colon A \mid\colon P_x\}\colon \mathbf{Type}\ \underline{\mathsf{by}} \\ \qquad x\colon \{\,x\colon A \mid\colon P_x\} \quad :=: \quad (x\colon A;\ \mathsf{prop}\,x\colon P_x)\ . \end{array}$$

One may use a pattern instead of the single variable x. So we have, if $a\colon \{\,x\colon A \mid\colon P_x\}$, then $a\colon A$ and $\mathsf{prop}\,a\colon P_a$. Furthermore, if for some $a\colon A$ there is an evident $p\colon P_a$, one may write just $a\colon \{\,x\colon A \mid\colon P_x\}$.

Here a problem appears: how should a subtype element a be noted when the corresponding proof p is not evident? Writing '$(a;p)$' is rather confusing. We have thought about '$(a|;p)$', where '$|;$' should be read as "because of", but leave it to informal notation, for now.

Note that subtypes admit only the declaration of variables of that type. To quantify over all subsets of a type one must use the subset type $\mathcal{P}T$ below, which is derived from **Prop**. A membership test for subtypes does not make much sense, because $a \in \{\,x\colon A \mid\colon P_x\}$ can always be replaced by P_a.

2.12.3 Subsets

The type of *subsets* $S{:}\,\mathcal{P}(A)$ is isomorphic to the type of predicates on A, but one writes $a \in S$ for the membership test rather than $S(a)$. We shall write $|P|$ for the subset that corresponds to predicate $P{:}\,\mathbf{Prop}^A$, by making the following definitions.

$$\begin{array}{rcl}
\mathcal{P}(A{:}\,\mathbf{Type}_i){:}\,\mathbf{Type}_i & ::= & |\,(P{:}\,\mathbf{Prop}^A)\,|\ . \\
S{:}\,\mathcal{P}A;\ a{:}\,A & \vdash & \underline{\mathsf{Define}}\ a \in S:\ \mathbf{Prop}\ \underline{\mathsf{by}} \\
 & & a \in |P|\ :=\ Pa \\
(x{:}\,A \vdash P_x{:}\,\mathbf{Prop}) \vdash \quad \{\,x\ |{:}\ P_x\}{:}\,\mathcal{P}A & := & |x :: P_x| \\
t{:}\,\mathsf{Fam}\,A \vdash \quad \{t\}{:}\,\mathcal{P}A & := & \{\,x\ |{:}\ \exists d{:}\,\mathsf{Dom}\,t :: x = t_d\} \\
R, S{:}\,\mathcal{P}A \vdash \quad R \subseteq S & := & \forall(x{:}\,A :: x \in R \Rightarrow x \in S)
\end{array}$$

This may be completed with all usual set operations, like the empty set $\emptyset$, binary operators $\cup$ and $\cap$, quantifiers $\bigcup$ and $\bigcap$ (intersection within A), and $\overline{S}$ (complement in A).

The third definition above introduces the usual suggestive notation for set comprehension, except that we write '$|{:}$' for "such that", just as with subtypes. The fourth definition introduces a notation similar to the replacement axiom of set theory. For example, $\{x{:}\,D :: t_x\,\}$ stands for the subset that contains t_x for any $x{:}\,D$.

The next definition tells how to interpret subsets as types themselves, that is, we have $\mathcal{P}T \subseteq_{\mathsf{t}} \mathbf{Type}$. Furthermore, a type may stand for its full subset. Finally, we shall sometimes omit the bars around a predicate P.

$$\begin{array}{rcl}
T{:}\,\mathbf{Type} \vdash \quad S \mapsto \{\,x{:}\,T\ |{:}\ x \in S\} & : & \mathcal{P}T \subseteq_{\mathsf{t}} \mathbf{Type} \\
T{:}\,\mathbf{Type} \vdash \quad T{:}\,\mathcal{P}T & := & \{\,x\ |{:}\ \mathsf{True}\} \\
T{:}\,\mathbf{Type} \vdash \quad P \mapsto |P| & : & \mathbf{Prop}^T \subseteq_{\mathsf{t}} \mathcal{P}T
\end{array}$$

The declaration $X{:}\subseteq T$ now stands for $X{:}\,\mathcal{P}T;\ \forall(x{:}\,T :: x \in X \Rightarrow \mathsf{True})$, which we read as just $X{:}\,\mathcal{P}T$.

2.12.4 Relational notations

A (binary) relation between two types A and B, notation $R{:}\,A \sim B$, should obviously be a subset $R{:}\,\mathcal{P}(A \times B)$. One easily defines converse, left and right domain, and (forward) composition of relations:

$$\begin{array}{rcl}
A, B{:}\,\mathbf{Type}_i \vdash \quad A \sim B{:}\,\mathbf{Type}_i & := & \mathcal{P}(A \times B) \\
R{:}\,A \sim B \vdash \quad R^{\cup}{:}\,B \sim A & := & \{\,(y, x)\ |{:}\ (x, y) \in R\}\ , \\
R^{<}{:}\,\mathcal{P}A & := & \{\,x\ |{:}\ \exists y :: (x, y) \in R\}\ , \\
R^{>}{:}\,\mathcal{P}B & := & \{\,y\ |{:}\ \exists x :: (x, y) \in R\} \\
R{:}\,A \sim B,\ S{:}\,B \sim C \vdash \quad R \cdot S{:}\,A \sim C & := & \{\,(x, z)\ |{:}\ \exists y{:}\,B :: (x, y) \in R \wedge (y, z) \in S\}
\end{array}$$

This composition is associative and has identity relations $|{=_A}|$.

Functions $f\colon A \to B$ are sometimes identified with their relational graph, and we define the relational image of a set:

$$\begin{array}{rcl}
f \mapsto \{x\colon A :: (x, f.x)\,\} & : & (A \to B) \subseteq_{\mathsf{t}} (A \sim B) \\
R\colon A \sim B;\ X\colon \mathcal{P}A \vdash \quad R[X] & := & \{\,y\colon B \mid\colon \exists x\colon\in X :: (x,y) \in R\} \\
R\colon A \sim B;\ x\colon A \vdash \quad R[x] & := & R[\{x\}]
\end{array}$$

So one has, e.g., for $X\colon\mathcal{P}A$, $Y\colon\mathcal{P}B$; $f\colon A \to B$, $g\colon B \to C$, $h\colon C \to D$; $R\colon B \sim C$, the following equalities:

$$\begin{array}{rcl}
f[X] & = & \{x\colon\in X :: f.x\,\}:\ \mathcal{P}B \\
f^{\cup}[Y] & = & \{\,x \mid\colon f.x \in Y\}:\ \mathcal{P}A \\
f \cdot g & = & f \,\bar{\circ}\, g:\ \ A \sim C \\
f \cdot R & = & \{\,(x,z) \mid\colon (f.x, z) \in R\}:\ \ A \sim C \\
R \cdot h & = & \{(y,z)\colon\in R :: (y, h.z)\,\}:\ \ B \sim D
\end{array}$$

In appendix D we will see how type constructors extend to relation constructors. The most important of these we give here for general use. For $R\colon A \sim A'$, $S\colon B \sim B'$, we define $R \to S$ to be the set of all pairs of functions that map related arguments to related results:

$$R \to S\colon (A \to A') \sim (B \to B') \ := \ \{\,(f,f') \mid\colon \forall (x,x')\colon\in R :: (fx, f'x') \in S\}$$

To obtain a function $f\colon A \to B$ given its graph $R\colon A \sim B$ with a proof that R is single-valued,

$$p\colon \forall x\colon A :: \exists! R[x]\ ,$$

one has to use the iota operator (subsection 2.11.1):

$$f.x \ := \ \iota_{R[x]}(px)\ . \tag{2.6}$$

2.12.5 Currying

We will sometimes consider $(z\colon \{\,(x\colon A; y\colon Bx)\,\} \triangleright Tz)$ to be equivalent to

$$(x\colon A;\ y\colon Bx \triangleright T(x;y))\ .$$

So tab is equivalent to $t(a;b)$ and we can write ta for $(y :: t(a;y))$.

Taking $A := 2$, this convention amounts to $(z\colon B + B' \triangleright Tz)$ being equivalent to

$$(y\colon B \triangleright T(0;y)) \times (y\colon B' \triangleright T(1;y))\ .$$

Thus, if we have $t\colon C^B$ and $t'\colon C^{B'}$, we can write $(t,t')\colon C^{B+B'}$ instead of $((x;y) :: (t,t')xy)$.

(*Currying* is the term used in functional programming for using functions that yield functions again, after an idea of H.B. Curry.)

2.12.6 Pattern matching

The notation of paragraph 2.8.3 for case analysis can be generalized to other patterns. We will be less restrictive than in subsection 2.3.2, for we admit any 'suitable' terms with some free variables to be used as a pattern.

Let meta-expressions $\xi\bar{u}{:}\, A$ stand for patterns containing a sequence of variables $\bar{u}$. Suppose we have a sequence of n patterns $\xi_i u$, where for simplicity we assume that each pattern has only one and the same variable $u{:}\, U_i$. If one knows that

$$\forall x{:}\, A :: \exists i{:}\, n;\ u{:}\, U_i :: x = \xi_i u \tag{2.7}$$

and a sequence of expressions $q_i u$ typed by

$$u{:}\, U_i \vdash q_i u{:}\, B(\xi_i u)$$

such that

$$\forall i, j{:}\, n;\ u{:}\, U_i;\ v{:}\, U_j :: (\xi_i u = \xi_j v \Rightarrow q_i u = q_j v) \tag{2.8}$$

then there is a unique tuple $p{:}\, \Pi(A; B)$ such that $p(\xi_i u) = q_i u$, which we note as

$$(\xi_0 u :: q_0 u \mid \cdots \mid \xi_{n-1} u :: q_{n-1} u)\ .$$

If the i and u in (2.7) are known to be unique, then (2.8) holds trivially.

A similar notation may be used for other infix abstractors, e.g.

$$\begin{aligned}(\xi_0(u{:}\, U_i) \triangleright B_0 u \mid \cdots) &:= (x{:}\, A \triangleright x^\backslash(\xi_0 u :: B_0 u \mid \cdots))\\ (\xi_0 u \mapsto q_0 u \mid \cdots) &:= (x \mapsto x^\backslash(\xi_0 u :: q_0 u \mid \cdots))\end{aligned}$$

A problem with these notations is that it might not be clear which identifiers are being bound, in case ξ_i contains free variables itself.

2.12.7 Linear proof notation

Statements $s\ R\ t$ that some transitive relation R, such as $=$ or $\Leftrightarrow$, holds between two objects or propositions are often derived through a linear sequence of steps. We may present these proofs in a three-column format:

$$\begin{array}{lll} & s & \\ R & s' & \{\,\langle\text{reason why } s\ R\ s'\rangle\,\}\\ R & t & \{\,\langle\text{reason why } s'\ R\ t\rangle\,\}\end{array}$$

During such a linear proof, we may sometimes give a definition that extends to the following proof lines, and even beyond.

Of course, any proposition P may be derived by a linear proof of $P \Leftarrow \mathsf{True}$.

2.13 Conclusion

We defined an extensive notation system based on Constructive Type Theory, partly using a two-level Van Wijngaarden grammar. Several of the more advanced features, that we consider valuable to obtain natural notational flexibility, appeared to be too complicated to be formally defined within the scope of this thesis. We shall apply the notations in the rest of the thesis to express our definitions and rules.

Chapter 3

Common induction and recursion principles

In this chapter, we present a summary of some common principles of inductive definition and recursion. This serves two purposes: to get introduced to inductive definitions and also to get acquainted with our notations.

In 3.1, we start with some simple examples, each one adding a new aspect of inductive definition: naturals, lists, trees, join lists, rose trees, ordinal notations, inductive relations, and infinite lists.

In 3.2, we formulate the main derived principles of induction and recursion on naturals.

In 3.3, we summarize the theory of inductively defining a subset of a given set.

In 3.4, we derive some recursion principles from the induction rules in 3.3.

3.1 Examples of inductive types

Before embarking on generalized formulations of induction principles, we list some examples of inductively defined types that illustrate several features.

Example 3.1 (Natural numbers) The type $\mathbb{N}$ is usually described by the five *Peano axioms* (where we ignore the presence of $\mathbb{N}$ in *ADAM*):

1. 0, *zero*, is a natural number: $0\colon \mathbb{N}$.
2. Whenever n is a natural number, then $\mathsf{s}\,n$, its *successor*, is also a natural number:
$$n\colon \mathbb{N} \vdash \mathsf{s}\,n\colon \mathbb{N}$$
3. No successor equals zero: $\mathsf{s}\,n \neq 0$.
4. Natural numbers with the same successor are equal: $\mathsf{s}\,n = \mathsf{s}\,m \Rightarrow n = m$.
5. Nothing else is a natural number. That is, if a predicate $P(n\colon \mathbb{N})$ holds for zero and is preserved by the successor operation, then it holds for all natural numbers:
$$P(0) \wedge \forall(n\colon \mathbb{N} :: P(n) \Rightarrow P(\mathsf{s}\,n)) \;\Rightarrow\; \forall(n\colon \mathbb{N} :: P(n)) \tag{3.1}$$

Regarding this definition, we distinguish a base clause 1, a step clause 2, equality rules 3 and 4, and an induction clause 5. The induction clause is an instance of the elimination principle for naturals (2.3), taking $Tn := Pn$.

Example 3.2 (Lists) For any type A, the type $\mathsf{Clist}\,A$ of *cons lists* is generated by:

$$\begin{array}{rcl} & \vdash & \square\colon \mathsf{Clist}\,A \\ e\colon A;\, l\colon \mathsf{Clist}\,A & \vdash & e \mathrel{+\!\!<} l\colon \mathsf{Clist}\,A \end{array}$$

As for naturals, we have to give clauses saying that two lists are only equal if they are constructed by the same constructor from equal arguments (*no confusion*):

$$\forall(e; l :: e \mathrel{+\!\!<} l \neq \square) \qquad \forall(e, e'; l, l' :: e \mathrel{+\!\!<} l = e' \mathrel{+\!\!<} l' \Rightarrow e = e' \wedge l = l')$$

and an induction clause saying that all lists can be built by repeated application of the construction rules (*no junk*): for predicates $P(l\colon \mathsf{Clist}\,A)$,

$$P\square \wedge \forall(e; l :: Pl \Rightarrow P(e \mathrel{+\!\!<} l)) \;\Rightarrow\; \forall(l :: Pl)\ .$$

Example 3.3 (Binary trees) The type $\mathsf{BTree}\,A$ is generated by:

$$\begin{array}{rcl} & \vdash & \square\colon \mathsf{BTree}\,A \\ x\colon A & \vdash & \diamond.x\colon \mathsf{BTree}\,A \\ s, t\colon \mathsf{BTree}\,A & \vdash & s +\!\!+ t\colon \mathsf{BTree}\,A \end{array}$$

where $\square$ is a unit of $+\!\!+$:

$$s\colon \mathsf{BTree}\,A \;\vdash\; \square +\!\!+ s = s = s +\!\!+ \square$$

No other trees are identical (no confusion), i.e.:

$$\begin{array}{rcl} s +\!\!+ t = \square & \Rightarrow & s = \square \wedge t = \square \\ s +\!\!+ t = u +\!\!+ v & \Rightarrow & (s = u \wedge t = v) \vee (s = \square \wedge t = u +\!\!+ v) \vee (s = u +\!\!+ v \wedge t = \square) \end{array}$$

Finally, nothing else is a tree (no junk). That is, for predicates $P(t\colon \mathsf{BTree}\,A)$, if

$$P\square \wedge \forall(x\colon A :: P(\diamond.x)) \wedge \forall(s, t\colon \mathsf{BTree}\,A :: Ps \wedge Pt \Rightarrow P(s +\!\!+ t))$$

then $\forall(s\colon \mathsf{BTree}\,A :: Ps)$.

We will see in example 4.5 that the notion of initial algebra avoids the formulation of complicated no-confusion and no-junk conditions.

Example 3.4 (Join lists) The constructors for join lists $\mathsf{JList}\,A$ are the same as those for $\mathsf{BTree}\,A$, but have the additional equation telling that $+\!\!+$ (called *join*) is associative,

$$s, t, u\colon \mathsf{JList}\,A \;\vdash\; (s +\!\!+ t) +\!\!+ u = s +\!\!+ (t +\!\!+ u).$$

We shall not spell out the complicated no-confusion condition; the no-junk condition is the same as for binary trees.

Example 3.5 (Rose trees) Rose trees and forests (e.g. Malcolm [52]) can be described either as two mutually inductive types or as a single one using lists.

Making the latter choice, rose trees over some type A are inductively generated by: any list f of rose trees together with some element x of A makes a rose tree noted $x \surd f$, i.e.

$$x\colon A, f\colon \mathsf{Clist}(\mathsf{RTree}\,A) \quad \vdash \quad x \surd f\colon \mathsf{RTree}\,A$$

This is an *iterated inductive definition* in the sense of [55], as it builds upon another inductively defined type, Clist. However, the use of iterated induction is not essential here.

The alternative is to define rose trees and lists of them (named forests) simultaneously, by mutual induction:

$$\begin{array}{rcl} x\colon A, f\colon \mathsf{Forest}\,A & \vdash & x \surd f\colon \mathsf{RTree}\,A \\ & \vdash & \square\colon \mathsf{Forest}\,A \\ t\colon \mathsf{RTree}\,A, f\colon \mathsf{Forest}\,A & \vdash & t \mathrel{+\!\!<} f\colon \mathsf{Forest}\,A \end{array}$$

Example 3.6 (Ordinal notations) A more typical iterated inductive definition is the following one, which builds upon the inductive type of naturals. Ordinal notations are constructed from zero, a successor operation, and taking the limit of an infinite but countable series of ordinal notations. They may be used to represent ordinals of the so-called second number class. This type Ord is our first infinitary inductive type.

$$\begin{array}{rcl} & \vdash & 0\colon \mathsf{Ord} \\ n\colon \mathsf{Ord} & \vdash & \mathsf{s}\,n\colon \mathsf{Ord} \\ u\colon \mathsf{Ord}^{\omega} & \vdash & \mathsf{lim}\,u\colon \mathsf{Ord} \end{array}$$

Our final two examples illustrate rather different kinds of inductive definitions.

Example 3.7 (An inductive relation) Given a relation $R\colon\subseteq T^2$, its transitive closure $R^{(+)}$ is specified by

$$\begin{array}{rcl} R & \subseteq & R^{(+)} \\ R^{(+)} \cdot R^{(+)} & \subseteq & R^{(+)} \end{array}$$

and: $R^{(+)}$ is the least relation w.r.t. $\subseteq$ that satisfies these two rules.

By Knaster-Tarski (theorem 3.6), the unique solution to this specification is the intersection of all relations that satisfy the two rules:

$$R^{(+)} \ := \ \bigcap(X\colon\subseteq R \mid\colon R \subseteq X \ \wedge \ X \cdot X \subseteq X)$$

Similarly, the reflexive and transitive closure is

$$R^{(*)} \ := \ \bigcap(X\colon\subseteq R \mid\colon |{=_T}| \subseteq X \ \wedge \ R \subseteq X \ \wedge \ X \cdot X \subseteq X)$$

Note the difference with the preceding examples: there we created new types by stating new operations and equations, here we define a subset of an existing type (T^2) by giving rules that refer only to existing operations.

Example 3.8 (Infinite lists) The type $A\infty$ of (total) infinite lists over A is isomorphic with A^ω, but we give a quite different axiomization. This is not an inductive type definition like the examples 3.1–3.7, but we shall see that it is its categorical dual, a final coalgebra instead of an initial algebra. The axioms are:

1. Any infinite list has a head: $\mathsf{hd}\colon A\infty \to A$.

2. Any infinite list has a tail: $\mathsf{tl}\colon A\infty \to A\infty$.

3. For any type X and mappings $h\colon X \to A$ and $t\colon X \to X$, there exists a family of infinite lists $l\colon (A\infty)^X$ such that:

$$\begin{array}{rcl} \mathsf{hd}.l_i & = & h.i \ , \\ \mathsf{tl}.l_i & = & l_{(t.i)} \ . \end{array}$$

4. Moreover, this family of lists l is unique: any other such family equals it.

Summary. Example 3.1 gave the most common form of inductive type definition; example 3.2 defined a type with a parameter; example 3.3 gave a type with parameter and equations; the join lists of example 3.4 added more equations; example 3.5 used mutual induction, example 3.6 iterated induction.

Example 3.7 showed the difference between inductive type and inductive set definitions, and example 3.8 displayed the categorical dual of inductive type definition.

3.2 More on natural numbers

Peano's 5th axiom (3.1) constitutes the first induction principle. Of course one can derive similar principles starting at a base different from zero. A well-known equivalent principle is total induction. Let $|{<}| := (\lambda\,\mathsf{s})^{(+)}$, the transitive closure of the successor relation $\lambda\,\mathsf{s} = \{n\colon \mathbb{N} :: (n, \mathsf{s}\,n)\,\}$.

Theorem 3.1 (Total induction) *If a property $P(n\colon \mathbb{N})$ can be proven on the assumption that it holds for smaller natural numbers, then it holds for all natural numbers:*

$$\forall(n\colon \mathbb{N};\ |{<}\,n| \subseteq |P| :: Pn) \ \Rightarrow\ \mathbb{N} \subseteq |P| \tag{3.2}$$

Note that $|{<}\,n| \subseteq |P|$ abbreviates $\forall(m\colon \mathbb{N} :: m < n \Rightarrow Pm)$.

Proof. Use (3.1) substituting $P(n) \ := \ |{<}\,n| \subseteq |P|$. ■

Note that no separate treatment of some base case is needed.

Definitions of a recursive function f on natural numbers usually consist of a base case, $f.0 = b$ and an induction step of the form $f.(n+1) = g.(n, f.n)$. This is called primitive recursion. Typed lambda calculi with natural numbers usually have a recursion construct which allows such definitions. With the help of the iota operation, one can derive it from the Peano axioms.

Theorem 3.2 (Primitive recursion)

$$\frac{\begin{array}{l} U\colon \mathbf{Type} \\ b\colon U \\ g\colon \mathbb{N}\times U \to U \end{array}}{\exists! f\colon \mathbb{N} \to U :: f.0 = b \;\wedge\; \forall n :: f.\mathsf{s}\,n = g.(n, f.n)} \tag{3.3}$$

Proof. This is an instance of theorem 3.7 below, for $T := \mathbb{N}$, $|{\prec}| := \lambda\mathsf{s}$, and $s := ((0;h) :: b \mid (\mathsf{s}\,n;h) :: g.(n,hn))$, for Peano's axiom (3.1) says exactly that this $\prec$ is well-founded (3.6). ■

Alternatively, existence of f is a special case of the elimination principle for naturals (2.3) for nondependent types, $Tn := U$. Uniqueness follows: assume $g\colon \mathbb{N} \to U$ satisfies the same equations, then prove $\forall n\colon \mathbb{N} :: g.n = f.n$ by Peano induction, which is an instance of (2.3) too.

3.3 Inductive subset definitions

We summarize the standard theory of inductive set definitions, following the basic definitions from the first section of Aczel's Introduction to Inductive Definitions [3]. This theory deals with subsets of a set (or type) that has been constructed prior to the inductive definition.

3.3.1 Sets inductively defined by rules

Example 3.7 defined $R^{(+)}$ as the least subset of T^2 that is closed under certain rules. The other examples, 3.1 till 3.6, introduced new types, not subsets, but each of these types can be characterized by saying that it equals its own least subset that is closed under certain rules. Let us give the general form of such definitions.

Let type T be given. We define:

1. A *rule* is a pair (X, x) where $X\colon \mathcal{P}T$ is called the set of *premisses* and $x\colon T$ is the *conclusion*. A set of rules, $\Phi\colon \mathcal{P}(\mathcal{P}T \times T)$, is also called a *rule set*.

2. If Φ is a rule set, then a set $S\colon\subseteq T$ is Φ-*closed* iff each rule in Φ whose premisses are in S also has its conclusion in S, i.e. iff
$$\forall (X, y)\colon\in \Phi :: X \subseteq S \Rightarrow y \in S\ .$$

3. If Φ is a rule set, then $I(\Phi)$, the *set inductively defined by* Φ, is given by
$$I(\Phi) \;:=\; \bigcap (S\colon\subseteq T \mid\colon S \text{ is } \Phi\text{-closed})\ . \tag{3.4}$$

Note. Φ-closed sets exists; e.g. the type T itself. Also, the intersection of any collection of Φ-closed sets is Φ-closed. In particular $I(\Phi)$ is Φ-closed and hence $I(\Phi)$ *is the smallest* Φ-*closed subset.*

From the definition (3.4) of $I(\Phi)$ we get immediately the principle of Φ-*induction*: If $P(x\colon T)$ is a predicate, such that whenever $(X, y)\colon\in \Phi$ and $X \subseteq |P|$ then Py, then Px holds for every $x\colon\in I(\Phi)$.

Definition (3.4) involves a second-order quantification. If one designs a logical calculus without quantification over arbitrary subsets, one might consider including inductive set (or predicate) definitions as a primitive rule [55]. By the way, if a calculus includes inductive mutually recursive type definitions in the style of section 5.2.2, then one gets inductive predicate definitions as well.

3.3.2 The well-founded part of a relation

A slightly less general scheme of inductive subset definitions is based on well-founded relations. The constructive idea of a well-founded relation is a generalization of the principle of total induction (3.2), but its classical definition is different.

Let $\prec$ be a binary relation on a type T. The *well-founded part* of T for $\prec$, $W(\prec) :\subseteq T$, is defined (classically) as the set consisting of those $x : T$ for which there is no infinite descending sequence $x \succ s_0 \succ s_1 \succ \cdots$. The relation $\prec$ is called *well-founded* iff $T = W(\prec)$, and it is a *well-ordering* iff it is both well-founded and transitive (but see (3.6) for the constructive definition of wellfoundedness). Note that the transitive closure $\prec^{(+)}$ of any well-founded relation is a well-ordering. The elements $y :\prec x$ are called the *predecessors* of x.

The set $W(\prec)$ can be defined inductively using the following rule set $\Phi_\prec$.

$$\Phi_\prec \;:=\; \{x : T :: (|\prec x|, x)\} \tag{3.5}$$

The set inductively defined by $\Phi_\prec$, $I(\Phi_\prec)$, is called the *reachable part* of T for $\prec$. The principle of total induction (3.2) can now be formulated as $\mathbb{N} \subseteq I(\Phi_<)$.

Theorem 3.3 *$W(\prec) = I(\Phi_\prec)$, classically.*

Proof. $\supseteq$: It suffices to show that $W(\prec)$ is $\Phi_\prec$-closed. So assume $|\prec x| \subseteq W(\prec)$. To show $x \in W(\prec)$, suppose $x \succ s_0 \succ \cdots$. Then $s_0 \in |\prec x| \subseteq W(\prec)$. But as $s_0 \succ s_1 \succ \cdots$, $s_0 \notin W(\prec)$, which gives a contradiction.

$\subseteq$: Let $x :\in W(\prec)$ and S be $\Phi_\prec$-closed. Supposing $x \notin S$, we shall derive a contradiction by finding $x \succ s_0 \succ \cdots$ showing that $x \notin W(\prec)$. As $x \notin S$, then $|\prec x| \not\subseteq S$. Hence there is an $s_0 :\prec x$ such that $s_0 \notin S$. Repeating indefinitely we obtain $s_{i+1} :\prec s_i$ such that $s_{i+1} \notin S$. ∎

Conversely, inductive definitions can often be rephrased in the form $\Phi_\prec$ for a suitable $\prec$. Let Φ be a rule set on a type T; we say Φ is *deterministic* iff

$$(X_0, y) \in \Phi \wedge (X_1, y) \in \Phi \;\Rightarrow\; X_0 = X_1 \;.$$

Let $(\prec_\Phi) :\subseteq T^2$ be the relation $|\in| \cdot \Phi \;\cup\; \{(x, y) \,|: y \notin \Phi^>\}$, i.e.:

$$x \prec_\Phi y \;\;:=\;\; y \in \Phi^> \Rightarrow \exists(X : \mathcal{P}T :: x \in X \wedge (X, y) \in \Phi) \;.$$

(Aczel [3, proposition 1.2.4] erroneously missed the condition $y \in \Phi^>$.)

Theorem 3.4 *For deterministic Φ, one has (classically)*

$$I(\Phi) = I(\Phi_{\prec_\Phi}) = W(\prec_\Phi) \;.$$

Proof. We have $I(\Phi_{\prec_\Phi}) = W(\prec_\Phi)$ from theorem 3.3. Next,

$$\begin{array}{rcl}
 & I(\Phi) \subseteq I(\Phi_{\prec_\Phi}) & \\
\Leftarrow & \text{all } \Phi_{\prec_\Phi}\text{-closed sets are } \Phi\text{-closed} & \{\text{def. } I \text{ (3.4)}\} \\
\Leftarrow & \Phi \subseteq \Phi_{\prec_\Phi} & \{\text{def. closedness}\} \\
\Leftrightarrow & \forall (X, y) {:}\in \Phi :: X = |{\prec_\Phi}\ y| & \{\text{def. } \Phi_\prec \text{ (3.5)}\} \\
\Leftrightarrow & \forall (X, y) {:}\in \Phi :: X = \{\, x \mid{:}\ (x, y) \in |{\in}| \cdot \Phi\} & \{\text{def. } \prec_\Phi,\ y \in \Phi^>\} \\
\Leftrightarrow & \Phi \text{ is deterministic} & \{\text{def. deterministic}\}
\end{array}$$

and:

$$\begin{array}{rcl}
 & I(\Phi_{\prec_\Phi}) \subseteq I(\Phi) & \\
\Leftarrow & I(\Phi) \text{ is } \Phi_{\prec_\Phi}\text{-closed} & \{\text{def. } I(\Phi_{\prec_\Phi})\} \\
\Leftrightarrow & \forall y :: |{\prec_\Phi}\ y| \subseteq I(\Phi) \Rightarrow y \in I(\Phi) & \{\text{def. } \Phi_\prec,\ \text{closedness}\} \\
\Leftarrow & \forall ((X, y) {:}\in \Phi :: X \subseteq I(\Phi) \Rightarrow y \in I(\Phi)) & \\
 & \wedge\, \forall (y \notin \Phi^> :: T \subseteq I(\Phi) \Rightarrow y \in I(\Phi)) & \{\text{def. } \prec_\Phi\} \\
\Leftrightarrow & \mathsf{True} & \{\text{def. } I(\Phi)\}
\end{array}$$

■

Theorem 3.3 suggests us a constructive interpretation of well-foundedness. Constructively, we even take $T \subseteq I(\Phi_\prec)$ as the definition of well-foundedness. Thus, we henceforth say $\prec$ is well-founded iff it admits *transfinite induction*:

$$\frac{\begin{array}{l} P{:}\,\mathbf{Prop}^T \\ \forall y{:}\,T;\ |{\prec}\ y| \subseteq |P| :: Py \end{array}}{T \subseteq |P|} \tag{3.6}$$

For natural numbers with $(\prec) := (<)$, this is exactly the total induction principle (3.2).

There are several other constructive interpretations of well-foundedness possible, which are classically equivalent:

Theorem 3.5 *The three properties: $\prec$ admitting transfinite induction, $\prec$ having no descending chains, and all nonempty subsets of T having a minimal element, are classically equivalent. Formally these are:*

$$T \subseteq I(\Phi_\prec) \tag{3.7}$$
$$\forall s{:}\,T^\omega :: \exists i{:}\,\omega :: s_{i+1} \not\prec s_i \tag{3.8}$$
$$\forall Q{:}\,\mathcal{P}T;\ \exists(Q) :: \exists q{:}\in Q :: \neg\exists(Q \cap |{\prec}\ q|) \tag{3.9}$$

((3.9) implies (3.8) constructively as well. (3.7) and (3.9) seem to be constructively independent.)

Proof. (3.7) $\Leftrightarrow$ (3.8) is a corollary of theorem 3.3, for (3.8) says just $T \subseteq W(\prec)$.

(3.8) $\Rightarrow$ (3.9): Assume $Q{:}\,\mathcal{P}T$, $q{:}\in Q$ and suppose $\forall q{:}\in Q :: \exists(Q \cap |{\prec}\ q|)$. Then we choose a descending sequence by taking $s_0 := q$, and given $s_i \in Q$, choosing an $s_{i+1}{:}\in Q \cap |{\prec}\ s_i|$. This contradicts (3.8).

(3.9) $\Rightarrow$ (3.8): Given $s\colon T^\omega$, take $Q := \{i\colon \omega :: s_i\}$. Then by (3.9), $\neg\exists(Q \cap |\prec s_i|)$ for some i, and in particular $s_{i+1} \not\prec s_i$. ■

A somewhat different use of well-founded relations is to conduct inductive proofs over some given type. To find suitable relations, Paulson [69] described a number of principles to construct well-founded relations.

3.3.3 Inductive definitions as operators

An "operator" (function) $\phi\colon \mathcal{P}T \to \mathcal{P}T$ is *monotonic* iff $(\phi,\phi) \in (\subseteq) \to (\subseteq)$, i.e. iff $X\colon\subseteq Y\colon \mathcal{P}T$ implies $\phi.X \subseteq \phi.Y$. Given ϕ, let rule set Φ_ϕ be defined by

$$\Phi_\phi \;:=\; \{\, (X,y) \mid\colon y \in \phi.X \} \;.$$

For monotonic ϕ, $S\colon \mathcal{P}T$ is Φ_ϕ-closed just in case $\phi.S \subseteq S$. So

$$I(\Phi_\phi) \;=\; \bigcap(\, S\colon \mathcal{P}T \mid\colon \phi.S \subseteq S) \;. \tag{3.10}$$

Hence it is natural to write, still following Aczel [3], $I(\phi)$ for $\bigcap(S\colon \mathcal{P}T \mid\colon \phi.S \subseteq S)$.

Conversely, all inductive definitions can be obtained using monotonic operators. For, if Φ is a rule set on T we may define ϕ by

$$\phi.Y \;:=\; \{\, y \mid\colon \exists X\colon\subseteq Y :: (X,y) \in \Phi\} \;.$$

Then Y is Φ-closed just in case $\phi.Y \subseteq Y$ so that $I(\Phi) = I(\phi)$.

An alternative characterization of $I(\phi)$ uses transfinite iterations $\phi^{(\lambda)}$ for ordinals λ. We skip this; see [3].

3.3.4 Fixed points in a lattice

The Knaster-Tarski theorem generalizes the fixed point property (3.10) of monotonic operators on sets to complete lattices.

Theorem 3.6 (Knaster-Tarski) *Any monotonic operator F in a complete lattice $(U;\sqsubseteq)$ has a least fixed point*

$$\mathsf{fix}\,F \;:=\; \sqcap(X\colon U \mid\colon F.X \sqsubseteq X)$$

and hence by duality a greatest fixed point

$$\sqcup(X\colon U \mid\colon X \sqsubseteq F.X) \;.$$

Proof. We have $F.(\mathsf{fix}\,F) \sqsubseteq \mathsf{fix}\,F$ because for any $X\colon U$, if $F.X \sqsubseteq X$ then $\mathsf{fix}\,F \sqsubseteq X$ so $F.(\mathsf{fix}\,F) \sqsubseteq F.X \sqsubseteq X$.

Conversely, $\mathsf{fix}\,F \sqsubseteq F.(\mathsf{fix}\,F)$ follows from $F.(F.(\mathsf{fix}\,F)) \sqsubseteq F.(\mathsf{fix}\,F)$, which holds by monotonicity of F. ■

We show the same proof in linear form:

$$\begin{array}{cll}
 & F.(\mathsf{fix}\,F) = \mathsf{fix}\,F & \\
\Leftrightarrow & \mathsf{fix}\,F \sqsubseteq F.(\mathsf{fix}\,F) \ \wedge\ F.(\mathsf{fix}\,F) \sqsubseteq \mathsf{fix}\,F & \{\text{lattice}\} \\
\Leftarrow & F.(F.(\mathsf{fix}\,F)) \sqsubseteq F.(\mathsf{fix}\,F) \ \wedge\ F.(\mathsf{fix}\,F) \sqsubseteq \mathsf{fix}\,F & \{\text{def. } \mathsf{fix}\} \\
\Leftrightarrow & F.(\mathsf{fix}\,F) \sqsubseteq \mathsf{fix}\,F & \{F \text{ monotonic}\} \\
\Leftrightarrow & \forall X;\ F.X \sqsubseteq X :: F.(\mathsf{fix}\,F) \sqsubseteq X & \{\text{def. } \mathsf{fix}\} \\
\Leftarrow & \forall X;\ F.X \sqsubseteq X :: F.(\mathsf{fix}\,F) \sqsubseteq F.X & \{\text{assumption } X\} \\
\Leftarrow & \forall X;\ F.X \sqsubseteq X :: \mathsf{fix}\,F \sqsubseteq X & \{F \text{ monotonic}\} \\
\Leftrightarrow & \mathsf{True} & \{\text{def. } \mathsf{fix}\}
\end{array}$$

3.4 From induction to recursion

Once one has a well-founded relation $\prec$ (or a deterministic rule set), one can define recursive functions provided one has the iota-correspondence of subsection 2.11.1 between functions and single-valued relations. The proof would be somewhat simpler in set theory, as functions and single-valued relations are identified there.

Theorem 3.7 (transfinite recursion) *If $(\prec){:}\subseteq T^2$ is well-founded, and one has a recursion step*

$$s(x{:}\,T;\ h{:}\,U^{|\prec x|}){:}\,U$$

then one can construct a unique $f{:}\,U^T$ such that

$$\forall x{:}\,T :: fx = s(x; f|_{\prec x})\ . \tag{3.11}$$

($f|_{\prec x}$ is the restriction of f to $|\prec x|$, which will henceforth be noted just f.)

Proof. Let the infix binary relation $R{:}\subseteq T \times U$, that is to become the graph of f, be inductively defined as the least relation X such that

$$\forall x{:}\,T;\ h{:}\,U^{|\prec x|}\ ::\ \forall(y{:}\prec x :: y\ X\ hy) \Rightarrow x\ X\ s(x;h)\ . \tag{3.12}$$

To prove single-valuedness of R we apply transfinite induction (3.6) to $Px := \exists!\,|x\ R|$ (the predicate stating that x has a unique R-image), and see that $\forall(x{:}\,T :: \exists!\,|x\ R|)$ holds provided

$$\forall x{:}\,T;\ \forall(z{:}\prec x :: \exists!\,|z\ R|) :: \exists!\,|x\ R|\ .$$

So assuming $x{:}\,T$ and induction hypothesis

$$\forall z{:}\prec x :: \exists!\,|z\ R|\ , \tag{3.13}$$

we have to prove $\exists!\,|x\ R|$. From (3.13) we get (using iota) a unique

$$g{:}\,!(z{:}\prec x \triangleright |z\ R|)\ . \tag{3.14}$$

Taking then $y := s(x;g)$, one has by (3.14) and R satisfying (3.12) that $x\ R\ y$, so $\exists\,|x\ R|$.

Now supposing some $z{:}\,T$ satisfies $x \; R \; z$ too, we must show $z = y$. By definition of R (3.12), we have $z = s(x; h)$ for some h with $\forall(y{:} \prec x :: y \; R \; hu)$. By uniqueness of g (3.14) it follows that $h = g$, and hence $z = s(x; h) = s(x; g) = y$.

This completes the constructive proof of single-valuedness of R. So let $p(x{:}\,T){:}\, \exists! \, |x \; R|$ be the corresponding proof term, and take $fx := \iota(px)$. Then $fx = s(x; f)$ holds by (3.12).

For uniqueness, assume $gx = s(x; g|_{\prec x})$. Then by transfinite induction one proves $\forall(x{:}\,T :: gx = fx)$. ■

We may note that this theorem can be generalized to dependent types $U{:}\,\mathbf{Type}^T$. We shall do this in theorem 6.2: given a recursion step

$$s(x{:}\,T;\; h{:}\,\Pi(y{:} \prec x :: Uy)){:}\, Ux \;,$$

there is a unique $f{:}\,\Pi(T; U)$ satisfying (3.11). The proof goes analogous; one replaces $|R|{:} \subseteq T \times U$ by $R{:} \subseteq \Sigma(T; U)$.

3.5 Conclusion

We have seen our language *ADAM* at work in some inductive definitions. In the rest of the thesis, we develop a general theory for inductive types, based on categorical notions which we introduce in the next chapter.

Chapter 4

Categories and algebra

In this chapter we develop within *ADAM* the categorical framework for manipulating inductive types in the style advocated by Hagino [37], i.e. as initial objects in a category of (F, G)-algebras, for functors F and G. This differs from the style used in the branch of mathematics called *universal algebra*, where inductive types are formed as *monads*, being themselves functors with appropriate natural transformations. In the next three chapters we shall concentrate on particular induction and recursion rules as they fit in this categorical framework.

In section 4.1 we introduce basic categorical notions in our notation, in 4.2 we introduce the categorical view of algebras over some signature. In 4.3 inductive types appear as initial F, G-algebras, and in section 4.4 we take these algebras modulo equations, which are formed either on an (abstract) syntactic or semantic level. Section 4.5 looks at the relationship with well-founded relations, and 4.6 relates the initial algebra approach to the monads of universal algebra. In 4.7, we make a comparison with the framework of Algebraic Specification.

4.1 Categorical notions

Category theory provides a number of general purpose concepts and theorems. We will use some of the most basic notions, which we introduce here in the notation of *ADAM*. For a gentle introduction to some of the basic concepts, see Rydeheard [76].

4.1.1 Categories. First the big type of *categories*. There are several equivalent definitions in use. We take a category $\mathcal{C}$ to be a type, also named $\mathcal{C}$, of *objects* together with for any pair of objects X, Y a type (or set) $X \to Y$ <u>in</u> $\mathcal{C}$ of *morphisms* (or arrows) from X to Y, called its *hom-set*, and with associative arrow composition $\bar{\circ}$ and identity arrows Id_X. This is formalized below in the notation of paragraph 2.6.3.

<u>Define</u> $\mathcal{C}\colon \mathbf{Cat}_i\colon \mathbf{Type}_{i+1} :=: ($
    $\mathcal{C}\colon \mathbf{Type}_i;$
    $X, Y\colon \mathcal{C} \vdash (X \to Y\ [\ $<u>in</u>$\ \mathcal{C}])\colon \mathbf{Type}_i;$
    <u>Variables</u> $X, Y, Z, U\colon \mathcal{C};$
        $f\colon X \to Y,\ g\colon Y \to Z,\ h\colon Z \to U;$

$\mathsf{Id}_X : X \to X$,
$f \mathbin{\bar{\circ}} g : X \to Z$;
$\mathsf{Id}_X \mathbin{\bar{\circ}} f = f, \quad f \mathbin{\bar{\circ}} \mathsf{Id}_Y = f$,
$(f \mathbin{\bar{\circ}} g) \mathbin{\bar{\circ}} h = f \mathbin{\bar{\circ}} (g \mathbin{\bar{\circ}} h)$
)

Note that any universe of types together with functions as morphisms form a category,

$$\mathbf{TYPE}_i : \mathbf{Cat}_{i+1} \ := \ (\mathbf{Type}_i ; (\to); \mathsf{I}, (\bar{\circ})) \ .$$

4.1.2 Functors. A *functor* between two categories is a mapping of both objects and arrows that preserves identities and composition:

<u>Define</u> $F : (\mathcal{C} : \mathbf{Cat}) \to (\mathcal{D} : \mathbf{Cat})$:=: (
$F : \mathcal{C} \to \mathcal{D}$ <u>in</u> $\mathbf{TYPE}$;
<u>Variables</u> $X, Y, Z : \mathcal{C}$; $f : X \to Y$, $g : Y \to Z$;
$F : (X \to Y$ <u>in</u> $\mathcal{C}) \to (F.X \to F.Y$ <u>in</u> $\mathcal{D})$;
$F.\mathsf{Id}_X = \mathsf{Id}_{F.X}$,
$F.(f \mathbin{\bar{\circ}} g) = F.f \mathbin{\bar{\circ}} F.g$
)

We have identity functors I, and composition of functors denoted by reverse juxtaposition, so the type of categories with functors forms itself a (big) category:

$$\begin{aligned} & \mathsf{I}_{\mathcal{C}} : \mathcal{C} \to \mathcal{C} && := \ (\mathsf{I}; \mathsf{I}) \\ F : \mathcal{C} \to \mathcal{D},\ G : \mathcal{D} \to \mathcal{E} \ \vdash \quad & GF : \mathcal{C} \to \mathcal{E} && := \ (F \mathbin{\bar{\circ}} G;\ F \mathbin{\bar{\circ}} G) \\ & \mathbf{CAT}_i : \mathbf{Cat}_{i+1} && := \ (\mathbf{Cat}_i ; (\to); \mathsf{I}, (F, G :: GF)) \end{aligned}$$

4.1.3 Natural transformations. For categories $\mathcal{C}$, $\mathcal{D}$ and functors $F, G : \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\phi : F \mathrel{\dot{\to}} G$ is a family of arrows $\phi_{X:\mathcal{C}} : F.X \to G.X$ <u>in</u> $\mathcal{D}$ such that, for any $f : X \to Y$ <u>in</u> $\mathcal{C}$, one has

$$\phi_X \mathbin{\bar{\circ}} G.f = F.f \mathbin{\bar{\circ}} \phi_Y : \ F.X \to G.Y \ .$$

In relational notation, this is $(\phi_X, \phi_Y) \in F.f \ \to \ G.f$. Note the similarity with the typing $\phi_X : F.X \to G.X$!

As natural transformations are easily composed with each other,

$$\phi : F \mathrel{\dot{\to}} G,\ \psi : G \mathrel{\dot{\to}} H \ \vdash \quad \phi \mathbin{\bar{\circ}} \psi : F \mathrel{\dot{\to}} H \ := \ (X :: \phi_X \mathbin{\bar{\circ}} \psi_X) \ ,$$

the class of functors $F : \mathcal{C} \to \mathcal{D}$ with natural transformations as arrows forms the *functor category* $\mathcal{D}^{\mathcal{C}}$.

Natural transformations $\phi : F \mathrel{\dot{\to}} G$ can be composed with functors in two ways:

$$\begin{aligned} H : \mathcal{D} \to \mathcal{E} \ \vdash \quad & H.\phi : HF \mathrel{\dot{\to}} HG && := \ (X :: H.\phi_X) \\ J : \mathcal{B} \to \mathcal{C} \ \vdash \quad & \phi_{J.} : FJ \mathrel{\dot{\to}} GJ && := \ (X :: \phi_{J.X}) \end{aligned}$$

4.1.4 Product and exponent categories. The product $\Pi(D;\mathcal{C})$ of a family of categories is a category whose objects and arrows are tuples,

$$\begin{array}{rll} D\colon \mathbf{Type};\, \mathcal{C}\colon \mathbf{Cat}^D \vdash \quad \Pi(D;\mathcal{C}) & := (& \Pi(D;\mathcal{C}); \\ & & X \to Y := \Pi(d\colon D :: X_d \to Y_d \ \underline{\mathsf{in}}\ \mathcal{C}_d); \\ & & \mathsf{Id}_X := (d :: \mathsf{Id}_{X_d}), \\ & & f \mathbin{\bar{\circ}} g := (d :: f_d \mathbin{\bar{\circ}} g_d) \\ & &) \\ D\colon \mathbf{Type};\, \mathcal{C}\colon \mathbf{Cat} \vdash \quad \mathcal{C}^D\colon \mathbf{Cat} & := & \Pi(d\colon D :: \mathcal{C}) \end{array}$$

4.1.5 Dualization and initiality. For any category $\mathcal{C} = (\mathcal{C}; (\to); \mathsf{Id}, (\bar{\circ}))$, there is a *dual* or *opposite* category where all arrows are reversed,

$$\begin{array}{rcl} \mathcal{C}^{\mathsf{op}} & := & (\mathcal{C}; (\leftarrow); \mathsf{Id}, (\circ)) \\ \underline{\mathsf{where}}\ (X \leftarrow Y) & := & (Y \to X \ \underline{\mathsf{in}}\ \mathcal{C}) \ , \\ f \circ g & := & g \mathbin{\bar{\circ}} f \ . \end{array}$$

So $(X \to Y \ \underline{\mathsf{in}}\ \mathcal{C}^{\mathsf{op}}) = (Y \to X \ \underline{\mathsf{in}}\ \mathcal{C})$.

An *initial* object X of a category $\mathcal{C}$ is one for which, for any object $Y\colon\mathcal{C}$, there is a unique morphism from X to Y, noted $(\![X \to Y]\!)_{\mathcal{C}}$ as in [30], or rather just $(\![Y]\!)$ when $\mathcal{C}$ and X are evident.

$$\underline{\mathsf{Define}}\ X\colon \mathsf{Init}\,\mathcal{C} :=: (X\colon\mathcal{C};\ (\![Y\colon\mathcal{C}]\!)\colon !(X \to Y)\,)$$

A *final* (or *terminal*) object of $\mathcal{C}$ is an initial object of $\mathcal{C}^{\mathsf{op}}$.

The notion of initiality, and all its derived notions, can be weakened: a *weakly initial* object X of $\mathcal{C}$ is one for which, for any object $Y\colon\mathcal{C}$ there is a morphism from X to Y, not necessarily unique.

4.1.6 Product and sum objects. A category $\mathcal{C}$ is said to have (binary) *products* iff for any pair of objects, $B\colon\mathcal{C}^2$, we have an object $B_0 \times B_1\colon\mathcal{C}$ and two morphisms $\pi_i\colon B_0 \times B_1 \to B_i$, such that for any $X\colon\mathcal{C};\ p\colon (X, X) \to B$ there is a (unique) mediating morphism $\langle p\rangle = \langle p_0, p_1\rangle\colon X \to B_0 \times B_1$ characterized by

$$f = \langle p_0, p_1\rangle \ \Leftrightarrow\ \forall i\colon 2 :: f \mathbin{\bar{\circ}} \pi_i = p_i \ .$$

Using the algebraic terminology of section 4.2, one can say that $\langle p\rangle$ is a homomorphism

$$\langle p\rangle\colon (X; p) \to (B_0 \times B_1;\ \pi) \ ,$$

so that $(B_0 \times B_1;\ \pi)$ is the final object in the category of algebras $(X; p)$.

More generally, given a type N, category $\mathcal{C}$ has *products over* N iff for any tuple $B\colon\mathcal{C}^N$ there is a final object $(\Pi(N; B);\ \pi)$ in the category of algebras $(X\colon\mathcal{C};\ p\colon \Delta.X \to B)$. Here, $\Delta\colon \mathcal{C} \to \mathcal{C}^N$ is the *diagonal functor* $X \mapsto (i :: X)$.

Similarly, $\mathcal{C}$ is said to have (binary) *sums* (or coproducts) iff $\mathcal{C}^{\mathsf{op}}$ has binary products, noted $(B_0 + B_1; \sigma)$. Given $B{:}\,\mathcal{C}^2$; $X{:}\,\mathcal{C}$; $s{:}\, B \to (X, X)$, the mediating morphism is noted as

$$[s_0, s_1]{:}\,(B_0 + B_1;\ \sigma) \to (X; s)\ .$$

Note that the category of types has products and sums over all types in the category indeed, and the notations $\langle p\rangle$ and $[s]$ were already introduced in paragraphs 2.5.3 and 2.6.1 for the mediating morphisms in this category.

4.1.7 Subobjects. For an object $A{:}\,\mathcal{C}$ of any category, we can define the category of *subobjects of* A,

$$\begin{array}{rcl} \mathcal{P}_{\mathcal{C}}A & := & \{\,(H{:}\,C;\ r{:}\,H \to A)\,\} \\ (H; r) \to (H'; r')\ \underline{\mathsf{in}}\ \mathcal{P}_{\mathcal{C}}A & := & \{\,f{:}\,H \to H'\ \underline{\mathsf{in}}\ \mathcal{C}\ |{:}\ r = f \mathbin{\bar{\circ}} r'\} \end{array}$$

We define a preorder $(\leq)$ on subobjects, and any functor on $\mathcal{C}$ extends to a functor on $P_{\mathcal{C}}A$ (which preserves $(\leq)$):

$$\begin{array}{rcl} (H; r) \leq (H'; r') & := & \exists((H; r) \to (H'; r')\ \underline{\mathsf{in}}\ \mathcal{P}_{\mathcal{C}}A) \\ F.(H; r) & := & (F.H; F.r) \end{array}$$

For $\mathcal{C} := \mathbf{TYPE}$, a subobject $(H; r)$ represents the subset $\{z{:}\,H :: r.z\,\}{:}\,\mathcal{P}A$, and any subset $S{:}\,\mathcal{P}A$ is represented by a subobject $(S; \mathsf{I})$. Relation $(\leq)$ represents subset inclusion $(\subseteq)$, and extended functors preserve not only inclusions $S \subseteq S'$, but also *inclusion maps*:

$$F.(\mathsf{I}{:}\,S \to S')\ =\ \mathsf{I}{:}\,F.S \to F.S'\ . \tag{4.1}$$

Note that the latter property is not automatic for functors on the subset category $(\mathcal{P}A; (\to))$ or on **SET**, for inclusion maps are not identity arrows when $S \neq S'$.

4.2 Algebras and signatures

An *algebra* is essentially a tuple of types T_i called the *carriers* or *sorts*, where i ranges over some type N called the set of *sort names*, together with a tuple of functions ϕ_j called the *operations*, where j ranges over a type M of operation names. The domain and codomain (range) of the operations is specified by a signature.

We use a more liberal notion of signature than in the tradition of Algebraic Specification, see section 4.7. The *signature* of an algebra consists of the types N and M, and two functors $F, G{:}\,\mathbf{TYPE}^N \to \mathbf{TYPE}^M$, specifying for each operation its domain and codomain, so that $\phi_j{:}\,(F.T)_j \to (G.T)_j$.

The type of signatures is thus

$$\mathbf{Sign}{:}\,\mathbf{Type}\ :=\ \{\,(N, M{:}\,\mathbf{Type};\ F, G{:}\,\mathbf{TYPE}^N \to \mathbf{TYPE}^M)\,\}$$

and the type of algebras with a given signature $\Sigma = (N, M; F, G)$ is

$$\mathbf{Alg}\,\Sigma\ :=\ \{\,(T{:}\,\mathbf{Type}^N;\ \phi{:}\,F.T \to G.T\ \underline{\mathsf{in}}\ \mathbf{TYPE}^M)\,\}\ .$$

These are called Σ-*algebras*, or (F,G)-algebras when N and M are evident. As a special case, if $N = M$ and the codomain functor G is the identity, one speaks about F-*algebras*, and their type is noted $\mathbf{Alg}\,F$.

The type of *homomorphisms* between two Σ-algebras $(T;\phi)$, $(U;\psi)$ is the subtype of those arrows $f{:}\,T \to U$ (that is, tuples of functions $f_i{:}\,T_i \to U_i$ for $i{:}\,N$) that preserve the operations,

$$(T;\phi) \to (U;\psi) \;:=\; \{\, f{:}\,T \to U \ \underline{\mathsf{in}}\ \mathbf{TYPE}^N \;|{:}\; \phi \mathbin{\bar{\circ}} G.f = F.f \mathbin{\bar{\circ}} \psi \} \ .$$

Using the relational interpretation of '$\to$' (section 2.12.4), this condition for $f{:}\,T \to U$ to be a homomorphism reads

$$(\phi,\psi) \in F.f \to G.f \ .$$

Note the similarity with the type $\phi{:}\,F.T \to G.T$!

One has identity and composition of homomorphisms, so algebras and homomorphisms form a category $\mathbf{ALG}\,\Sigma$ for any $\Sigma{:}\,\mathbf{Sign}$.

This notion of algebra is easily generalized by replacing $\mathbf{TYPE}^N$ and $\mathbf{TYPE}^M$ by arbitrary categories $\mathcal{C}$, $\mathcal{D}$. Thus one has signatures $\Sigma = (\mathcal{C},\mathcal{D}{:}\,\mathbf{Cat};\ F,G{:}\,\mathcal{C} \to \mathcal{D})$ and Σ-algebras $(T{:}\,\mathcal{C};\ \phi{:}\,F.T \to G.T\ \underline{\mathsf{in}}\ \mathcal{D})$. This corresponds to the notion of F,G-dialgebra of Hagino [37].

Example 4.1 The ring of naturals with zero, one, addition and multiplication,

$$(\mathbb{N};\mathsf{K}\,0,\mathsf{K}\,1,(+),(\cdot)) \ ,$$

forms an algebra of signature

$$\Sigma \;:=\; (N := 1,\ M := 4;\ F.X := (1,1,X^2,X^2),\ G.X := (X,X,X,X)) \ . \tag{4.2}$$

This same algebra can also be given as an F-algebra, where $F.X := 1{+}1{+}X^2{+}X^2$, using the fact that the type of morphisms $A \to (X,X)\ \underline{\mathsf{in}}\ \mathbf{TYPE}^2$ is (naturally) isomorphic to $A_0 + A_1 \to X\ \underline{\mathsf{in}}\ \mathbf{TYPE}$.

4.3 Initial algebras, catamorphisms

Let $\Sigma = (\mathcal{C},\mathcal{D};F,G)$ be a (generalized) signature. According to the definition of initial objects in 4.1.5, a Σ-algebra $(T;\tau)$ is initial iff there exists a unique homomorphism (noted $(\![U;\psi]\!)$) from $(T;\tau)$ to any other Σ-algebra $(U;\psi)$, i.e.

$$(\![U;\psi]\!) : \ \ !(\,(T;\tau) \to (U;\psi)\,) \ .$$

Such homomorphisms, further abbreviated to $(\![\psi]\!)$, are called *catamorphisms* as in the Bird-Meertens formalism [57]. $(\![\psi]\!)$ satisfies the property that, for all $f{:}\,T \to U$,

$$\tau \mathbin{\bar{\circ}} G.f = F.f \mathbin{\bar{\circ}} \psi \ \Leftrightarrow\ f = (\![\psi]\!) \tag{4.3}$$

from which one has immediately the following *characteristic equation* of $(\![\psi]\!)$:

$$\tau \mathbin{\bar{\circ}} G.(\![\psi]\!) \;=\; F.(\![\psi]\!) \mathbin{\bar{\circ}} \psi \ .$$

Example 4.2 We can build an initial Σ-algebra where Σ is given by (4.2). Take the type A^* of strings over the alphabet $A := \{\mathtt{0}, \mathtt{1}, \mathtt{+}, \mathtt{*}\}$, and the operations

$$\begin{array}{rcl} z.0 & := & \mathtt{0} \\ o.0 & := & \mathtt{1} \\ x \oplus y & := & \mathtt{+}xy \\ x \otimes y & := & \mathtt{*}xy \end{array}$$

Let $T{:}\subseteq A^*$ be the least subset that is closed under these operations. Then by theorem 4.3 below, the algebra $(T; z, o, (\oplus), (\otimes))$ is an initial object of the category $\mathbf{ALG}\,\Sigma$. That is, it has a unique arrow f to any other algebra $(U; \psi)$ in this category, inductively defined by equations like $f.(\mathtt{+}xy) = \psi_2.(f.x, f.y)$ for $x, y{:}\in T$.

Thus, initial algebras, when they exist, are often thought of as sets of syntactic terms.

Example 4.3 From the primitive recursion principle (3.3) we get immediately the following *iteration principle*.

$$\frac{\begin{array}{l} U{:}\,\mathbf{Type} \\ b{:}\,U \\ g{:}\,U \to U \end{array}}{\exists! f{:}\,\mathbb{N} \to U :: f.0 = b \wedge f.\mathsf{s}\,n = g.(f.n)} \tag{4.4}$$

Now this says exactly that the natural numbers, with zero and successor, form an initial algebra $\Upsilon := (\mathbb{N}; \mathsf{K}\,0, \lambda\,\mathsf{s})$ of signature

$$\Sigma \ := \ (1, 2;\ F.X := (1, X),\ G.X := (X, X)) \ ,$$

for the equations say that f is a homomorphism from Υ to $(U; \mathsf{K}\,b, g)$. Equivalently, $(\mathbb{N}; [\mathsf{K}\,0, \lambda\,\mathsf{s}])$ is an initial F-algebra where $F.X := 1 + X$. We will often omit the brackets in such cases, writing just $(\mathbb{N}; \mathsf{K}\,0, \lambda\,\mathsf{s})$.

We will see in chapter 6 that the iteration principle in generalized form is equivalent to other recursion principles. (End of example)

Many other inductive types also form initial F-algebras for some F. In fact, we can take the notion of initial algebra in the category $\mathbf{TYPE}^N$ as our basic notion of inductive type.

Example 4.4 The algebra of lists of example 3.2,

$$(\mathsf{Clist}\,A; [\mathsf{K}\,\Box, +\!\!<]) \ ,$$

is the initial F-algebra where $F.X := 1 + A \times X$. And the algebra of rose trees and forests of example 3.5,

$$(\mathsf{RTree}\,A, \mathsf{Forest}\,A;\ (\surd), [\mathsf{K}\,\Box, (+\!\!<)]) \ ,$$

is the initial F-algebra where $F{:}\,\mathbf{TYPE}^2 \to \mathbf{TYPE}^2$ is

$$F.(X, Y) := (A \times Y, 1 + (X \times Y)) \ .$$

(End of example)

Identity is a catamorphism, and constructors are always isomorphisms here:

Theorem 4.1 (Lambek's lemma) *For any initial F-algebra $(T;\tau)$, Id is the catamorphism in $(T;\tau) \to (T;\tau)$.*

Proof. $\tau \mathbin{\bar{\circ}} \mathsf{Id} = F.\mathsf{Id} \mathbin{\bar{\circ}} \tau$, so Id is a homomorphism and hence the unique one. ■

The main use of this fact is in proving that a morphism $f\colon T \to T$ equals identity: $f = \mathsf{Id} \Leftrightarrow \tau \mathbin{\bar{\circ}} f = F.f \mathbin{\bar{\circ}} \tau$.

Theorem 4.2 *The constructor $\tau\colon F.T \to T$ of an initial F-algebra (in any category) is an isomorphism.*

Proof. We seek to define some $\delta\colon T \to F.T$. It should be a pre-inverse of τ:

$$\begin{array}{cll}
 & \delta \mathbin{\bar{\circ}} \tau = \mathsf{Id} & \\
\Leftrightarrow & \delta \mathbin{\bar{\circ}} \tau \in (T;\tau) \to (T;\tau) & \{\text{theorem } 4.1\} \\
\Leftrightarrow & \tau \mathbin{\bar{\circ}} (\delta \mathbin{\bar{\circ}} \tau) = F.(\delta \mathbin{\bar{\circ}} \tau) \mathbin{\bar{\circ}} \tau & \\
\Leftarrow & \tau \mathbin{\bar{\circ}} \delta = F.\delta \mathbin{\bar{\circ}} F.\tau & \\
\Leftrightarrow & \delta \in (T;\tau) \to (F.T;F.\tau) &
\end{array}$$

So taking for δ the catamorphism $(\![F.T;F.\tau]\!)$ will do. It remains to check that this δ is a post-inverse as well:

$$\begin{array}{cll}
 & \tau \mathbin{\bar{\circ}} \delta = \mathsf{Id} & \\
\Leftrightarrow & F.\delta \mathbin{\bar{\circ}} F.\tau = \mathsf{Id} & \{\tau \mathbin{\bar{\circ}} \delta = F.\delta \mathbin{\bar{\circ}} F.\tau \text{ above}\} \\
\Leftrightarrow & F.(\delta \mathbin{\bar{\circ}} \tau) = F.\mathsf{Id} & \{\text{functors}\} \\
\Leftarrow & \delta \mathbin{\bar{\circ}} \tau = \mathsf{Id} & \\
\Leftrightarrow & \mathsf{True} & \{\delta \text{ is a pre-inverse}\}
\end{array}$$

Thus, we can indeed use $\tau^{\cup} := (\![F.T;F.\tau]\!)$. ■

We now prove that an F-algebra being initial coincides with having no junk and no confusion. Let F be a functor in **TYPE**, extended to subsets (paragraph 4.1.7) and $(T;\tau)$ an F-algebra; we say that $(T;\tau)$ has *no confusion* iff τ is injective, i.e. $\tau \cdot \tau^{\cup} = \mathsf{Id}_{F.T}$, and *no junk* iff T is minimal, i.e. iff for all $S\colon\subseteq T$:

$$\tau \in F.S \to S \;\Rightarrow\; T \subseteq S \; . \tag{4.5}$$

Theorem 4.3 *F-algebra $(T;\tau)$ is initial iff it has no junk and no confusion.*

Proof. $\Rightarrow$: Let $(T;\tau)$ be initial, then theorem 4.2 comprises that τ is injective. Furthermore, assume $\tau \in F.S \to S$, then $(\![\tau]\!) \in (T;\tau) \to (S;\tau)$. But theorem 4.1 says $(\![\tau]\!) = \mathsf{I}$, so $T \subseteq S$.

$\Leftarrow$: Let $(U;\psi)$ be another F-algebra; we seek to define the unique homomorphism $f\colon (T;\tau) \to (U;\psi)$. The homomorphism condition $(\tau,\psi) \in F.f \to f$ gives rise to the inductive definition of f as being the smallest subset $f\colon\subseteq T \times U$ such that:

$$\forall g\colon\subseteq T \times U;\ g \text{ single-valued};\ (x,y)\colon\in F.g \,::\, g \subseteq f \Rightarrow (\tau.x, \psi.y) \in f \; .$$

It then follows from minimality of T and injectivity that f is total and single-valued, using (4.5) with

$$S := \{ x{:}\, T \mid: \exists! f[x] \} \ .$$

One clearly has that f is a homomorphism $f{:}\,(T;\tau) \to (U;\psi)$, and one proves by extensionality and minimality that f equals any other such homomorphism.

■

4.4 Algebras with equations

Some types that have an inductive character, can be seen as F-algebras, but not initial ones, because they violate the no-confusion condition. These can often be described by using a subcategory of algebras that satisfy certain equations.

Example 4.5 Considering example 3.4, the type of joinlists with its constructors, note that $(\mathsf{JList}\, A; [\mathsf{K}\,\Box, \Diamond, (+\!\!+)])$, is an F-algebra with $F.X := 1 + A + X^2$. But, as its constructors are not injective, we see that this algebra is not initial in $\mathbf{ALG}\,F$.

One can, however, form the subcategory of those F-algebras $(T; [\mathsf{K}\, e, f, (\oplus)])$ that satisfy the equations

$$\begin{aligned} s{:}\,T &\ \vdash\ \ e \oplus s = s,\ \ s \oplus e = s \\ s, t, u{:}\,T &\ \vdash\ \ (s \oplus t) \oplus u = s \oplus (t \oplus u). \end{aligned}$$

One may check that the algebra of joinlists is initial in this subcategory.

(End of example)

Now, what is the general form of a family of equations? The first idea is to view an equation as a pair of “syntactic” terms with free variables. We shall introduce syntactic terms in a rather abstract style, that allows for infinitary terms.

The second, even more abstract, view of equations is that an equation may be anything that establishes in a uniform way, for any algebra of the given signature, a binary relation on the carrier of the algebra. This idea gives rise to the notion of semantic equations.

We took the distinction between syntactic and semantic equations from Manes [54]. Fokkinga [30, chapter 5] introduced “transformers” and “laws” to describe equations; these are in fact a slight generalization of Manes’ semantic operations and equations.

4.4.1 Syntactic terms. Let $\Sigma = (N, M; F, G)$ be a signature, so $\mathcal{C} = \mathbf{Type}^N$; we can give an inductive definition of the sets of terms for this signature Σ. First, fix for each carrier index $i{:}\,N$ the set (type) V_i of (substitutable) variable names for this carrier. So V is an object in $\mathcal{C}$. We define the type of syntactic terms over V for carrier i, $(\mathsf{T}.V)_i$, as follows:

1. The sets of variable names are embedded in the sets of terms through $\eta_V{:}\, V \to \mathsf{T}.V$ $\underline{\mathsf{in}}$ $\mathbf{TYPE}^N$
2. For each operation index $j{:}\,M$, there is a syntactic operation τ_{Vj} building composite terms, so that $\tau_V{:}\, F\mathsf{T}.V \to G\mathsf{T}.V$ $\underline{\mathsf{in}}$ $\mathbf{TYPE}^M$

3. These terms are all distinct and there are no more, i.e., $(\mathsf{T}.V; \tau_V, \eta_V)$ is an initial algebra of signature

$$\Sigma' := (N, M+N; \langle F, \mathsf{K}\, V\rangle, \langle G, \mathsf{I}\rangle)$$

(for simplicity, we identify $\mathbf{Type}^{M+N}$ with $\mathbf{Type}^M \times \mathbf{Type}^N$ as in 2.12.5.)

4. The choice of $\mathsf{T}.V$ is uniform with respect to V, i.e., $\mathsf{T}{:}\, \mathcal{C} \to \mathcal{C}$ is a functor and τ and η are natural transformations $\tau{:}\, F\mathsf{T} \dot{\to} G\mathsf{T}$ and $\eta{:}\, \mathsf{I} \dot{\to} \mathsf{T}$.

Given any Σ-algebra $(X; \phi)$ and a valuation of the variables $v{:}\, V \to X$ $\underline{\text{in}}$ $\mathcal{C}$, one can interpret terms over V as denoting elements of X via the catamorphism $(\![\phi, v]\!){:}\, \mathsf{T}.V \to X$. Check that $(X; \phi, v)$ is another Σ'-algebra!

4.4.2 Syntactic equations. For a syntactic *equation* over signature Σ one needs two terms for one common carrier index i and over a common set of variables $V{:}\, \mathbf{Type}^N$. We require that each equation has its own set of (relevant) variables V, in order that a valuation has to specify values only for relevant variables.

Thus, an equation is an instance of

$$(V{:}\, \mathbf{Type}^N;\ i{:}\, N;\ s, t{:}\, (\mathsf{T}.V)_i)$$

where $(\mathsf{T}.V; \tau, \eta)$ is the term algebra over V. An algebra $(X; \phi)$ *satisfies* such an equation iff, for all valuations $v{:}\, V \to X$, the denoted elements are equal:

$$(\![\phi, v]\!)_i.s = (\![\phi, v]\!)_i.t \ ;$$

or, put differently, iff the relation

$$\{ v{:}\, V \to X :: (\![\phi, v]\!)^2_i.(s, t) \}$$

is contained in the equality relation $|{=}_{X_i}|$.

In general, one needs a family of equations, $(d{:}\, D :: (V_d; i_d; s_d, t_d))$, to delimit the required subcategory. For the example of joinlists above, taking $[\mathsf{K}\, e, f, (\oplus)] := \tau$, the three equations would be

$$\langle\ (1; e \oplus \eta.0, \eta.0),\quad (1; \eta.0 \oplus e, \eta.0),\quad (3; (\eta.0 \oplus \eta.1) \oplus \eta.2, \eta.0 \oplus (\eta.1 \oplus \eta.2))\ \rangle\ .$$

4.4.3 Semantic equations. In the category of types, a semantic equation should provide, for any Σ-algebra Φ, a relation $E\Phi{:} \subseteq \Phi^2$. Uniformity requires at least that any homomorphism $f{:}\, \Phi \to \Psi$ respects the relation, i.e. $f^2 \in E\Phi \to E\Psi$.

In an arbitrary category $\mathcal{C}$ with binary products, one can represent relations $R{:}\, A \sim B$ by subobjects $(H; r){:}\, \mathcal{P}_{\mathcal{C}}(A \times B)$. To avoid products, we will use *spans*:

$$(H{:}\, \mathcal{C};\ r{:}\, (H, H) \to (A, B)\ \underline{\text{in}}\ \mathcal{C}^2)\ .$$

In **TYPE**, such a span $(H; r)$ corresponds to the relation $\{h{:}\, H :: (r_0.h, r_1.h)\}$. We have a preorder $\leq$ on spans as on subobjects:

$$(H; r) \leq (H'; r')\ :=\ \exists m{:}\, H \to H' :: r = (m, m) \mathbin{\bar{\circ}} r'$$

The identity relation on $A{:}\,\mathcal{C}$ is represented by the span $(A; \mathsf{Id})$.

Now, we define a *semantic equation* (or *law*) E over a signature $\Sigma = (\mathcal{C}, \mathcal{D}; F, G)$ to be, for any Σ-algebra $(X;\phi)$, a span $E(X;\phi){:}\, X \sim X$ which is uniformly defined in the sense that

$$E(X;\phi) = (H.X; r(X;\phi))$$

for a functor $H{:}\,\mathcal{C} \to \mathcal{C}$ together with two natural transformations $r_0, r_1{:}\, HU \dot{\to} U$, where $U{:}\, \mathbf{ALG}\,\Sigma \to \mathcal{C}$ is the *forgetful functor* $(X;\phi) \mapsto X$. That is, one has

$$r{:}\, ((X;\phi){:}\, \mathbf{ALG}\,\Sigma \triangleright H.X \to X)^2$$

satisfying the promotion law that, for all Σ-algebras Φ, Ψ,

$$\forall f{:}\, \Phi \to \Psi \ \underline{\mathsf{in}}\ \ \mathbf{ALG}\,\Sigma :: r_j\Phi \,\bar{\circ}\, f = H.f \,\bar{\circ}\, r_j\Psi \ .$$

The r_j are called *semantic operations.* Algebra $(X;\phi)$ *satisfies* equation E iff $E(X;\phi)$ is contained in the identity relation, $E(X;\phi) \le (X;\mathsf{Id})$, which is equivalent to (4.6):

$$\begin{array}{rc}
 & (H.X; r(X;\phi)) \le (X;\mathsf{Id}) \\
\Leftrightarrow & \exists m{:}\, H.X \to X :: r(X;\phi) = (m,m) \,\bar{\circ}\, \mathsf{Id} \\
\Leftrightarrow & r_0(X;\phi) = r_1(X;\phi)
\end{array} \qquad (4.6)$$

One checks easily that our first uniformity requirement, that homomorphisms respect semantic equations, is satisfied indeed. The set of all algebras that satisfy a law forms a subcategory,

$$\mathbf{ALG}(\Sigma; E) \ := \ \{\, \Phi{:}\, \mathbf{ALG}\,\Sigma \mid{:}\ r_0\Phi = r_1\Phi\} \ .$$

Fokkinga [30, chapter 5] calls a natural transformation $r{:}\, HU \dot{\to} JU$ a *transformer of type* $(F,G) \to (H,J)$. The current notion of semantic operation (following Manes) corresponds to transformers of type $(F,G) \to (H,\mathsf{I})$. Transformers can be composed both horizontally and vertically, which gives rise to algebraic manipulations:

$$\begin{array}{rrcl}
r{:}\,(F,G) \to (H,J);\ r'{:}\,(F,G) \to (J,K) \vdash & r \,\bar{\circ}\, r' & := & (\Phi :: r\Phi \,\bar{\circ}\, r'\Phi) : \ (F,G) \to (H,K) \\
r{:}\,(F,G) \to (H,J);\ s{:}\,(H,J) \to (K,L) \vdash & sr & := & (\Phi :: s(U.\Phi; r\Phi)) : \ (F,G) \to (K,L)
\end{array}$$

We now show that semantic equations really generalize syntactic ones, and moreover, that a single semantic equation suffices.

Theorem 4.4 *Any family of syntactic equations corresponds to a single semantic equation, provided the base category has sums and exponents.*

Proof. Given a family of syntactic equations $(d{:}\,D :: (V_d; s_d, t_d))$, we have to find a law $(H;r)$ such that

$$\forall (d{:}\,D;\ v{:}\,V_d \to X :: (\![\phi, v]\!).s_d = (\![\phi, v]\!).t_d\,) \ \Leftrightarrow\ r_0(X;\phi) = r_1(X;\phi) \ . \qquad (4.7)$$

That's fairly simple; take

$$\begin{array}{rcl}
H.X & := & \Sigma(d{:}\,D :: X^{V_d}) \ ; \\
r(X;\phi) & := & (d;v) \mapsto ((\![\phi, \lambda v]\!).s_d, (\![\phi, \lambda v]\!).t_d) \ ,
\end{array}$$

then the righthand side of (4.7) unfolds to the lefthand side. ■

We shall see in section 8.3 that for certain signatures, called polynomial, initial $(\Sigma;E)$-algebras do generally exist.

4.5 Initial algebras related to well-founded relations

For bijective F-algebras, we will define a predecessor relation that is well-founded just when the algebra is initial. We cannot do this for algebras with equations, nor does an arbitrary well-founded relation correspond to some initial algebra.

Let $F\colon \mathbf{TYPE} \to \mathbf{TYPE}$ be a functor, extended to subsets, that preserves nonempty intersections,

$$\exists A;\ X_{i:A}\colon \mathcal{P}T \vdash \quad F.\bigcap(i\colon A :: X_i) = \bigcap(i\colon A :: F.X_i)\ ,$$

and $(T;\tau)$ a bijective F-algebra. We define the set of predecessors of $x\colon T$ as the least set $X\colon\subseteq T$ for which $x \in \tau[F.X]$. So we have a relation on T:

$$|\prec x| \ := \ \bigcap(X\colon\subseteq T \mid\colon x \in \tau[F.X])$$

We must check that this set satisfies $x \in \tau[F.X]$ itself:

$$\begin{array}{lll}
 & \forall x\colon T :: x \in \tau[F.|\prec x|] & \\
\Leftrightarrow & \forall y\colon F.T :: y \in F.\bigcap(X \mid\colon \tau.y \in \tau[F.X]) & \{\tau \text{ is surjective: } x = \tau.y\} \\
\Leftrightarrow & \forall y\colon F.T :: y \in F.\bigcap(X \mid\colon y \in F.X) & \{\tau \text{ is injective}\} \\
\Leftrightarrow & \forall y\colon F.T :: y \in \bigcap(X\colon\subseteq T;\ y \in F.X :: F.X) & \{F \text{ preserves } \bigcap\} \\
\Leftrightarrow & \mathsf{True} &
\end{array}$$

Theorem 4.5 *For $F, T, \tau, \prec$ as above, T has no junk iff $\prec$ is well-founded.*

Proof.

$$\begin{array}{lll}
 & T \text{ has no junk} & \\
\Leftrightarrow & \forall S\colon\subseteq T;\ \tau \in F.S \to S :: T \subseteq S & \\
\Leftrightarrow & \forall S\colon\subseteq T;\ \forall(x\colon T;\ x \in \tau[F.S] :: x \in S) :: T \subseteq S & \\
\Leftrightarrow & \forall S\colon\subseteq T;\ \forall(x\colon T;\ |\prec x| \subseteq S :: x \in S) :: T \subseteq S & \{\text{See below}\} \\
\Leftrightarrow & \prec \text{ is well-founded} &
\end{array}$$

It remains to prove $x \in \tau[F.S] \Leftrightarrow |\prec x| \subseteq S$.

$\Rightarrow$: Immediate, by definition of $\prec$.

$\Leftarrow$: When $|\prec x| \subseteq S$, then $\tau[F.|\prec x|\,] \subseteq \tau[F.S]$. As $x \in \tau[F.|\prec x|\,]$, we are done. ■

(Classically, $\prec$ being well-founded implies τ being surjective, so that requirement could be dropped.)

If one has an algebra with equations, τ is no longer injective so this construction of $\prec$ would not make sense. Indeed, take the initial algebra $(T; a, b, f)$ in the category of algebras

$$\{\,(T\colon \mathbf{TYPE};\ a, b\colon T,\ f\colon T \to T) \mid\colon f.a = f.b\}\ .$$

Then the set $|\prec f.a|$ of immediate predecessors of $f.a$ would have to be $\{\,a\,\}$ and $\{\,b\,\}$ at the same time.

Conversely, not all well-founded relations $(\prec) : \subseteq T^2$ correspond to some initial F-algebra with equations (or without). For, take the type $T := \{0, 1\}$ with the ordering $0 \prec 1$. A corresponding algebra should have some operation $f : T \to T$ with $f.0 = 1$. But then it would also have an element $f.1$ with $1 \prec f.1$, which T has not.

4.6 An aside: monads

Universal (categorical) algebra usually talks about inductive types in the form of monads, see Manes [54]. A nice introduction is also given by Lambek and Scott in [46, page 27–34]. As a side trip in our exposition, we summarize this concept here and establish its relationship to initial F-algebras.

A *monad* on a category $\mathcal{C}$ is a triple $(T; \eta, \mu)$ consisting of a functor and two natural transformations, typed by

$$\begin{array}{rcl} T & : & \mathcal{C} \to \mathcal{C} \\ \eta & : & \mathsf{I}_{\mathcal{C}} \dot{\to} T \\ \mu & : & TT \dot{\to} T \ , \end{array}$$

that satisfy the three equations

$$T.\eta \mathbin{\bar{\circ}} \mu = \mathsf{Id}_T = \eta_{T.} \mathbin{\bar{\circ}} \mu \ , \qquad \mu_{T.} \mathbin{\bar{\circ}} \mu = T.\mu \mathbin{\bar{\circ}} \mu \ . \tag{4.8}$$

If $\mathcal{C}$ is **TYPE**, one may think of $T.V$ as a type of structured values with sub-values drawn from V. Transformation η creates a value consisting of a single subvalue, and transformation μ merges a structured value with its structured subvalues..

Example 4.6 The monad of lists in the category of types is given by $T.X := X^*$, $\eta_X.x := \langle x \rangle$, and μ being the join function that concatenates a list of lists into a single list, so $\mu.\langle a, b\rangle = a +\!\!+ b$. To get some understanding of the monad equations (4.8), we apply them to respectively the list $\langle x, y\rangle$ and the list of lists of lists $\langle\langle a, b\rangle, \langle c, d\rangle\rangle$, and get:

$$\mu.\langle\langle x\rangle, \langle y\rangle\rangle = \langle x, y\rangle = \mu.\langle\langle x, y\rangle\rangle \ , \qquad \mu.\mu.\langle\langle a, b\rangle, \langle c, d\rangle\rangle = \mu.\langle \mu.\langle a, b\rangle, \mu.\langle c, d\rangle\rangle \ .$$

The following theorem gives a link with initial F-algebras.

Theorem 4.6 1. *For any functor F, if $(T.V; [\tau_V, \eta_V])$ is a uniformly defined initial $(F + \mathsf{K}\,V)$-algebra (so that τ and η are natural transformations), then define*

$$\mu_V : TT.V \to T.V \ := \ (\![T.V; [\tau_V, \mathsf{Id}_{T.V}]]\!)$$

to make a monad $(T; \eta, \mu)$.

2. *Conversely, any monad $(T; \eta, \mu)$ can be made into a $T{+}\mathsf{K}\,V$-algebra $(T.V; [\mu_V, \eta_V])$, but not necessarily into an initial one.*

Proof 1. Check that μ_V is correctly typed (its domain is an initial $F + \mathsf{K}(T.V)$-algebra, so see that $(T.V; [\tau_V, \mathsf{Id}_V])$ is an algebra of the same signature). As μ is clearly polymorphic, it is a natural transformation by the naturality theorem D.1. Then we have to check (4.8). By the catamorphism property we have the following.

$$\begin{aligned} \tau_{T.} \mathbin{\bar\circ} \mu &= F.\mu \mathbin{\bar\circ} \tau && (4.9)\\ \eta_{T.} \mathbin{\bar\circ} \mu &= \mathsf{Id}_{T.} && (4.10)\end{aligned}$$

The equality $\eta_{T.} \mathbin{\bar\circ} \mu = \mathsf{Id}_{T.}$ goes

$$\begin{aligned} & \eta_{T.} \mathbin{\bar\circ} \mu \\ = \quad & \sigma_1 \mathbin{\bar\circ} [\tau_{T.}, \eta_{T.}] \mathbin{\bar\circ} \mu \\ = \quad & \sigma_1 \mathbin{\bar\circ} (F.\mu + \mathsf{Id}_{T.}) \mathbin{\bar\circ} [\tau, \mathsf{Id}_{T.}] \quad \{\mu \text{ is a catamorphism}\} \\ = \quad & \mathsf{Id}_{T.} \mathbin{\bar\circ} \sigma_1 \mathbin{\bar\circ} [\tau, \mathsf{Id}_{T.}] \\ = \quad & \mathsf{Id}_{T.} \end{aligned}$$

Then to check $T.\eta \mathbin{\bar\circ} \mu = \mathsf{Id}_{T.}$, we use theorem 4.1 saying that $f = \mathsf{Id}_{T.V}$ iff

$$(F.f + \mathsf{Id}_V) \mathbin{\bar\circ} [\tau, \eta] \;=\; [\tau, \eta] \mathbin{\bar\circ} f$$

which is equivalent to

$$F.f \mathbin{\bar\circ} \tau = \tau \mathbin{\bar\circ} f \;\wedge\; \eta = \eta \mathbin{\bar\circ} f \;.$$

Instantiating this to $f := T.\eta \mathbin{\bar\circ} \mu$, we calculate first

$$\begin{aligned} & \tau \mathbin{\bar\circ} T.\eta \mathbin{\bar\circ} \mu \\ = \quad & FT.\eta \mathbin{\bar\circ} \tau_{T.} \mathbin{\bar\circ} \mu \quad \{\tau \text{ is natural}\} \\ = \quad & FT.\eta \mathbin{\bar\circ} F.\mu \mathbin{\bar\circ} \tau \quad \{(4.9)\} \\ = \quad & F.(T.\eta \mathbin{\bar\circ} \mu) \mathbin{\bar\circ} \tau \quad \{\text{functor}\} \end{aligned}$$

and second

$$\begin{aligned} & \eta \mathbin{\bar\circ} T.\eta \mathbin{\bar\circ} \mu \\ = \quad & \eta \mathbin{\bar\circ} \eta_{T.} \mathbin{\bar\circ} \mu \quad \{\eta \text{ is natural}\} \\ = \quad & \eta \mathbin{\bar\circ} \mathsf{Id}_{T.} \quad \{(4.10)\} \\ = \quad & \eta \end{aligned}$$

Finally we derive $\mu_{T.V} \mathbin{\bar\circ} \mu_V = T.\mu_V \mathbin{\bar\circ} \mu_V$ by proving that both sides are equal to $(\!|T.V; [\tau_V, \mu_V]|\!)$. Note that $f = (\!|T.V; [\tau_V, \mu_V]|\!)$ iff

$$\tau_{TT.V} \mathbin{\bar\circ} f = F.f \mathbin{\bar\circ} \tau_V \;\wedge\; \eta_{TT.V} \mathbin{\bar\circ} f = \mu_V \;.$$

So together we have four proof obligations to check.

$$\begin{aligned} & \tau_{TT.} \mathbin{\bar\circ} \mu_{T.} \mathbin{\bar\circ} \mu \\ = \quad & F.\mu_{T.} \mathbin{\bar\circ} \tau_{T.} \mathbin{\bar\circ} \mu \quad \{(4.9)\} \end{aligned}$$

$$\begin{array}{rcl} & = & F.\mu_{T.} \mathbin{\bar\circ} F.\mu \mathbin{\bar\circ} \tau \quad \{(4.9)\} \\ & = & F.(\mu_{T.} \mathbin{\bar\circ} \mu) \mathbin{\bar\circ} \tau \ , \end{array}$$

$$\begin{array}{rcll} & & \eta_{TT.} \mathbin{\bar\circ} \mu_{T.} \mathbin{\bar\circ} \mu & \\ & = & \mathsf{Id}_{TT.} \mathbin{\bar\circ} \mu & \{(4.10)\} \\ & = & \mu \ , & \end{array}$$

$$\begin{array}{rcll} & & \tau_{TT.} \mathbin{\bar\circ} T.\mu \mathbin{\bar\circ} \mu & \\ & = & FT.\mu \mathbin{\bar\circ} \tau_{T.} \mathbin{\bar\circ} \mu & \{\tau \text{ is natural}\} \\ & = & FT.\mu \mathbin{\bar\circ} F.\mu \mathbin{\bar\circ} \tau & \{(4.9)\} \\ & = & F.(T.\mu \mathbin{\bar\circ} \mu) \mathbin{\bar\circ} \tau & , \end{array}$$

$$\begin{array}{rcll} & & \eta_{TT.} \mathbin{\bar\circ} T.\mu \mathbin{\bar\circ} \mu & \\ & = & \mu \mathbin{\bar\circ} \eta_{T.} \mathbin{\bar\circ} \mu & \{\eta \text{ is natural}\} \\ & = & \mu \mathbin{\bar\circ} \mathsf{Id}_{T.} & \{(4.10)\} \\ & = & \mu & \end{array}$$

2. It is obviously a $T + \mathsf{K}\,V$-algebra. It need not be initial, as shown by the following counterexample. Let $\mathcal{C} := \mathbf{TYPE}$, $T := \mathsf{K}\,1$, let η and μ be the unique transformations to $\mathsf{K}\,1$, and $V := 1$. The carrier of the initial $T + \mathsf{K}\,V$-algebra is then 2, not 1. ■

4.7 Algebraic Specification

Let us make a few remarks about the relation between the approach sketched here and the tradition of Algebraic Specification (A.S.) as reviewed by Wirsing [87].

A.S. uses a notion of signature that is more restrictive than our categorical formulation, in that the argument and result types of operations must be sorts from the algebra itself. Such an algebra is called *plain* in 5.2.2. Furthermore, the sets of sorts, operations, and arguments of each operation are often required to be finite. Thus, a signature Σ consists of finite numbers or name sets n, m for the sorts and operations, and arities $(d_j, c_j){:}\, n^* \times n$ specifying the domain and codomain of the operation named by j :

$$\Sigma \ = \ (n, m{:}\,\mathbb{N};\ d{:}\,(n^*)^m, c{:}\,n^m)$$

The algebras of signature Σ are given by:

$$(T{:}\,\mathbf{Type}^n;\ \tau_{j:m}{:}\,\Pi(k{:}\,\#d_j :: T_{(djk)}) \to T_{(cj)})$$

A data type specification is given in both approaches by means of a signature with axioms. A.S. is more liberal in that the axioms may contain inequations, conditional equations, or even unrestricted axioms in first order logic. The main difference lies in the interpretation and derivational use of such a specification.

In our approach, we define the data type explicitly to be an initial algebra of the given signature, which needs to contain only the basic constructors and if necessary additional equations. The signature should be of a form that guarantees existence of such an algebra. We use full logic, even higher order if needed, to exploit initiality in defining derived functions and in deriving theorems. The logic may be restricted for specific purposes.

In A.S., one should distinguish between the formal application and the semantic interpretation of a specification. Formal derivations may only use the specified axioms, so that derived theorems hold for all algebras of the given signature. As a consequence, the signature must contain additional operations and axioms describing their behavior. A specification of lists for example must either contain operations yielding the head, tail, and length of a list, or a recursion operator. There is no formal guarantee that the signature axioms are consistent. The logic is often quite restricted, for example purely equational, allowing only finitary operations, and not containing function types.

One can prove consistency by meta-reasoning on the semantic level, where one distinguishes initial and terminal interpretations. For example, one can point out some of the operations as being constructors, prove that the sub-signature of these constructors has an initial model, and prove that this model has indeed operations that satisfy the remaining axioms. For terminal interpretations, see section 7.5.

A.S. has the advantage of admitting several alternative semantic concepts, like partial or continuous algebras. A simple logic is of course simpler to implement, to master, and to analyze.

4.8 Concluding remarks

Now that we have the basic categories of algebras, the stage is set for discussing the various forms that inductive and recursive definitions may take. The next chapter studies forms of inductive type definitions themselves, the subsequent one recursive function definitions over inductive types. The dual forms follow in chapter 7.

Chapter 5

Specifying inductive types

In the previous chapter, we introduced the notion of initial algebra. By viewing inductive types as initial algebras, we can define them up to isomorphism by giving the appropriate signature. However, not all algebra signatures have initial algebras. In this chapter we seek schemes for signatures that do have initial algebras, such that concrete inductive type definitions fit into the scheme.

We discuss only the abstract form that such schemes may take, not the concrete syntax that has to be defined in order to write signatures in a concrete language. We start with single inductive types, where an admissible functor F is specified by means of a family of sets. The generalization to mutually inductive types in section 5.2 diverges into several alternatives. In section 5.3 we have a look at producing admissible functors through inductive rules themselves. The use of positive type expressions may be seen as a special case of this.

The name '**Type**' as we use it stands for any universe $\mathbf{Type}_i$ from the hierarchy. But it is relevant that inductive types in one universe $\mathbf{Type}_i$ are understood to be initial algebras in all higher universes too, in order that one can use recursion to define other types.

5.1 Single inductive types

Suppose we want to introduce a single inductive type, $T\colon\mathbf{Type}$. In sections 4.3 and 4.4 we have seen how giving a signature consisting of a collection of constructors and equational axioms suffices to describe T. We discuss two ways to abstractly specify this collection.

We postpone the introduction of equations. So, in this section, we stipulate that objects built either by different constructors, or by one constructor from different arguments, are always different.

5.1.1 Operator domains

The most common approach in classical set theory (see for example Manes [54]) of giving a general scheme for initial algebras, is to identify for each constructor θ its arity as a cardinal number p: **Card** so that $\theta: T^p \to T$. Note that a constructor with arguments of other types, say $\theta: A \times T^p \to T$, can be regarded as a family of constructors $\theta_{a:A}: T^p \to T$.

Subsequently, all constructors with equal arity p have to be collected in a single family τ_p indexed by a set Ω_p, so we get the typing

$$\tau_p: (T^p \to T)^{\Omega_p}$$

or equivalently

$$\tau_p: \Omega_p \times T^p \to T\ .$$

Thus, the family of sets

$$\Omega: \mathbf{Set}^{\mathbf{Card}}$$

determines the number and arity of all constructors. It is called an *operator domain.* The pair $(T; \tau)$ forms an F-algebra where:

$$F.X := \Sigma(p: \mathbf{Card} :: \Omega_p \times T^p)$$

Unfortunately, taking a functor of this form does not guarantee that an initial F-algebra exists in **SET**. We shall see that it does exist when Ω is *bounded*, i.e. Ω_n is empty for all n above some cardinal m. Also, when Ω_0 is empty, then the empty algebra, consisting of $T := \emptyset$, τ_0 the empty tuple, and all other τ_p being tuples of (empty) functions $t \mapsto t_0$, is trivially initial. But:

Theorem 5.1 *When Ω is not bounded and Ω_0 is nonempty, then Ω does not have an initial algebra in* **SET**.

Proof. Suppose $(T; \tau)$ were initial; we shall define (with choice) an injection $f: \mathbf{Set} \to T$ by means of set-recursion (recursion over the wellfounded relation $\in$, (A.7)).
For any set s, let p be its cardinality and choose a surjection $\phi_s: p \to s$. Let q be the least cardinal $q \geq p$ such that Ω_q is nonempty, and choose $e_s: \Omega_q$. Then define

$$f.s \ := \ \tau_q e_s.t \ \underline{\mathsf{where}}\ t_{r:<q} := \begin{cases} f.(\phi_s.r) & \text{if } 0 \leq r < p \\ f.(\phi_s.0) & \text{if } p \leq r < q \end{cases}$$

Note that in the second case, $\phi_s.0$ is defined because if $q > p$ then Ω_p is empty so $p > 0$.
One checks easily that $f.s = f.s'$ implies $s = s'$. But such an injection cannot exist. ■

Objections against the use of operator domains in type theory are the following:

- As we just showed, extra conditions are needed to guarantee existence of an initial algebra with operator domain Ω.
- It may be unnatural or in constructive logic impossible to identify the arity as a cardinal number. To overcome this, one can use types rather than cardinals, so that $\Omega: \mathbf{Type}^{\mathbf{Type}}$.
- It may be unnatural to group constructors according to their arity. For example, an algebra may contain a family of constructors $\theta_{n:\omega}: T^n \to T$ which one would prefer to keep apart from other constructors for type T.

5.1.2 Operators with arity

An approach better fitted to type theory is to specify the arity of each constructor τ_j as a type. This is really the algebraic specification approach, but with finite sets replaced by possibly infinite types. The general form is:

$$\tau_{a:A}\colon T^{Ba} \to T \tag{5.1}$$

So, the signature is characterized by the type A, the index domain of the constructors, and the tuple of types B, Ba being the index domain of the constructor with index $a\colon A$. We call the pair $(A; B)\colon \mathsf{Fam}\,\mathbf{Type}$ an *operator specification*, and the corresponding signature is $(1, A; (X \mapsto (a :: X^{Ba})), (X \mapsto (a :: X)))$.

If we have an inductive type characterized by a list of constructors with arguments of other types as well, we can transform these into the form of (5.1) as follows. First, write the constructor types as:

$$\tau_{j:M}\colon \Sigma(x\colon A'_j :: T^{B'jx}) \to T \tag{5.2}$$

Next, the index j can be eliminated by transforming τ into (5.1) where we substitute

$$\begin{aligned} A\colon \mathbf{Type} &\;:=\; \Sigma(M; A') \\ B(j; x) &\;:=\; B'_j x \\ \tau_{(j;x)} &\;:=\; t \mapsto \tau_j.(x; (y :: t(j; y))) \end{aligned}$$

The types of τ in (5.2) and in (5.1) are isomorphic:

$$\begin{aligned} \Pi(j\colon M :: \Sigma(x\colon A'_j :: T^{B'jx}) \to T) &\;\cong\; \Pi(j\colon M :: \Pi(x\colon A'_j :: T^{B'jx} \to T)) \\ &\;\cong\; \Pi((j; x)\colon \Sigma(M; A') :: T^{B'jx} \to T) \end{aligned}$$

Example 5.1 In example 3.1 (natural numbers), we had two constructors, namely “zero” with 0 arguments, and “successor” with 1 argument. That is, $\tau_0\colon T^0 \to T$ and $\tau_1\colon T^1 \to T$, which results in the operator specification $(A := 2;\ B := (0, 1))$.

For example 3.2 (lists over E), we have originally

$$\begin{aligned} &\tau_0\colon 1 \to T \\ &\tau_1\colon E \times T \to T \end{aligned}$$

which is first transformed into

$$\begin{aligned} &\tau_0\colon \Sigma(0\colon 1 :: T^0) \to T \\ &\tau_1\colon \Sigma(e\colon E :: T^1) \to T \end{aligned}$$

and next into (5.1) where $A := 1 + E$ and $B := ((0; 0) :: 0 \mid (1; e) :: 1)$. We might use a labeled sum for A, for example:

$$A ::= \mathsf{empty} \mid \mathsf{cons}(E)\,;\ B := (\mathsf{empty} :: 0 \mid \mathsf{cons}(e) :: 1)$$

(End of example)

As a third step, one can replace the constructor family τ by a single constructor,

$$\tau\colon \Sigma(x\colon A :: T^{Bx}) \to T \ . \tag{5.3}$$

Note that we can define a functor $F\colon \mathbf{TYPE} \to \mathbf{TYPE}$ by:

$$\begin{array}{rcl} F.(U\colon \mathbf{Type}) & := & \Sigma(x\colon A :: U^{Bx}) \ , \\ F.(f\colon U \to V) & := & (x;t) \mapsto (x;(y :: f.t_y)) \ . \end{array}$$

We will call a functor *polynomial* iff it is (naturally) isomorphic with a functor of this form, as it is a sum of products. It is well known that for polynomial F, an initial F-algebra does always exist, and we will name it μF. Two different constructions of μF are given in chapter 8. A final F-coalgebra exists too and is named νF.

The type inductively defined by F is the carrier of the algebra μF, which is called μF too. This type is a fixed point of functor F, modulo isomorphism. Not all functors do have fixed points; for example the powerset functor (with $\mathcal{P}.f := X \mapsto \{x\colon\in X :: f.x\}$) cannot have one for cardinality reasons. But most type-construction principles, such as generalized sum and product, and taking fixed points itself, transform polynomial functors into polynomial functors. This is exploited in the next subsection. Also, for many theoretical purposes, it's simpler to deal with a functor than with A and B explicitly.

An operator specification that corresponds to an operator domain Ω is

$$(A := \Sigma(\mathbf{Card};\Omega);\ B_{(n;a):A} := n) \ .$$

Note that $\Sigma(\mathbf{Card};\Omega)$ is a set indeed if and only if Ω is bounded.

Conversely, given an operator specification $(A;B)$, a corresponding operator domain is

$$\Omega_p := \{\, a\colon A \mid\colon B_a \cong p\} \ .$$

This assumes that every type is isomorphic to some cardinal p, which requires the axiom of choice.

5.1.3 The wellordering of a single inductive type

The inductive type T characterized by operator specification $(A;B)$ is wellordered by the subterm relation $|{<}| := |{\prec}|^{(+)}$, where $\prec$ is the immediate subterm relation:

$$|{\prec}| \ := \ \{x\colon A;\ t\colon T^{Bx};\ y\colon Bx :: (t_y, \tau.(x;t))\} \ . \tag{5.4}$$

Type T is actually Martin-Löf's so-called wellordering type $\mathsf{W}(A,B)$ described in [56] where the constructor is called sup after *supremum* as, for $a\colon A$ and $t\colon \mathsf{W}(A,B)^{Ba}$, $\mathsf{sup}(a;t)\colon \mathsf{W}(A,B)$ is the supremum of the family $(y\colon Ba :: t_y)$ with respect to $<$.

5.2 Mutually inductive types

By *mutually inductive types* we mean a family of types $T_{i:N}$ for some index domain N, that are inductively generated by constructors τ whose arguments may come from any

T_i from the family. The cardinality of N is normally greater than 1. Algebra $(T;\tau)$ can again be viewed as an initial $(N,M;F,G)$-algebra.

A well-founded relation over a family of types is simply a well-founded relation over the sum type of the family, $\Sigma(N;T)$. Specifying a family of N types by simultaneous induction is somewhat more complicated. We give two alternatives for the abstract specification of mutually inductive types. The first is the generalization of the polynomial functor approach of 5.1.2 to exponential categories, but the required generalization of "polynomial" is not evident. The second alternative is the abstract rendering of plain (multi-sorted) algebra signatures, mentioned in 4.7.

5.2.1 Using an exponential category

We may use an endo-functor on the exponential category $\mathbf{TYPE}^N$, i.e. $F\colon \mathbf{TYPE}^N \to \mathbf{TYPE}^N$, that is polynomial as defined below. Then the tuple of mutually inductive types and their constructors appears as an initial algebra $\mu F = (T\colon \mathbf{Type}^N;\ \tau\colon F.T \to T)$. As arrows in an exponential category are tuples of arrows in the component categories, τ consists of functions $\tau_i\colon (F.T)_i \to T_i$ for each $i\colon N$.

We give two ways to characterize such polynomial functors F on $\mathbf{TYPE}^N$. In both ways, a type A_i indexes the constructors for type T_i.

The first way, perhaps the most straightforward one, lets type $B((i;a),j)$ (also written $Biaj$) index the arguments of type T_j that the constructor for type T_i indexed by $a\colon A_i$ shall get. Thus, for some

$$A\colon \mathbf{Type}^N;\ B\colon \mathbf{Type}^{\Sigma(N;A)\times N}\ ,$$

we let the constructors be typed by

$$\tau_{i:N;a:A_i}:\ \ \Pi(j\colon N :: T_j^{Biaj}) \to T_i\ . \tag{5.5}$$

Equivalently, $\tau\colon F.T \to T$ where F is given by

$$(F.X)_i\ :=\ \Sigma(a\colon A_i :: \Pi(j\colon N :: X_j^{Biaj}))\ .$$

With the second way, $B_i a\colon \mathbf{Type}$ indexes all arguments that the constructor τ_{ia} shall get, and $si(a;k)\colon N$ (or $siak$) indicates what the type of the argument indexed by $k\colon B_i a$ must be. Thus, for some

$$A\colon \mathbf{Type}^N;\ B\colon \Pi(i\colon N :: \mathbf{Type}^{Ai});\ s\colon \Pi(i\colon N :: N^{\Sigma(Ai;Bi)})\ ,$$

we let the constructors be typed by

$$\tau_{i:N;a:Ai}:\ \ \Pi(k\colon B_i a :: T_{(siak)}) \to T_i\ . \tag{5.6}$$

Equivalently, $\tau\colon F.T \to T$ where F is given by

$$(F.X)_i\ :=\ \Sigma(a\colon A_i :: \Pi(k\colon B_i x :: X_{(siak)}))\ .$$

This is the approach proposed by Petersson [72, section 4]. An advantage over the plain algebra approach below is that one gets a simpler case-distinction construct when no recursion is needed.

A context-free grammar corresponds to an algebra specification of form (5.6), where all A_i and $B_i a$ are finite:

Example 5.2 The rose trees and forests of example 3.5 can be described by a context-free grammar, given a nonterminal E:

$$\begin{array}{rcl} \mathit{RTree} & ::= & E \surd \mathit{Forest} \\ \mathit{Forest} & ::= & \Box \mid \mathit{RTree} \mathbin{+\!\!<} \mathit{Forest} \end{array}$$

This corresponds to an algebra (5.6), where

$$\begin{array}{rcl} N & := & 2; \\ A & := & (E, 2); \\ B & := & ((e :: 1),\ (0, 2)\,); \\ s & := & (((e; 0) :: 1),\ ((1; 0) :: 0 \mid (1; 1) :: 1)\,) \end{array}$$

Theorem 5.2 *Both characterizations* (5.5) *and* (5.6) *are reducible to each other, provided one has equality types.*

Proof. (5.6) $\Rightarrow$ (5.5): We can find an initial solution to (5.5) for some given $(A; B)$ by finding an initial solution to (5.6) with

$$B_i a := \Sigma(j\colon N :: Biaj);\ sia(j; k) := j\ ,$$

and then taking $\tau_{ia} := p \mapsto \tau_{ia}.((j; k) :: pjk)$.

(5.5) $\Rightarrow$ (5.6): Using equality, apply (5.5) with $Biaj := \{k\colon B_i a \mid\colon siaj = k\}$.

5.2.2 Plain algebra signatures

The choice above, that τ be an arrow in the category $\mathbf{TYPE}^N$, implied that each type T_i had its own (family of) constructor(s). Alternatively, one can give a single family of constructors and indicate for each one its codomain. This makes us return to the notion of *plain algebra* as we used in connection with Algebraic Specification in section 4.7, but here infinite families of constructors and arguments are allowed, rather than finite sequences. So given

$$M\colon \mathbf{Type};\ B\colon \mathbf{Type}^M;\ d\colon N^{\Sigma(M;B)};\ c\colon N^M\ ,$$

let τ be typed by

$$\tau_{j:M}\colon \Pi(k\colon B_j :: T_{(djk)}) \to T_{(cj)}\ . \tag{5.7}$$

So $(T; \tau)$ is not an F-algebra, but it is an initial $(N, M; F, G)$-algebra where

$$\begin{array}{rcl} (F.X)_j & := & \Pi(k\colon B_j :: X_{(djk)}) \\ (G.X)_j & := & X_{(cj)} \end{array}$$

Theorem 5.3 *Formulations* (5.6) *and* (5.7) *reduce to each other, provided one has equality types.*

Proof. (5.7) $\Rightarrow$ (5.6): Apply (5.7) with $M := \Sigma(N; A)$; $B := B$; $d(i;a)k := siak$; $c(i;a) := i$.

(5.6) $\Rightarrow$ (5.7): Here one needs equality. Apply (5.6) with $A_i := \{\, j\colon M \mid\colon cm =_N i\}$; $Bij := B_j$; $dijk := djk$. ■

This formulation is particularly suited to build types of proof trees for inductively defined relations:

Example 5.3 The type $P(n,m)$ of proof trees for $n < m$ as defined in example 3.7, together with appropriate constructors, is initial in the category of algebras

$$(P\colon \mathbf{Type}^{\mathbb{N}^2};\ \tau_{(0;n)}\colon 1 \to P(n, \mathsf{s}\, n),\ \tau_{(1;\, n,m)}\colon P(n,m) \to P(n, \mathsf{s}\, m))\ .$$

So here we have (5.7) with:

$$\begin{array}{rcl} N & := & \mathbb{N}^2 \\ M & := & \mathbb{N} + \mathbb{N}^2 \\ B & := & (0; n) :: 0 \mid (1;\ n, m) :: 1 \\ d & := & (1;\ n, m;\ 0) :: (n, m) \\ c & := & (0; n) :: (n, \mathsf{s}\, n) \mid (1;\ n, m) :: (n, \mathsf{s}\, m)\ . \end{array}$$

(End of example)

This example gives us an alternative way to define $(<)$: first define $(P;\tau)$ to be an initial algebra in the above category, and then define $n < m$ to hold iff $P(n,m)$ is nonempty. If one's calculus has inductive type definitions as a primitive, then this avoids the second order quantification occurring in example 3.7. Or, inductive relation definitions can be allowed as primitive themselves by stating that the category of predicates $T\colon \mathbf{PROP}^N$ with constructors typed by (5.7) has an initial object.

A drawback of this plain algebra approach is that it seems less suited for dualization; see paragraph 7.1.2.

5.3 Production rules for polynomial functors

Instead of requiring that the typings of operations have exactly a polynomial form like (5.1) or (5.3), the class of polynomial type expressions may be defined by production rules. This is based on the fact that the class of polynomial functors contains projections and constant functors, and is closed under taking sums, products and inductive types.

Let $\mathsf{PF}(N,M)\colon\subseteq (\mathbf{TYPE}^N \to \mathbf{TYPE}^M)$ be the subtype of polynomial functors as defined in subsection 5.2.1 and closed under isomorphism; $\mathsf{PF}(N)$ is $\mathsf{PF}(N,1)$; note that $\mathsf{PF}(N,M) \cong \mathsf{PF}(N)^M$. We state the following fact:

Theorem 5.4 *Type* $\mathsf{PF}(N)$ *is closed under the following rules:*

$$\begin{array}{rcll} i\colon N & \vdash & (X \mapsto X_i) \in \mathsf{PF}(N) & (5.8) \\ T\colon \mathbf{Type} & \vdash & (X \mapsto T) \in \mathsf{PF}(N) & (5.9) \\ F\colon \mathsf{Fam}\,\mathsf{PF}(N) & \vdash & \Sigma F \in \mathsf{PF}(N),\ \Pi F \in \mathsf{PF}(N) & (5.10) \\ F\colon \mathsf{PF}(N+M, M) & \vdash & (X \mapsto \mu(Y\colon \mathbf{Type}^M \mapsto F.(X,Y))) \in \mathsf{PF}(N,M), & (5.11) \\ & & (X \mapsto \nu(Y\colon \mathbf{Type}^M \mapsto F.(X,Y))) \in \mathsf{PF}(N,M) & (5.12) \end{array}$$

where ΣF abbreviates the sum functor $(X \mapsto \Sigma(i{:}\,\mathsf{Dom}\,F :: F_i.X))$, and similarly for ΠF.

To understand (5.11), let $F{:}\,\mathsf{PF}(M+N, M)$, and note that for $X{:}\,\mathbf{Type}^N$ we have that $(Y \mapsto F.(X,Y))$ is an endofunctor in $\mathbf{TYPE}^M$ indeed, so $\mu(Y \mapsto F.(X,Y))$ is an M-tuple of types. We write this tuple as $\mu F.X$. To see that $\mu F{:}\,\mathbf{Type}^N \to \mathbf{Type}^M$ is a functor, let $g{:}\,X \to X'$ be an arrow in $\mathbf{TYPE}^N$; we have to find an arrow $\mu F.g$. We take

$$\mu F.g{:}\,\mu F.X \to \mu F.X' \ := \ (\!|\mu F.X'; \phi|\!)$$

where

$$\phi{:}\,F.(X, \mu F.X') \to \mu F.X' \ := \ F.(g, \mathsf{I}) \mathbin{\bar{\circ}} \tau \ .$$

Proving that each $(\mu F.X)_j$, for $j{:}\,M$, is polynomial in X is quite complicated; we do not try it here.

5.3.1 Positive type expressions

One possibility for employing the above observation in a language design is to syntactically distinguish polynomial functors as type expressions E_X that are (strictly) *positive* in X. We say that a type expression is positive in X iff all free occurrences of type variables X are positive, i.e. not within the domain of any product or function type, nor in the parameter or argument of any user-defined operation.

This is done for example in Nuprl [18], where one can form the inductive type $\mu(X \mapsto E_X)$, and in Paulin-Mohring's extension [22, 68] of CC, where one can form the inductive type that is closed under a finite number of constructors with positive argument types.

A drawback is that in building E_X from X, one can neither apply user-defined type constructors (as we did in our example 3.5, where a user-defined type of lists was applied in defining the type of rose trees), nor apply other constants or variables to X. (Normally one can replace the constant with its definition.)

5.3.2 A type of polynomial functors

As an alternative, one might include the class of polynomial functors as a primitive type itself, say $\mathsf{PF}(N, M)$, which is closed under composition and the operations listed in theorem 5.4. This would allow user definitions of polynomial functors, and polymorphic operations to be instantiated to polynomial functors. One has to find easy notations, for example noting the projection functor $(X \mapsto X_i)$ by the name of parameter i.

5.4 Adding equations

To any of the three approaches above, one can add syntactic or semantic equations as in section 4.4. Semantic equations can be used if one's language admits such a semantic approach, otherwise one can develop concrete syntax for syntactic terms and equations.

5.5 Conclusion

We characterized the signatures that may be admitted for defining inductive types, in the following ways.

1. Using a bounded operator domain–this approach does not fit very well to type theory
2. Giving an arity for each constructor, by which means one defines a polynomial functor
3. For mutually inductive types, generalizing the notion of polynomial functor to an exponential category in either of two ways
4. Using generalized plain algebra signatures, which leaves a bit more freedom for the type definition
5. Using polynomial functors characterized by production rules, or by syntactic conditions on type expressions

Chapter 6

Recursors in constructive type theories

In this chapter we present several styles of introducing recursive functions on an inductive type. The inductive type may be characterized in two ways:

1. as a well-founded relation $(\prec) : \subseteq T^2$, for which we have recursion principle (3.11):

$$\frac{\begin{array}{l} U : \mathbf{Type} \\ s(x{:}\, T;\ h{:}\, U^{|\prec x|}) : U \end{array}}{\exists! f : U^T :: \forall x : T :: fx = s(x; f)}$$

2. or as an initial F-algebra $(T; \tau)$ (section 4.3), for which we have

$$\frac{\begin{array}{l} U : \mathcal{C} \\ \psi : F.U \to U \end{array}}{(\![U; \psi]\!) : !\{\, f : T \to U \mid : f \circ \tau = \psi \circ F.f \}} \ . \tag{6.1}$$

   This is sometimes called the iteration principle, and $(\![U; \psi]\!)$, or $(\![\psi]\!)$, is called a catamorphism [57]. If $\mathcal{C}$ is **TYPE**, we have a well-founded predecessor relation $\prec$ as given in section 4.5.

We shall first derive the recursion principle for initial F-algebras, then we derive in 6.2 the construction of dependent functions using either kind of inductive type characterization. In 6.3 we consider Mendler's style of recursion which comes closer in form to the unrestricted form of recursive equation, $f(\tau.y) = E_{f,y}$. Finally we shall see in 6.4 how each style of recursion generalizes to mutual recursion, and how an alternative form of mutual recursion may be useful.

6.1 Algebraic recursion, or paramorphisms

The function $\psi : F.U \to U$ in (6.1) that determines the value $(\![\psi]\!).x$ of a catamorphism cannot use the predecessors $y : \prec x$ directly but only the values $(\![\psi]\!).y$. To overcome this, the following recursion principle is derived, which corresponds more closely to (3.11). We formulate name it in the style of Meertens [58].

Theorem 6.1 (Paramorphisms) *An F-algebra $(T;\tau)$ in a category with binary products is initial iff one has*

$$\frac{\begin{array}{l} U:\mathcal{C} \\ \psi: F.(T\times U)\to U \end{array}}{\exists! f: T\to U :: f\circ\tau = \psi\circ F.\langle \mathsf{Id}, f\rangle} \tag{6.2}$$

The function f produced by this rule is called a paramorphism, *and is noted $[\![\psi]\!]$.*

Proof. $\Rightarrow$:

$$\begin{array}{cll} & f\circ\tau = \psi\circ F.\langle \mathsf{Id}, f\rangle & \\ \Leftrightarrow & \langle \mathsf{Id}, f\rangle\circ\tau = \langle \tau\circ F.\pi_0,\ \psi\rangle\circ F.\langle \mathsf{Id}, f\rangle & \{\text{functor properties}\} \\ \Leftrightarrow & \langle \mathsf{Id}, f\rangle = (\![T\times U;\ \phi]\!) & \{(6.1),\ \text{defining } \phi := \langle \tau\circ F.\pi_0, \psi\rangle\} \\ \Leftrightarrow & f = \pi_1\circ (\![\phi]\!) & \{\text{as } \mathsf{Id} = \pi_0\circ (\![\phi]\!)\} \end{array}$$

So we have:

$$[\![\psi]\!] \ := \ \pi_1\circ(\![\phi]\!) \ \underline{\mathsf{where}}\ \phi := \langle \tau\circ F.\pi_0, \psi\rangle\ . \tag{6.3}$$

$\Leftarrow$: Assume (6.2), and $\psi: F.U\to U$. We seek to find a unique homomorphism:

$$\begin{array}{cll} & f\circ\tau = \psi\circ F.f & \\ \Leftrightarrow & f\circ\tau = \psi\circ F.(\pi_1\circ\langle \mathsf{Id}, f\rangle) & \{\text{products}\} \\ \Leftrightarrow & f\circ\tau = (\psi\circ F.\pi_1)\circ F.\langle \mathsf{Id}, f\rangle & \{\text{functor}\} \\ \Leftrightarrow & f = [\![\psi\circ F.\pi_1]\!] & \{\ (6.2)\ \} \end{array}$$

■

See section 7.1 for dual catamorphisms called *anamorphisms*. Malcolm [52] and Fokkinga [30] give these and some more schemes of recursive functions with names like *zygomorphisms*, *mutumorphisms*, *prepromorphisms*, and *postpromorphisms*.

Example 6.1 For natural numbers, with $F.X := 1+X$ and $[\mathsf{K}\,0, \lambda\,\mathsf{s}] = \tau$, one can get the usual recursor $R_U: U\to(\mathbb{N}\to U\to U)\to\mathbb{N}\to U$ of typed lambda calculus, that satisfies

$$R_U a g 0 = a \qquad R_U a g (\mathsf{s}\,n) = g n (R_U a g n)\ ,$$

by taking for $R_U a g$ the paramorphism $[\![\psi]\!]$ where

$$\psi := [\mathsf{K}\,a, ((n,u)\mapsto gnu)] : \ 1+T\times U\to U\ .$$

(End of example)

It should be noted that, though $[\![\psi]\!]$ as defined by (6.3) satisfies the equation given by (6.2), the reduction rule that we actually get is somewhat different:

$$[\![\psi]\!]\circ\tau \ \texttt{=>}\ \pi_1\circ(\![\phi]\!)\circ\tau \ \texttt{=>}\ \pi_1\circ\phi\circ F.(\![\phi]\!) \ \texttt{=>}\ \psi\circ F.(\![\phi]\!)$$

6.2 Recursive dependent functions

We now specialize to the category of types. Usage of dependent types allows the transfinite induction principle (3.6) and transfinite recursion (theorem 3.7) for well-founded relations to be unified into a single dependent recursion principle. We have two similar principles for initial F-algebras, described in 6.2.2 and 6.2.3. In all cases, the principle does not need to state that the constructed function f is unique, for this is derivable by an auxiliary application of the very same principle!

6.2.1 Dependent recursion over a well-founded relation.

Theorem 6.2 *A relation $(\prec)\colon\subseteq T^2$ on a type T is well-founded iff one has:*

$$\frac{\begin{array}{l} U\colon \mathbf{Type}^T; \\ s(x\colon T;\ h\colon (z\colon\prec x \rhd U z))\colon U x \end{array}}{\exists f\colon \Pi(T;U) :: \forall x\colon T :: f x = s(x; f|_{\prec x})} \tag{6.4}$$

Proof. $\Rightarrow$: This runs parallel to theorem 3.7. First we inductively define a subset $R\colon\subseteq \Sigma(T;U)$ by

$$\forall x\colon T;\ h\colon (z\colon\prec x \rhd U z)\ ::\ \forall (z\colon\prec x :: (z; hz) \in R) \Rightarrow (x; s(x;h)) \in R\ .$$

Then one proves by transfinite induction (3.6) that R is single-valued in the sense that

$$\forall x\colon T :: \exists! u\colon U x :: (x;u) \in R\ ,$$

which proof we skip here. Letting p be the corresponding proof term, we can take $f x := \iota(px)$.

$\Leftarrow$: Rule (6.4) subsumes (3.6) by taking $Ux := Px$. Note also that it subsumes theorem 3.7 by substituting $Ux := U$. ■

6.2.2 Dependent recursion on an initial F-algebra. To formulate a rule for dependent recursion on an initial F-algebra, one has to find a way to encode the hypothesis of the induction step. This hypothesis should contain, for some $y\colon F.T$, for each predecessor $z\colon T$ of $\tau.y$ the function value $fz\colon Uz$. One possibility is to replace each predecessor z by the pair $(z; fz)\colon \Sigma(T;U)$, so the hypothesis becomes

$$h\colon F.\Sigma(T;U)\ .$$

A second possibility, which we shall consider in 6.2.3, is to add to y the tuple of all these function values fz. Pursuing the first possibility, we get the following.

Theorem 6.3 *An F-algebra $(T;\tau)$ in the category of types is initial iff the following rule holds.*

$$\frac{\begin{array}{l} U\colon \mathbf{Type}^T; \\ s(h\colon F.\Sigma(T;U))\colon U(\tau.(F.\lambda\mathsf{fst}.h)) \end{array}}{\exists f\colon \Pi(T;U) :: \forall y\colon F.T :: f(\tau.y) = s(F.(z \mapsto (z; fz)).y)} \tag{6.5}$$

Proof. $\Rightarrow$: assume that $(T;\tau)$ is initial, and that the rule premises hold. To get the dependent function f, we seek some $f'\colon T \to \Sigma(T;U)$ with $\mathsf{fst}(f'.x) = x$, so that we can take $fx := \mathsf{snd}(f'.x)$. We derive f' from the specification of f as follows.

$$\begin{array}{ll}
& \lambda\mathsf{fst} \circ f' = \mathsf{I} \ \wedge\ \forall(y :: \mathsf{snd}(f'.(\tau.y)) = s(F.f'.y)\,) \\
\Leftrightarrow & \lambda\mathsf{fst} \circ f' \circ \tau = \tau \circ F.(\lambda\mathsf{fst} \circ f') \ \wedge\ \forall(y :: \mathsf{snd}(f'.(\tau.y)) = s(F.f'.y)\,) \quad \{\text{th. } 4.1\} \\
\Leftrightarrow & \forall y :: f'.(\tau.y) = (\tau.(F.\lambda\mathsf{fst}.(F.f'.y)); s(F.f'.y)) \\
\Leftrightarrow & f' \circ \tau = (h \mapsto (\tau.(F.\lambda\mathsf{fst}.h); s(h))) \circ F.f' \\
\Leftrightarrow & f' = (\![h \mapsto (\tau.(F.\lambda\mathsf{fst}.h); s(h))]\!) \qquad \{\text{catamorphism}\}
\end{array}$$

(This proves also that f', and hence f, is unique.)

$\Leftarrow$: assuming (6.5), we prove that $(T;\tau)$ is initial by deriving the paramorphism rule (6.2). Given some U: **Type** and $\psi\colon F.(T \times U) \to U$, apply (6.5) to $Ux := U$; $s(h) := \psi.(F.((z;u) \mapsto (z,u)).h)$. Say this yields $f'\colon \Pi(T;U)$, then take $f.x := f'x$, which clearly satisfies the requirement $f \circ \tau = \psi \circ F.\langle \mathsf{I}, f\rangle$.

It remains to check that this f is unique. So, assuming that some $g\colon T \to U$ satisfies $g \circ \tau = \psi \circ F.\langle \mathsf{I}, g\rangle$ too, we prove that f and g are equal:

$$\begin{array}{ll}
& f = g \\
\Leftrightarrow & \forall x\colon T :: f.x = g.x \\
\Leftrightarrow & \exists\,\Pi(T;U') \qquad \underline{\text{where}}\ U'x := (f.x = g.x) \\
\Leftarrow & \exists\,\Pi(h\colon F.\Sigma(T;U') :: U'(\tau.(F.\lambda\mathsf{fst}.h)) \qquad \{(6.5)\} \\
\Leftarrow & \forall h\colon F.\Sigma(T;U') :: f.(\tau.(F.\lambda\mathsf{fst}.h)) = g.(\tau.(F.\lambda\mathsf{fst}.h)) \\
\Leftrightarrow & \psi \circ F.(\langle \mathsf{I}, f\rangle \circ \lambda\mathsf{fst}) = \psi \circ F.(\langle \mathsf{I}, g\rangle \circ \lambda\mathsf{fst}) :\ F.\Sigma(T;U') \to T \quad \{\text{property } f \text{ and } g\} \\
\Leftarrow & \langle \mathsf{I}, f\rangle \circ \lambda\mathsf{fst} = \langle \mathsf{I}, g\rangle \circ \lambda\mathsf{fst} :\ \Sigma(T;U') \to T \\
\Leftrightarrow & \forall x\colon T;\ f.x = g.x :: (x, f.x) = (x, g.x) \\
\Leftrightarrow & \mathsf{True}
\end{array}$$

■

As the $f\colon \Pi(T;U)$ in (6.5) is unique, we can give it a name: $\mu\_\mathsf{rec}(U;s)$. We remark that the proof for the '$\Leftarrow$'-part in theorem 6.3 contains *two* abstract applications of (6.5), one to construct a paramorphism and one to prove that it is unique.

Most typical examples of dependent recursion arise from inductive proofs: given a property P: $\mathbf{Prop}^T$ and an inductive proof of $\forall x :: Px$, the proof object (or tree) corresponding to this proof is given by a dependent recursion. A simple concrete example is the following.

Example 6.2 Consider the natural numbers as an initial $(\mathsf{K}\,1 + \mathsf{I})$-algebra, $(\mathbb{N}; [\mathsf{K}\,0, \lambda\mathsf{s}])$. We construct, for any $n\colon \mathbb{N}$, the function $f_n\colon \mathbb{N}^2 \to \mathbb{N}^n$ that transforms (a,b) into the n-tuple $(a, a\cdot b, \ldots, a\cdot b^{n-1})$. The recursion equations are

$$\begin{array}{lcl}
f_0 & = & (a,b) \mapsto () \\
f_{\mathsf{s}\,n} & = & (a,b) \mapsto (a, f_n.(a\cdot b, b))
\end{array}$$

Now, equation (6.5) says that for any s of appropriate type, there exists an f such that $f_0 = s(0;0)$, $f_{\mathsf{s}n} = s(1;n;f_n)$. So we just have to take

$$\begin{array}{rcl} U_n & := & \mathbb{N}^n \\ s(0;0) & := & (a,b) \mapsto () \\ s(1;n;f') & := & (a,b) \mapsto (a, f'.(a \cdot b, b)) \end{array}$$

and obtain an $f := \mu\_\mathsf{rec}(U;s)$ that satisfies our recursion equations.

6.2.3 Dependent recursion in Paulin style. The second possibility is to keep $y\colon F.T$ separate from the tuple of values fz. This is done in most languages with dependent recursion, where the inductive type is usually defined by a finite set of production rules. A formulation based on a functor F seems only possible for polynomial F, and requires us to extend F to operate on families of types and on dependent functions. This was done by Coquand and Paulin in [22], as follows.

Let $F.X = \Sigma(a\colon A :: X^{Ba})$. We extend F to operate on families $U\colon \mathbf{Type}^T$ and on dependent functions $f\colon \Pi(T;U)$, in such a way that:

$$\frac{U\colon \mathbf{Type}^T}{F'.U\colon \mathbf{Type}^{F.T}} \qquad \frac{f\colon \Pi(T;U)}{F.f\colon \Pi(F.T;F'.U)}$$

For $y\colon F.T$, $(F.f)y$ has to be the tuple of function values fz for all components $z\colon T$ of y, and $(F'.U)y$ is the type of this tuple. Thus:

$$\begin{array}{rcl} (F'.U)(a;t) & := & \Pi(y\colon Ba :: U(t_y)) \\ (F.f)(a;t) & := & (y :: f(t_y)) \end{array}$$

You may note that $F.\Sigma(T;U) \cong \Sigma(F.T;F'.U)$. Now, the rule becomes (we leave the proof to the reader):

$$\frac{\begin{array}{l} U\colon \mathbf{Type}^T \\ s(y\colon F.T;\ h\colon (F'.U)y)\colon U(\tau.y) \end{array}}{\exists f\colon \Pi(T;U) :: \forall y\colon F.T :: f(\tau.y) = s(y;(F.f)y)} \tag{6.6}$$

6.3 Mendler's approach

Mendler [59] introduces a somewhat different style of recursion over an initial F-algebra μF. The idea is here that in order to define a (dependent) function $f\colon \Pi(\mu F;U)$, one may assume that the function is already available on some subset $X\colon\subseteq \mu F$ while defining it on $F.X$. This gives a recursive equation for f that is simpler than the one appearing in (6.5). Mendler's thesis [59] uses a distinguished inclusion relation on types that is defined by separate production rules, and which we note $(\subseteq_{\mathsf{m}})\colon\subseteq \mathbf{Type}^2$. The rule looks like:

$$\frac{\begin{array}{l} U\colon \mathbf{Type}^T; \\ X\colon \mathbf{Type};\ X \subseteq_{\mathsf{m}} T;\ h\colon \Pi(X;U) \vdash s(h)\colon \Pi(y\colon F.X :: U(\tau.y)) \end{array}}{\exists f\colon \Pi(T;U) :: \forall y\colon F.T :: f(\tau.y) = sfy} \tag{6.7}$$

Note that, as in (6.4) and (6.5), rule (6.7) does not need to state that the constructed f is unique, for this can be derived by employing the dependency in the type of U, again taking $U'x := (f.x = g.x)$.

Example 6.3 Rule (6.7) yields the recursion equations of example 6.2 when we define s simply by:

$$\begin{aligned} sf(0;0) &:= (a,b) \mapsto () \\ sf(1;n) &:= (a,b) \mapsto (a, f(n).(a \cdot b, b)) \end{aligned}$$

(End of example)

Derivation of (6.7) requires a semantical analysis of the predicate $\subseteq_{\mathsf{m}}$, which we will not do here. But in his paper [60] Mendler replaced the inclusion relation $X \subseteq_{\mathsf{m}} T$ by an explicit function $i{:}\, X \to T$, and used only non-dependent functions. Correctness of this principle requires that the polymorphic dependency on X, h, and i be uniform in a certain way. This is covered by the *naturality* principle, described in appendix D for languages without dependent types. The resulting rule holds in any category $\mathcal{C}$ with binary products. Thus we get:

$$\frac{\begin{array}{l} U{:}\, \mathcal{C}; \\ X{:}\, \mathcal{C};\ i{:}\, X \to T;\ h{:}\, X \to U \vdash s_X(i,h){:}\, F.X \to U \\ \text{where } s \text{ is } \textit{natural} \end{array}}{\exists! f{:}\, T \to U :: f \circ \tau = s_T(\mathsf{Id}, f)} \tag{6.8}$$

where 's is natural' means that, for all $p{:}\, X \to X'$; $i'{:}\, X' \to T$; $h'{:}\, X' \to U$, one has

$$s_X(i' \circ p,\ h' \circ p) \ = \ s_{X'}(i', h') \circ F.p \ .$$

As indicated in appendix D, any lambda-definable s is natural. Therefore, this requirement can be omitted in calculi where one has only lambda-definable objects.

Example 6.4 We construct a non-dependent variant of the function of example 6.2, namely $f{:}\, \mathbb{N} \to \mathbb{N}^2 \to \mathsf{Clist}\, \mathbb{N}$ satisfying

$$\begin{aligned} f.0 &= (a,b) \mapsto \square \\ f.\mathsf{s}\, n &= (a,b) \mapsto a \mathbin{+\!\!<} f.(a \cdot b, b) \end{aligned}$$

This function is produced by (6.8) when we take for s:

$$\begin{aligned} s_X(i,h).(0;0) &:= (a,b) \mapsto \square \\ s_X(i,h).(1;x) &:= (a,b) \mapsto a \mathbin{+\!\!<} h.(a \cdot b, b) \end{aligned}$$

Theorem 6.4 *An F-algebra $(T;\tau)$ in any category with binary products is initial iff it satisfies (6.8).*

Proof. $\Rightarrow$: Let $(T;\tau)$ be initial, and assume an s that satisfies the premises of (6.8). We calculate the unique solution for f by showing that $\langle \mathsf{Id}, f\rangle$ is a homomorphism, as follows.

$$\begin{array}{cll}
 & f\circ\tau = s_T(\mathsf{Id}, f) & \\
\Leftrightarrow & f\circ\tau = s_T(\pi_0\circ\langle\mathsf{Id}, f\rangle,\ \pi_1\circ\langle\mathsf{Id}, f\rangle) & \{\text{products}\} \\
\Leftrightarrow & f\circ\tau = s_{T\times U}(\pi_0,\pi_1)\circ F.\langle\mathsf{Id}, f\rangle & \{s \text{ is natural}\} \\
\Leftrightarrow & \langle\tau,\ f\circ\tau\rangle = \langle\tau,\ s(\pi_0,\pi_1)\circ F.\langle\mathsf{Id}, f\rangle\rangle & \{\text{products}\} \\
\Leftrightarrow & \langle\mathsf{Id}, f\rangle\circ\tau = \langle\tau\circ F.\pi_0,\ s(\pi_0,\pi_1)\rangle\circ F.\langle\mathsf{Id}, f\rangle & \{\text{products, } F \text{ a functor}\} \\
\Leftrightarrow & \langle\mathsf{Id}, f\rangle = (\!|T\times U;\ \langle\tau\circ F.\pi_0,\ s(\pi_0,\pi_1)\rangle|\!) & \{\text{initiality}\} \\
\Leftrightarrow & f = \pi_1\circ(\!|T\times U;\ \langle\tau\circ F.\pi_0,\ s(\pi_0,\pi_1)\rangle|\!) & \{\text{fact below}\}
\end{array}$$

Writing $\phi := \langle\tau\circ F.\pi_0,\ s(\pi_0,\pi_1)\rangle$, we used the fact:

$$\begin{array}{cll}
 & \mathsf{Id} = \pi_0\circ(\!|\phi|\!) & \\
\Leftrightarrow & \pi_0\circ(\!|\phi|\!)\circ\tau = \tau\circ F.(\pi_0\circ(\!|\phi|\!)) & \{\text{theorem 4.1}\} \\
\Leftrightarrow & \pi_0\circ\phi\circ F.(\!|\phi|\!) = \tau\circ F.(\pi_0\circ(\!|\phi|\!)) & \{\text{catamorphism}\} \\
\Leftrightarrow & \tau\circ F.\pi_0\circ F.(\!|\phi|\!) = \tau\circ F.(\pi_0\circ(\!|\phi|\!)) & \{\text{definition } \phi\} \\
\Leftrightarrow & \mathsf{True} & \{F \text{ a functor}\}
\end{array}$$

$\Leftarrow$: Simple; given $\phi\colon F.U\to U$, apply (6.8) to $s_X(i,h) := \phi\circ F.h$ which is obviously natural. ■

As with paramorphisms, the actual reduction rule that we get when f is defined as $\pi_1\circ(\!|\phi|\!)$ is not $f\circ\tau \Rightarrow s(\mathsf{Id}, f)$, but rather:

$$f\circ\tau\ \ {=}{>}\ \ s(\pi_0,\pi_1)\circ F.(\!|\phi|\!)\ .$$

A Mendler rule for dependent functions that uses an explicit inclusion function can be given, but it appears to be too complicated to be practical:

$$\frac{\begin{array}{l} U\colon \mathbf{Type}^T;\\ X\colon\mathbf{Type};\ i\colon X\to T;\ h\colon\Pi(x\colon X :: U(i.x))\vdash s_X(i,h)\colon\Pi(y\colon F.X :: U(\tau.(F.i.y)))\\ \text{where } s \text{ is natural}\end{array}}{\exists f\colon\Pi(T;U) :: \forall y\colon F.T :: f(\tau.y) = s_T(\mathsf{I}, f)y} \tag{6.9}$$

It is probably not possible to derive rule (6.7) directly, because of the special role of the inclusion relation. Rather, one would have to prove that any construction made under an inclusion assumption $X\subseteq_{\mathsf{m}} T$ can be transformed into one using a function $i\colon X\to T$. We will not try to do so.

Parameter h in premise s in rules (6.7) and (6.8) gives access to the function value on immediate predecessors $x\colon X$ of the function argument $\tau.y$. Either rule can be strengthened to allow access to the function value on non-immediate predecessors. For (6.7), this is done by adding a hypothesis $X \subseteq_{\mathsf{m}} F.X$, for (6.8) by adding a parameter $d\colon X \to F.X$. Assuming that $\tau^{\cup}\colon T \to F.T$ is available, the latter rule becomes:

$$\frac{\begin{array}{l} U\colon \mathcal{C}; \\ X\colon \mathcal{C};\ i\colon X \to T;\ d\colon X \to F.X;\ h\colon X \to U \vdash s_X(i,d,h)\colon F.X \to U \\ \text{where } s \text{ is natural} \end{array}}{\exists! f\colon T \to U :: f \circ \tau = s_T(\mathsf{Id}, \tau^{\cup}, f)} \tag{6.10}$$

To derive this rule, one has to instantiate X not to $T \times U$, but to some type that encodes the function value on all predecessors of $y\colon F.X$. An initial $(F \times \mathsf{K}\,U)$-algebra, say $(V\colon \mathcal{C}; \kappa\colon F.V \times U \to V)$, would suit well, for then we can instantiate

$$\begin{array}{rcl} i\colon V \to T & := & (\!|\pi_0 \mathbin{\bar\circ} \tau|\!) \\ d\colon V \to F.V & := & \kappa^{\cup} \mathbin{\bar\circ} \pi_0 \\ h\colon V \to U & := & \kappa^{\cup} \mathbin{\bar\circ} \pi_1 \end{array}$$

Further proof details are left to the reader.

6.4 Recursors for mutual induction and recursion

Considering mutual recursion, we have to distinguish between mutually recursive functions on a single inductive type and recursive functions on a family of mutually inductive types.

6.4.1 Mutual recursion on a single inductive type. Regarding the first kind of mutual recursion, note that a tuple of functions on a single inductive type, e.g. $f_0\colon T \to B_0$, $f_1\colon T \to B_1$, is equivalent to a single function with a cartesian product as codomain, $f\colon T \to B_0 \times B_1$. Therefore, in a calculus that has cartesian products (finite or infinite), any recursion principle can be employed to construct mutually recursive functions.

6.4.2 Standard recursion on mutually inductive types. We modeled mutually inductive types (section 5.2) by several forms of initial algebras in an exponential category. All categorical recursion principles for initial algebras that we presented: (6.1), (6.2), and (6.7), can be interpreted in these categories, yielding arrows $f\colon T \to U$ $\underline{\text{in}}$ $\mathbf{TYPE}^N$. The recursors for dependent functions, (6.5) and (6.7), can easily be accommodated in an exponential category too; for example, rule (6.5) becomes

$$\frac{\begin{array}{l} U_{i:N}\colon \mathbf{Type}^{Tn}; \\ s_{i:N}(h\colon F_i.\Sigma(T;U))\colon U_i(\tau_i.(F_i.\lambda\mathsf{fst}^N.h)) \end{array}}{\exists f\colon \Pi(N;\Pi(T;U)) :: \forall i\colon N;\ y\colon F_i.T :: f_i(\tau_i.y) = s_i(F_i.(n' :: z \mapsto (z; f_i z)).y)}$$

where Σ en Π have to be lifted: $\Pi(T;U)_i := \Pi(T_i;U_i)$.

6.4.3 Liberal mutual recursion. Standard recursion on a family of N inductive types above requires that the recursive functions f consist of one function f_i for each type T_i. As an alternative recursion scheme, it is sometimes more convenient to index the functions over some type M that is different from N, and use a mapping $d(j{:}\, M){:}\, N$ to indicate the domain of function f_j. The defining equation for $f_j.(\tau_{dm}.y)$ may assume that, for every predecessor $x{:}\, T_i$ of y, the function results $f_{m'}.x$ for each $m'{:}\, M$ with $dm' = n$ are available. We name the type of this tuple of function results $U|_{=n}$, so $U|_=$ is the tuple of all these types. We dub the rule "liberal mutual recursion", as the function index type M is not fixed to be the index type N.

Theorem 6.5 (Liberal mutual recursion) *For any endofunctor F on $\mathbf{TYPE}^N$, an F-algebra $(T{:}\, \mathbf{TYPE}^N;\tau{:}\, F.T \to T)$ is initial iff rule (6.11) below holds for any $M{:}\, \mathbf{Type}$; $d{:}\, N^M$; $U{:}\, \mathbf{Type}^M$. We abbreviate:*

$$\begin{array}{rcl}
U|_= : \mathbf{Type}^N & := & (n :: \Pi(\{j{:}\, M \mid{:}\ dm = n\};\, U));\\
T_d{:}\, \mathbf{Type}^M & := & (m :: T_{(dm)})\\
F_d{:}\, \mathbf{TYPE}^M \to \mathbf{TYPE}^M & := & S \mapsto (m :: F_{(dm)}.S)\\
\tau_d{:}\, F_d.T_d \to T_d \ \underline{\mathsf{in}}\ \mathbf{TYPE}^M & := & (m :: \tau_{(dm)})\\
f{:}\, T_d \to U;\ i{:}\, N \vdash \quad f|_{=n}{:}\, T_i \to U|_{=n} & := & x \mapsto (m :: f_j.x)
\end{array}$$

$$\frac{\psi{:}\, F_d.(T \times U|_=\ ) \to U \ \underline{\mathsf{in}}\ \mathbf{TYPE}^M}{\exists! f{:}\, T_d \to U :: f \circ \tau_d = \psi \circ F_d.\langle \mathsf{Id}, f|_=\ \rangle} \tag{6.11}$$

Proof. $\Rightarrow$: We calculate the unique f that satisfies the specification, by translating it into an equation in category $\mathbf{TYPE}_N$:

$$\begin{array}{clr}
 & \forall m :: f_j \circ \tau_{dm} = \psi_j \circ F_{dm}.\langle \mathsf{Id}, f|_=\ \rangle & \\
\Leftrightarrow & \forall n;\, m;\, dm = n :: f_j \circ \tau_i = \psi_j \circ F_i.\langle \mathsf{Id}, f|_=\ \rangle & \{\text{introduce } n = dm\}\\
\Leftrightarrow & \forall n :: f|_{=n} \circ \tau_i = \psi|_{=n} \circ F_i.\langle \mathsf{Id}, f|_=\ \rangle & \{\text{definition } |_{=n}\}\\
\Leftrightarrow & f|_=\ = [\![\psi|_=\ ]\!] & \{(6.2)\}\\
\Leftrightarrow & \forall m :: f_j = x \mapsto ([\![\psi|_=\ ]\!]_{dm}.x)_j &
\end{array}$$

$\Leftarrow$: This rule subsumes the paramorphism principle (6.2) with $\mathcal{C} := \mathbf{TYPE}^N$, by instantiating $M := N$, $dm := m$. ∎

The rule of liberal mutual recursion made use of the equality type. However, in a calculus without explicit equality, one might still allow restricted forms of this rule by using syntactic checks for the equality $dm = n$, as the following example illustrates.

Example 6.5 Suppose we have a family of inductive types $T{:}\, \mathbf{Type}^{\mathbb{N}\times\mathbb{N}}$, and we wish to simultaneously define two families of recursive functions,

$$g_n{:}\, T_{nn} \to \mathbb{N}$$
$$h_{nm}{:}\, T_{nm} \to T_{mn}$$

This is possible by rule (6.11), taking

$$\begin{array}{rcl} M & := & \mathbb{N} + \mathbb{N}^2 \\ d(0;n) & := & (n,n) \\ d(1;n,m) & := & (n,m) \\ U(0;n) & := & \mathbb{N} \\ U(1;n,m) & := & T_{mn} \end{array}$$

After selecting a suitable ψ, the rule yields an f from which we can obtain $g_n := f_{(0;n)}$ and $h_{nm} := f_{(1;n,m)}$. Using the currying convention of subsection 2.12.5, we can write $(g,h) = f$. The characteristic equations become

$$\begin{array}{rcl} g_n \circ \tau_{nn} & = & \psi_{0n} \circ F_{0n}.\langle \mathsf{Id}, (g,h)|_= \rangle \\ h_{nm} \circ \tau_{nm} & = & \psi_{1nm} \circ F_{1nm}.\langle \mathsf{Id}, (g,h)|_= \rangle \end{array}$$

Inspection of the right-hand side of these equations reveals that the expressions which define $g_n.(\tau_{nn}.y)$ and $h_{nm}.(\tau_{nm}.y)$ may contain reference to y, to $h_{n'm'}.z$ for any immediate predecessor $z{:}\,T_{n'm'}$ of $\tau_{nm}.y$, and also to $g_{n'}.z$ when it happens that $m' = n'$. If one allows the latter only when m' and n' are equal by definitional equality, no explicit equality predicate is necessary.

6.5 Summary

This chapter completed our expedition of describing ordinary inductive types: chapter 2 introduced our language, chapter 4 our categorical machinery, chapter 5 surveyed schemes for inductive type definitions, and this chapter finished with describing the forms of recursion over an inductive type.

We described the following forms:

1. Catamorphisms (6.1), obtained directly from initiality.
2. Paramorphisms (6.2), which follow the scheme of simple or algebraic recursion.
3. Dependent (algebraic) recursion (6.5) and (6.6)
4. Mendler recursion (6.8), using a quantifiction over types. Any of the recursors 1–3 above can be formulated in Mendler form, giving six combinations.
5. Liberal mutual recursion (6.11). Any of the six combinations above can be generalized to either standard or liberal mutual recursion, giving twelve forms of mutual recursion.

Furthermore, any of these may appear either in a weak form, giving just a typing rule and an equality (or reduction) rule, or in a strong form, giving also a uniqueness condition. The latter is only possible when the calculus has an explicit equality predicate.

The strong forms of the recursors and the weak form of dependent recursion are all equivalent (with respect to extensional equality), with these remarks:

- Equality types are required in order to formulate and derive the general form of liberal mutual recursion.
- Equality types are also required to derive any strong recursor from weak dependent recursion.
- Generalized sums are required to derive strong or weak dependent recursion.

"Equivalent" means here, that any application of one recursor can be translated into an application of the other recursor that satisfies the same equation. However, the actual reduction behavior may differ.

Similarly, the weak forms of the catamorphism, paramorphism, Mendler, and liberal mutual recursion rules are equivalent, with the same remarks.

Chapter 7

Co-inductive types

We have noted in section 4.3 and 4.4 that the categories $\mathbf{TYPE}^N$ have initial F-algebras and initial $(F;E)$-algebras. Quite remarkably, the same holds for the opposite categories $(\mathbf{TYPE}^N)^{\mathsf{op}}$. Initial algebras in the opposite category are final co-algebras in the original category, and may be called *co-inductive types*. While the elements of initial F-algebras are like trees with finite branches only, elements of final F-coalgebras are like trees with possibly infinitely deep branches. Final coalgebras are introduced in 7.1.

In 7.2 we have a look at the various shapes which the unique homomorphism to a final coalgebra may take. We present the interesting example of infinite processes.

Section 7.3 shows how all recursion constructs that do not involve dependent functions dualize.

While adding equations to an initial F-algebra has the effect of identifying some trees (elements of the F-algebra), we prove in 7.4 that adding equations to a final F-coalgebra has the effect of removing some trees from the algebra. Section 7.5 contrasts this with the Algebraic Specification idea of final or terminal interpretation of equations over an initial algebra.

7.1 Dualizing F-algebras

Let $\mathcal{C}$ be the category $\mathbf{TYPE}^N$ for some type N. An F-algebra in $\mathcal{C}^{\mathsf{op}}$, say $(X{:}\,\mathcal{C}^{\mathsf{op}};\ \phi{:}\, F.X \to X \ \underline{\mathsf{in}}\ \mathcal{C}^{\mathsf{op}})$, is, by definition of $^{\mathsf{op}}$, a (Id, F)-algebra $(X{:}\,\mathcal{C};\ \phi{:}\, X \to F.X \ \underline{\mathsf{in}}\ \mathcal{C})$, which is also called an F-*coalgebra*. In section 8.2 we shall see that, if F is polynomial, then there exists a final F-coalgebra, which we name νF. Thus, for any F-coalgebra $(X;\phi)$ there is a unique homomorphism $f{:}\,(X;\phi) \to (U;\delta)$, with characteristic equation:

$$\delta \circ f \ = \ F.f \circ \phi\ . \tag{7.1}$$

This homomorphism is noted '$[\!(X;\phi)\!]$', and called an *anamorphisms*, as devised by Erik Meijer. Note that δ is an isomorphism by theorem 4.2, as $(U;\delta)$ is initial in $\mathcal{C}^{\mathsf{op}}$.

Theorem 7.1 *A final F-coalgebra $(U;\delta)$ contains an initial F-algebra $(V;\delta^{\cup})$.*

Proof. Take $V{:}\,\mathcal{P}U := \bigcap(X \mid{:}\ \delta^{\cup} \in F.X \to X)$. Then $(V;\delta^{\cup})$ is initial by theorem 4.3 (no junk and no confusion). ■

Whereas, in **TYPE**, elements of initial algebras are thought of as well-founded trees, final coalgebras do also contain all non-wellfounded trees, having infinitely deep branches.

Example 7.1 (Infinite lists) The algebra of streams or infinite lists $(E\infty; \langle \mathsf{hd}, \mathsf{tl}\rangle)$, as specified in example 3.8, is the final $(E\times)$-coalgebra.

To define the list $e\colon \mathbb{N}\infty$ of all even numbers, we let $f.n$ be the arithmetic sequence $\langle n, n+2, \ldots\rangle$, using an equation of the shape (7.1).

$$\langle \mathsf{hd}, \mathsf{tl}\rangle.(f.n) \;=\; (n, f.(n+2)) \;=\; (\,(\mathsf{I}_{\mathbb{N}} \times f) \circ \langle \mathsf{I}, (+2)\rangle\,).n$$

So $f := [\![\mathbb{N}; \langle \mathsf{I}, (+2)\rangle]\!]$. Then we take $e := f.0$.

Let us now define a bijection $g\colon E^\omega \leftrightarrow E\infty$. We wish

$$\begin{aligned} g.e \;&=\; f.0 \;\underline{\text{where}} \\ &\quad\;\; \delta.(f.n) = (e_n, f.(n+1)) \end{aligned}$$

where δ is $\langle \mathsf{hd}, \mathsf{tl}\rangle$, so we define

$$g.e \;:=\; [\![\mathbb{N};\, (n \mapsto (e_n, n+1))]\!].0 \;.$$

For the inverse, we define

$$g^\cup.l \;:=\; (n :: \mathsf{hd}.(\mathsf{tl}^{(n)}.l)) \;.$$

We prove $g \circ g^\cup = \mathsf{I}$:

$$\begin{array}{lcl} & g.(g^\cup.l) = l & \\ \Leftrightarrow & f.0 = l \;\underline{\text{where}}\; f := [\![\mathbb{N};\, (n \mapsto ((g^\cup.l)_n, n+1))]\!] & \{\text{definition } g\} \\ \Leftarrow & f = (n \mapsto \mathsf{tl}^{(n)}.l) & \{\text{generalization}\} \\ \Leftrightarrow & \forall n :: \delta.(\mathsf{tl}^{(n)}.l) = (\mathsf{I} \times (n \mapsto \mathsf{tl}^{(n)}.l)).((g^\cup.l)_n, n+1) & \{\text{anamorphism}\} \\ \Leftrightarrow & \forall n :: \mathsf{hd}.(\mathsf{tl}^{(n)}.l) = (g^\cup.l)_n \;\wedge\; \mathsf{tl}.(\mathsf{tl}^{(n)}.l) = \mathsf{tl}^{(n+1)}.l & \{\text{pairing}\} \\ \Leftrightarrow & \mathsf{True} & \end{array}$$

And $g^\cup \circ g = \mathsf{I}$:

$$\begin{array}{lcl} & g^\cup.(g.e) = e & \\ \Leftrightarrow & \forall n :: \mathsf{hd}.(\mathsf{tl}^{(n)}.(f.0)) = e_n \quad \underline{\text{where}}\; f := [\![\mathbb{N};\, (n \mapsto (e_n, n+1))]\!] & \\ \Leftarrow & \forall n, m :: \mathsf{hd}.(\mathsf{tl}^{(n)}.(f.m)) = e_{n+m} & \\ \Leftarrow & \forall (m :: \mathsf{hd}.(\mathsf{tl}^{(0)}.(f.m)) = e_{m+0}\,) & \\ & \wedge\, \forall (n, m :: \mathsf{hd}.(\mathsf{tl}^{(n+1)}.(f.m)) = \mathsf{hd}.(\mathsf{tl}^{(n)}.(f.(m+1))\,) & \{\text{induction on } n\} \\ \Leftrightarrow & \mathsf{True} & \{\text{definition } f\} \end{array}$$

7.1.1 Final F-algebras. Looking at final F-algebras in any category with final objects is not very interesting, for these are always that object (for $\mathbf{TYPE}^N$, the trivial unit algebra, all carriers having exactly one element). Dually, initial F-coalgebras are the initial object (for $\mathbf{TYPE}^N$, the empty algebra).

7.1.2 Dualizing plain algebras. In subsection 5.2.2 we gave an alternative scheme for mutually inductive types, plain algebras. This scheme seems less suited for dualization.

The operations were typed by:

$$\tau_{j:M} \colon \Pi(k \colon B_j :: T_{(djk)}) \to T_{(cj)} \ .$$

A dual algebra $(U;\delta)$ would be typed by

$$\delta_{j:M} \colon T_{(cj)} \to \Sigma(k \colon B_j :: T_{(djk)}) \ ,$$

for Σ is the generalized product constructor in the category $\mathbf{TYPE}^{\mathsf{op}}$. This is no longer a plain algebra. It seems not to be very useful because neither operations δ nor $\delta^{\cup}$ admit multiple arguments, and each operation having alternative single result types that are unrelated seems difficult to make sense of.

7.2 Anamorphism schemes

Equation (7.1) for an anamorphism $f \colon (X;\phi) \to (U;\delta)$ can take on a somewhat different shape if it is combined with pattern matching, and furthermore if F happens to be the sum of other functors.

First we eliminate δ on the left-hand side:

$$f \;=\; \delta^{\cup} \circ F.f \circ \phi \ . \tag{7.2}$$

Now, suppose that X can be split up over n patterns $\xi_i(x' \colon X'_i) \colon X$, so that:

$$\forall x \colon X :: \exists! i \colon < n;\ x' \colon X'_i :: x = \xi_i x' \ .$$

Then (7.2) may be written as a list of equations

$$f \circ \lambda\xi_i \;=\; \delta^{\cup} \circ F.f \circ \phi'_i \tag{7.3}$$

where $\phi'_i := \phi \circ \lambda\xi_i$.

Next, if F is a finite sum, $F.Y = \Sigma(j \colon < m :: F'_j.Y)$, one has actually a list of constructors

$$\tau_j \colon F'_j.U \to U \;:=\; \delta^{\cup} \circ \sigma_j$$

or equivalently $[\tau] := \delta^{\cup}$. Then, if some ϕ'_i has the shape $\sigma_j \circ \phi''_i$, the respective equation (7.3) reduces to

$$f \circ \lambda\xi_i \;=\; \tau_j \circ F'_j.f \circ \phi''_i \ . \tag{7.4}$$

Example 7.2 (Processes) Co-inductive types offer elegant models for (indefinitely proceeding) processes, viewing these as incremental stream transformers. For example, the type of simple processes that can input data values from channels $i \colon I$ and output data values over channels $o{:}\,O$, can make finitary nondeterministic choices and silent steps, and can halt, is the final coalgebra $(\mathsf{Sproc}(I,O);\delta)$ with $\mathsf{Sproc}(I,O){:}\,\mathbf{Type}$ and

$$\begin{array}{rcl}
\delta{:}\,\mathsf{Sproc}(I,O) \to & (& I \times (\mathsf{Data} \triangleright \mathsf{Sproc}(I,O)) \\
& + & O \times \mathsf{Data} \times \mathsf{Sproc}(I,O) \\
& + & \Sigma(n{:}\,\mathbb{N} :: \mathsf{Sproc}(I,O)^n) \\
& + & \mathsf{Sproc}(I,O) \\
& + & 1 \\
&) & \\
[\mathsf{outp},\mathsf{inp},\mathsf{choose},\mathsf{step},\mathsf{halt}] & := & \delta^{\cup}
\end{array}$$

Here, $\mathsf{inp}.(i,s)$ represents a process that requires an input from channel i and continues with process si; $\mathsf{outp}.(o,c,s)$ outputs value c over channel o and continues with s; $\mathsf{choose}.(n;s)$ chooses some arbitrary $k{:} < n$ and continues with sk, and $\mathsf{halt}.0$ represents the process that just halts. $\mathsf{step}.s$ represents a process that performs some internal steps without external action, and continues with s.

The choose alternative may be omitted if one represents a nondeterministic process by a set of deterministic processes.

Now, consider the following program, written in CSP notation (Communicating Sequential Processes, Hoare [41]). It reads a number x from input channel A, and then repeats x times reading a number y from A and outputting y^2 to channel B.

$$A?x;\ *|[\ x > 0 \longrightarrow A?y;\ B!(y^2);\ x := x - 1\ ]|\ .$$

To give the process defined by this program, we first define X to be a suitable state space. Between each input or output action there has to be a distinguished state.

$$X{:}\,\mathbf{Type}\ :=\ 1 + \mathbb{N} + \mathbb{N}^2$$

The mapping of states to processes, $f{:}\,X \to \mathsf{Sproc}(I,O)$, is then given by the following equations of the form (7.4).

$$\begin{array}{rcl}
f.(0;0) & = & \mathsf{inp}.(A,(x :: f.(1;x))) \\
f.(1;x) & = & \underline{\mathsf{if}}\ x > 0\ \underline{\mathsf{then}}\ \mathsf{inp}.(A,(y :: f.(2;x,y)))\ \underline{\mathsf{else}}\ \mathsf{halt}.0 \\
f.(2;x,y) & = & \mathsf{outp}.(B,y^2,f.(1;x-1))
\end{array}$$

The intended process is now $f.(0;0)$, as the initial state is $(0;0){:}\,X$.

Processes that operate in an environment that may be changed by the process itself can be modeled by final coalgebras in the category $\mathbf{TYPE}^N$, where type N is the set of possible environment states. This makes it possible to let the range of possible actions depend on the current environment state. (End of example)

Within a domain theory of partial and infinite objects (section 9.1), and hence in programming languages with partial objects, processes can be represented as continuous functions from lazy streams to lazy streams, which are also called *stream transformers*. Dybjer and Sander [25] represented a system of concurrent processes using the stream approach. One needs a "network transfer function" to combine the separate agents into a single stream transformer. They used a functional calculus in which types are basically predicates so that final coalgebras can be obtained as the greatest fixed points of monotonic predicate transformers (as in section 10.2).

We refer to Malcolm [52] for some more examples and properties of final coalgebras, called "terminal data structures" there.

7.3 Dual recursion

We have a look at how the derived non-dependent recursors of chapter 6 dualize. Dependent recursors are not dualizable, as the dual of a dependent function would have to be something where the type of the argument depends on the function result, which is impossible.

All the following dual recursors can be transformed into a pattern-matching scheme like 7.4.

7.3.1 Algebraic recursion. As the scheme of algebraic recursion in section 6.1 works for any category with products, and products in category $\mathcal{C}^{\mathsf{op}}$ are sums in $\mathcal{C}$, we have the following corollary.

Corollary 7.2 *If $(T;\delta)$ is a final F-coalgebra in a category $\mathcal{C}$ with binary sums, then*

$$\frac{\begin{array}{l} U{:}\,\mathcal{C} \\ \psi{:}\,U \to F.(T+U) \end{array}}{\exists! f{:}\,U \to T :: \delta \circ f = F.[\mathsf{Id}, f] \circ \psi} \tag{7.5}$$

Example 7.3 Let $(T;\delta) := \nu(X \mapsto E \times X^2)$ be the coalgebra of non-wellfounded labeled binary trees. Given a label $e{:}\,E$ and a tree $t{:}\,T$, we can construct a tree s with

$$\delta.s = (e, s, t)$$

by taking $s := f.0$ where $f{:}\,1 \to T$ is, using (7.5), the unique solution to:

$$\delta \circ f = (\mathsf{I}_E \times [\mathsf{I}_T, f]^2) \circ \mathsf{K}(e, ((1;0),(0;t))) \ .$$

So we apply (7.5) to $\psi := \mathsf{K}(e, ((1;0),(0;t)))$.

7.3.2 Mendler recursion. Mendler's recursor dualizes too (only for non-dependent functions of course) to:

$$\frac{\begin{array}{l} U{:}\,\mathbf{Type}; \\ X{:}\,\mathbf{Type};\ i{:}\,T \to X;\ h{:}\,U \to X \vdash s_X(i,h){:}\,U \to F.X \\ \text{where } s \text{ is } \textit{natural} \end{array}}{\exists! f{:}\,U \to T :: \delta \circ f = s_T(\mathsf{I}, f)} \tag{7.6}$$

Example 7.4 The same $s{:}\,T$ as in example 7.3 is obtained by taking for $f{:}\,1 \to T$ the unique solution to:

$$\delta \circ f = \mathsf{K}(e, f.0, \mathsf{I}.t) \ .$$

So we apply (7.6) to $s_X(i,h) := \mathsf{K}(e, h.0, i.t)$. Check that this is well-typed.

7.3.3 Liberal mutual recursion. Rule (6.11) dualizes trivially, yielding an $f{:}\,U \to T_d$ $\underline{\mathsf{in}}$ $\mathbf{TYPE}^M$ that satisfies $\delta_d \circ f = F_d.[\mathsf{Id}, f_=] \circ \psi$.

7.4 Dual equations

We shall now look at the effect of adding equations to a final coalgebra, by applying the categorical notion of an algebra with equations, as described in paragraph 4.4.3, to the dual category $\mathcal{C}^{\mathsf{op}}$ where $\mathcal{C} := \mathbf{TYPE}^N$. This is not to be confused with the terminal interpretation of equations in algebraic specification, described in section 7.5.

A law E in the category $\mathcal{C}^{\mathsf{op}}$ is a functor $H{:}\,\mathcal{C} \to \mathcal{C}$ with two natural transformations $E_j{:}\,U \mathrel{\dot{\to}} HU$, where U is the forgetful functor $(X;\phi) \mapsto X$. An F-coalgebra $(X;\phi)$ satisfies law E when $E_0(X;\phi) =_{X\to H.X} E_1(X;\phi)$, that is, when for all $i{:}\,N$; $x{:}\,X_i$,

$$E_0(X;\phi)_i.x \;=_{H_i.X}\; E_1(X;\phi)_i.x\;. \tag{7.7}$$

Now, we prove that a final algebra with equations is obtained from a final algebra without equations by removing all elements that do not satisfy the equations and the elements that contain these elements.

Theorem 7.3 *If, for a functor $F{:}\,\mathcal{C} \to \mathcal{C}$ and law E, there exists a final F-coalgebra $(T;\delta)$, then the final $(F;E)$-coalgebra exists as well and is the greatest subalgebra of $(T;\delta)$ that satisfies law E, namely coalgebra $(T';\delta)$ where:*

$$T' \;:=\; \bigcup(X{:}\subseteq T \mid{:}\; \forall i{:}\,N;\ x{:}\in X_i :: \delta_i.x \in F_i.X \wedge (7.7)\,)$$

where the union and subset on tuples should be taken pointwise, and the functor is extended to subsets (par. 4.1.7).

Proof. We consider only the case $N = 1$, so we can forget about the subscripts i. $(T';\delta)$ is clearly an $(F;E)$-coalgebra. Let Ψ be another one; we must exhibit a unique homomorphism $\Psi \to (T';\delta)$ $\underline{\mathsf{in}}$ $\mathbf{ALG}(F;E)$.
We have a unique F-homomorphism $f{:}\,\Psi \to (T;\delta)$, because $(T;\delta)$ is final. As E is a natural transformation and Ψ satisfies E, the range $f[\Psi]$ satisfies E too. So $f \in \Psi \to (T';\delta)$. As any other homomorphism to $(T';\delta)$ is a homomorphism to $(T;\delta)$ as well, it must equal f. ■

Actually, no really useful example of a final $(F;E)$-coalgebra is known to me.

7.5 Terminal interpretation of equations

In the tradition of "algebraic specification" [87], which we sketched in 4.7, one distinguishes between the initial and terminal interpretation of an algebraic specification. A specification consists of an algebra signature Σ together with a set of equations and sometimes inequations ($\neq$). Normally, only finitary signatures are allowed.

The *initial interpretation* of a specification is just the initial object in the category of all Σ-algebras that satisfy the equations. This corresponds to our notion of initial algebra with equations. Any inequations are superfluous: if they are not satisfied in this initial algebra, the specification is inconsistent.

The *terminal interpretation* of a specification is different, though. This is the final object in the category of all "term-generated" Σ-algebras that satisfy both the equations and inequations. An algebra Φ is *term-generated* iff the unique homomorphism from the initial Σ-algebra to Φ is surjective. Put otherwise, the terminal interpretation is the initial Σ-algebra modulo the greatest equivalence relation that satisfies the (in-)equations, when this exists. Thus, these terminal algebras are definitely not co-inductive types.

Presence of inequations is essential here; otherwise the terminal interpretation would trivially be the unit algebra. Normally, an algebraic specification includes some standard specification of one or more types whose elements are required to be distinct, like booleans, characters, or integers.

7.6 Conclusion

We described the dualization of inductive types, which are types with infinitely deep objects, for example indefinitely proceeding processes. The rigid form of recursive equation, needed to construct objects of these types, could be transformed into a more natural definition scheme.

The notion of algebra with equations can be dualized too, and is meaningful in the category of types, but this dual form of equation does not seem to be very useful.

Co-inductive types are not to be confused with the terminal interpretation of an algebraic specification.

Chapter 8

Existence of inductively defined sets

We characterized inductive types in chapter 5 by means of polynomial functors; now we shall show that for a polynomial functor F, in set theory, an initial F-algebra and a final F-coalgebra indeed exist. The axioms of set theory are listed in section A.1. We outline two alternative proofs. Manes [54, p. 74] lists a number of works that present a rigorous construction of (an equivalent of) initial F-algebras for polynomial functors F.

The first proof, in section 8.1, is the standard construction of an initial F-algebra, by taking the transfinite limit $F^{(u)}.\emptyset$ for some ordinal number u.

Section 8.2 gives a more elementary proof, after an idea of Kerkhoff [45], which works in type theory too, and which dualizes yielding final co-algebras.

In section 8.3, we add equations to an initial F-algebra. For adding equations to a final F-coalgebra, we refer to section 7.4.

8.1 Using transfinite ordinal induction

Given a polynomial F, we are going to define a transfinite sequence of sets so that its limit gives an initial F-algebra.

For κ (*kappa*) a cardinal number, we define

$$Y \subseteq_\kappa X \;:=\; Y \subseteq X \wedge \mathsf{card}\, Y \le \kappa \;.$$

A functor $F\colon \mathbf{SET} \to \mathbf{SET}$ is *bounded* iff it has some *rank*. It has rank κ (or is κ*-based*) iff for all $X\colon \mathbf{Set}$,

$$F.X = \bigcup(Y\colon \subseteq_\kappa X :: F.Y) \;.$$

Note that bounded functors are monotonic: they preserve $(\subseteq)$.

Theorem 8.1 1. *Any polynomial functor F is bounded.*

2. *For any bounded functor F there exists an initial F-algebra $(T; \tau)$. Actually, τ can be the identity, so that $F.T = T$.*

Proof 1. Let $F.X = \Sigma(x\colon A :: X^{Bx})$. We calculate a rank κ of F.

$$\begin{array}{rc}
 & \Sigma(x\colon A :: X^{Bx}) \subseteq \bigcup(Y\colon \subseteq_\kappa X :: \Sigma(x\colon A :: Y^{Bx})) \\
\Leftrightarrow & \Sigma(x\colon A :: X^{Bx}) \subseteq \Sigma(x\colon A :: \bigcup(Y\colon \subseteq_\kappa X :: Y^{Bx})) \\
\Leftarrow & \forall x\colon A ::\ X^{Bx} \subseteq \bigcup(Y\colon \subseteq_\kappa X :: Y^{Bx}) \\
\Leftrightarrow & \forall x\colon A;\ u\colon X^{Bx} :: \exists Y\colon \subseteq_\kappa X ::\ u \in Y^{Bx} \\
\Leftarrow & \forall x\colon A;\ u\colon X^{Bx};\ Y := \{i\colon Bx :: u_i\} ::\ Y \subseteq_\kappa X\ \wedge\ u \in Y^{Bx} \\
\Leftrightarrow & \forall x\colon A;\ u\colon X^{Bx} ::\ \mathsf{card}\{i\colon Bx :: u_i\} \le \kappa\ \wedge\ \mathsf{True} \\
\Leftarrow & \kappa = \mathsf{max}(x\colon A :: \mathsf{card}\, Bx)
\end{array}$$

2. Let the rank of F be κ. For some ordinal u, we define a transfinite sequence of sets $T\colon \mathbf{Set}^{\mathsf{s}\,u}$ by ordinal recursion:

$$\begin{array}{rcll}
T_0 & := & \emptyset & \\
T_{\mathsf{s}\,n} & := & F.T_n & \\
T_v & := & \bigcup(w\colon < v :: T_w) & \text{for limit ordinals } v
\end{array}$$

Now, T_u is the limit of the whole sequence, and if it satisfies $F.T_u \subseteq T_u$ then $(T_u; \mathsf{I})$ is an F-algebra. So we try:

$$\begin{array}{rl}
 & F.T_u \subseteq T_u \\
\Leftrightarrow & \bigcup(Y\colon \subseteq_\kappa T_u :: F.Y) \subseteq T_u \\
\Leftrightarrow & \forall(Y\colon \subseteq_\kappa T_u :: F.Y \subseteq T_u)
\end{array}$$

To prove this condition, assume $Y\colon \subseteq_\kappa T_u$. Requiring that u is a limit ordinal (requirement 1), we have that for all $y\colon \in Y$ there is some $v_y < u$ with $y \in T_{v_y}$. Hence $Y \subseteq T_{\mathsf{max}\, v}$, and:

$$F.Y \subseteq F.T_{\mathsf{max}\, v} = T_{\mathsf{s}(\mathsf{max}\, v)}\ . \tag{8.1}$$

Now note that if $\kappa < u$ (2) then:

$$\mathsf{card}(\mathsf{Dom}\, v) = \mathsf{card}\, Y \le \kappa < u\ .$$

So if u is a regular cardinal (3) (see section A.4), then:

$$\mathsf{max}\, v \le \sum v < u\ ,$$

hence $\mathsf{s}(\mathsf{max}\, v) < u$ as u is a limit. Combined with (8.1) and monotony of T, we obtain our present goal, $F.Y \subseteq T_u$.

So we are done if we find a u that is a limit ordinal (requirement 1), that is bigger than κ (2), and that is a regular cardinal (3). Taking $u := \mathsf{max}\langle \kappa^+, \omega \rangle$ satisfies all this. (κ^+ is the least regular cardinal greater than κ, see section A.4.) ■

This proof does not dualize to final coalgebras, because T_0 would then have to be a set of all sets, which does not exist. Indeed, final coalgebras of the form $(U; \mathsf{I})$ generally do not exist in ZFC. But within Aczel's set theory with anti-foundation (section A.7), it is possible to build such algebras; see [4].

8.2 Kerkhoff's proof

An alternative construction is the following one, somewhat simplified from Kerkhoff [45]. It needs no ordinal recursion but only natural numbers and powersets. Furthermore, it can be dualized to model co-inductive types, and it can be formalized within our extended type theory as well.

We want to construct an initial algebra $(T\colon \mathbf{Set};\ \tau\colon \Sigma(x\colon A :: T^{Bx}) \to T)$. Elements of T are built from some $a\colon A$ and a tuple $s\colon T^{Ba}$ of sub-elements, so we think of them as trees, where each node has a label $x\colon A$ and a tuple of subtrees indexed over Bx. The idea is to represent such a tree by its set of nodes, where each node is characterized by its label together with the sequence of indices from $\bigcup(x\colon A :: Bx)$ that leads to the node.

Theorem 8.2 *(Kerkhoff) For polynomial F, there exists an initial F-algebra.*

Proof. As in paragraph 2.9.2, let $X^* := \Sigma(n\colon \mathbb{N} :: X^n)$ be the type of finite sequences, so that $\langle\rangle\colon X^*$, and $\langle x\rangle +\!\!+\, l\colon X^*$ for $x\colon X$, $l\colon X^*$. We'll define S to be the type of arbitrary sets of node representations, τ to be the operator that combines a tuple of such sets into a new one with a single root node, so that $(S;\tau)$ is an F-algebra, and then define T to be the subalgebra of S generated by τ.

Let $F.X = \Sigma(x\colon A :: X^{Bx})$.

$$\begin{array}{rcl} S & := & \mathcal{P}(\bigcup(A;B)^* \times A\,) \\ \tau.(a\colon A; s\colon S^{Ba})\colon S & := & \{\,(\langle\rangle, a)\,\} \,\cup\, \{y\colon Ba;\ (l,x)\colon\in s_y :: (\langle y\rangle +\!\!+\, l, x)\,\} \\ T & := & \bigcap(X\colon\subseteq S \mid\colon \tau[F.X] \subseteq X) \end{array}$$

Now, theorem 4.3 says that $(T;\tau)$ is initial, provided that τ is injective. To prove this, assume:

$$\tau.(a;s) = \tau.(a';s')$$

First, as we have $(\langle\rangle, a) \in \tau.(a;s)$, and as it is not possible that $(\langle\rangle, a) = (\langle y'\rangle +\!\!+\, l', x')$ for some $y'\colon Ba$; $(l',x')\colon\in s'_{y'}$, it follows that $(\langle\rangle, a) = (\langle\rangle, a')$ so $a = a'$.

Secondly, we prove $s_y \subseteq s'_y$ for arbitrary $y\colon Ba$.

$$\begin{array}{lcl} & (l,x) \in s_y & \\ \Rightarrow & (\langle y\rangle +\!\!+\, l, x) \in \tau.(a;s) & \{\text{def. } \tau\} \\ \Leftrightarrow & (\langle y\rangle +\!\!+\, l, x) \in \tau.(a;s') & \{\text{assumption}\} \\ \Leftrightarrow & \exists y'\colon Ba;\ (l';x')\colon\in s'_{y'} :: (\langle y\rangle +\!\!+\, l, x) = (\langle y'\rangle +\!\!+\, l', x') & \{\text{def. } \tau\} \\ \Leftrightarrow & \exists y'\colon Ba;\ (l';x')\colon\in s'_{y'} :: y = y' \wedge l = l' \wedge x = x' & \\ \Leftrightarrow & (l,x) \in s'_y & \end{array}$$

By symmetry we have $s'_y \subseteq s_y$, so that $s = s'$. ■

The difference with Kerkhoff is that he constructed the "free" F-algebra over a set C, which is the initial $(F + \mathsf{K}\,C)$-algebra; he had

$$S \ := \ \mathcal{P}(\,(C \cup A \cup \bigcup(A;B))^*\,)$$

$$\begin{array}{rcl} \tau.(a;s) & := & \{\langle a\rangle\} \cup \{y\colon Ba;\ l\colon\in s_y :: \langle a,y\rangle +\!\!+\, l\} \\ \eta.c & := & \{\langle c\rangle\} \\ T & := & \bigcap(X\colon\subseteq S \mid\colon \tau[F.X] \cup \eta[C] \subseteq X) \end{array}$$

A dual construction (dual with respect to set inclusion) yields an F-coalgebra $(U;\delta)$. The proof that this coalgebra is final is very different, though.

Theorem 8.3 *For polynomial F, there exists a final F-coalgebra.*

Proof. Let S and τ be as above. Then define:

$$\begin{array}{rcl} U & := & \bigcup(X\colon\subseteq S \mid\colon X \subseteq \tau[F.X]) \\ \delta.(t\colon U) & := & (a;s)\ \underline{\text{where}}\ \ (\langle\rangle, a) \in t\ , \\ & & \qquad\qquad s_y := \{(l,x) \mid\colon (\langle y\rangle +\!\!+\, l, x) \in t\}\ . \end{array}$$

Note that $\delta\colon U \to F.U$ is the inverse of τ (on U, not on S). This gives an F-coalgebra $(U;\delta)$; we'll prove that it is final. Let $(V;\gamma)$ be another F-coalgebra; we have to construct a unique homomorphism $f\colon(V;\gamma) \to (U;\delta)$, so that $\delta \circ f = F.f \circ \gamma$, or:

$$\forall v\colon V :: f.v\ =\ \tau.(F.f.(\gamma.v)) \tag{8.2}$$

An inductive definition of f would yield only a partial function. Rather, we define the collection of subsets $f.v\colon S$ for $v\colon V$ by simultaneous induction as the least tuple of sets such that

$$\forall v\colon V :: \tau.(F.f.(\gamma.v)) \subseteq f.v\ .$$

That is, for $v\colon V$ and $(a;w) := \gamma.v$:

$$\begin{array}{rccl} & & & (\langle\rangle, a) \in f.v \\ y\colon Ba;\ (l,x)\colon S\ \vdash & (l,x) \in f.w_y & \Rightarrow & (\langle y\rangle +\!\!+\, l, x) \in f.v \end{array}$$

This has the form of a fixed point equation on the lattice $(S;\subseteq)^V$, so by Knaster-Tarski (theorem 3.6) we have indeed $f\colon V \to S$ and (8.2).

We first check the type of f:

$$\begin{array}{rcl} & f \in V \to U & \\ \Leftrightarrow & f[V] \subseteq U & \\ \Leftarrow & f[V] \subseteq \tau[F.f[V]] & \{\text{definition } U\} \\ \Leftarrow & (F.f \circ \gamma)[V] \subseteq F.f[V] & \{f = \tau \circ F.f \circ \gamma\} \\ \Leftrightarrow & F.f \circ \gamma \in V \to F.f[V] & \\ \Leftarrow & F.f \in F.V \to F.f[V] & \{\gamma\colon V \to F.V\} \\ \Leftarrow & f \in V \to f[V] & \\ \Leftrightarrow & \mathsf{True} & \end{array}$$

Thus, f is a homomorphism indeed. For uniqueness, suppose g is a homomorphism too. As $\tau.(F.g.(\gamma.v)) = g.v$ and f is minimal, we have $f.v \subseteq g.v$. But then $f.v = g.v$, because of the following lemma, and we are done.

Lemma. *If $u, u' \in U$ and $u \subseteq u'$, then $u = u'$.*
We prove for $l\colon \bigcup(A; B)^*$, $x\colon A$ the following, by induction on the length of the finite sequence l:

$$\forall u, u'\colon U;\ u \subseteq u' :: (l, x) \in u' \Rightarrow (l, x) \in u \tag{8.3}$$

First we note that for any u, u', by definition of U we have $u = \tau.(a; s)$ and $u' = \tau.(a'; s')$ for certain $a, a'\colon A$, $s\colon U^{Ba}$, $s'\colon U^{Ba'}$. Given $u \subseteq u'$ and the definition of τ, it follows then that $a = a'$ and $s_y \subseteq s'_y$ for all $y\colon Ba$.

We check (8.3) for the empty list: if $(\langle\rangle, x) \in u'$, then we have $x = a' = a$ so $(\langle\rangle, x) \in u$.

Then, assume (8.3) as induction hypothesis. If $(\langle y\rangle \mathbin{+\!\!+} l, x) \in u'$, then we have $(l, x) \in s'_y$, so by hypothesis $(l, x) \in s_y$, hence $(\langle y\rangle \mathbin{+\!\!+} l, x) \in u$. This completes the induction, the lemma, and the theorem. ■

8.3 Algebras with equations

In section 4.4, we introduced equations or laws. We show now that one can always add laws to an initial F-algebra in **TYPE** (and also in **SET**), when quotient-types are available. The dual theorem, that one can always add laws to a final coalgebra, was already shown in section 7.4 .

Theorem 8.4 *For any polynomial endofunctor F on* **TYPE**, *and law $E = (H; r)$, if* **ALG** *F has an initial object, then* **ALG**$(F; E)$ *has one as well.*

Proof. Let $F.X = \Sigma(x\colon A :: X^{Bx})$, and $(T; \tau)$ be initial in **ALG** F. We define a congruence relation $R\colon \mathcal{P}(T^2)$ as follows. It is inductively defined by the clauses:

$$\begin{array}{rrcl} h\colon H.T \vdash & & & r(T;\tau).h \in R \qquad (8.4)\\ a\colon A;\ t, t'\colon T^{Ba} \vdash & \forall y\colon Ba :: (t_y, t'_y) \in R & \Rightarrow & (\tau.(a;t), \tau.(a;t')) \in R \qquad (8.5)\\ & |{=_T}| & \subseteq & R\\ & R^{\cup} & \subseteq & R\\ & R \cdot R & \subseteq & R \end{array}$$

The first two clauses may be written as $r(T; \tau) \in H.T \to R$ and $(\tau, \tau) \in F.R \to R$.

Using the quotient types of C.4.2, we take T' to be T modulo this congruence, and we define τ' so that $/\!/\_\mathsf{in}_R\colon (T \rhd T/\!/R)$ is a homomorphism, $\lambda/\!/\_\mathsf{in}_R\colon (T; \tau) \to (T'; \tau')$:

$$\begin{array}{lcl} T' & := & T/\!/R\\ \tau' & := & \{a\colon A;\ t\colon T^{Ba} :: ((a; (y :: /\!/\_\mathsf{in}_R\, t_y)), /\!/\_\mathsf{in}_R(\tau.(a;t)))\} \end{array}$$

First we have to show that this τ' is really a function. So assume $a\colon A$, $t, t'\colon T^{Ba}$, and $/\!/\_\mathsf{in}_R\, t_y = /\!/\_\mathsf{in}_R\, t'_y$ for $y\colon Ba$. As R is an equivalence relation, we have $(t_y, t'_y) \in R$ for all y, hence $(\tau.(a;t), \tau.(a;t')) \in R$ by the last clause of R, and $/\!/\_\mathsf{in}_R(\tau.(a;t)) = /\!/\_\mathsf{in}_R(\tau.(a;t'))$.

Secondly, F-algebra $(T'; \tau')$ should satisfy law E, that is, $r_0(T'; \tau') =_{H.T' \to T'} r_1(T'; \tau')$. As the $r_j\colon HU \dot\to U$ are natural transformations, and as $\lambda/\!/\_\mathsf{in}_R$ is a homomorphism, we have

$$\lambda/\!/\_\mathsf{in}_R \circ r_j(T; \tau) \;=\; r_j(T'; \tau') \circ H.(\lambda/\!/\_\mathsf{in}_R)\ .$$

But by the first clause of R, we have

$$\lambda /\!/ \_\mathsf{in}_R \circ r_0(T;\tau) \;=\; \lambda /\!/ \_\mathsf{in}_R \circ r_1(T';\tau') \;,$$

so we are done if $H.(\lambda /\!/ \_\mathsf{in}_R)$ is surjective, that is, has a right-inverse. Now note that $\lambda /\!/ \_\mathsf{in}_R$ must have a right-inverse g by the axiom of choice, and then $H.g$ is a right-inverse of $H.(\lambda /\!/ \_\mathsf{in}_R)$.

Thirdly, supposing that $(U;\psi)$ is another $(F;E)$-algebra, we must provide a unique homomorphism $f\colon (T';\tau') \to (U;\psi)$. For a function $f\colon T' \to U$ we have:

$$\begin{array}{cll}
 & f \text{ is a homomorphism} & \\
\Leftrightarrow & f \circ \tau' = \psi \circ F.f & \\
\Leftrightarrow & f \circ \tau' \circ F.(\lambda /\!/ \_\mathsf{in}) = \psi \circ F.f \circ F.(\lambda /\!/ \_\mathsf{in}) & \{\lambda /\!/ \_\mathsf{in} \text{ is surjective}\} \\
\Leftrightarrow & f \circ \lambda /\!/ \_\mathsf{in} \circ \tau = \psi \circ F.(f \circ \lambda /\!/ \_\mathsf{in}) & \{\text{def. } \tau'\} \\
\Leftrightarrow & f \circ \lambda /\!/ \_\mathsf{in} = (\![U;\psi]\!) & \{\text{initiality } (T;\tau)\ \}
\end{array}$$

So we can take

$$f \;:=\; \lambda /\!/ \_\mathsf{elim}\, (\![U;\psi]\!). \;,$$

where we must prove that for $(x,x')\colon\in R$, one has $(\![\psi]\!).x = (\![\psi]\!).x'$. For this we need the minimality of R. So defining

$$S \;:=\; \{\, (x,x')\colon T^2 \mid\colon (\![\psi]\!).x = (\![\psi]\!).x' \} \;,$$

we prove that $R \subseteq S$ by checking that S satisfies the five clauses that define R. Relation S is clearly reflexive, symmetric, and transitive. To check (8.4), we have for $h\colon H.T$

$$(\![\psi]\!).(r_0(T;\tau).h) \;=\; (\![\psi]\!).(r_1(T;\tau).h)$$

because $(\![\psi]\!) \circ r_j(T;\tau) = r_j(U;\psi) \circ H.(\![\psi]\!)$ by naturality of r_j, and $r_0(U;\psi) = r_1(U;\psi)$ as $(U;\psi)$ satisfies law E.
To check (8.5), when $(t_y, t'_y) \in S$ for $y\colon Ba$, then we have $(\tau.(a;t), \tau.(a;t')) \in S$ because $(\![\psi]\!)$ is a homomorphism, i.e. $(\![\psi]\!).(\tau.(a;t)) = \psi.(a;(y :: (\![\psi]\!).t_y))$. ■

Chapter 9

Partiality

Up till now, all objects were fully defined. Functional programming languages that employ so-called *lazy* (non-strict) evaluation require quite different recursive types. In these languages objects can be defined, some parts of which are undefined. "Undefined" means here that the part is given by a program whose computation proceeds indefinitely, without producing any output. (Note that a non-terminating program may still produce fully defined infinite objects.)

Classically, such types are modeled by adding a special value $\perp$ to represent an undefined (sub-)value. A recursive type T with one constructor $\tau\colon F.T \to T$ may then be represented by an initial algebra $(T\colon \mathbf{Type};\ [\tau, \mathsf{K}\perp]\colon F.T + 1 \to T)$. Constructively, this does not work, as it is in general undecidable whether a program will produce anything or not. We shall give an alternative representation in 9.3.

First, in section 9.1, we give a brief overview of the standard theory of complete partial orders (*cpo*'s). This theory can be used to interpret recursive object definitions, classically as well as constructively.

In 9.2, we treat the simplified case of optional objects, which are either undefined or fully defined. These may be used to model partial functions or procedures in a language without lazy evaluation.

In 9.3, we give a new constructive representation of the cpo of recursive types with lazy parts, using final coalgebras in the category of strict types.

Finally, section 9.4 shows how recursive object definitions can be interpreted in this representation without using the cpo structure.

9.1 Domain theory

There are many constructions of categories of domains that model recursive data types, see e.g. Scott [77]. Smyth and Plotkin set up [79] a categorical framework that generalizes the construction of recursive domains in most categories occurring in computational semantics.

One of these categories is the category of *complete partial orders* (cpo's). We define this category, and give three fixed-point theorems: 9.1 shows how to interpret recursive object definitions, 9.3 shows how to interpret recursive type definitions, and theorem 9.4 shows how to prove properties of recursively defined objects.

We define first the category of *partial orders*, then *ω-chains* over a partial order, then the category of cpo's. The object and arrow parts of these categories are given by means of structure definitions (paragraph 2.6.3); but the arrow part might in fact be obtained from the definition of the object part by means of a straightforward procedure.

$\underline{\mathsf{Define}}$ $\mathbf{PoSet}\colon\mathbf{Cat}$ $\underline{\mathsf{by}}$
$X\colon\mathbf{PoSet}$ $:=:$ $(X\colon\mathbf{TYPE};$
$\quad(\leq)\colon\subseteq X^2;$
$\quad(\leq)\cdot(\leq)\ \subseteq\ (\leq);$
$\quad(\leq)\cap(\leq)^{\cup}\ =\ (=_X)$
$\quad)$
$f\colon X\to Y$ $\underline{\mathsf{in}}$ $\mathbf{PoSet}$ $:=:$ $(f\colon X\to Y$ $\underline{\mathsf{in}}$ $\mathbf{TYPE};$
$\quad(f,f)\in(\leq)\to(\leq)$
$\quad);$

$\omega\_\mathsf{chain}(X\colon\mathbf{PoSet}) := \{\, s\colon X^\omega \mid\colon \forall i\colon\omega :: s_i \leq s_{i+1}\}$

$\underline{\mathsf{Define}}$ $\mathbf{CPO}\colon\mathbf{Cat}$ $\underline{\mathsf{by}}$
$\mathcal{D}\colon\mathbf{CPO}$ $:=:$ $((\mathcal{D};(\sqsubseteq_{\mathcal{D}}))\colon\mathbf{PoSet};$
$\quad\bot_{\mathcal{D}}\colon\mathcal{D};\ \forall(x\colon\mathcal{D} :: \bot\sqsubseteq x);$
$\quad s\colon\omega\_\mathsf{chain}\,\mathcal{D}\ \vdash\quad \bigsqcup s\colon\mathcal{D};\ \forall x\colon\mathcal{D} :: (\bigsqcup s\sqsubseteq x\ \Leftrightarrow\ \forall i\colon\omega :: s_i\sqsubseteq x)$
$\quad);$
$f\colon\mathcal{D}\to\mathcal{E}$ $\underline{\mathsf{in}}$ $\mathbf{CPO}$ $:=:$ $(f\colon\mathcal{D}\to\mathcal{E}$ $\underline{\mathsf{in}}$ $\mathbf{PoSet};$
$\quad f.\bot_{\mathcal{D}}=\bot_{\mathcal{E}};$
$\quad s\colon\omega\_\mathsf{chain}\,\mathcal{D}\vdash\quad f.\bigsqcup s=\bigsqcup(f^\omega.s)$
$\quad)$

Relation '$x\sqsubseteq y$' may be understood as 'Partial object x is an approximation of y', and $\bot$ is the undefined object, which approximates everything.

A *continuous function* $f\colon\mathcal{D}\to_{\mathsf{c}}\mathcal{E}$ between two cpo's is a function that is monotonic with respect to $\sqsubseteq$ and that preserves limits of ω-chains. Note that arrows $f\colon\mathcal{D}\to\mathcal{E}$ $\underline{\mathsf{in}}$ $\mathbf{CPO}$ are continuous functions that preserve $\bot$ too.

Theorem 9.1 (fixed points in a cpo) *Any equation $x=f.x$ where $f\colon\mathcal{D}\to_{\mathsf{c}}\mathcal{D}$, has a least solution in $\mathcal{D}$, called $\mathsf{fix}\,f$. That is, one has*

$$\begin{aligned} f.(\mathsf{fix}\,f) &= \mathsf{fix}\,f\\ f.x\sqsubseteq x\quad\Rightarrow\quad \mathsf{fix}\,f &\sqsubseteq x\end{aligned}$$

Proof. Take $s_0:=\bot$, $s_{i+1}:=f.s_i$. By induction one has $\forall i\colon\omega :: s_i\sqsubseteq s_{i+1}$, so we can define $\mathsf{fix}\,f:=\bigsqcup s$. Then

$$f.(\bigsqcup s)\ =\ \bigsqcup(i :: f.s_i)\ =\ \bigsqcup(i :: s_{i+1})\ =\ \bigsqcup(i :: s_i)\ ,$$

and if $f.x\sqsubseteq x$, then by a simple induction $\forall i\colon\omega :: s_i\sqsubseteq x$, so $\bigsqcup s\sqsubseteq x$. ■

Category **CPO** is closed under products, sums, continuous function space, and taking fixed points (modulo isomorphism) of suitable endofunctors, by Scott's inverse limit (colimit) construction, as follows.

A *cochain* in any category $\mathcal{C}$ with ω-products is an ω-tuple, $T{:}\,\mathcal{C}^\omega$, with arrows $\phi_i{:}\,T_{i+1} \to T_i$. A *cocone* is a structure $(T;\phi;S;\psi)$ where $(T;\phi)$ is a cochain, S an object, and where arrows $\psi_{i:\omega}{:}\,S \to T_i$ commute with ϕ_i:

$$\psi_i \;=\; \psi_{i+1} \mathbin{\bar{\circ}} \phi_i \ .$$

Given a cocone $(T;\phi;S;\psi)$, we call $(S;\psi)$ a *colimit* of $(T;\phi)$ iff for any cocone $(T;\phi;S';\psi')$ there is a unique homomorphism $(S';\psi') \to (S;\psi)$.

Theorem 9.2 *Every cochain $(T;\phi))$ has a colimit.*

Proof. Take

$$S := \{\, t{:}\,\Pi(\omega;T) \mid{:}\ \forall i{:}\,\omega :: t_i = \phi_i.t_{i+1} \} \ ,$$

then $(T;\phi;S;\pi)$ is a cocone, and for any cocone $(T;\phi;S';\psi')$ one has $\langle\psi'\rangle{:}\,(S';\psi') \to (S;\pi)$.
Assuming $\chi{:}\,(S';\psi') \to (S;\pi)$ too, one has $\langle\psi'\rangle = \chi$ because $\psi'_i = \chi \mathbin{\bar{\circ}} \pi_i$. ■

Theorem 9.3 (fixed points in CPO) *(Scott) Any functor $F{:}\,\mathbf{CPO} \to \mathbf{CPO}$ that preserves colimits of cochains has a unique fixed point (modulo isomorphism) μF, $F(\mu F) \cong \mu F$, yielding both an initial F-algebra and a final F-coalgebra.*

Proof. (sketch) Given functor F, we define a cochain $(T;\phi)$ by:

$$\begin{aligned} T_0 &:= \{\bot\} \\ T_{i+1} &:= F.T_i \\ \phi_0 &:= x \mapsto \bot \\ \phi_{i+1} &:= F.\phi_i \end{aligned}$$

Let $(S;\psi)$ be its colimit, take $\mu F := S$. Now we define a constructor:

$$\tau{:}\,F.(\mu F) \to \mu F \ := \ x \mapsto (0 :: \bot \mid i+1 :: F.\psi_i.x)$$

To construct $\tau^\cup$, note that $(F.\mu F; F^\omega.\psi)$ is a colimit of

$$(F^\omega.T; F^\omega.\phi) \;=\; ((i :: F.T_i);(i :: F.\phi_i)) \;=\; ((i :: T_{i+1});(i :: \phi_{i+1})) \ .$$

But as $((i :: T_{i+1});(i :: \phi_{i+1});\,\mu_F;(i :: \psi_{i+1}))$ is a cocone as well, there must be a unique homomorphism

$$\tau^\cup{:}\,(\mu F;\ (i :: \psi_{i+1})) \to (F.\mu F;\ (i :: F.\psi_i)) \ .$$

The unique homomorphisms required for initiality and finality are (see also Paterson [67], where they are called $\mathbf{reduce}_F\,\phi$ and $\mathbf{generate}_F\,\psi$)

$$\mathsf{fix}(g \mapsto \tau^\cup \mathbin{\bar{\circ}} F.g \mathbin{\bar{\circ}} \phi) : \ (\mu F;\tau) \to (X;\phi)$$

and

$$\mathsf{fix}(g \mapsto \psi \mathbin{\bar{\circ}} F.g \mathbin{\bar{\circ}} \tau) : \ (X;\psi) \to (\mu F;\tau^\cup) \ .$$

(end of proof sketch) ■

Theorem 9.4 (fixed point induction)

$$\begin{array}{l} f\colon \mathcal{D} \to_{\mathsf{c}} \mathcal{D} \\ P(x\colon \mathcal{D})\colon \mathbf{Prop} \\ s\colon \omega\_\mathsf{chain}\,\mathcal{D} \vdash \forall(i\colon \omega :: P(s_i)) \Rightarrow P(\bigsqcup s) \\ P(\bot) \\ \forall(x\colon \mathcal{D} :: P(x) \Rightarrow P(f.x)) \\ \hline P(\mathsf{fix}\, f) \end{array}$$

Proof. Take $s_0 := \bot$, $s_{i+1} := f.s_i$ as in theorem 9.1, then one has $\forall i\colon \omega :: P(s_i)$ and $\mathsf{fix}\, f = \bigsqcup s$, so $P(\mathsf{fix}\, f)$. ■

One of the first computer verification systems was LCF [35, 70], Logic of Computable Functions. It has fixed point induction as a primitive rule, using a syntactic check for chain-completeness.

9.2 Optional objects

We wish to lift a type A to a type $\uparrow A$ with $A \subseteq_{\mathsf{t}} \uparrow A$ and a special value $\bot\colon \uparrow A$, called "undefined". (Such a type $\uparrow A$ is often named '$A_\bot$', but we do not want to use subscripts for this.) Partial functions from A to B may then be represented by total functions from A to $\uparrow B$.

The postfix predicate $x\!\downarrow$ means 'x is defined', and $\uparrow A$ is partially ordered, as follows:

$$\begin{array}{rrcl} x\colon \uparrow A \vdash & x\!\downarrow\colon \mathbf{Prop} & := & \exists a\colon A :: x =_{\uparrow A} a \\ x, y\colon \uparrow A \vdash & x \sqsubseteq y & := & (x\!\downarrow \Rightarrow x = y) \ , \end{array}$$

and this gives a cpo (classically), for

$$s\colon \omega\_\mathsf{chain} \uparrow A \vdash \quad \bigsqcup s \ := \begin{cases} s_i & \text{if } s_i\!\downarrow\text{, for some } i \\ \bot & \text{if } \forall i :: s_i = \bot \end{cases} .$$

So $(\uparrow B)^A$ is a cpo too, and any continuous function $e\colon (\uparrow B)^A \to_{\mathsf{c}} (\uparrow B)^A$ must have a unique least fixed point $f = e.f$. This gives us the possibility of recursive definition of partial functions.

There are several ways to define such a type $\uparrow A$ within our calculus:

9.2.1 Explicit options. Classically, the idea is to simply add a singleton type to A, using a sum:

$$\begin{array}{rcl} \mathsf{Opt}\, A & := & A + 1 \ , \\ \bot & := & \sigma_1.0 \\ \sigma_0 : \ A & \subseteq_{\mathsf{t}} & \mathsf{Opt}\, A \ . \end{array}$$

Constructively, this does not give a cpo. For, to construct a value of type $\mathsf{Opt}\, A$ one must effectively decide whether it has to be $\bot$ or some $a\colon A$. But the limit of an ω-chain s should be $\bot$ if and only if all s_i equal $\bot$, and this cannot be effectively decided.

Similarly, a mapping $e\colon (\mathsf{Opt}\,B)^A \to (\mathsf{Opt}\,B)^A$ that is monotonic (which means that a more defined argument gives a more defined result) can classically be shown to have a unique least fixed point $f = e.f$, but this fixed point is not constructively definable, in general. For, given an argument a, it cannot be effectively decided whether computation of $f\,a$ will terminate.

9.2.2 Propositional options. An alternative is to think of an optional object as a proposition that tells whether the object is defined or not, together with the actual value in case the proposition is true. Thus, one can only access the value if one has a proof of the proposition.

$$\begin{array}{rcl} \uparrow\! A & := & \Sigma(D\colon \mathbf{Prop} :: A^D) \\ \bot & := & (\mathsf{False}; ()) \\ x \mapsto (\mathsf{True}; (() :: x)) & : & A \subseteq_{\mathsf{t}} \uparrow\! A \end{array}$$

This $\uparrow\! A$ gives (using strong existential quantifier elimination) a constructive cpo, for we can define:

$$\bigsqcup(s\colon \omega\_\mathsf{chain}\uparrow\! A) \;:=\; (\,\exists(i\colon \omega :: \mathsf{fst}\,s_i);\ \exists\_\mathsf{elim}((i;d) :: \mathsf{snd}(s_i d))\,)$$

(Check that for $(i;d),(i';d')\colon \Sigma(i\colon \omega :: \mathsf{fst}\,s_i)$, one has that $\mathsf{snd}(s_i d) = \mathsf{snd}(s_{i'} d')$.) So continuous mappings $e\colon (\uparrow\! B)^A \to_{\mathsf{c}} (\uparrow\! B)^A$ have (constructible) fixed points.

This construction is not possible in Nuprl [18], for its type theory does not have strong $\exists\_\mathsf{elim}$. However, Nuprl has recursive definition of partial functions as a primitive rule. It employs a (restrictive) syntactic test for continuity of recursive definitions.

9.2.3 Lazy options. If one has co-inductive types, there is another alternative. Given a type A, we define a co-inductive type T that represents computations of objects of A. It has constructors $\eta\colon A \to T$ and $\zeta\colon T \to T$, so that a nonterminating computation can be represented by $\zeta.(\zeta.(\ldots))$, repeating ζ indefinitely. Type $\uparrow\! A$ is then the quotient type (section C.4.2) of T modulo the relation given by ζ, so that $\zeta.x$ and x are identified.

$$\begin{array}{rcl} (T;\delta) & := & \nu(\mathsf{K}\,A + \mathsf{Id}), \\ [\eta,\zeta] & := & \delta^{\cup}; \\ \bot\colon T & := & \zeta.\bot \\ \uparrow\! A & := & T/\!/\zeta \\ \bot\colon \uparrow\! A & := & /\!/\_\mathsf{in}(\bot\colon T) \\ (x \mapsto /\!/\_\mathsf{in}(\eta.x))\colon\ A & \subseteq_{\mathsf{t}} & \uparrow\! A \end{array}$$

Note that ζ induces an equivalence $\equiv_\zeta$ on T, and that T is partially ordered by:

$$\begin{array}{rcl} (x\colon T)\!\downarrow & := & \exists a\colon A :: x \equiv_\zeta \eta.a \\ x \sqsubseteq y & := & (x\!\downarrow \Rightarrow x \equiv_\zeta y)\ . \end{array}$$

To get a constructive definition of $\bigsqcup(s\colon \omega\_\mathsf{chain}(\uparrow\! A))$, we have to do a simultaneous quotient elimination on all $s_i\colon \uparrow\! A$. This is possible by a construction similar to the one in C.4.5. It then suffices to construct $\bigsqcup(t\colon \omega\_\mathsf{chain}\,T)\colon T$.

Note that if there are some $i, k{:}\,\omega$ and $a{:}\,A$ such that $t_i = \zeta^{(k)}.(\eta.a)$, then for any j, one necessarily has $t_{i+j} \equiv_\zeta \eta.a$, and we should have $\bigsqcup t \equiv_\zeta \eta.a$.

Now, one cannot effectively decide whether such i and k exist. Rather, we define an anamorphism $f{:}\,(\mathbb{N}, \phi) \to (T; \delta)$ such that $f.n$ tries only i and k up to bound n, and yields $\zeta.(f.(n+1))$ if that does not succeed. So, making improper use of $\underline{\mathsf{if}}$, we define informally:

$$f.n \ := \ \underline{\mathsf{if}}\ t_i = \zeta^{(k)}.(\eta.a) \text{ for some } i, k{:} \leq n,\ a{:}\,A\ \underline{\mathsf{then}}\ \eta.a\ \underline{\mathsf{else}}\ \zeta.(f.(n+1))\ .$$

The proper definition requires two local recursions over $\mathbb{N}$, and is left to the reader. Finally we define $\bigsqcup t := f.0$.

9.3 Building recursive cpo's by co-induction

We now generalize the construction in paragraph 9.2.3 of types with lazy optional objects to recursive type definitions. That is, we build a solution to the domain equation $T \cong {\uparrow}(F.T)$, where F is polynomial, using co-induction.

The value $\bot$ should represent an object that takes infinite time to compute. If we add a constructor $\zeta{:}\,T \to T$ (zeta) to represent a value that takes one step more than its argument, then $\bot$ can be represented by an infinite sequence of ζ's.

$$\begin{array}{rcl} (T; \delta) & := & \nu(F + \mathsf{Id}) \\ [\tau, \zeta] & := & \delta^\cup \\ \bot{:}\,T & := & \zeta.\bot \end{array}$$

Actually, T should be taken modulo a congruence $\simeq$ that identifies any $\zeta.x$ with x. To obtain this congruence, we first define the approximation relation $(\sqsubseteq){:} \subseteq T^2$ as the greatest relation such that, for all $x{:}\,F.T;\ y{:}\,T$:

$$\exists(n{:}\,\mathbb{N} :: \zeta^{(n)}.(\tau.x) \sqsubseteq y) \ \Rightarrow\ \exists(m{:}\,\mathbb{N};\ z{:}\,F.T :: y = \zeta^{(m)}.(\tau.z)\ \wedge\ x\ F.(\sqsubseteq)\ z)\ .$$

Put in relational calculus:

$$\tau \cdot \zeta^{(*)} \cdot (\sqsubseteq) \ \subseteq\ F.(\sqsubseteq) \cdot \tau \cdot \zeta^{(*)}\ . \tag{9.1}$$

Here, $F.(\sqsubseteq){:} \subseteq (F.T)^2$ stands for F lifted to relations as in section D.3, applied to $(\sqsubseteq)$, and $\zeta^{(*)}$ is the reflexive, transitive closure of ζ, i.e.

$$\zeta^{(*)} \ = \ \bigcup(n{:}\,\mathbb{N} :: \zeta^{(n)}) \ = \ \bigcap(Q{:}\,\mathcal{P}T^2 \mid{:}\ \zeta \cup (=) \cup Q \cdot Q \subseteq Q)\ .$$

Note that $\bot \sqsubseteq x$, for any $x{:}\,T$. Then we define $(\simeq) := (\sqsubseteq) \cap (\sqsubseteq)^\cup$.

Theorem 9.5 1. *Relation $\sqsubseteq$ is a preorder.*

2. *When we extend $\sqsubseteq$ to $T/\!/\zeta$, then $(T/\!/\zeta; (\sqsubseteq), \bot)$ is an ω-complete partial order, at least if F is a polynomial, $F.X = \Sigma(a{:}\,A :: X^{Ba})$.*

Proof 1. Relation $\sqsubseteq$ is reflexive, for $(=_T)$ satisfies (9.1) so $(=) \subseteq (\sqsubseteq)$. And $\sqsubseteq$ is transitive, i.e. $(\sqsubseteq)\cdot(\sqsubseteq) \subseteq (\sqsubseteq)$, follows again from $\sqsubseteq$ being maximal, for

$$\begin{array}{cll} & \tau\cdot\zeta^{(*)}\cdot((\sqsubseteq)\cdot(\sqsubseteq)) & \\ \subseteq & F.(\sqsubseteq)\cdot\tau\cdot\zeta^{(*)}\cdot(\sqsubseteq) & \{\ (9.1)\ \} \\ \subseteq & F.(\sqsubseteq)\cdot F.(\sqsubseteq)\cdot\tau\cdot\zeta^{(*)} & \{\ (9.1)\ \} \\ = & F.((\sqsubseteq)\cdot(\sqsubseteq))\cdot\tau\cdot\zeta^{(*)} & \end{array}$$

2. By definition of $\simeq$, preorder $\sqsubseteq$ gives a partial order on $T//(\simeq)$.

Proving that ω-chains over $\sqsubseteq$ have limits is rather complicated. Let $s\colon T^\omega$ be a chain, $s_i \sqsubseteq s_{i+1}$, we have to define $\bigsqcup s\colon T$. We follow the method of paragraph 9.2.3.

The idea is that, if there are some $i, k_0\colon\omega$ and $u_0\colon F.T$ such that $s_i = \zeta^{(k0)}.(\tau.u_0)$, then all s_{i+j} must necessarily equal $\zeta^{(kj)}.(\tau.u_j)$ for certain k_j and u_j, and $u_j\ F.(\sqsubseteq)\ u_{j+1}$ (exercise). Hence all u_j equal $(a; v_j)$ for a fixed a and $v_j\colon T^{Ba}$, and $v_j y \sqsubseteq v_{j+1} y$ for $y\colon Ba$. So $(j :: v_j y)$ are chains again, and $\bigsqcup s$ should equal, for some n,

$$\zeta^{(n)}.(\tau.(a; (y :: \bigsqcup(j :: v_j y)))) \ .$$

As trying an unbounded number of i and k cannot be done constructively, we define a homomorphism $f\colon(\omega\_\mathsf{chain}\,T \times \mathbb{N}) \to T$ such that $f.(s,n)$ tries only values of i and k up to bound n, and yields $\zeta.(f.(s,n+1))$ if that does not succeed.

```
$f.(s,n)$ := if $s_i = \zeta^{(k)}.(\tau.(a;v_0))$ for some $i,k\colon\leq n$ and $(a;v_0)\colon F.T$
             then let $v_j$ be such that $s_{i+j} = \zeta^{(k')}.(\tau.(a;v_j))$ in
                $\tau.(a;(y :: f.((j :: v_j y), 0)))$
             else $\zeta.(f.(s,n+1))$
```

We leave the constructive definition of i, k, a and the v_j to the reader. Finally we define $\bigsqcup s := f.(s,0)$. ∎

9.4 Recursive object definitions

The use of a cpo in section 9.3 to define recursive objects in T is something of a detour. In this section we give a more direct construction of T-elements out of recursive object definitions, a construction which does not use the partial order at all.

Suppose that we have a system of mutually recursive tree expressions. We wish to construct the (infinite or partial) trees that are defined by these expressions. For this purpose we need, given a type V representing tree variables, a type E_V that represents tree expressions with variables from V. The family of types E_V with its operations is defined as a co-inductive algebra, and includes a constructor $\zeta\colon E_V \to E_V$ to accommodate nontermination.

Let F be polynomial, $F.X = \Sigma(x\colon A :: X^{Bx})$ and $(T;\delta) := \nu(F + \mathsf{Id})$ as in 9.3. Given a valuation, i.e. a binding of expressions to variables, $t\colon T^V$, any expression $e\colon E_V$ should define a tree $\mathsf{eval}_t.e\colon T$. Elements of E_V may have one of the following forms:

- $\tau.(x\colon F.E_V)$, representing a tree constructed from subtrees

- $\zeta.(e{:}\, E_V)$, equivalent to just e
- $\eta.(v{:}\, V)$, representing a variable occurrence
- $\gamma.(d{:}\, E_V,\, c{:}\, \Pi(x{:}\, A :: E_{V+Bx}))$, representing a case analysis on the result of tree expression d. If d evaluates to some tree $\tau.(a;u)$, then evaluation of $\gamma.(d,c)$ should boil down to evaluation of c_a under the valuation $(t,u){:}\, T^{V+Ba}$. A suggestive program notation for $\gamma.(d,c)$ might be:

  $$\underline{\mathsf{case}}\ d\ \underline{\mathsf{is}}\ \tau.(a;u) \Longrightarrow c_a$$

  where expression c_a may contain variables referring to the tuple of trees $u{:}\, T^{Ba}$.

Thus, we define E as follows.

$$\begin{array}{rcl} F' & : & \mathbf{TYPE}^{\mathbf{Type}} \to \mathbf{TYPE}^{\mathbf{Type}} \\ (F'.X)_V & := & F.X_V + X_V + V + (X_V \times \Pi(x{:}\, A :: X_{V+Bx})) \\ (E;\delta') & := & \nu F' \\ [\tau,\zeta,\eta,\gamma] & := & \delta'^{\cup} \end{array}$$

We omit the index V of the operations. Note that there is an embedding $[\![V :: \delta \mathbin{\bar{\circ}} [\sigma_0,\sigma_1]\,]\!]_V{:}\, T \subseteq_{\mathsf{t}} E_V$.

To define the evaluation function as a homomorphism, we need a substitution operation. For a fixed type W, we define the tuple of substitution functions $\mathsf{subst}_V{:}\, (E_{V+W} \times E_V^W) \to E_V$ by the equations:

$$\begin{array}{rcl} \mathsf{subst}_V.(\tau.x,t) & = & \tau.(F.(\mathsf{subst}_V \circ \langle \mathsf{I}, \mathsf{K}\,t\rangle).x) \\ \mathsf{subst}_V.(\zeta.d,t) & = & \zeta.(\mathsf{subst}_V.(d,t)) \\ \mathsf{subst}_V.(\eta.(0;v),t) & = & \eta.v \\ \mathsf{subst}_V.(\eta.(1;w),t) & = & \zeta.tw \\ \mathsf{subst}_V.(\gamma.(d,c),t) & = & \gamma.(\mathsf{subst}_V.(d,t),(x :: \mathsf{subst}_{V+Bx}.(cx^{\backslash}E_r, t^{\backslash}E_{\sigma_0}^W))) \end{array}$$

where we use that E_V is functorial in its subscript V, and

$$r{:}\, (V+W)+Bx \to (V+Bx)+W \ := \ [\,[\sigma_0\circ\sigma_0,\sigma_1],\sigma_1\circ\sigma_0]\ .$$

To see the need for this, note that subst_{V+Bx} requires an argument of type $(E_{V+Bx+W} \times E^W_{V+Bx})$ whereas we have $cx{:}\, E_{V+W+Bx}$ and $t{:}\, E_V^W$. For well-definedness of subst, note that these equations can be given the shape of a dual recursion (paragraph 7.3.1)

$$\mathsf{subst} = \delta^{\cup} \circ F'.[\mathsf{I},\mathsf{subst}] \circ \phi$$

for some $\phi{:}\, D \to E + F'.D$ $\underline{\mathsf{in}}$ $\mathbf{TYPE}^{\mathbf{Type}}$ where $D_V := E_{V+W} \times E_V^W$.

Now suppose we have a tuple of recursive expressions,

$$t{:}\, E_V^V\ .$$

In context t we define, for any expression $e{:}\, E_V$, the tree $\mathsf{eval}_t.e$ that is denoted by e when parameters $v{:}\, V$ are bound to tv. This eval_t is defined as the unique homomorphism

$$\mathsf{eval}_t \colon (E_V;\phi) \to (T;\delta)$$

for some $\phi\colon E_V \to F.E_V + E_V$ by:

$$\begin{array}{rcl}
\mathsf{eval}_t.(\tau.x) & = & \tau.(F.\mathsf{eval}_t.x) \\
\mathsf{eval}_t.(\zeta.e) & = & \zeta.(\mathsf{eval}_t.e) \\
\mathsf{eval}_t.(\eta.v) & = & \zeta.(\mathsf{eval}_t.tv) \\
\mathsf{eval}_t.(\gamma.(\tau.(a;u),c)) & = & \zeta.(\mathsf{eval}_t.(\mathsf{subst}_V.(ca,u))) \\
\mathsf{eval}_t.(\gamma.(\zeta.e,c)) & = & \zeta.(\mathsf{eval}_t.(\gamma.(e,c))) \\
\mathsf{eval}_t.(\gamma.(\eta.v,c)) & = & \zeta.(\mathsf{eval}_t.(\gamma.(tv,c))) \\
\mathsf{eval}_t.(\gamma.(\gamma.(e,d),c)) & = & \zeta.(\mathsf{eval}_t.(\gamma.(e,(x :: \gamma.(dx,c)))))
\end{array}$$

One may prove that if $\mathsf{eval}_t.d \simeq \tau.(a;u)$, then $\mathsf{eval}_t.(\gamma.(d,c)) \simeq \mathsf{eval}_{(t,u)}.ca$.

9.5 Conclusion

We introduced the most basic notions of recursive domain theory, and defined simple classical and constructive domains for representing optional objects. We then developed in 9.2 a representation for lazy optional objects by means of co-induction within the strict type theory of *ADAM*, which we generalized in 9.3 to lazy recursive types. While this representation of types is quite elegant, the representation of actual recursive object definitions is not. If one wishes to use such objects in a constructive type theory, it seems preferable to include them in the theory as primitives.

Chapter 10

Related subjects

10.1 Impredicative type theories

Second-order, or impredicative, or polymorphic, type theories like the Calculus of Constructions [21] and second-order typed lambda calculus allow the formation of types in the lowest universe, which we call $\mathbf{Data}\colon\mathbf{Type}$ here, by quantification over types from a higher universe:

$$\frac{\begin{array}{l}A\colon\mathbf{Type}\\ D\colon\mathbf{Data}^A\end{array}}{\Pi(A;D)\colon\mathbf{Data}}$$

Thus, **Data** is very much like our **Prop**, except that **Prop** has additional equality rules stating that equivalent propositions are equal, and that all proofs of the same proposition are equal. Furthermore, their use is different, for objects in **Data** are used for actual computation, while objects in **Prop** are used for stating properties only. The impredicative quantification allows one to define all kinds of weakly initial and final algebras, without using further primitive notions.

Example 10.1 The type of booleans can be defined by

$$\begin{array}{rcl}
\mathsf{Bool} & := & \Pi(X\colon\mathbf{Data};\ x\colon X;\ y\colon X :: X)\\
\mathsf{true} & := & (X;x;y :: x)\\
\mathsf{false} & := & (X;x;y :: y)\\
b\colon\mathsf{Bool};\ E\colon\mathbf{Data};\ e_0,e_1\colon E \vdash \quad \underline{\mathsf{if}}\ b\ \underline{\mathsf{then}}\ e_0\ \underline{\mathsf{else}}\ e_1 & := & bEe_0e_1
\end{array}$$

(End of example)

A drawback of such impredicative encodings is that dependent types like $T\colon\Pi(x\colon\mathsf{Bool} :: \mathbf{Data})$ cannot use an elimination on the impredicative object x, because expression '$x\mathbf{Data}T_0T_1$' would be wrongly typed.

Luo's ECC [48] extends the Calculus of Constructions with generalized sums and a hierarchy of universes as in *ADAM*. Ore [66] extended ECC further with disjoint sums (sums over a finite type) and inductive types at the predicative level. We may call this system ECCI. It is equivalent to *ADAM* without equality types and strong proof elimination; data types are to be built at the predicative level rather than in the impredicative universe **Prop**.

10.1.1 Weak initial algebras

The weak initial algebra $(T;\theta)$ that has a sequence of constructors $\theta_j\colon F_j.T \to T$ can be impredicatively defined by: ($\Delta\colon \mathcal{C} \to \mathcal{C}^N$ is the diagonal functor $X \mapsto (i :: X)$)

$$\begin{array}{rcl} T\colon \mathbf{Data} & := & \Pi(X\colon \mathbf{Data};\ \phi\colon F.X \to \Delta.X :: X) \\ \theta_j\colon F_j.T \to T & := & y \mapsto (X;\phi :: \phi_j.(F_j.(x \mapsto xX\phi).y)\,) \end{array}$$

In particular, a weak initial F-algebra for $F\colon \mathbf{Data} \to \mathbf{Data}$ is given by:

$$\begin{array}{rcl} \mu_{\mathsf{w}}F\colon \mathbf{Data} & := & \Pi(X\colon \mathbf{Data};\ \phi\colon F.X \to X :: X) \\ \tau\colon F.\mu_{\mathsf{w}}F \to \mu_{\mathsf{w}}F & := & y \mapsto (X;\phi :: \phi.(F.(x \mapsto xX\phi).y)\,) \end{array}$$

Given another F-algebra $(U;\psi)$, there is a homomorphism

$$(\![U;\psi]\!)\colon \mu_{\mathsf{w}}F \to U \ := \ x \mapsto xU\psi$$

for which we have the reduction rule $(\![U;\psi]\!)\circ\tau \ => \ \psi\circ F.(\![U;\psi]\!)$. One cannot prove that this homomorphism is unique. For instance, given (weak) binary products, we cannot construct a weak paramorphism $[\![\psi]\!]$ such that $[\![\psi]\!]\circ\tau => \psi\circ F.\langle\mathsf{Id}, [\![\psi]\!]\rangle$, nor even a true $\tau^{\cup}$ such that $\tau^{\cup}.(\tau.y) => y$, nor a (transfinite) induction property like (6.5).

In section D.7 we give an impredicative definition of $\mathbb{N}$ in typed lambda calculus, and prove that induction holds by naturality for all terms of type $\mathbb{N}$. This generalizes easily to type $\mu_{\mathsf{w}}F$, and one may expect that the naturality property holds for generalized calculi like CC too. Yet this would not yield an induction theorem *within* the calculus.

One might restrict, as suggested in [73], all quantifications over $\mu_{\mathsf{w}}F$ as to use only its *standard* elements, being those elements that satisfy transfinite induction:

$$\mathsf{St}(x\colon \mu_{\mathsf{w}}F) \ := \ \Pi(U\colon \mathbf{Data}^{\mu_{\mathsf{w}}F};\ h\colon \Pi(y\colon F.\mu_{\mathsf{w}}F :: (F'.U)(y) \to U(\tau.y)) :: Ux)$$

where $(F'.U)(y)$ stands, as in paragraph 6.2.3, for the product of Uz for all immediate predecessors z of $\tau.y$. In particular, if $F.X = \Sigma(a\colon A :: X^{Ba})$, then

$$(F'.U)(a;t) \ = \ \Pi(y\colon Ba :: U(t_y))\ .$$

A declaration '$x\colon \mu F$' may then be replaced by '$x\colon \mu_{\mathsf{w}}F;\ \mathsf{St}\,x$ ' Now, if one's calculus has subtypes, one can use the subtype $\{\,x\colon \mu_{\mathsf{w}}F \mid\colon \mathsf{St}\,x\}$ for μF. If it does not have subtypes, as the Calculus of Constructions, this restriction of quantifications to standard elements does not give a satisfactory solution, for then a quantification over all types cannot be applied to the class of all standard elements of $\mu_{\mathsf{w}}F$.

So it will be more satisfactory to extend the calculus with inductive types as a primitive notion (at the impredicative level). This is done by Coquand and Paulin in [22], using type definitions as described in subsection 5.3.1 and a recursor as we described in paragraph 6.2.3.

Ore [66] discusses extending CC with inductive types either at the impredicative or predicative level.

10.1.2 Weak final algebras

An analogous treatment as in 10.1.1 is possible for co-inductive types. The dual impredicative definition of weak final coalgebras utilizes Σ, but this Σ can be translated into a double use of Π:

$$\begin{aligned}
\nu_{\mathsf{w}} F\colon \mathbf{Data} &:= \Sigma_{\mathsf{w}}(X\colon \mathbf{Data};\, \phi\colon X \to F.X :: X) \\
&:= \Pi(Y\colon \mathbf{Data};\, \Pi(X\colon \mathbf{Data};\, \phi\colon X \to F.X;\, x\colon X :: Y) :: Y) \\
\delta\colon \nu_{\mathsf{w}} F \to F.\nu_{\mathsf{w}} F &:= (X;\phi;x) \mapsto F.(z \mapsto (X;\phi;z)).(\phi.x) \\
&:= u \mapsto u(F.\nu_{\mathsf{w}} F)(X;\phi;x :: F.(z \mapsto (X;\phi;z)).(\phi.x)\,)
\end{aligned}$$

Given another F-coalgebra $(U;\phi)$, the mediating morphism (anamorphism) is

$$[\!(U;\phi)\!]\colon U \to \nu_{\mathsf{w}} F \;:=\; u \mapsto (U;\phi;u)\ .$$

10.2 Using type-free values

The notion of type as mainly used in this thesis comprises that types are introduced together with their values; there are no values without types. An alternative is to define types as sets of basically type-free values. A suitable universe of values is the set of untyped lambda terms, to be taken modulo conversion.

When types may be built by unrestricted comprehension, i.e. $\{\, x \mid\colon \Psi(x)\}$ where $\Psi(x)$ is a formula from second-order logic, then one gets inductive types by taking simply

$$\mu F \;:=\; \bigcap(\, X \mid\colon F.X \subseteq X)\ .$$

Such a system with unrestricted second-order comprehension cannot admit types as values themselves, because this would lead to inconsistency.

10.2.1 Henson's calculus TK

Martin Henson [39] introduced a calculus with kinds organized into a hierarchy of levels, in order to avoid inconsistency. Unlike the hierarchy of universes in type theory, kinds of a higher level do not collect kinds of previous levels, nor do they admit greater cardinalities, for the kind that contains everything, $\{\, x \mid\colon \mathsf{True}\}$, is already of level zero. Rather, kinds of a higher level admit a greater definitional complexity.

- Terms are built from constants c, application $(t\,t)$, lambda abstraction $\lambda x.t$ and $\lambda X.t$ where x is a term variable and X a kind variable of some specific level, and may furthermore contain kind expressions and logical formulae as primitive values
- Atomic formulae include $t \in T$, $t = t'$, and $t\!\downarrow$ (meaning "t is defined"), where t, t' are terms and T is a kind
- Formulae are built from atomic formulae by the ordinary propositional connectives and by quantification over either all terms or all kinds of some specific level
- Types are kinds of level 0

- Kinds of level n are either (1) kind variables of level at most n, (2) comprehensions $\{ x \mid: \Psi(x)\}$ over lambda terms where formula $\Psi(x)$ may contain kind-quantifications over levels below n only, or (3) inductive kinds $\Xi(\Phi, K)$ of level n

Inductive kinds of level n have the form $\Xi(\Phi, K)$, where $\Phi(z, x)$ is a formula that contains kind quantifications below level n only, and K is a kind of level n. Kind $\Xi(\Phi, K)$ is the smallest kind X such that

$$K \subseteq X \quad \text{and} \quad \{ z \mid: \forall x :: \Phi(z, x) \Rightarrow x \in X\} \subseteq X \ . \tag{10.1}$$

So, when there would be no level restriction on comprehension, Ξ would be definable by

$$\Xi(\Phi, K) \ := \ \bigcap(X \mid: (10.1)\,) \ .$$

Put otherwise, $\Xi(\Phi, K)$ is the well-ordered type generated by the relation

$$x \prec z \ := \ z \notin K \wedge \Phi(z, x) \ .$$

An an initial F-algebra, it is

$$\mu(X \mapsto K \cup \{ z \mid: \forall x :: \Phi(z, x) \Rightarrow x \in X\}) \ .$$

Note that the second argument of Ξ is really superfluous, as by taking $\Phi'(z, x) := x \prec z$ we have

$$\Xi(\Phi, K) \ = \ \Xi(\Phi', \emptyset) \ .$$

Therefore we find this type unnecessarily complicated. The comprehension and induction rules given in [39] and some other publications are actually erroneous — and the claimed consistency proof flawed. For example, the induction rule draws a conclusion $\forall z{:}\in \Xi(\Phi, K) :: \psi z$ without a premise $\forall z{:}\in K :: \psi z$. By taking $\Phi(z, x) := \mathsf{False}$, one would obtain $\forall z :: \psi z$ for every formula ψz. Later publications, like [40], had correct rules.

A more fundamental objection is that the usefulness of kinds and formulae as values is negligible, because terms may not be used in place of kinds or formulae. Henson seems to miss this point. We note that the higher level abstraction facilities are quite limited: one can abstract over kinds, but not over functions on kinds. Finally, we remark that the intuitive basis for this hierarchy of kinds is rather weak.

10.3 Inductive universe formation

Predicative universes of types or sets, such as our $\mathbf{Type}_i$, are described by listing the rules for constructing their elements. Then one might add a principle that the universe is actually the least (or initial) one that is closed under these rules, by giving a universe elimination (or recursion) principle. This would make the universe an initial algebra in a category of families of types and extensions between them. N.P. Mendler discusses the categorical semantics of such recursion rules in [61].

One might strengthen type theory by adding a rule for introducing new inductive universes *inside the system* by listing their introduction rules. The difference between universes and ordinary inductive types is that introduction rules for universes, like Π-formation (B.9), when they have a premise that A be a type in the universe, may in subsequent premises quantify over the type (associated with) A itself.

For this purpose it is best to treat universes *à la* Tarski, namely as a pair $(U; T)$ where U is a type and T assigns to each "code" $A{:}\,U$ a type TA. So a universe is a family of types.

For any type S, we define a category $\mathsf{FAM}\,S$. Its object are families in $\mathsf{Fam}\,S$, and its morphisms are given by:

$$(D; s) \to (D'; s') \ \underline{\text{in}}\ \ \mathsf{FAM}\,S \ := \ \{\, f{:}\, D \to D' \mid: \forall d{:}\, D :: sd = s'(f.d)\} \ . \qquad (10.2)$$

A universe formation principle might read: For any endofunctor F on $\mathsf{FAM}\,\mathbf{Type}$ that satisfies some constraints, there is an initial F-algebra. The constraints that are required here are much more difficult to express than for ordinary inductive types, and we will not try to do so.

Example 10.2 The type constructor Π gives rise to an endofunctor P on $\mathsf{FAM}\,\mathbf{Type}$, such that there is a morphism $p{:}\, P.(U; T) \to (U; T)$ just when the universe is closed under Π, i.e. when for $a{:}\, U$ and $b{:}\, U^{Ta}$ there is some $c{:}\, U$ with $Tc \cong \Pi(x{:}\, Ta :: T(bx))$. This P is given by:

$$\begin{array}{rcl} P.(U; T) & := & (\,\Sigma(a{:}\, U :: U^{Ta});\\ & & \quad ((a; b) :: \Pi(x{:}\, Ta :: T(bx)))\ ) \end{array}$$

and for $f{:}\,(U; T) \to (U'; T')$ $\underline{\text{in}}$ $\mathsf{FAM}\,\mathbf{Type}$ the definition of $P.f$ is obtained from (10.2):

$$\begin{array}{ll} & P.f \in P.(U; T) \to P.(U'; T') \\ \Leftrightarrow & \forall a{:}\, U;\ b{:}\, U^{Ta};\ (a'; b') := P.f.(a; b) :: \\ & \quad \Pi(x{:}\, Ta :: T(bx)) = \Pi(x{:}\, T'a' :: T'(b'x)) \qquad \{(10.2) \text{ for } P.f\} \\ \Leftarrow & \forall a{:}\, U;\ b{:}\, U^{Ta};\ (a'; b') := P.f.(a; b) :: \\ & \qquad f.a = a' \ \wedge\ \forall x{:}\, Ta :: f.bx = b'x \qquad \{\forall a{:}\, U :: Ta = T'(f.a)\} \\ \Leftrightarrow & \ \forall a{:}\, U;\ b{:}\, U^{Ta} :: P.f.(a; b) = (f.a;\ f^{Ta}.b) \end{array}$$

We can do the same for other type constructors, and define an endofunctor F such that the carrier of an initial F-algebra may serve as the definition of the universe $\mathbf{Type}_0$. The object part of F is as follows:

$$\begin{array}{rcll} F.(U; T){:}\, \mathsf{FAM}\,\mathbf{Type} & := & (\{\, \mathsf{N} \mid \mathsf{Fin}(n{:}\, \mathbb{N}) \mid \mathsf{Prop} \mid \mathsf{Holds}(P{:}\, \mathbf{Prop}) & \\ & & \ \mid \mathsf{Pi}(a{:}\, U;\ b{:}\, U^{Ta}) \mid \mathsf{Sigma}(a{:}\, U;\ b{:}\, U^{Ta})\,\}; & \\ & & \ (\mathsf{N} :: & \mathbb{N} \\ & & \ \mid \mathsf{Fin}(n) :: & n \\ & & \ \mid \mathsf{Prop} :: & \mathbf{Prop} \\ & & \ \mid \mathsf{Holds}(P) :: & P \\ & & \ \mid \mathsf{Pi}(a; b) :: & \Pi(x{:}\, Ta :: T(bx)) \\ & & \ \mid \mathsf{Sigma}(a; b) :: & \Sigma(x{:}\, Ta :: T(bx)) \\ & & \ )\,) & \end{array}$$

10.4 Bar recursion

Though the scheme of bar recursion introduced by Spector [80] has little to do with inductive types, we include it here because it is so remarkably different from our other recursion schemes. It defines a function $f\colon A^* \to B$ on finite sequences by recursive application to *longer* sequences, until a special termination condition holds.

Well-definedness of such a function depends on a property that a computable function $c\colon A^\omega \to \mathbb{N}$ is *continuous* in the sense that its value on an infinite sequence t depends only on a finite prefix $t|_n$ of t:

$$\forall t\colon A^\omega :: \exists n\colon \mathbb{N} :: \forall u\colon A^\omega :: (t|_n = u|_n \Rightarrow c.t = c.u) \ . \tag{10.3}$$

Let a type A with some default value $a\colon A$ be given. We define an embedding of finite into infinite sequences. (Alternatively, we may restrict the definition to nonempty sequences and replace 'a' by s_0.)

$$s\colon A^* \vdash \quad [s]\colon A^\omega \quad := \quad (i :: \underline{\mathsf{if}}\ i < \#s\ \underline{\mathsf{then}}\ s_i\ \underline{\mathsf{else}}\ a\,)$$

The termination condition of f mentioned above is $c.[s] < \#s$. The point is that, as s grows in length, $c.[s]$ must become constant and the condition will be satisfied when s is long enough.

Theorem 10.1 (Bar recursion) *Classically, one can derive:*

$$\frac{\begin{array}{l} c\colon A^\omega \to \mathbb{N} \\ c \textit{ is continuous} \\ b\colon A^* \to B \\ e(\_\colon B^A)\colon A^* \to B \end{array}}{\begin{array}{l} \exists! f\colon A^* \to B :: \\ \quad \forall s\colon A^* :: f.s = \underline{\mathsf{if}}\ c.[s] < \#s\ \underline{\mathsf{then}}\ b.s\ \underline{\mathsf{else}}\ e(x :: f.(s +\!\!+ \langle x\rangle)\,).s \end{array}}$$

Proof. The equation for f obviously has a least solution $f\colon A^* \to \uparrow B$ in the domain of partial functions. Then for any $s\colon A^*$ such that $f.s = \bot$ one has

$$c.[s] \geq \#s \ \wedge \ \exists(x\colon A :: f.(s +\!\!+ \langle x\rangle) = \bot) \ .$$

Let $g\colon \Pi(s\colon A^*;\ f.s = \bot :: \{x\colon A \mid\colon f.(s +\!\!+ \langle x\rangle) = \bot\})$ be a corresponding choice operator.

To prove totality of f, suppose $f.s = \bot$ for some $s\colon A^*$. Define $t\colon A^\omega$ using total induction by

$$t_i \ := \ \underline{\mathsf{if}}\ i < \#s\ \underline{\mathsf{then}}\ s_i\ \underline{\mathsf{else}}\ g(t|_{i-1})$$

and see that, for $i\colon\geq \#s$, one has $f.(t|_i) = \bot$, hence $c.[t|_i] \geq \#t|_i = i$.
Let n be according to (10.3) for t, then for $i\colon \geq n$ we have $c.t = c.[t|_i]$ as $t|_n = [t|_i]|_n$.
Taking $i := \mathsf{max}\langle \#s, n, c.t+1\rangle$, it follows that $c.t = c.[t|_i] \geq i \geq c.t+1$, contradiction.
Thus $f.s = \bot$ cannot be, and f is total. ■

Function c actually defines a well-founded relation by

$$|\!\prec\!| \ := \ \{s\colon A^*;\ c.[s] \geq \#s;\ x\colon A :: (s +\!\!+ \langle x\rangle, s)\,\} \ .$$

In a constructive calculus, the continuity condition for c is automatically satisfied and may be omitted from the premises. In that case, the principle of bar recursion is essentially stronger than algebraic recursion in typed polymorphic lambda calculus, as Barendsen and Bezem prove [10].

Chapter 11

Reflections and conclusion

In this thesis, we played and experimented with language notations, language definition, and constructive type theories, employing these in studying abstract formulations of principles for inductive types, and the relationships between these. It was not our primary aim to solve specific problems, but rather to unify different approaches and to obtain an overall perspective on them. Looking back we can make a number of reflections on the areas mentioned.

11.1 Mathematical language

We developed the language *ADAM* as a medium to express principles of inductive types. We want to make the following remarks.

11.1.1 Ambiguity. One of the characteristics of *ADAM* is the great amount of ambiguity that we allow in defining and extending the language. We found this comfortable in shaping notations and identifiers that are easy to use, but it may make automatic checking of concrete text difficult or unfeasible. However, actual writing in a formal calculus usually takes place through interactive proof editing, where the author can immediately indicate how to resolve any ambiguities, and proofs are stored in a format that represents the internal structure rather than the concrete appearance. Thus, ambiguity is not insurmountable, but it requires attention of the author not to obscure his text.

11.1.2 Generalized typing. *ADAM* is based on constructive type theory. The availability of generalized type constructors made it possible to treat many constructive calculi as sublanguages, and allowed a unified treatment of parametrization and finite and infinite products. Whether it is desirable or necessary to have a constructive type theory as the logical foundation of the language remains to be seen; we discuss this in section 11.2.

11.1.3 Proof notation. We did not define a formal proof notation, but it is clear that it should be a structured, readable representation of natural deduction style proofs, without the obligation to write down all intermediate results. The very terse proof representation of pure type theory is generally too unwieldy to read, but may be held available to be used for small, almost evident proofs and for proofs that the reader is expected to skip.

Sometimes a linear proof style is convenient. Such a style can very well be embedded within a natural deduction framework, but it can never replace it fully. More remarks on proof notation appear in section 11.4.

11.1.4 Refinement and scope rules. Creative thought has to be given to the subject of scope rules. Often, during the construction of an object or proof, one makes definitions that one would like to extend beyond the current (sub-)proof. This conflicts with the scope rules as used in any modern programming language. Linear proofs have even more difficulty with this (see e.g. U' on page 77), because a definition made within a line of a linear proof would by ordinary rules not even extend to the following lines that lay outside the local expression.

Somewhat related is the representation of stepwise refinement. Part of a construction may be left open to be filled in later on, perhaps guided by side conditions on the construction. In appendix C we experiment a bit with "goal variables" to fill temporary gaps, which are given a value further on in the proof. These are not to be confused with the "place holders" that may be used in interactive editing [50] to temporally hold open places, for these disappear from the proof when they are filled in. Again there are scope problems, for it is not evident which identifiers that were visible at the open place may be used in its refinement.

11.1.5 Variable abstraction. We introduced a double-colon notation to be used both for simple variable abstraction $(x :: b_x)$, quantification $\forall x\colon A :: P_x$, families $(x\colon A :: b_x)$, simple case distinction on an enumerated type $(\mathsf{false} :: b_0 \mid \mathsf{true} :: b_1)$, and pattern matching $(\Box :: b \mid (x, y) \mathrel{+\!\!<} z :: c_{xyz})$. We found it comfortable to work with and clearer than a little dot (as in $\forall x\colon A.P_x$) when the declarations $x\colon A$ take up a bit more space, especially as we can extend these with propositional assumptions and local definitions. It extends nicely to pattern matching, unlike the dot or bracket abstraction $[x]b_x$. We liked the equivalence between finite tuples $(t_0, t_1)\colon T_0 \times T_1$ and abstractions $(i :: t_i)\colon \Pi(i\colon 2 :: T_i)$.

11.2 Constructive Type Theory

11.2.1 Objections. We have used type theory as the mathematical foundation of our research. There are some problems connected with this. The first is the gap between a single-valued predicate and a term denoting the same object. We had to introduce some extra machinery to bridge it (appendix C). It arises as soon as propositions are distinguished from data types. Original Martin-Löf type theory did not make this distinction, but it is needed for higher order quantification. Our solution was satisfactory, but we had to give up the property that any closed expression of some type is reducible to head canonical form for that type, for reasons discussed in subsection C.3.2. Taken together, it removed part of the original simplicity of the propositions-as-types idea.

Secondly, the representation in type theory of equality proofs and type conversion either is rather clumsy, or just omitted from proof terms.

Thirdly, generalized type theory requires parameters to be used for instantiating polymorphic objects and for supplying proofs to operations that have conditions on their arguments. If we understand the meaning of a term to be its computational content, these parameters are superfluous and make object expressions unnecessarily complicated.

11.2.2 Universes. In this thesis, we assumed a hierarchy of universes $\mathbf{Type}_i$, but usually we did not specify in which universe we worked. Most developments could be given in any universe, and it would be desirable if the calculus supported a formalization of this, by means of some kind of "universe parameters". For example, the description of category theory should be parametrized with the universe from which the classes of objects and arrows may be chosen. Next, the theory may be applied to the big category of categories itself by instantiating it to a higher universe.

Rather than having a fixed hierarchy, universes might be formed *inside* the calculus by specifying their basic types and type constructors. The latter may either be chosen from a fixed set, or perhaps, as suggested in section 10.3, be user-defined.

A simpler and easily realisable solution is to allow the formation of the *parametrized universe* $\mathbf{Type}\langle\Upsilon\rangle$ of all types generated from a family of basic types Υ. Then one could define $\mathbf{Type}_0 := \mathbf{Type}\langle\rangle$; $\mathbf{Type}_1 := \mathbf{Type}\langle\mathbf{Type}_0\rangle$, etc.

11.2.3 Alternatives. When selecting a foundation for mathematical language, I would make the following observations.

- Do not use proof information in terms. This gives unnecessary overhead and is counterintuitive for most mathematicians.
- Use classical logic by default, for most mathematicians do not care about constructivism. Constructive arguments may be specially distinguished, if needed.
- An interesting simple type theory is given by Lambek and Scott [46, p. 128]. Its class of types contains only the singleton type 1, binary products $A \times B$, infinity $\mathbb{N}$, powertypes $\mathcal{P}A$, and a type of propositions Ω.

11.3 Language definition mechanism

We used a form of two-level grammar, or Definite Clause Grammar with equations, to define part of our language *ADAM*. We find it both very elegant and powerful, as it subsumes Horn clause logic. It is really a form of logic programming with equations, but note that we regard predicates as special syntax classes rather than translating syntax classes into predicates on strings of characters, as is more common. A lot of research is going on in this area, see e.g. [24].

The mechanism can be used to define both the basic foundational theory and the concrete language with its semantics, but also search strategies for finding missing parts of proofs. It is possible that a (prototype) implementation of a language be automatically generated from the language definition, provided that the definition is set up with executability in mind. One ingredient of the definition mechanism we did not touch upon is the following.

11.3.1 Proving grammar properties. We defined the basic theory around a predicate, '$\Gamma \vdash t{:}T$', and the concrete language around classes like '$\mathit{Term}(\Gamma, \gamma, t, T)$'. We claimed the definition to be such that the following holds:

$$\text{Whenever one has } \mathit{Term}_{\Gamma,\gamma}(t, T)\text{, then } \Gamma \vdash t{:}T\ ,$$

yet we did not prove this. There is need for a formal notation for stating and proving such grammar properties, for example by checking that each production rule for *Term* corresponds to one or a few rules for $\vdash$.

Besides simple implications, one has to check well-definedness of syntactic operations, like:

$$\text{For any } \mathit{Term}\, t\text{, } \mathit{Subst}\, \phi\text{, one has } \mathit{Term}\, t[\phi]\ .$$

This involves an induction on the structure of terms. Existential properties "For all x, there is a y with $p(x, y)$" can be eliminated (as is often done in automatic theorem proving) by introducing an additional syntactic operation: "For all x, $p(x, f(x))$". This transformation is called a "Skolemization". Elementary automatic theorem proving can probably check all grammar properties we need, when we provide a list of them and indicate on which variables to perform induction.

11.4 Proofs and proof notation

11.4.1 What is a proof? A (formal) proof of a statement in the basic theory is normally its derivation tree. There is no necessity to encode this tree as an object in the theory. If we have developed a mathematical language and verified that a proof formulated in this language guarantees derivability of the statement in the basic theory, then the derivation tree of such a proof can be accepted as a formal proof indeed. The textual representation of such a proof can only be accepted when there is some reasonable upper bound on the amount of computation needed to verify it.

11.4.2 Modularization. The basic theory should be embedded within a logical framework [42] for the modularization and parametrization of theories. The framework may also provide facilities for information hiding, like "Abstract Data Types".

11.4.3 Proof format. Leslie Lamport [47] has designed a format for "Structured Proofs" in natural deduction style, featuring a neat numbering scheme for referring to proof steps and assumptions, and allowing linear proofs where appropriate. Such a format may very well be used as a standard format for proofs of theorems.

11.4.4 Local structure. Proofs for distinct steps of a deduction could be given by means of an expression that gives complete combinatorial information of all derivation steps and required facts, but normally one would be satisfied with giving just hints. Checking these hints would require some proof search, which can be defined through logic programming. It may be useful for some facts and assumptions to be marked as "active", meaning that they may be used without being hinted at.

11.4.5 Equality proofs. Suppose we have (a reference to) a proof p proving an equality $a = b$ (or a sequence of such proofs). A proof of $t_a = t_b$, where t_x may be a complicated term, say $f(g(x), c)$, has a number of intermediate steps like $g(a) = g(b)$, derived by extensionality of g. A natural notation for the full proof might be to insert p for variable x in t_x, properly marked, e.g. as

$$\%f(g(\cdot\, p\, \cdot), c)\ .$$

Here, '%' marks the start of the extensionality proof format, and '$(\cdot\quad \cdot)$' marks the insertion of ordinary proof notation.

Essentially the same format can be used for naturality proofs: given a polymorphic term

$$x\colon S[\alpha] \vdash t[x]\colon T[\alpha]$$

and a relation $R\colon A \sim B$, the extended relation (section D.3) might be noted $\%T(\cdot\, R\, \cdot)$, and given a proof p proving $(a, b) \in \%S(\cdot\, R\, \cdot)$, we would have by naturality (theorem D.1):

$$\%t(\cdot\, p\, \cdot) \text{ proving } (t[a], t[b]) \in \%T(\cdot\, R\, \cdot)\ .$$

11.5 Inductive types

We started with inductive subset definitions given by means of rule sets, well-founded relations, or monotonic operators. We went on with the description of inductive types, defined by means of construction and elimination rules or as the initial object of the category of algebras of appropriate signature.

The initial algebra approach allowed us to separate the investigation of forms of inductive type definition from the study of forms of (structural) induction and recursion over an inductive type. For an overview of these forms, we refer to the conclusions of chapter 5 (page 73) and 6 (page 83). Here we list some of the decisions to be made when including inductive types in a language definition.

- Do the inductive types appear as fixed points of functors, or as initial algebras of appropriate signatures? The first option is a special case of the second.
- Are inductive types to be generated by
  1. providing actual parameters for one of the admissible forms of signature or functor, or
  2. writing down the desired signature or functor, which has to be matched against the admissible forms, or
  3. writing down the signature or functor according to (inductive) production rules (section 5.3)?
- For families of mutually inductive types, does each type from the family have a fixed (set of) constructors, or may one have "plain" algebra signatures where the codomain of a constructor may depend on its parameters?

Similarly, for recursive function definitions:

- Are they to be given by providing actual parameters for some recursor, or by writing down the desired recursion equations where the correctness checker has to match these to the admissible forms of recursion?
- Does one allow dependent recursion, and if not, does one include a uniqueness condition?
- Does one allow liberal mutual recursion, using either equality types or syntactic equality checks?

11.6 Directions for further research

The work and ideas presented in this thesis call for further research in the following directions.

1. **Language definition method.** Research on the method of using two-level grammar to define a semi-decidable language which is formally reduced to a foundational theory:
   (a) experiment with using the method on a small language;
   (b) formally define the method.
2. **Mathematical language.**
   (a) A suitable foundation remains to be established, especially when constructive and classical reasoning are to be combined in a single system.
   (b) Develop proof notations for *ADAM*, including e.g. readable notations for structuring the argumentation, modularization, easy use of equality, better scoping rules for definitions made within a proof.
   (c) Formally define a usable subset of *ADAM*.
   (d) Write a readable manual for *ADAM*.
3. **Inductive types.**
   (a) Analyze and compare how inductive types as included in current languages follow our schemes.
   (b) Define a good concrete notation for inductive types in a general language like *ADAM*.
4. **Relational notation.**
   (a) The naturality theorem for simple types should be generalized to dependent types. This will require a non-standard interpretation of generalized type expressions, where type variables are replaced by relations as in 11.4.5.
   (b) The same interpretation may be used for replacing type variables by categorical arrow sets. Thus, one type expression can define both the object and arrow part of a category.

(c) Internalize naturality: naturality of objects in *ADAM* should be available within *ADAM* itself.

Appendix A

Set theory

This appendix contains some basic notions of set theory, and two models of set theory within type theory. First we list the axioms of Zermelo-Fraenkel set theory with choice (ZFC). Then we present two well-known classes of infinite numbers, namely ordinals and cardinals. (Here we used the outline by Manes [54, pp. 71–72], who cites Monk [63].)

In type theory one can define a big type in $\mathbf{Type}_1$ that gives a model of ZFC. We present two models in A.5 and A.6; the latter one uses an inductive type with equations.

Finally, non-wellfounded sets are described in A.7.

A.1 ZFC axioms

We introduce **Set** as a primitive sort, together with a binary predicate *membership*, $(\in)\colon\subseteq \mathbf{Set}^2$, and abbreviate

$$x \subseteq y \;:=\; \forall(z{:}\in x :: z \in y)\ .$$

Please do not get confused by the overloaded use of '$\in$', '$\subseteq$' and other set operations: they are defined for subset types, for families, and for models of ZFC as well. The first interpretation will not be used in this appendix.

The axiom of *extensionality* is about equality of sets:

$$s, t\colon \mathbf{Set} \;\;\vdash\;\; \forall(x :: x \in s \Leftrightarrow x \in t) \Rightarrow s = t \tag{A.1}$$

There are five axioms stating the existence of primitive sets, each accompanied with axioms describing the members of these sets.

Separation: If $P(x)$ is a propositional formula with parameter x:

$$\begin{aligned} P\colon \mathbf{Prop}^{\mathbf{Set}}, s\colon \mathbf{Set} \;\;\vdash\;\; & \{\, x{:}\in s \mid: P(x)\}\colon \mathbf{Set}; \\ & x \in \{\, x{:}\in s \mid: P(x)\} \;\Leftrightarrow\; x \in s \wedge Px \end{aligned} \tag{A.2}$$

Just as in *ADAM*, we use the symbol '$|$:', read "such that", instead of the more conventional '$|$' or ':' because we find the latter two too symmetric and want to use them for other purposes. Note also that '$:\in$' is used for introducing a variable that ranges over a set.

Union, *Power set*, and *Infinity*:

$$\begin{array}{rcll} s\colon \mathbf{Set} & \vdash & \bigcup s\colon \mathbf{Set}; & \\ & & x \in \bigcup s \;\Leftrightarrow\; \exists(y\colon\in s :: x \in y) & \text{(A.3)} \\ s\colon \mathbf{Set} & \vdash & \mathcal{P}(s)\colon \mathbf{Set}; & \\ & & x \in \mathcal{P}(s) \;\Leftrightarrow\; x \subseteq s & \text{(A.4)} \\ & \vdash & \omega\colon \mathbf{Set}; & \\ & & \exists(y\colon\in \omega :: \forall z :: z \notin y) \wedge \forall(y\colon\in \omega :: y \cup \{y\} \in \omega) & \text{(A.5)} \end{array}$$

where $y \cup \{y\} \in \omega$ abbreviates the formula $\exists z\colon\in \omega :: \forall u :: u \in z \Leftrightarrow u \in y \vee u = y$.

Replacement. If F is a unary operation (that may be given as a predicate):

$$\begin{array}{rcll} s\colon \mathbf{Set}, F\colon \mathbf{Set}^{\mathbf{Set}} & \vdash & \{x\colon\in s :: Fx\}\colon \mathbf{Set}; & \\ & & y \in \{x\colon\in s :: Fx\} \;\Leftrightarrow\; \exists(x\colon\in s :: y = Fx) & \text{(A.6)} \end{array}$$

Many more operations on sets may be derived, for example:

$$\begin{array}{rrcl} & \emptyset\colon \mathbf{Set} & := & \{x\colon\in \omega \mid\colon \mathsf{False}\} \\ x\colon \mathbf{Set} \vdash & \{x\} & := & \{z\colon\in \omega :: x\} \\ x, y\colon \mathbf{Set} \vdash & \{x, y\} & := & \{z\colon\in \omega :: Fz\} \\ & & & \underline{\mathsf{where}}\ Fz := \begin{cases} x & \text{if } z = \emptyset \\ y & \text{if } z \neq \emptyset \end{cases} \\ & x \cup y & := & \bigcup\{x, y\} \\ & x \cap y & := & \{z\colon\in x \mid\colon z \in y\} \end{array}$$

The axiom of *foundation* or *regularity* says that the membership relation $\in$ is well-founded. We formulate it in a constructive form:

$$P\colon \mathbf{Prop}^{\mathbf{Set}} \;\vdash\; \forall(x :: \forall(y\colon\in x :: Py) \Rightarrow Px) \Rightarrow \forall(x :: Px) \qquad \text{(A.7)}$$

The *axiom of choice* says that, given a mapping to nonempty sets, there exists a function picking one element of each set. (We use the encodings for functions given below.)

$$s\colon \mathbf{Set} \;\vdash\; (\forall x\colon\in \mathsf{dom}\, s :: s(x) \neq \emptyset) \;\Rightarrow\; (\exists f :: \forall x\colon\in \mathsf{dom}\, s :: f(x) \in s(x)) \qquad \text{(A.8)}$$

A.2 Set encodings

We may use the following standard encodings of pairs, functions, and naturals, using only the above axioms and operations.

$$\begin{array}{rcl} \langle x, y\rangle & := & \{\{x\}, \{x, y\}\} \\ \mathsf{fst}\, p & := & \bigcup\{x\colon\in \bigcup p \mid\colon \exists y\colon \mathbf{Set} :: p = \langle x, y\rangle\} \\ \mathsf{snd}\, p & := & \bigcup\{y\colon\in \bigcup p \mid\colon \exists x\colon \mathbf{Set} :: p = \langle x, y\rangle\} \end{array}$$

$$
\begin{array}{rcl}
X \times Y & := & \bigcup\{x{:}\in X :: \{y{:}\in Y :: \langle x,y\rangle\}\} \\
\mathsf{dom}\, f & := & \{p{:}\in f :: \mathsf{fst}\, p\} \\
\mathsf{cod}\, f & := & \{p{:}\in f :: \mathsf{snd}\, p\} \\
Y^X & := & \{f{:}\, \mathcal{P}(X \times Y) \mid: \forall x{:}\in X :: \exists! y{:}\, \mathbf{Set} :: \langle x,y\rangle \in f\} \\
f(x) & := & \bigcup\{y{:}\in \mathsf{cod}\, f \mid: \langle x,y\rangle \in f\} \\
0 & := & \emptyset \\
n+1 & := & n \cup \{n\}
\end{array}
$$

A.3 Ordinals

Ordinals are special sets, but the class **Ord** of all ordinals is too big to be a set itself. We give two definitions of this class.

A.3.1 Inductive definition of ordinals. The class **Ord** is the least class (i.e. the intersection of all classes) $X{:}\subseteq \mathbf{Set}$ such that:

1. For any x in X, its successor $x \cup \{x\}$ is in X;

2. The union of any *set* of X-members is in X.

Note that, as the empty set $\emptyset$ is the union of the empty set of ordinals, it is an ordinal by clause 2. It is named 0 as well.

This definition gives us a principle of transfinite induction: any predicate on sets that is closed under the clauses above holds for all ordinals. Unfortunately, it is a second order definition that cannot be given in first order logic. However, the following one is equivalent [63]:

A.3.2 First order definition of ordinals. A set (of sets) x is $\in$-*transitive* iff whenever $y \in x$ and $z \in y$ then $z \in x$. An *ordinal* is an $\in$-transitive set x such that all $y \in x$ are also $\in$-transitive.

If x, y are ordinals then (using the axiom of foundation) exactly one of $x \in y$, $x = y$, $y \in x$ occurs (classically), so that **Ord** is linearly ordered via:

$$x \le y \quad := \quad x = y \vee x \in y$$

If X is a nonempty set (or class) of ordinals then $\bigcap X$ is an ordinal and is in X; in particular, X has a least element. Further, for ordinals x, y, $x \le y$ holds iff $x \subseteq y$.

A.4 Cardinals

A *cardinal* is an ordinal which is not equipotent (*equipotent* means "in bijective correspondence") with a smaller ordinal. The class of cardinals is noted '**Card**'.

Given any set A there exists (classically) a unique cardinal $\mathsf{card}(A)$ that is equipotent with A. So cardinals are useful for measuring the size of sets.

If x is a cardinal, x^+ denotes the next largest cardinal. There is no largest cardinal, that is, x^+ always exists. A cardinal x is *regular* iff x is infinite and for every family $y\colon \mathsf{Fam}\,\mathbf{Card}$ with each $y_i < x$ and $\mathsf{card}(\mathsf{Dom}\,y) < x$, it is the case that $\mathsf{card}\,\Sigma y < x$. The first infinite cardinal, ω, is regular, and for any infinite cardinal x, x^+ is regular.

A.5 A model of ZFC

Within type theory, one may represent a set together with all its element sets by a directed graph $(N\colon \mathbf{Type}_0;\ S\colon\subseteq N^2)$ together with a designated root $n\colon N$.

Given graph $(N;S)$, a node $x\colon N$ corresponds to the set of those sets that correspond to the nodes in $S[x] = \{\,y\colon N \mid\colon (x,y) \in S\}$. A partial interpretation as sets of such triples $(N;S;n)$ is recursively specified by:

$$[\![N;S;n]\!] \;=\; \{m\colon\in S[n] :: [\![N;S;m]\!]\,\}\ .$$

We define the type T of directed rooted graphs, followed by an inductive definition of a partial equivalence relation $\equiv$ on T. Only triples where the root starts a wellfounded tree appear in $\equiv$.

$$\begin{array}{rcl}
T\colon \mathbf{Type}_1 & := & \{\,(N\colon \mathbf{Type}_0;\ S\colon\subseteq N^2;\ n\colon N)\,\} \\
R\colon \mathcal{P}T^2 & \vdash & \underline{\mathsf{Define}}\ (\preceq_R)\colon \mathcal{P}T^2\ \underline{\mathsf{by}} \\
(N;S;n) \preceq_R (N';S';n') & := & \forall x\colon\in S[n] :: \exists x'\colon\in S'[n'] :: ((N;S;x),(N';S';x')) \in R \\
(\equiv)\colon \mathcal{P}T^2 & := & \bigcap(R\colon \mathcal{P}T^2 \mid\colon (\preceq_R)\cap(\succeq_R) \subseteq R)
\end{array}$$

Now $\mathsf{ZFC}\colon\subseteq \mathcal{P}T$ will be the subtype of all $\equiv$-equivalence classes. (For non-wellfounded sets, see A.7.)

$$\begin{array}{rrcl}
 & \mathsf{ZFC}\colon\subseteq \mathcal{P}T & := & \{t\colon T;\ t\equiv t :: |t\equiv|\,\} \\
P,Q\colon \mathsf{ZFC} \vdash & P \in_{\mathsf{ZFC}} Q & := & \exists(N;S;n)\colon\in Q;\ x\colon\in S[n] :: (N;S;x) \in P
\end{array}$$

We now define the ZFC set constructions on the type T of triples. So, let $(N;S;n)\colon T$ be a triple, $P\colon \mathcal{P}(T)$ a predicate and $F\colon T^T$ an operation on T.

```
{_T t ∈ (N;S;n) |: P(t)} :=
    ( 1 + N;
      { x:∈ S[n]; P(N;S;x) :: ((0;0), (1;x))
      | (x,y):∈ S :: ((1;x), (1;y))
      }; (0;0) )

⋃_T(N;S;n) :=
    ( 1 + N;
      { x:∈ S[n]; y:∈ Sx :: ((0;0), (1;y))
      | (x,y):∈ S :: ((1;x), (1;y))
      }; (0;0) )

𝒫_T(N;S;n) :=
```

$$\begin{array}{l} (\ \ 1+\mathcal{P}(S[n])+N; \\ \quad \{\ \ P{:}\,\mathcal{P}(S[n]) :: ((0;0),(1;P)) \\ \quad |\ \ P{:}\,\mathcal{P}(S[n]);\ x{:}\in P :: ((1;P),(2;x)) \\ \quad |\ \ (x,y){:}\in S :: ((2;x),(2;y)) \\ \quad \};(0;0)\ ) \end{array}$$

$$\begin{array}{l} \{_T\, x \in (N;S;n) :: Fx\}\ \ := \\ \quad \underline{\mathsf{let}}\ (M_y;Q_y;m_y) := F(N;S;y)\ \underline{\mathsf{in}} \\ \quad (\ \ 1+\Sigma(y{:}\in S[n] :: M_y); \\ \qquad \{\ \ y{:}\in S[n] :: ((0;0),(1;y;m_y)) \\ \qquad |\ \ y{:}\in S[n];\ (u,v){:}\in Q_y :: ((1;y;u),(1;y;v)) \\ \qquad \};(0;0)\ ) \end{array}$$

$$\begin{array}{l} \omega_T\ \ := \\ \quad (\ \ 1+\mathbb{N}; \\ \qquad \{\ \ i{:}\,\mathbb{N} :: ((0;0),(1;i)) \\ \qquad |\ \ i{:}\,\mathbb{N};\ j{:}<i :: ((1;i),(1;j)) \\ \qquad \};(0;0)\ ) \end{array}$$

It is straightforward to extend these constructions to ZFC, e.g. $\bigcup_{\mathsf{ZFC}} Q := \bigcup(t{:}\in Q :: \bigcup_T |t \equiv|\,)$. We leave it to the reader to check that they satisfy the axioms, and that the axioms of extensionality, foundation, and choice are satisfied.

A.6 An inductive model of ZFC

Yet a simpler model uses an inductive type. Note that $\mathsf{Fam}_0{:}\,\mathbf{TYPE}_1 \to \mathbf{TYPE}_1$ is a polynomial functor, with

$$\begin{array}{rcl} \mathsf{Fam}_0(X{:}\,\mathbf{TYPE}_1) & := & \Sigma(D{:}\,\mathbf{Type}_0 :: X^D)\ , \\ \mathsf{Fam}_0(h{:}\,A \to B) & := & (D;a) \mapsto (D;h^D.a)\ . \end{array}$$

We define a membership relation $(\in_{\mathsf{f}}){:}\subseteq X \times \mathsf{Fam}_0\, X$ and a subfamily relation $(\subseteq_{\mathsf{f}}){:}\subseteq \mathsf{Fam}_0^2\, X$:

$$\begin{array}{rrcl} x{:}\,T;\ t{:}\,\mathsf{Fam}\,T \vdash & x \in_{\mathsf{f}} t & := & \exists d{:}\,\mathsf{Dom}\,t :: x = t_d \\ t,t'{:}\,\mathsf{Fam}\,T \vdash & t \subseteq_{\mathsf{f}} t' & := & \forall x{:}\in_{\mathsf{f}} t :: x \in_{\mathsf{f}} t' \end{array}$$

An initial Fam_0-algebra contains families of families of families *ad infinitum.* Modulo the appropriate equation these families model sets.

$$\begin{array}{rcl} (\mathsf{ZFC};\tau) & := & \mu(\mathsf{Fam}_0;E)\ \underline{\mathsf{where}} \\ E(X;\phi) & := & \{f{:}\,\mathsf{Fam}_0^2\,X;\ f_0 \subseteq_{\mathsf{f}} f_1 \wedge f_1 \subseteq_{\mathsf{f}} f_0 :: (\phi.f_0,\phi.f_1)\,\} \\ x \in_{\mathsf{ZFC}} y & := & \exists(f{:}\,\mathsf{Fam}_0\,\mathsf{ZFC};\ i{:}\,\mathsf{Dom}\,f :: y = \tau.f \wedge x = f_i) \end{array}$$

The (total) interpretation $[\![\cdot]\!]{:}\,(\mathsf{ZFC} \triangleright \mathbf{Set})$ is recursively defined by:

$$[\![\tau.f]\!]\ =\ \{x{:}\in_{\mathsf{f}} f :: [\![x]\!]\,\}$$

The extensionality axiom (A.1) follows easily from $E(\mathsf{ZFC};\tau) \subseteq (=_{\mathsf{ZFC}})$ and the following lemma.

Lemma A.1 *For $x{:}\,\mathsf{ZFC}$, $f{:}\,\mathsf{Fam\,ZFC}$, one has:*

$$x \in_{\mathsf{ZFC}} \tau.f \;\Leftrightarrow\; x \in_{\mathsf{f}} f$$

Proof. $\Leftarrow$ is trivial.

$\Rightarrow$: we must prove that if $\tau.f = \tau.f'$ and $x \in_{\mathsf{f}} f'$, then $x \in_{\mathsf{f}} f$. Note that, as $E(X;\phi)$ is an equivalence, one has $(=_{\mathsf{ZFC}}) = E(\mathsf{ZFC};\tau)$, so if $\tau.f = \tau.f'$ then $f' \subseteq_{\mathsf{f}} f$. ■

We define the operations required by the axioms (A.3) till (A.6), writing '$\in$' for $\in_{\mathsf{ZFC}}$, by:

$$\begin{array}{rcl}
\{\,x{:}\in s \mid{:}\; P(x)\} & := & \tau.(x{:}\in s;\; Px :: x\,) \\
\bigcup s & := & \tau.(y{:}\in s;\; x{:}\in y :: x\,) \\
\mathcal{P}(s) & := & \tau.(P{:}\,\mathcal{P}(\mathsf{ZFC}) :: \tau.(x{:}\in s;\; Px :: x)\,) \\
\omega & := & \tau.(i{:}\,\mathbb{N} :: \tau.(h.i)\,)\ \underline{\mathsf{where}} \\
 & & h.0 := \langle\rangle, \\
 & & h.(k+1) := h.k \mathbin{+\!\!+} \langle \tau.(h.k)\rangle \\
\{x{:}\in s :: Fx\,\} & := & \tau.(x{:}\in s :: Fx\,)
\end{array}$$

To check their properties, one applies lemma A.1 over and again.

Finally, the foundation axiom (A.7) holds by induction over the definition of ZFC, and the lemma.

A.7 Anti-foundation

Peter Aczel [4] proposed an alternative view on sets, in which non-wellfounded sets are permitted. He removed the foundation axiom (A.7) from ZFC, and replaced it by an *anti-foundation axiom* (AFA), stating that, given a directed graph, each node x in it corresponds to a set such that the elements of the set are the sets that correspond to the subnodes of x in the graph. Thus:

$$s{:}\,\mathbf{Set};\; R{:}\,\mathcal{P}(s\times s) \vdash \quad \exists! f{:}\,\mathbf{Set}^s :: \forall x{:}\in s :: fx = f[R[x]] \tag{A.9}$$

We may call this system ZFA. A noninductive model of ZFA can be obtained from our T as defined in section A.5, by using the dual equivalence relation:

$$\begin{array}{rcl}
(\equiv') & := & \bigcup(R{:}\,\mathcal{P}T^2 \mid{:}\; R \subseteq (\preceq_R) \cap (\succeq_R)) \\
\mathsf{ZFA} & := & \{t{:}\,T :: |t \equiv'|\,\}
\end{array}$$

Remark that $\equiv'$ is total, that is, $(=_T) \subseteq (\equiv')$.

Appendix B

ADAM's Type Theory

In this appendix we define the type theory ATT that forms the foundation of the language *ADAM* described in chapter 2. It includes impredicative propositions, a hierarchy of universes, strong sums for non-propositions, naturals, finite types, and equality types *à la* Martin-Löf. Without the naturals, finite types, and equality types, it would be Luo's Extended Calculus of Constructions [48].

As this calculus serves merely as a logical foundation, notational convenience is not of primary importance. However, we include a few derived notations to make direct employment of the calculus, as in the examples of appendix C, more comfortable.

We also outline a set-theoretical semantics of the calculus in B.10, assuming a sufficiently strong set theory.

B.1 Abstract syntax

We have the following abstract syntax for terms and contexts, where '::=', '|', '{', '}', and '.' are metasymbols, and T^* stands for a possibly empty list of expressions from class T. We assume a syntax class *Var* of variables, a class *Const* of constants, and a class *Nat* of naturals, with addition '+' and comparison '<', to be used for indexing constants.

$$\begin{array}{rcl} \mathit{Term} & ::= & \mathit{Var} \\ & | & \mathit{Const}(\mathit{Term},^*) \\ & | & (\mathit{Var} :: \mathit{Term}) \ . \\ \mathit{Context} & ::= & \{\mathit{Var}\colon \mathit{Term};\}^* \ . \\ \mathit{Statement} & ::= & \mathit{Term}\colon \mathit{Term} \ . \end{array}$$

So (abstract) *terms* are built from variables, constants with a list of arguments, and abstractions $(v :: t)$, which are like $\lambda v.t$ in lambda calculus. As in typed lambda calculus *à la* Curry, abstractions do not carry the type of the bound variable with them, so terms will not have unique types.

A *context* Γ consists of a sequence of assumptions of the form $v\colon T$, where v is a variable and T a term (representing the type of v). We will define a derivability relation

$\Gamma \vdash t{:}T$. The intended meaning is that, for any correct assignment of values to the typed variables in Γ, term t represents a value of type T.

We write '_' for an anonymous variable. Among the constants in *Const* we use the following primitive ones, listed with their arity, notational sugar, and intended meaning.

Prop	0		Type of all propositions
$\mathbf{Type}_i$	0	for any $Nat\,i$	Type of all types of level i
Π	1		Cartesian product of a family of types
@	2	$f(a)\ :=\ @(f,a)$	Selecting a component from a tuple or function
Σ	1		Disjoint sum of a family of types
$(;)$	2	$(a;b)\ :=\ (;)(a,b)$	Element of a disjoint sum
Σ_elim	1		Elimination on a disjoint sum
n	0	for any $Nat\,n$	Finite type with elements $0,\ldots,n-1$
$(,_n)$	n	for any $Nat\,n$; $(a_0,\ldots,a_{n-1})\ :=\ (,_n)(a_0,\ldots,a_{n-1})$	Element of a cartesian product over finite type n
$\mathbb{N}$	0		Type of natural numbers
s	1		Successor of a natural
$\mathbb{N}$_rec	2		Recursion over the naturals
$\forall$	1		Universal quantification of a family of propositions
hyp	1		Proof of a universal proposition by hypothesis
app	1		Application of a proof of a universal proposition
$\exists$	1		Prop. stating existence of an inhabitant of a type
$\exists$_in	1		Proof of an existential proposition
$\exists$_elim	1		Elimination on an existential proposition
$=$	1	$(a =_A a')\ :=\ @(=(A),(,_2)(a,a'))$	Equality predicate on any type
eq	0		Trivial proof of equality
ac	1		Proof by the axiom of choice

Parentheses may be omitted when no ambiguity arises.

B.2 Meta-predicates

The calculus defines a substitution operation and two predicates on terms and contexts.

B.2.1 Substitution. $s[v := t]$ for terms s, t and variable v, or more generally $s[\phi]$ where ϕ is a list of single substitutions, $\{Var := Term,\}^*\,\phi$, is defined as usual:

$$\begin{array}{rcll} v[\phi] & := & t & \text{if } in(\{v := t\},\phi)\text{, for some } t \\ v[\phi] & := & v & \text{otherwise} \\ c(t_0,\ldots,t_{n-1})[\phi] & := & c(t_0[\phi],\ldots,t_{n-1}[\phi]) & \\ (w :: s)[\phi] & := & (w' :: s[w := w'][\phi]) & (w' \text{ free in neither } (w :: s) \text{ nor } \phi) \end{array}$$

The use of named variables may of course be replaced by some numbering scheme.

B.2.2 Reduction. Reduction is a reflexive and transitive binary predicate $t => t'$ on terms. It is inductively defined by structural rules

$$\frac{}{t => t}$$
$$\frac{t => t' \quad t' => t''}{t => t''}$$
$$\frac{t_i => t'_i \quad (i = 0, \ldots, n-1)}{c(t_0, \ldots, t_{n-1}) => c(t'_0, \ldots, t'_{n-1})}$$
$$\frac{t => t'}{(v :: t) => (v' :: t'[v := v'])} \qquad (v' \text{ not free in } (v :: t)\ )$$

and rules related to specific constants:

$$\begin{array}{rcl} (v :: b)(a) & => & b[v := a] \\ \Sigma\_\mathsf{elim}(t)(a; b) & => & t(a)(b) \\ (t_0, \ldots, t_{n-1})(k) & => & t_k \\ \mathbb{N}\_\mathsf{rec}(b, t)(0) & => & b \\ \mathbb{N}\_\mathsf{rec}(b, t)(\mathsf{s}\, x) & => & t(x)(\mathbb{N}\_\mathsf{rec}(b, t)(x)) \end{array}$$

Two terms are called *convertible* when they reduce to the same thing: $t == t'$ when there is some term t'' such that $t => t''$ and $t' => t''$.

Reduction has the Church-Rosser property: if $t => t'$ and $t => t''$, then $t' => t'''$ and $t'' => t'''$ for some term t'''. Hence we have that convertibility is transitive.

B.2.3 Derivable judgements. A *judgement* consists of a context Γ and two terms t, T. *Derivability* of judgements is denoted by an infix turnstyle and colon, '$\Gamma \vdash t{:}\, T$', and is defined by a set of *rules*. Intuitively, such a judgement represents the assertion that for any assignment to the variables in Γ of values of the respective type, the term t denotes a value of the type denoted by T. See section B.10 for a formal semantics.

The structural rules for variable occurrences are:

$$\frac{\Gamma \vdash A{:}\, \mathbf{Type}_i}{\Gamma;\, v{:}\, A \vdash v{:}\, A} \tag{B.1}$$

$$\frac{\begin{array}{l}\Gamma \vdash t{:}\, T \\ \Gamma \vdash A{:}\, \mathbf{Type}_i\end{array}}{\Gamma;\, v{:}\, A \vdash t{:}\, T} \tag{B.2}$$

and the type T of a judgement may be replaced by a type that is convertible to T:

$$\frac{\begin{array}{l}\Gamma \vdash t{:}\, T \\ T == T'\end{array}}{\Gamma \vdash t{:}\, T'} \tag{B.3}$$

The rule for introducing a bound variable is

$$\frac{\Gamma;\, x{:}\, A \vdash b{:}\, Bx}{\Gamma \vdash (x :: b){:}\, \Pi(A; B)} \tag{B.4}$$

All other rules are of the form

$$\frac{\begin{array}{l}\Gamma \vdash t_0 \colon T_0 \\ \quad\vdots \\ \Gamma \vdash t_{n-1} \colon T_{n-1}\end{array}}{\Gamma \vdash t' \colon T'}$$

for arbitrary Γ, and we will write them on a single line, as

$$t_0 \colon T_0;\ \ldots;\ t_{n-1} \colon T_{n-1}\ \vdash\ t' \colon T'$$

As a derived rule, derivability is closed under substitution:

$$\frac{\begin{array}{l}\Gamma;\ x \colon A;\ \Gamma' \vdash t \colon T \\ \Gamma \vdash a \colon A\end{array}}{\Gamma;\ \Gamma'[x := a] \vdash t[x := a] : T[x := a]}$$

B.3 Universes

A universe is a type whose elements are types themselves. There is a universe **Prop** of propositions and a cumulative hierarchy of universes $\mathbf{Type}_i$, each being an inhabitant of the next one.

$$\begin{array}{rcll} & \vdash & \mathbf{Prop} \colon \mathbf{Type}_0 & \quad (\text{B.5}) \\ & \vdash & \mathbf{Type}_i \colon \mathbf{Type}_{i+1} & \quad (\text{B.6}) \\ P \colon \mathbf{Prop} & \vdash & P \colon \mathbf{Type}_i & \quad (\text{B.7}) \\ T \colon \mathbf{Type}_i & \vdash & T \colon \mathbf{Type}_j \qquad \text{when } i < j & \quad (\text{B.8}) \end{array}$$

If the subscript i of **Type** is irrelevant, it will not be shown.

B.4 Products

The type $\Pi(A; B)$ is thought of as the product of all Bx for $x \colon A$; see rule (B.4) for introducing its elements.

In the following rules exponentiation of types is used, T^A, which stands itself for a product type $\Pi(A;\ (\_ :: T))$. So we have $\vdash (x :: t) \colon T^A$ when $x \colon A \vdash t \colon T$ and x doesn't occur free in T. Rule (B.14) (extensionality) refers to the equality predicate described in section B.8.

$$\begin{array}{rcll} A \colon \mathbf{Type}_i;\ B \colon \mathbf{Type}_i^A & \vdash & \Pi(A; B) \colon \mathbf{Type}_i & \quad (\text{B.9}) \\ A \colon \mathbf{Type}_i;\ P \colon \mathbf{Prop}^A & \vdash & \forall(A; P) \colon \mathbf{Prop} & \quad (\text{B.10}) \\ f \colon \Pi(A; P) & \vdash & \mathsf{hyp}\, f \colon \forall(A; P) & \quad (\text{B.11}) \\ f \colon \Pi(A; B);\ a \colon A & \vdash & fa \colon Ba & \quad (\text{B.12}) \\ p \colon \forall(A; P);\ a \colon A & \vdash & \mathsf{app}(p, a) \colon Pa & \quad (\text{B.13}) \\ f, g \colon \Pi(A; B);\ h \colon \forall(A;\ (x :: fx =_{Bx} gx)) & \vdash & h \colon (f =_{\Pi(A;B)} g) & \quad (\text{B.14}) \end{array}$$

A pair $(A; B)$ as in (B.9) is called a *family of types*. We define the following alternative notations. In the first one, variable v may occur in B.

$$\begin{aligned} (v{:}\, A :: B) &:= (A; (v :: B)) \\ B^A &:= \Pi(\_{:}\, A :: B) \\ P \Rightarrow Q &:= \forall(\_{:}\, P :: Q) \end{aligned}$$

B.5 Sums

In the rules for strong Σ, we use a primitive constant $\mathsf{\Sigma\_elim}$ rather than operations fst and snd.

$$A{:}\, \mathbf{Type}_i;\ B{:}\, \mathbf{Type}_i^A \vdash \Sigma(A; B){:}\, \mathbf{Type}_i \tag{B.15}$$
$$B{:}\, \mathbf{Type}^A;\ a{:}\, A;\ b{:}\, Ba \vdash (a; b){:}\, \Sigma(A; B) \tag{B.16}$$
$$T{:}\, \mathbf{Type}^{\Sigma(A;B)};\ t{:}\, \Pi(x{:}\, A :: \Pi(y{:}\, Bx :: T(x; y))) \vdash \mathsf{\Sigma\_elim}\, t{:}\, \Pi(\Sigma(A; B); T) \tag{B.17}$$

Thus, the expression '$\mathsf{\Sigma\_elim}(x :: (y :: t_{xy}))$' denotes the function that maps $(x; y)$ to t_{xy}. A pattern-matching notation suggestive of this is given in section B.11.

B.6 Finite types

Any $Nat\, n$ denotes a type with n elements, named by the Nat's 0 till $n - 1$.

$$\vdash n{:}\, \mathbf{Type}_i \tag{B.18}$$
$$\vdash k{:}\, n \qquad \text{where } k < n \tag{B.19}$$
$$T{:}\, \mathbf{Type}^n;\ t_i{:}\, T(i) \text{ for } i < n \vdash (t_0, \ldots, t_{n-1}){:}\, \Pi(n; T) \tag{B.20}$$

Sequences of arbitrary length are denoted using angle brackets. This allows elegant definitions of finite products and sums:

$$\begin{aligned} \langle t_0, \ldots, t_{n-1} \rangle &:= (n; (t_0, \ldots, t_{n-1})) \\ B_0 \times B_1 &:= \Pi\langle B_0, B_1 \rangle \\ B_0 + B_1 &:= \Sigma\langle B_0, B_1 \rangle \\ Q_0 \wedge Q_1 &:= \forall\langle Q_0, Q_1 \rangle \end{aligned}$$

B.7 Naturals

The rules for naturals are exactly as in *ADAM*, paragraph 2.9.1.

$$\vdash \mathbb{N}{:}\, \mathbf{Type}_i \tag{B.21}$$
$$\vdash 0{:}\, \mathbb{N} \tag{B.22}$$
$$x{:}\, \mathbb{N} \vdash \mathsf{s}\, x{:}\, \mathbb{N} \tag{B.23}$$
$$T{:}\, \mathbf{Type}^{\mathbb{N}};\ b{:}\, T(0);\ t{:}\, \Pi(x{:}\, \mathbb{N} :: \Pi(h{:}\, Tx :: T(\mathsf{s}\, x))) \vdash \mathsf{\mathbb{N}\_rec}(b, t){:}\, \Pi(\mathbb{N}; T) \tag{B.24}$$

B.8 Equality

Rules for the equality predicate are the following.

$$\begin{aligned} a\colon A;\ b\colon A &\ \vdash\ (a =_A b)\colon \mathbf{Prop} && \text{(B.25)}\\ a\colon A &\ \vdash\ \mathsf{eq}\colon (a =_A a) && \text{(B.26)}\\ \_\colon (A =_{\mathbf{Type}} B);\ a\colon A &\ \vdash\ a\colon B && \text{(B.27)}\\ P, Q\colon \mathbf{Prop};\ h\colon (P \Rightarrow Q) \wedge (Q \Rightarrow P) &\ \vdash\ h\colon (P =_{\mathbf{Prop}} Q) && \text{(B.28)}\\ P\colon \mathbf{Prop};\ p, q\colon P &\ \vdash\ \mathsf{eq}\colon (p =_P q) && \text{(B.29)} \end{aligned}$$

This simple version of type conversion (B.27) has a drawback: correct terms need not normalize, because in an inconsistent context any two types can be proven equal. One has, for example,

$$A\colon \mathbf{Type};\ h\colon (A =_{\mathbf{Type}} (A \to A)) \vdash (x :: xx)(x :: xx)\colon A\ .$$

An explicit conversion construct, as suggested in C.3.2, third point, would prevent this. In any case, terms that are correct in the empty context reduce to head normal form.

There must be an equality rule for all language constructs, stating that two terms constructed from equal subterms are equal. We do not list all these, but one simple and two more complicated cases are:

$$\begin{aligned} a =_A a';\ b =_B b' &\ \vdash\ (a, b) =_{A\times B} (a', b') && \text{(B.30)}\\ A =_{\mathbf{Type}} A';\ B =_{\mathbf{Type}^A} B' &\ \vdash\ \Pi(A; B) =_{\mathbf{Type}} \Pi(A'; B') && \text{(B.31)}\\ A =_{\mathbf{Type}} A';\ (a, b) =_{A^2} (a', b') &\ \vdash\ (a =_A b) =_{\mathbf{Prop}} (a' =_{A'} b') && \text{(B.32)} \end{aligned}$$

This suffices to derive symmetry and transitivity of equality. But rules like (B.31) have a snag: the type of $B'\colon \mathbf{Type}^{A'}$ has to be converted via (B.27) to get $B'\colon \mathbf{Type}^{A}$. An alternative would be to use an equality predicate indexed by *two* type expressions, which have to denote equal types, thus:

$$\_\colon A = B;\ a\colon A;\ b\colon B\ \vdash\ (a\ {}_A{=}_B\ b)\colon \mathbf{Prop}$$

B.9 Existential propositions

As discussed in appendix C, we add strong existential propositions and the axiom of choice. For A a type, $\exists A$ means 'A is inhabited'.

$$\begin{aligned} A\colon \mathbf{Type}_i &\ \vdash\ \exists A\colon \mathbf{Prop} && \text{(B.33)}\\ a\colon A &\ \vdash\ \exists\_\mathsf{in}\, a\colon \exists A && \text{(B.34)}\\ T\colon \mathbf{Type}^{\exists A}&;\\ t\colon \Pi(x\colon A :: T(\exists\_\mathsf{in}\, x))&;\\ d\colon \forall(x, y\colon A :: tx = ty) &\ \vdash\ \exists\_\mathsf{elim}(t)\colon \Pi(\exists A; T) && \text{(B.35)}\\ B\colon \mathbf{Type}^A;\ p\colon \forall(x\colon A :: \exists(Bx)) &\ \vdash\ \mathsf{ac}\, p\colon \exists\, \Pi(A; B) && \text{(B.36)} \end{aligned}$$

Note: in appendix C we write $\exists\_\mathsf{elim}(t|;d)$, rather than $\exists\_\mathsf{elim}\,t$, to make the proof obligation d explicit.

As discussed in C.3.2, we cannot use a reduction rule $\exists\_\mathsf{elim}(t)(\exists\_\mathsf{in}\,a) \Rightarrow ta$. Rather we add an equation:

$$\vdash \mathsf{eq}\colon (\exists\_\mathsf{elim}(t)(\exists\_\mathsf{in}\,a) = ta) \tag{B.37}$$

B.10 Semantics

We wish to assign a simple set-theoretical semantics $[\![t]\!]\sigma$ to terms t (under valuation σ), such that any function $f\colon A \to B$ simply denotes the set of pairs $\{x\colon\in [\![A]\!]\sigma :: \langle x, [\![f]\!]\sigma.x\rangle\}$. Unfortunately, an abstraction $(v :: t)$ doesn't show up the type A of its bound variable v. Therefore we introduce *annotated terms*, where abstractions $(v ::_A t)$ are annotated with this type A. Annotated terms are given by:

$$\begin{array}{rcl} \mathit{ATerm} & ::= & \mathit{Var} \\ & | & \mathit{Const}(\mathit{ATerm},^*) \\ & | & (\mathit{Var} ::_{\mathit{ATerm}} \mathit{ATerm}) \ . \end{array}$$

For any *Term* t, we define its class of *annotations*, a subclass of *Aterm*.

- The only annotation of a variable v is v;
- An annotation of $\mathsf{c}(t_0, \ldots, t_{n-1})$ is $\mathsf{c}(t'_0, \ldots, t'_{n-1})$ where each t'_i is an annotation of t_i.
- An annotation of an abstraction $(v :: t)$ is $(v ::_{A'} t')$ where A' is any *ATerm* and t' is an annotation of t.

All propositions denote subsets of $\{\emptyset\}$. In particular, we will have $[\![\mathsf{False}]\!]\sigma = \emptyset$ and $[\![\mathsf{True}]\!]\sigma = \{\emptyset\}$, and all terms that denote proofs are mapped onto the empty set $\emptyset$. Thus, our semantics pays no respect to the computational contents of proofs.

For any *Const* c and any list of sets $\bar{s}$ of length arity of c, we define a set $[\![\mathsf{c}]\!](\bar{s})$. Empty argument lists are omitted. Note that we use the set encoding of section A.2.

$$\begin{array}{rcl} [\![\mathbf{Prop}]\!] & := & \mathcal{P}\{\emptyset\} \\ [\![\forall]\!](\langle A, P\rangle) & := & \{\,\emptyset \mid\colon \forall x\colon\in A :: \emptyset \in P(x)\} \\ [\![\exists]\!](A) & := & \{\,\emptyset \mid\colon \exists x\colon\in A :: \mathsf{True}\} \\ \multicolumn{3}{l}{[\![\mathsf{hyp}]\!](x), [\![\mathsf{app}]\!](x,y), [\![\exists\_\mathsf{in}]\!](x), [\![\mathsf{ac}]\!](x), [\![\mathsf{eq}]\!]} \\ & := & \emptyset \\ [\![\exists\_\mathsf{elim}]\!](t) & := & \{p\colon\in t :: \langle\emptyset, \mathsf{snd}\,p\rangle\,\} \\ [\![\Pi]\!](\langle A, B\rangle) & := & \{\,f\colon\in (\bigcup \mathsf{cod}\,B)^A \mid\colon \forall x\colon\in A :: f(x) \in B(x)\} \\ [\![@]\!](f, a) & := & f(a) \\ [\![\Sigma]\!](\langle A, B\rangle) & := & \{x\colon\in A;\ y\colon\in B(x) :: \langle x, y\rangle\,\} \\ [\![\,;\,]\!](a, b) & := & \langle a, b\rangle \\ [\![\Sigma\_\mathsf{elim}]\!](t) & := & \{x\colon\in \mathsf{dom}\,t;\ y\colon\in \mathsf{dom}\,t(x) :: \langle\langle x, y\rangle, t(x)(y)\rangle\,\} \end{array}$$

$$\begin{array}{rcl}
[\![n]\!] & := & n \\
[\![\ ,_n]\!](t_0,\ldots,t_{n-1}) & := & \{\langle 0,t_0\rangle,\ldots,\langle n-1,t_{n-1}\rangle\} \\
[\![\mathbb{N}]\!] & := & \omega \\
[\![\mathsf{s}]\!] & := & \{x{:}\in\omega :: \langle x,x+1\rangle\,\} \\
[\![\mathbb{N}\_\mathsf{rec}]\!](b,t) & := & \bigcap(\,f{:}\subseteq\omega\times\mathsf{dom}\,t(0)\ |{:} \\
 & & \qquad \langle 0,b\rangle\in f\wedge\forall\langle x,y\rangle{:}\in f :: \langle x+1,t(x)(y)\rangle\in f) \\
[\![=]\!](A) & := & \{x,x'{:}\in A :: \langle[\![\ ,_2]\!](x,x'),\{\,\emptyset\ |{:}\ x=x'\}\rangle\,\} \\
[\![\mathbf{Type}_i]\!] & := & \bigcap(U\ |{:}\quad \omega\subseteq U\ \wedge\ [\![\mathbb{N}]\!],[\![\mathbf{Prop}]\!]\in U \\
 & & \qquad \wedge\,\forall A{:}\in U;\ B{:}\in U^A :: [\![\Pi]\!]\langle A,B\rangle,[\![\Sigma]\!]\langle A,B\rangle\in U \\
 & & \qquad \wedge\,\forall j{:}<i :: [\![\mathbf{Type}_j]\!]\in U)
\end{array}$$

The last line gives an iterated inductive definition of $[\![\mathbf{Type}_i]\!]$ of the form $\bigcap(U\ |{:}\ F.U\subseteq U)$ for a monotonic functor F that is not bounded in the sense of section 8.1. Existence of such an inductive set cannot be shown in ZFC, for $[\![\mathbf{Type}_0]\!]$ yields already a model of ZFC (see section A.5). So this requires a strengthening of ZFC, that allows one to give inductive set definitions with clauses of the form

$$\forall\,A{:}\in U;\ B{:}\in U^A :: \phi(A;B)\in U\ .$$

We think a suitable large-cardinal axiom will do.

Substituting a special constant ω for i with $j<\omega$ for all $Nat\,j$, the resulting set $D:=[\![\mathbf{Type}_\omega]\!]$ may serve to model types, and $E:=\bigcup D$ to model values.

A *valuation* is a partial function $\sigma\colon Var\to E$. The semantics of an annotated term t assigns to any valuation σ a set $[\![t]\!]\sigma\in E$:

$$\begin{array}{rcl}
[\![v]\!]\sigma & := & \sigma(v)\qquad\text{for } Var\,v \\
[\![\mathsf{c}(t_0,\ldots,t_{n-1})]\!]\sigma & := & [\![\mathsf{c}]\!]([\![t_0]\!]\sigma,\ldots,[\![t_{n-1}]\!]\sigma) \\
[\![(v ::_A b)]\!]\sigma & := & \{x{:}\in[\![A]\!]\sigma :: \langle x,[\![b]\!](\sigma\mid v\mapsto x)\rangle\,\}
\end{array}$$

The semantics of an annotated context Γ' is a set $[\![\Gamma']\!]$ of valuations:

$$\begin{array}{rcl}
[\![\{\}]\!] & := & \{\emptyset\} \\
[\![\Gamma';\ v{:}T']\!] & := & \{\sigma{:}\in[\![\Gamma']\!];\ x{:}\in[\![T']\!]\sigma :: (\sigma\mid v\mapsto x)\,\}
\end{array}$$

We define an annotated term to be *correct* in an annotated context Γ', as follows.

- Any variable v is correct in Γ'
- A term $\mathsf{c}(t_0,\ldots,t_{n-1})$ is correct in Γ' if each t_i is correct in Γ' and moreover, if c is @ and n is 2, then
$$\forall\sigma{:}\in[\![\Gamma']\!] :: [\![t_1]\!]\sigma\in\mathsf{dom}[\![t_0]\!]\sigma$$
and if c is $\forall$, Π, or Σ, and n is 1, then
$$\forall\sigma{:}\in[\![\Gamma']\!] :: \mathsf{fst}[\![t_0]\!]\sigma=\mathsf{dom}\,\mathsf{snd}[\![t_0]\!]\sigma$$

- An abstraction $(v ::_A b)$ is correct in Γ' if b is correct in $\Gamma'; v\colon A$

Validity is defined by:

$$\begin{array}{rcl} \Gamma' \models t'\colon T' & := & \forall \sigma\colon\in [\![\Gamma']\!] :: [\![t']\!]\sigma \in [\![T']\!]\sigma \\ \Gamma' \models s' = t' & := & \forall \sigma\colon\in [\![\Gamma']\!] :: [\![s']\!]\sigma = [\![t']\!]\sigma \end{array}$$

Now, we hope to have a theorem like the following, but the details of assigning annotations are tricky:

Conjecture (Soundness).

1. If $\Gamma \vdash T\colon \mathbf{Type}$, and Γ', T' are correct annotations of Γ, T, and if $s == t$ and s', t' are correct annotations of s, t in Γ' and $\Gamma' \models s', t'\colon T'$, then $\Gamma' \models s' = t'$
2. If $\Gamma \vdash t\colon T$, then for any correct annotations Γ', T' of Γ, T, there is a correct annotation t' of t such that $\Gamma' \models t'\colon T'$

B.11 More derived notations

B.11.1 Σ-elimination. One may use the following notations for elimination on a Σ-type.

$$\begin{array}{rcl} ((x;y) :: t_{xy}) & := & \mathsf{\Sigma\_elim}(x :: (y :: t_{xy})) \\ \mathsf{fst} & := & ((x;y) :: x) \\ \mathsf{snd} & := & ((x;y) :: y) \end{array}$$

B.11.2 Subtypes. When A is a type, and P a predicate on A, then $\Sigma(A;P)$ represents the type of all $a\colon A$ that come with a proof $p\colon Pa$. As all proofs of a proposition are equal, we have that $(a;p) = (a';p')$ just when $a = a'$. This type gets a special notation.

$$\begin{array}{rcl} \{x\colon A \mid\colon P_x\} & := & \Sigma(x\colon A :: P_x) \\ (a|;p) & := & (a;p) \end{array}$$

One may read '|:' as "such that" and '|;' as "because of". Both bind weaker than '::'. As the proof component p of $(a|;p)$ is irrelevant, we sometimes write just a.

Appendix C

Proof elimination in Type Theory

When Type Theory is to be used as a fully fledged foundation of mathematics, presence of powersets, or equivalently impredicative propositions, is indispensable. We remark that, e.g., the 'iota' or Frege's description operator denoting the element of a one-element set is not representable in current type theories. We propose an existential quantifier with a new elimination rule, and show how the iota operator and quotient types are then representable. We use a version of type theory that unifies finite and infinite products and sums in a particularly elegant way.

This material was distributed earlier, together with appendix B, as report [15].

C.1 Introduction

The basic thought of Brouwer's intuitionistic logic was, a proposition should only be acknowledged as true if we have a construction validating its truth. Martin-Löf's Intuitionistic Type Theory (ITT) [56] was developed to clarify this: a proposition was identified with its set of constructions, called a type, and proofs were identified with constructions. From a construction for the existential statement $\exists(x\colon A :: B_x)$, which is identified in ITT with the generalized sum type $\Sigma(x\colon A :: B_x)$, one can construct the witnessing element of A by the function $\mathsf{fst}\colon \Sigma(x\colon A :: B_x) \to A$.[1]

ITT does not allow impredicative quantification; it makes no sense to the orthodox intuitionist to quantify over the class of all propositions before this class is completed. If $\mathbf{Type}_0$ is a universe of types, then types involving $\mathbf{Type}_0$ cannot reside in $\mathbf{Type}_0$ itself.

In traditional set theory, on the other hand, even in a constructive version, one *can* construct the powerset $\mathcal{P}T$ of any set T. This is a set whose elements are definable by arbitrary predicates on T, even those involving quantification over the powerset $\mathcal{P}T$ itself. Thus, the class of propositions is considered to be understood *a priori*.

Coquand and Huet introduced a type theory, the Calculus of Constructions (CC) [21], that has a type of propositions ($\mathbf{Prop}$, residing inside $\mathbf{Type}_0$) that allows impredicative

[1]Subscripts stand for variables that may occur in an expression. The symbol '::' separates typed bound variables from the body of a quantification. We have $(x :: b_x)\colon \Pi(x\colon A :: B_x)$.

quantification. The system has no Σ-constructor. While Luo [48] added (strong) Σ for the higher type universes, it cannot be consistently added for **Prop** [43]. (Propositions in **Prop** are normally interpreted as sets that have at most one element, but Σ builds types with more elements.) One can only define weak existential quantification:

$$\exists_{\mathsf{w}}(x{:}\, A :: P_x) \ := \ \forall(X{:}\,\mathbf{Prop} :: \forall(x{:}\, A :: P_x \Rightarrow X) \Rightarrow X)$$

From a proof of such an existential proposition one cannot construct the witnessing $x{:}\, A$ inside the system, even if this witness is provably unique. Formally, there is no function $f{:}\, \exists_{\mathsf{w}}(x{:}\, A :: P_x) \to A$, and not even a function $f{:}\, \exists_{\mathsf{w}}!(x{:}\, A :: P_x) \to A$, as CC does not allow object construction using proof information. Addition of the latter f would be perfectly valid in the standard set interpretation.

Now, our purpose is to develop type theory into a complete alternative to traditional set theory as a foundation of mathematics. It is not our purpose to extract programs from constructions by omitting redundant proof information. Any kind of reasoning representable in set theory (but not specific to sets) should be representable in our type theory. This involves:

- An impredicative universe of propositions (**Prop**) should be present, so that powertypes are definable: $\mathcal{P}T := (T \to \mathbf{Prop})$. This makes a type theory into a *topos*. Two propositions are equal if they are equivalent. Two proofs of the same proposition are always equal. We will use some appropriate set notations, particularly '$\in$' for subset membership.

- Extra rules for equality types should be present, including type conversion and extensionality. This is standard in ITT, not in CC. A readable notation for fully formal equality proofs is missing. We will use a semi-formal notation, which should guarantee the existence of a proof object.

- For any ordinarily definable object, there should be an expression in type theory denoting it. Equivalently, *function comprehension* should be possible: from a proof that a relation $R{:}\, \mathcal{P}(A \times B)$ is single valued, the corresponding function $f{:}\, A \to B$ should be constructible.

In this paper we take a type theory that satisfies the first two requirements, and study the last point. Traditionally, a (constructive) object definition may consist of a *description*, being a predicate together with a (constructive) proof that the predicate is satisfied by one and only one object. Gottlob Frege [31, § 11] introduced a *description operator* 'ι' (iota) into predicate calculus to denote this object. In a type theory where propositions are distinguished from types, like CC, one cannot obtain the object from the proof.

Let T be the subtype of objects satisfying the predicate. Assume we have

$$p{:}\, \exists x{:}\, T :: \forall y{:}\, T :: x =_T y \ .$$

We need an expression that extracts the object, say $\iota_T(p){:}\, T$. Rather than adding primitive rules for ι, we propose an equivalent principle in section C.3 for making use of proof information in object expressions.

An essentially equivalent principle is proposed by Pavlović in [71, par. 32], but not worked out. The 'new set type' $\{x : A||P_x\}$ proposed by Constable [19, section 3.1] for Nuprl is based on the same idea too, but doesn't really increase the strength of the system: as Nuprl has no impredicative propositions, one can use a Σ-type modulo the appropriate equivalence.

C.2 The basic system

In this paper, we use the variant of type theory described in appendix B, which includes impredicative propositions, a hierarchy of universes, strong sums for non-propositions, finite types and equality types *à la* Martin-Löf.

Although proofs of propositions always have a proof expression, we won't often show this expression. A really practical formal language for proofs would be much more elaborate. Rather, we use the usual informal language to describe proofs.

In our expressions we will also use *goal variables*, starting with a question mark like '?1', and usually followed by a typing. These represent subexpressions to be defined later on. All names that are visible in the context of a goal variable may be used in its definition as well.

C.3 Strong existence

C.3.1 New rules

We introduce a new existential quantifier with a stronger elimination rule than one has for $\exists_{\mathsf{w}}$ as defined in section C.1. Actually, we define $\exists$ not as a quantifier, but as a constructor operating on types, see rule (C.1). The quantifier is then recovered via the subset type by

$$\exists(A;P) \ := \ \exists\{x\colon A \mid\colon Px\} \ .$$

The rules are suggested by viewing the proposition $\exists T$ as the quotient type of T modulo the equivalence relation that identifies everything in T. This quotient type contains at most one equivalence class indeed.

$$A\colon \mathbf{Type} \ \vdash \ \exists A\colon \mathbf{Prop} \tag{C.1}$$

$$a\colon A \ \vdash \ \exists\_\mathsf{in}\, a\colon \exists A \tag{C.2}$$

$$\begin{array}{ll} T\colon \mathbf{Type}^{\exists A}; & \\ t\colon \Pi(x\colon A :: T(\exists\_\mathsf{in}\, x)); & \\ d\colon \forall x,y\colon A :: tx = ty & \vdash \ \exists\_\mathsf{elim}(t|;d)\colon \Pi(\exists A; T) \end{array} \tag{C.3}$$

Note that d is not always shown. The expected reduction rule is

$$\exists\_\mathsf{elim}(t|;d)(\exists\_\mathsf{in}\, a) \ \texttt{=>} \ (ta) \ , \tag{C.4}$$

but see the next subsection.

C.3.2 Difficulties with reduction to canonical form

Looking at (C.4), we see that in order to reduce an *object* expression '$(\exists\_\mathsf{elim}\, t)(p)$' one must obtain the canonical form of the *proof* expression p. Therefore, *if* we wish to preserve the attractive property of constructive type theory that any closed expression of some type is reducible to head canonical form for that type, we must take care of the following points.

- Proof information is no longer irrelevant and cannot be removed from some language constructs. For example, objects of a subtype $\{x\colon A \mid\colon P_x\}$ cannot be just single $a\colon A$ for which there exists a proof $p\colon P_a$, but must really be pairs $(a|;p)$.

- There should be reduction rules for all noncanonical constructs for proof objects. For example, if we have the *axiom of choice* (which holds trivially in pure ITT, without **Prop**, by the identification of proofs and constructions),

$$B\colon \mathbf{Type}^A;\ p\colon \forall(x\colon A :: \exists(Bx)) \ \vdash\ \mathsf{ac}\, p\colon \exists\,\Pi(A;B)\ , \tag{C.5}$$

  then we must also add a reduction rule to the effect that

$$\mathsf{ac}(x :: \exists\_\mathsf{in}\, b_x) \ \texttt{=>}\ \exists\_\mathsf{in}(x :: b_x)\ . \tag{C.6}$$

  Here we encounter a serious problem: not all $p\colon \forall(x\colon A :: \exists\, B_x)$ reduce to the form $(x :: \exists\_\mathsf{in}\, b_x)$. An ad-hoc solution is to make both $\exists\_\mathsf{in}$ and ac implicit, so that reduction rule (C.6) becomes void. An alternative is to use an *untyped* $\exists\_\mathsf{out}$, and reduction rules:

$$\begin{aligned} \mathsf{ac}\, p \ &\texttt{=>}\ \exists\_\mathsf{in}(x :: \exists\_\mathsf{out}(px)) \\ \exists\_\mathsf{out}(\exists\_\mathsf{in}\, a) \ &\texttt{=>}\ a \end{aligned}$$

- Rule (B.28) says that equivalent propositions are equal. However, a canonical proof term of a proposition need not be a canonical term of an equivalent proposition. Therefore, the type-conversion rule (B.27) has to specify a conversion on term a too. The rule might look like

$$e\colon (A =_{\mathbf{Type}} B);\ a\colon A \ \vdash\ (e \gt a)\colon B$$

  together with a bunch of reduction rules for all constructs that prove equality between types.

In short, the system appears to become rather ugly.

We aim for elegance and therefore choose to part with this property of reduction to canonical form. However, as the calculus is still constructive, one may devise a procedure to extract from a given proof a separate term containing its computational content. Several implemented type theories, including Nuprl and the Calculus of Constructions, do already use such a procedure.

C.4 Applications

C.4.1 Iota

C.4.1 From 'exists' to 'iota'. Now, using $\exists\_\mathsf{elim}$ we can define ι, the construct that, from a proof that a type has a unique element, constructs that element. First, $!A$ is the subtype that, if A has a unique element, contains that single element, and is empty otherwise. Next ι is defined by a $\exists\_\mathsf{elim}$ on $\mathsf{fst}\colon !A \to A$, with a very simple proof that fst does always yield the same result on $!A$.

$$\begin{array}{rcl} !(A\colon \mathbf{Type}) & := & \{x\colon A \mid\colon \forall y\colon A :: x =_A y\} \\ \iota_A\colon \exists!A \to A & := & \exists\_\mathsf{elim}(\,\mathsf{fst} \mid; (x|;p), (y|;q)\colon !A :: py(\colon x = y)) \end{array}$$

C.4.2 From 'iota' to 'exists'. The converse is also possible: we can derive (C.1–C.3) with $\exists$ defined as $\exists_{\mathsf{w}}$ when we assume $\iota_A\colon \exists!A \to A$.

$$\begin{array}{rcl} \exists(A\colon \mathbf{Type}) & := & \forall(X\colon \mathbf{Prop} :: \forall(x\colon A :: X) \Rightarrow X) \\ \exists\_\mathsf{in}(x\colon A) & := & (X :: (h :: hx)) \\ \exists\_\mathsf{elim}(t|;d) & := & (p\colon \exists A :: ?1\colon Tp) \end{array}$$

The context of the goal ?1 is

$t\colon \Pi(x\colon A :: T(\exists\_\mathsf{in}\, x))$;
$d\colon \forall x, y\colon A :: tx = ty$;
$p\colon \exists A$.

To solve $?1\colon Tp$, we define S to be the subtype containing all tx for $x\colon A$,

$$S := \{u\colon Tp \mid\colon \exists x\colon A :: u = tx\} .$$

Such a type might be noted as $\{x\colon A :: tx\}$. For $x\colon A$ let $\mathsf{si}\, x\colon S := (tx|; \exists\_\mathsf{in}(x|;\mathsf{eq}))$ be the corresponding S-element.

Using p and d we can prove that S has a unique element:

$$s\colon \exists!S := p(\exists!S)(x\colon A :: \exists\_\mathsf{in}(\mathsf{si}\, x \mid; (y|;e)\colon S :: ?2\colon tx = y)) .$$

The definition of ?2, using d and $e\colon \exists(x\colon A :: y = tx)$, is left to the reader. Then we take

$$?1 := \mathsf{fst}(\iota_S s) .$$

C.4.2 Quotient types

Another application arises with quotient types. These are sometimes added to type theory as primitives [18], but with the strong-existence construct we can *define* them in much the same way as they are defined in set theory. Furthermore, there is a construction dealing with quotient types that should follow from the rules, but which is not derivable in ordinary type theory.

C.4.3 Specification. We wish quotient types to satisfy the following rules:

$$\begin{array}{rcll}
A\colon \mathbf{Type};\ R\colon \mathcal{P}A^2 & \vdash & A/\!/R\colon \mathbf{Type} & \text{(C.7)}\\
a\colon A & \vdash & /\!/\_\mathsf{in}_R\, a\colon A/\!/R & \text{(C.8)}\\
x,y\colon A;\ r\colon (x,y)\in R & \vdash & /\!/\_\mathsf{in}_R\, x = /\!/\_\mathsf{in}_R\, y & \text{(C.9)}\\
\multicolumn{4}{l}{T\colon \mathbf{Type}^{A/\!/R};}\\
\multicolumn{4}{l}{t\colon \Pi(x\colon A :: T(/\!/\_\mathsf{in}\, x));}\\
d\colon \forall(x,y)\colon\in R :: tx = ty & \vdash & /\!/\_\mathsf{elim}(t|;d)\colon \Pi(A/\!/R;\ T) & \text{(C.10)}\\
/\!/\_\mathsf{elim}(t\,|;d)(/\!/\_\mathsf{in}\, a) & => & ta & \text{(C.11)}
\end{array}$$

A typing '$(x,y)\colon\in R$' abbreviates $x,y\colon A;\ r\colon (x,y)\in R$. Note that it is not necessary to require that R be an equivalence relation. Note also that the rules for $\exists$ are exactly those for $/\!/$ with R instantiated to the total relation $((x,y) :: \mathsf{True})$, except that $\exists A$ is a proposition rather than only a type.

C.4.4 Implementation. We present a definition of quotient types satisfying the specification above. It corresponds to the normal set-theoretic construction: $A/\!/R$ is the subtype of those subsets of A that are equivalence classes of some $x\colon A$, where the equivalence class of x is the least subset of A that contains x and is closed under R.

Let $\mathsf{Equiv}(Q\colon \mathcal{P}A^2)$ be the proposition stating that Q is an equivalence relation. First we define the infix relation $\equiv_R$, read 'equivalent modulo R' as the least equivalence containing R. The so-called 'section' $(x \equiv_R)\colon \mathcal{P}A$ stands for the predicate (or subset) $(y :: x \equiv_R y)$, which is the equivalence class of x.

$$\begin{array}{rcl}
(\equiv_R) & := & \bigcap(Q\colon \mathcal{P}A^2\ |\colon R\subseteq Q \wedge \mathsf{Equiv}\,Q)\\
A/\!/R & := & \{P\colon \mathcal{P}A\ |\colon \exists x\colon A :: P = (x\equiv_R)\,\} \qquad (= \{x\colon A :: (x\equiv_R)\})\\
/\!/\_\mathsf{in}_R(x\colon A) & := & ((x\equiv_R)\ |;\exists\_\mathsf{in}(x|;\mathsf{eq}))\\
/\!/\_\mathsf{elim}(t\,|;d) & := & ((P|;e) :: \exists\_\mathsf{elim}((x|;p) :: (?1)tx\ |;(x|;p),(y|;q) :: ?2\colon tx = ty)\,e\,)
\end{array}$$

The goal variables in this last definition still have to be filled in. From the required typing (C.10) of $/\!/\_\mathsf{elim}$ one can deduce that the types of the bound variables are:

$t\colon \Pi(x\colon A :: T(/\!/\_\mathsf{in}\,x))$
$d\colon \forall(x,y)\colon\in R :: tx = ty$
$P\colon \mathcal{P}A$
$e\colon \exists x\colon A :: P = (x\equiv_R)$
$x\colon A;\ p\colon (P = (x\equiv_R))$
$y\colon A;\ q\colon (P = (y\equiv_R))$

So the subexpression tx has type $T(/\!/\_\mathsf{in}_R\,x)$, while type $T(P|;e)$ is required. A type conversion can be inserted at ?1, for:

$$\begin{array}{cll}
 & /\!/\_\mathsf{in}_R\,x & \\
= & ((x\equiv_R)\ |;\exists\_\mathsf{in}(x|;\mathsf{eq})) & \{\text{definition } /\!/\_\mathsf{in}\}\\
= & (P|;e) & \{\text{by } p\colon (P = (x\equiv_R)),\text{ and proof equality}\}
\end{array}$$

Next, a proof for $tx = ty$ has to be inserted at ?2. Remark that we have $(x \equiv_R) = (y \equiv_R)$ by assumptions p and q, hence $x \equiv_R y$ follows from $y \equiv_R y$.

$$\begin{array}{lcl} & tx = ty & \\ \Leftarrow & \forall x, y{:}\, A :: (x \equiv_R y \Rightarrow tz = ty) & \{\text{because } x \equiv_R y\} \\ \Leftrightarrow & (\equiv_R) \subseteq Q & \{\text{taking } Q := ((x,y) :: tx = ty)\ \} \\ \Leftarrow & R \subseteq Q \wedge \mathsf{Equiv}\, Q & \{\text{definition } \equiv_R\} \\ \Leftrightarrow & \mathsf{True} & \{\text{by } d\text{, and } Q \text{ being an equivalence}\} \end{array}$$

This completes the definition of $//\_\mathsf{elim}$.

Finally, (C.9) is derivable:

$$\begin{array}{lcl} & //\_\mathsf{in}_R\, x = //\_\mathsf{in}_R\, y & \\ \Leftarrow & (x \equiv_R) = (y \equiv_R) & \{\text{definition } //\_\mathsf{in}\text{, and proof equality}\} \\ \Leftrightarrow & \forall z{:}\, A :: (x \equiv_R z) = (y \equiv_R z) & \{\text{extensionality}\} \\ \Leftarrow & x \equiv_R y & \{\equiv_R \text{ is an equivalence}\} \\ \Leftarrow & (x,y) \in R & \{R \subseteq (\equiv_R)\} \end{array}$$

C.4.5 A problem with quotients. Let's return to the basic system, without our strong existence rules. Rules for quotient types (C.7–C.11) may be (and have been) added as primitive rules. We present a specification that cannot be solved by these rules, presumably. The rules from section C.3, including the axiom of choice (C.5), do solve it.

Assume we have a quotient type $A//R$, where R is an equivalence relation, and a function f on infinite A-tuples that respects R:

$$\begin{array}{rcl} & & A{:}\, \mathbf{Type};\ R{:}\, \mathcal{P}A^2;\ \mathsf{Equiv}\, R \\ & & f{:}\, A^\omega \to A \\ u, v{:}\, A^\omega & \vdash & \forall (i{:}\, \omega :: (u_i, v_i) \in R) \Rightarrow (fu, fv) \in R \end{array}$$

(In relational notation, the latter property may be expressed as $(f, f) \in R^\omega \to R$.)

The problem is to construct a corresponding function on $A//R$:

$$f'{:}\, (A//R)^\omega \to A//R \text{ such that } \forall u{:}\, A^\omega :: f'(i :: //\_\mathsf{in}\, u_i) = //\_\mathsf{in}(fu)$$

We would naturally expect this to be possible as follows. Suppose we have a tuple $x{:}\, (A//R)^\omega$. Then:
1. For any $i{:}\,\omega$ there exists a $u{:}\, A$ with $x_i = //\_\mathsf{in}\, u$.
2. Thus there exists, by the axiom of choice, a tuple $y = (i :: y_i)$ with $x_i = //\_\mathsf{in}\, y_i$.
3. For such a y, one has $//\_\mathsf{in}(fy){:}\, A//R$.
4. The property of f says that the value of $//\_\mathsf{in}(fy){:}\, A//R$ is independent of the particular choice of the y_i — had we chosen other values z_i with $x_i = //\_\mathsf{in}\, z_i$, then would any z_i be in the same equivalence class as y_i, so $(y_i, z_i) \in R$ for all i, and hence $(fy, fz) \in R$, so $//\_\mathsf{in}(fy) = //\_\mathsf{in}(fz)$.
5. Thus, we can define $f'(x) := //\_\mathsf{in}(fy)$ where y is chosen as in step 2.

When we try to formalize this, we get stuck. The problem is that the quotient elimination rule (C.10) eliminates only a *single* element at a time. Repeated application works for a finite number of elements, but we have to eliminate an *infinite* number.

One might replace (C.10) with a stronger rule, but that would miss the point as the present rules already determine $A/\!/R$ uniquely, up to isomorphism.

C.4.6 Our solution. We show how the $\exists$-rules together with the axiom of choice (C.5) solve the problem.

Assume $x\colon (A/\!/R)^\omega$. We make the following steps, mirroring the proof above.

$$\begin{aligned}
&s1\colon \forall x\colon A/\!/R :: \exists u\colon A :: x = /\!/\_\mathsf{in}\, u\\
&\qquad := \quad /\!/\_\mathsf{elim}(u :: \exists\_\mathsf{in}(u|; \mathsf{eq}))\\
&s2\colon \exists\, \Pi(i\colon\omega :: \{u\colon A \mid\colon x_i = /\!/\_\mathsf{in}\, u\})\\
&\qquad := \quad \mathsf{ac}(i :: s1(x_i))\\
&s3\colon \Pi(i\colon\omega :: \{u\colon A \mid\colon x_i = /\!/\_\mathsf{in}\, u\}) \to A/\!/R\\
&\qquad := \quad (y :: /\!/\_\mathsf{in}(f(i :: \mathsf{fst}\, y_i)))\\
&s4\colon \forall (y, z\colon \Pi(i\colon\omega :: \{u\colon A \mid\colon x_i = /\!/\_\mathsf{in}\, u\}) :: s3(y) = s3(z))\\
&\qquad := \quad ?1\\
&s5\colon A/\!/R \; := \quad s2^{\setminus}\, \exists\_\mathsf{elim}(s3 \mid; s4)
\end{aligned}$$

The skipped proof ?1 runs as follows, for given y, z:

$$\begin{array}{lcl}
 & s3(y) = s3(z) & \\
\Leftarrow & (f(i :: \mathsf{fst}\, y_i), f(i :: \mathsf{fst}\, z_i)) \in R & \{\text{by (C.9)}\}\\
\Leftarrow & \forall i\colon\omega :: (\mathsf{fst}\, y_i, \mathsf{fst}\, z_i) \in R & \{\text{for } f \text{ respects } R\}\\
\Leftarrow & \forall i :: \forall (u|; e), (v|; e')\colon \{u\colon A \mid\colon x_i = /\!/\_\mathsf{in}\, u\} :: (u, v) \in R & \{\text{Type of } y_i,\ z_i\}\\
\Leftarrow & \forall u, v\colon A;\ /\!/\_\mathsf{in}\, u = /\!/\_\mathsf{in}\, v :: (u, v) \in R &
\end{array}$$

To prove this last proposition, assuming $/\!/\_\mathsf{in}\, u = /\!/\_\mathsf{in}\, v$:

$$\begin{array}{lcl}
 & (u, v) \in R & \\
\Leftrightarrow & /\!/\_\mathsf{in}\, u^{\setminus}\, /\!/\_\mathsf{elim}(u :: (u, v) \in R \mid; ?2) & \{\text{by } /\!/\text{-reduction and ?2 below}\}\\
\Leftrightarrow & /\!/\_\mathsf{in}\, v^{\setminus}\, /\!/\_\mathsf{elim}(u :: (u, v) \in R \mid; ?2) & \{\text{assumption}\}\\
\Leftrightarrow & (v, v) \in R & \{\text{by } /\!/\text{-reduction}\}\\
\Leftrightarrow & \mathsf{True} & \{R \text{ is reflexive}\}
\end{array}$$

Finally, $?2\colon \forall (u, u')\colon\in R :: ((u, v) \in R) = ((u', v) \in R)$ is solvable by symmetry and transitivity of R, and propositions being equal when they are equivalent.

C.4.3 Inductive types

In set theory, the existence of a single infinite set ω suffices to construct all inductive sets, for example by the construction of Kerkhoff [45]. The new rules allow us to mirror this construction in our strong type theory, as is shown in section 8.2 of this thesis.

C.5 Conclusion

Set theory is embeddable in type theory, by defining a big type $\mathbf{Set}\colon \mathbf{Type}_1$ and a relation $(\in)\colon \mathcal{P}\mathbf{Set}^2$ such that all ZF axioms are formally derivable. Such a model is defined in section A.5. This implies that type theory is stronger than ZF set theory (because of the extra universes), but only at the level of first order propositions about sets.

However, as to construct objects of other types, or to construct new types (within the universe $\mathbf{Type}_0$), there are constructions possible in set theory that cannot be done in current type theories. We have shown how the addition of a rule for strong proof elimination fills this deficiency. Some type constructors (quotients, inductive types) that have been proposed as primitives become derivable.

We have seen that, if one wishes to use the logic of type theory as a reduction system, our new principle resists the idea of proof irrelevance (section C.3.2). To avoid complications in the proof system, we suggested to distinguish terms as they occur in the proof system from reducible terms as they may be extracted from proofs.

Appendix D

Naturality of Polymorphism

From the type of a polymorphic object we derive a general uniformity theorem that the object must satisfy. We use a scheme for arbitrary inductive type constructors. Applications include natural and dinatural transformations, promotion and induction theorems. We employed the theorem in section 6.3 to prove equivalence between different recursion operators.

The issue was suggested to us by J.G. Hughes of Glasgow University who mentioned property (D.1) during a lecture in Groningen in October 1988. It was discussed in Backhouse's research club which led to our theorem. After sending Hughes an earlier version of our notes we received the draft of a paper [83] from Philip Wadler who derives the same main theorem as we do but in a more formal setting, and gives many promotion-like applications. His paper gave us some entries to the literature.

This is a revision of our report [14]. The notation has been partly adapted to this thesis, and the proofs of theorems D.2 and D.4 have been simplified through replacing inductive arguments by additional applications of the naturality theorem.

D.1 Introduction

Objects of a parametric polymorphic type in a polymorphic functional language like Miranda or Martin-Löf-like systems enjoy the property that instantiations to different types must have a similar behavior. To state this specifically, we use greek letters as type variables, and $T[\alpha]$ stands for a type expression possibly containing free occurrences of α. Stating $t{:}\,T[\alpha]$ means that term t has polymorphic type $T[\alpha]$ (in some implicit context), and hence $t{:}\,T[A]$ for any particular type A. Such instances are sometimes written with a subscript for clarity: $t_A{:}\,T[A]$. Thus the quantification $\forall\alpha$ is implicit. Some type expressions $T[\alpha]$ are *functorial* in α. I.e., there is a polymorphic expression $T[p]{:}\,T[\alpha] \to T[\beta]$ when $p{:}\,\alpha \to \beta$, such that $T[\mathsf{I}_A] = \mathsf{I}_{T[A]}$ and $T[p \mathbin{\bar\circ} q] = T[p] \mathbin{\bar\circ} T[q]$, where I is the identity function and $(\bar\circ)$ denotes forward function composition. It has often been observed (e.g. [76]) that polymorphic functions $f{:}\,U[\alpha] \to V[\alpha]$, where U, V are functorial, must be natural transformations:

$$\text{For any } p{:}\,A \to A', \text{ one has } f_A \mathbin{\bar\circ} V[p] = U[p] \mathbin{\bar\circ} f_{A'} :\ \ U[A] \to V[A'] \tag{D.1}$$

Illustration:

$$\begin{tikzcd} U[A] \arrow[r, "f_A"] \arrow[d, "U[p]"] & V[A] \arrow[d, "V[p]"] \\ U[A'] \arrow[r, "f_{A'}"] & V[A'] \end{tikzcd}$$

For example, any function $\mathsf{rev}\colon \alpha^* \to \alpha^*$, where α^* is the type of lists over α, must satisfy for $p\colon A \to A'$:

$$\mathsf{rev}_A \,\bar{\circ} p^* = p^* \,\bar{\circ}\, \mathsf{rev}_{A'}$$

Unfortunately, type expression $(U[\alpha] \to V[\alpha])$ is generally not functorial itself, because $(\to)$ is *contravariant* in its first argument. One can extend statement (D.1) to *dinatural transformations* in the sense of Mac Lane [51, pp. 214-218], but such a statement is still not provable by induction on the derivation of type correctness of f. In this appendix we develop a generalization to arbitrary types that is provable for all lambda-definable objects of some type. The generalization allows one to derive many properties of polymorphic objects from their type alone, properties which are conventionally proven by induction using the definition of the object, for example "promotion" theorems on functions like:

$$\mathsf{foldr}\colon (\alpha \times \beta \to \beta) \times \beta \to (\alpha^* \to \beta)$$

This promotion theorem says for $p\colon A \to A'$, $q\colon B \to B'$, $c\colon A \times B \to B$ and $c'\colon A' \times B' \to B'$, if

$$c \,\bar{\circ}\, q = (p \times q) \,\bar{\circ}\, c' : \quad A \times B \to B'$$

then:

$$\mathsf{foldr}(c, b) \,\bar{\circ}\, q = p^* \,\bar{\circ}\, \mathsf{foldr}(c', qb) : \quad A^* \to B'$$

If one identifies natural numbers with objects of polymorphic type $(\alpha \to \alpha) \to (\alpha \to \alpha)$, one can even derive Peano's induction axiom.

The essential theorem is in fact Reynolds' abstraction theorem [75]. (The inconsistency in his modelling of polymorphic objects as set-theoretic functions on the class of all types is rather irrelevant.) The basic idea had already been given by Plotkin [74], and a more complicated variant is formed by the logical relations of Mitchell and Meyer [62]. It was generally regarded as merely a representation independence theorem for datatype implementations, while its implications for deriving properties of functional programs seem to have been unrecognized at that time. Quite different approaches are used in Bainbridge, Freyd et al. [9, 32], based on dinatural transformations in a category called **LIN** of certain coherent spaces and linear maps, and in Carboni et al. [17] where the so-called Realizability Universe is constructed. Both approaches appear to be conceptually far more complicated than Reynolds'. There is also a paper by John Gray [36] which deals only with naturality of some particular operations like currying.

Our contribution consists of the inclusion of arbitrary initial types, some applications of a different kind than Wadler's applications, a very attractive proof of the dinaturality property, and a proof of equivalence between two different recursion operators. We expect the theorem to hold for generalized typed lambda calculus too, but leave this to further research.

Survey. In section D.2 we shall define a typed lambda calculus, and in D.3 we show how type constructors correspond to relation constructors. In D.4 we derive the main naturality theorem; in D.5 we give some simple applications; in D.6 we derive a dinaturality result which could not be proven directly. In D.7 we add second-order quantification and derive mathematical induction for the encoded natural numbers. In D.8 we see how polymorphism over types that support certain operations can be treated. For a final application, proving the equivalence between categorical recursion and Mendler recursion, we refer to theorem 6.4 in this thesis.

D.2 Polymorphic typed lambda calculus

The proof of the naturality theorem is by induction on the derivation of the type of an object. So we need to specify the derivation rules of the polymorphic functional language that we shall use. This language may be either a programming language with fixpoints, where the semantic domain of a type is a cpo, or a purely constructive language where types are flat sets and all functions are total.

We will use typed lambda calculus. The syntax for types and for terms is:

$$\begin{array}{rcl} T & ::= & \alpha \mid (T \to T) \mid \Theta\, T^* \ . \\ t & ::= & x \mid (t\, t) \mid \lambda x.t \mid \theta_j \mid \Theta\_\mathrm{elim}(t^*) \ . \end{array}$$

Besides type variables α, we have $(\to)$ as the function type constructor, and an unspecified number of other type constructors Θ, each one constructing from a sequence of types U of some fixed length the least datatype that is closed under a number of object constructors θ_j. Each constructor θ_j has a sequence of arguments of types $F_{jl}[U, \Theta U]$. The types $F_{jl}[\alpha, \beta]$ must be functorial in β, so for $q\colon B \to B'$ we have $F_{jl}[A, q]\colon F_{jl}[A, B] \to F_{jl}[A, B']$. The functions $F_{jl}[A, q]$ are required to preserve relations too (D.3), but this is guaranteed by naturality. We use a categorical elimination construct Θ_elim (compare Hagino [37]), from which other eliminators may be defined.

Note that we often write a single meta-variable, like 'U', to stand for a sequence, '$U_0, \ldots, U_{n-1}$'. Furthermore, a type expression '$U \to V$', where U is a sequence, stands for '$U_0 \to \cdots (U_{n-1} \to V)$'.

The derivability judgement 't has type T under the assumptions $x_j\colon S_j$' is noted $x\colon S \vdash t\colon T$, and is generated by the following rules:

$$\begin{array}{rcll} & & x\colon S \vdash x_j\colon S_j & \{\text{Var-intro}\} \\ x\colon S \vdash f\colon U \to V;\ x\colon S \vdash u\colon U & \Rightarrow & x\colon S \vdash (f\, u)\colon V & \{(\to)\text{-elim}\} \\ x\colon S, y\colon U \vdash v\colon V & \Rightarrow & x\colon S \vdash \lambda y.v\colon U \to V & \{(\to)\text{-intro}\} \\ & & x\colon S \vdash \theta_j\colon F_j[U, \Theta U] \to \Theta U & \{\Theta\text{-intro}\} \\ x\colon S \vdash v_j\colon F_j[U, V] \to V \ (\text{each } j) & \Rightarrow & x\colon S \vdash \Theta\_\mathrm{elim}(v)\colon \Theta U \to V & \{\Theta\text{-elim}\} \end{array}$$

There is an untyped congruence relation $(==)$ on terms, called *conversion*, which is generated by:

$$\begin{array}{rcl} (\lambda y.v\, u) & == & v[y := u] \\ (\Theta\_\mathrm{elim}(v)\, (\theta_j d)) & == & (v_j\, (F_j[\alpha, \Theta\_\mathrm{elim}(v)]\, d)) \end{array}$$

We will define a typed extensional equivalence in section D.3. In section D.7 we will add second-order quantification (in terms of which initial and final datatypes can be defined).

Note that we derive the theorem using only lambda terms, without any reference to models. One can derive the same results as we do in an arbitrary model, see Wadler [83].

D.3 Turning type constructors into relation constructors

Observe that, although we cannot extend a function p from A to A' to a function from $(U[A] \to V[A])$ to $(U[A'] \to V[A'])$, property (D.1) suggests us to consider the binary relation between $f\colon U[A] \to V[A]$ and $f'\colon U[A'] \to V[A']$ that is given by $f \mathbin{\bar{\circ}} V[p] = U[p] \mathbin{\bar{\circ}} f'$. We will use this observation to extend a relation (instead of a function) between A and A' to one between $T[A]$ and $T[A']$.

Definition. A *relation* $R\colon\subseteq A \times A'$ is a set of pairs of terms of types A and A', taken modulo conversion. (Had we used a programming language permitting the construction of non-terminating programs then we would use elements of the corresponding cpo's and R would be required to be closed under directed limits: if $V \subseteq R$ is (pairwise) directed, then $\bigsqcup V \in R$. In particular, $(\bot_A, \bot_{A'}) \in R$ as $\bot$ is the limit of the empty set.)
While we use a colon $(:)$ for typing, $(\in)$ denotes relation-membership.

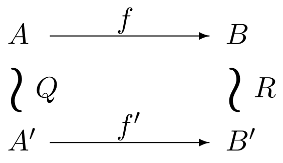
Definition. We will lift any type-constructor Θ to a relation-constructor such that for any relation sequence $R\colon\subseteq A \times A'$ (i.e. $R_i\colon\subseteq A_i \times A'_i$) one has:

$$\Theta R\colon\subseteq \Theta A \times \Theta A'$$

First, as $(A \to B)$ is the greatest type such that for $f\colon A \to B$ one has $\forall x\colon A\,.\,fx\colon B$, we define for $Q\colon\subseteq A \times A'$, $R\colon\subseteq B \times B'$:

$$Q \to R \quad := \quad \{(f,f')\colon (A \to B) \times (A' \to B') \mid\colon \forall (x,x')\colon\in Q\,.\,(fx, f'x') \in R\} \tag{D.2}$$

That is to say, the pair of functions should map related arguments to related results, as illustrated by:


$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, phantom, "\wr\, Q"] & B \arrow[d, phantom, "\wr\, R"] \\ A' \arrow[r, "f'"] & B' \end{tikzcd}$$

If $\Theta\alpha$ is the initial type that is closed under object-constructors

$$\theta_j\colon F_j[\alpha, \Theta\alpha] \to \Theta\alpha$$

where each F_{jl} may be interpreted as a relation constructor that satisfies for relations $P, Q\colon\subseteq \Theta A \times \Theta A'$, $R\colon\subseteq A \times A'$

$$\forall (g,g')\colon\in P \to Q\,.\,(F_{jl}[A,g], F_{jl}[A',g']) \in F_{jl}[R,P] \to F_{jl}[R,Q]\ , \tag{D.3}$$

and preserves identity, then we define ΘR to be the least relation that is closed under:

$$(\theta_j, \theta_j) \in F_j[R, \Theta R] \to \Theta R \tag{D.4}$$

This makes sense since for $P \subseteq Q$ we have $F_j[R,P] \subseteq F_j[R,Q]$ by (D.3), taking g and g' to be the identity I.

Thus, the induction principle for ΘR is: if we wish to prove a predicate P for all pairs in ΘR, then, considering P as a relation $P{:}\subseteq \Theta A \times \Theta A'$, we must show for each j:

$$\forall (d, d'){:}\in F_j[R, P] \,.\, (\theta_j d, \theta_j d') \in P \tag{D.5}$$

Check for example that all pairs in ΘR are convertible to $(\theta_j d, \theta_j d')$ for some j and $(d, d'){:}\in F_j[R, \Theta R]$

Example. Some relation-constructors corresponding to common type-constructors are

$$\begin{array}{rcl} Q + R & := & \{ (x, x'){:}\in Q :: (\mathsf{inl}(x), \mathsf{inl}(x'))\, \} \\ & & \cup \{ (y, y'){:}\in R :: (\mathsf{inr}(y), \mathsf{inr}(y'))\, \} \\ Q \times R & := & \{ (x, x'){:}\in Q, (y, y'){:}\in R :: ((x, y), (x', y'))\, \} \\ \mathsf{Bool} & := & \{ (\mathsf{true}, \mathsf{true}), (\mathsf{false}, \mathsf{false}) \} \end{array}$$

and $\mathbb{N}$ and R^* are the least relations such that:

$$\begin{array}{rcl} \mathbb{N} & = & \{(0, 0)\} \cup \{ (z, z'){:}\in \mathbb{N} :: (\mathsf{s}(z), \mathsf{s}(z'))\, \} \\ R^* & = & \{(\mathsf{nil}, \mathsf{nil})\} \cup \{ (x, x'){:}\in R, (z, z'){:}\in R^* :: (x \mathrel{+\!\!<} z, x' \mathrel{+\!\!<} z')\, \} \end{array}$$

Now, for any type-expression $T[\alpha]$ containing only type-variables from the sequence α, we have extended a relation-sequence $R{:}\subseteq A \times A'$ to a relation $T[R]{:}\subseteq T[A] \times T[A']$. However, T is not necessarily a functor on relations, for it need not preserve composition.

Notice that the same schemes (D.2) and (D.4) describe extensional equality on $A \to B$ and ΘA in terms of the equality on the A and B. Thus we can use the relational interpretation of a type to define extensional equality:

Definition. Extensional equality on a closed type T is given by $T[]$ as a relation:

$$(\models t = t'{:}T) \quad := \quad (t, t') \in T[]$$

We will often write just $t = t'$ rather than $\models t = t'{:}T$. We shall consider only relations that are closed under this extensional equality. Equality for terms of types containing variables will be defined in the next section.

D.4 Naturality of expressions

If we can derive $t{:}T[\alpha]$ then not only $t_A{:}T[A]$ for any type-sequence A, by an appropriate substitution theorem, but also $(t_A, t_{A'}) \in T[R]$ for any relation-sequence $R{:}\subseteq A \times A'$. (It is to be understood that *overloaded* operators, like the effective equality-test $(==_A){:}\, A \times A \to \mathsf{Bool}$ in Miranda, may not be used as if they where polymorphic.) Taking the context into account, we have the following main theorem, similar to the abstraction theorem in [75], the fundamental theorem of Logical Relations in [62], and the parametricity result in [83]:

Theorem D.1 (Naturality) *If $x{:}\, S[\alpha] \vdash t[x]{:}\, T[\alpha]$ then for any sequences A, A', R, s, s', where $R{:}\subseteq A \times A'$ respects extensional equality, and $(s_j, s'_j) \in S_j[R]$, one has:*

$$(t_A[s], t_{A'}[s']) \in T[R]$$

Remark: we say that $t[x]$ is *natural*. The theorem may be generalized to relations of arbitrary arity.

Proof. By a straightforward induction on the derivation of $x\colon S[\alpha] \vdash t[x]\colon T[\alpha]$. We check all rules:

Var-intro. The judgement is $x\colon S[\alpha] \vdash x_j\colon S_j[\alpha]$. By assumption we have $(s_j, s'_j) \in S_j[R]$ indeed.

$(\to)$-elim. The hypotheses say $(f[s], f[s']) \in U[R] \to V[R]$ and $(u[s], u[s']) \in U[R]$. Then by definition of $(\to)$ on relations we obtain:

$$(f[s]u[s], f[s']u[s']) \in V[R]$$

$(\to)$-intro. The hypothesis for the premise $x\colon S[\alpha], y\colon U[\alpha] \vdash v[x,y]\colon V[\alpha]$ says that for any $(s,s')\colon\in S[R]$ and $(u,u')\colon\in U[R]$ one has $(v[s,u], v[s',u']) \in V[R]$.
So $((\lambda y.v[s,y])u, (\lambda y.v[s',y])u') \in V[R]$ as relations are closed under conversion. Hence:

$$(\lambda y.v[s,y], \lambda y.v[s',y]) \in U[R] \to V[R]$$

Θ-intro. We must show:

$$(\theta_j, \theta_j) \in F_j[U[R], \Theta U[R]] \to \Theta U[R]$$

This is an instance of (D.4).

Θ-elim. The (global) hypothesis is: $(v_j[s], v_j[s']) \in F_j[U[R], V[R]] \to V[R]$ for each j. We must show:

$$(\Theta\_\text{elim}(v[s]), \Theta\_\text{elim}(v[s'])) \in \Theta U[R] \to V[R]$$

We use a local induction on the generation of $\Theta U[R]$. Thus we will prove $\Theta U[R] \subseteq P$ where:

$$P \quad := \quad \{(t,t')\colon \Theta U[A] \times \Theta U[A'] \mid\colon (\Theta\_\text{elim}(v[s])t, \Theta\_\text{elim}(v[s'])t') \in V[R]\}$$

Note that:

$$(\Theta\_\text{elim}(v[s]), \Theta\_\text{elim}(v[s'])) \in P \to V[R] \tag{D.6}$$

We check (D.5) for each j:

$$\begin{array}{cll}
 & (d,d') \in F_j[U[R], P] & \\
\Rightarrow & (F_j[\alpha, \Theta\_\text{elim}(v[s])]d, F_j[\alpha, \Theta\_\text{elim}(v[s'])]d') \in F_j[U[R], V[R]] & \{(\text{D.3}) \text{ on } (\text{D.6})\} \\
\Rightarrow & (v_j[s](F_j[\alpha, \Theta\_\text{elim}(v[s])]d), v_j[s'](F_j[\alpha, \Theta\_\text{elim}(v[s'])]d')) \in V[R] & \{\text{global hyp.}\} \\
\Leftrightarrow & (\Theta\_\text{elim}(v[s])(\theta_j d), \Theta\_\text{elim}(v[s'])(\theta_j d')) \in V[R] & \{\text{conversion}\} \\
\Leftrightarrow & (\theta_j d, \theta_j d') \in P & \{\text{def. } P\}
\end{array}$$

■

The theorem suggests the following definition:

Definition. Extensional equality under a sequence of assumptions $x_j\colon S_j$, noted

$$x\colon S[\alpha] \;\models\; t[x] = t'[x]\colon T[\alpha] \;,$$

holds iff, for all sequences A, A'; $R\colon\subseteq\; A \times A'$; $(s, s')\colon\in\; S[R]$ one has $(t_A[s], t'_{A'}[s']) \in T[R]$.

Convention. All of the following definitions and theorems may be taken in the context of a set of assumptions, and all terms should be taken modulo extensional equality under these assumptions.

Definition. A type expression $T[\alpha]$ is called *functorial (in α)* if there is an expression $T[p]$ that satisfies

$$p\colon \alpha \to \beta \vdash T[p]\colon T[\alpha] \to T[\beta] \tag{D.7}$$

and that preserves identity, $\models T[\mathsf{I}] = \mathsf{I}\colon \alpha \to \alpha$. We shall shortly proof that because of naturality, T preserves composition too.

Many applications arise by using a sequence of function-like relations:

Definition. For $p\colon A \to A'$ let the *graph* of p be:

$$(p) \;:=\; \{x\colon A :: (x, px)\,\}$$

A relation $R\colon\subseteq A \times A'$ is called *function-like* if $R = (p)$ for some $p\colon A \to A'$. (Note that, in a cpo, (p) is closed under directed limits iff p is continuous and strict, i.e. $p\bot_A = \bot_{A'}$.)

Fact. For function-like relations we have, if $p\colon A \to A'$, $q\colon B \to B'$:

$$(p) \to (q) \;=\; \{(f, f') \mid\colon f \mathbin{\bar\circ} q = p \mathbin{\bar\circ} f'\colon A \to B'\} \tag{D.8}$$

$$(p)^{\cup} \to (q) \;=\; \{f\colon A \to B :: (f,\, p \mathbin{\bar\circ} f \mathbin{\bar\circ} q)\,\} \tag{D.9}$$

$$\text{using } R^{\cup} \;:=\; \{(x, y)\colon\in R :: (y, x)\,\}$$

Also useful might be, for $f\colon A' \to B$:

$$(p \mathbin{\bar\circ} f,\; f \mathbin{\bar\circ} q) \in (p) \to (q) \tag{D.10}$$

For functorial type expressions, the functorial interpretation must coincide with the relational interpretation:

Theorem D.2 *If $T[\alpha]$ is functorial, then for any $p\colon A \to B$,*

$$(T[p]) = T[(p)] \;.$$

Proof. We derive:

$$\begin{array}{cll}
 & (\mathsf{I}, p) \in (\mathsf{I}_A) \to (p) & \{(\text{D.8})\} \\
\Rightarrow & (T[\mathsf{I}], T[p]) \in T[(\mathsf{I}_A)] \to T[(p)] & \{\text{naturality } T\} \\
\Leftrightarrow & (\mathsf{I}, T[p]) \in (\mathsf{I}_{T[A]}) \to T[(p)] & \{\text{preservation of identity}\} \\
\Leftrightarrow & (T[p]) \subseteq T[(p)] & \{\text{definition } (T[p]),\, \to\}
\end{array}$$

and:

$$\begin{array}{rcl} & (p, \mathsf{I}) \in (p) \to (I_B) & \{(\text{D.8})\} \\ \Rightarrow & (T[p], T[\mathsf{I}]) \in T[(p)] \to T[(\mathsf{I}_B)] & \{\text{naturality } T\} \\ \Leftrightarrow & (T[p], \mathsf{I}) \in T[(p)] \to (\mathsf{I}_{T[B]}) & \{\text{preservation of identity}\} \\ \Leftrightarrow & T[(p)] \subseteq (T[p]) & \{\text{definition } (T[p]), \to\} \end{array}$$

■

So we have, for example, $(p \times q) = (p) \times (q)$ and can safely omit the parentheses. Notice that $p \to q$ can only be read as $(p) \to (q)$, for $\alpha \to \beta$ is not functorial.

Theorem D.3 *Any functorial $T[\alpha]$ must preserve composition, i.e., if $p\colon A \to B$ and $q\colon B \to C$, in some context, then $\models T[p \,\bar{\circ}\, q] = T[p] \,\bar{\circ}\, T[q]\colon A \to C$.*

Proof.

$$\begin{array}{rcl} & (p \,\bar{\circ}\, q, q) \in (p) \to (\mathsf{I}_C) & \{(\text{D.8})\} \\ \Rightarrow & (T[p \,\bar{\circ}\, q], T[q]) \in T[(p)] \to T[(\mathsf{I}_C)] & \{\text{naturality } T\} \\ \Leftrightarrow & (T[p \,\bar{\circ}\, q], T[q]) \in (T[p]) \to (\mathsf{I}_{T[C]}) & \{\text{theorem D.2}\} \\ \Leftrightarrow & T[p \,\bar{\circ}\, q] = T[p] \,\bar{\circ}\, T[q] & \{(\text{D.8})\} \end{array}$$

■

The naturality theorem specializes for polymorphic $t\colon T[\alpha]$ and $p\colon A \to A'$ to:

$$(t_A, t_{A'}) \in T[(p)] \qquad \text{(D.11)}$$

So if $f\colon U[\alpha] \to V[\alpha]$, and hence $(f, f) \in U[p] \to V[p]$ by (D.11), then f is a natural transformation indeed.

D.5 Applications

Example D.1 A simple application is to prove that $f = \lambda x.x$ is the only polymorphic function of type $f\colon \alpha \to \alpha$. Naturality of f says: for any $R\colon\subseteq A \times A'$ we have $(f, f) \in R \to R$.
Now, fix type A and $a\colon A$. Taking $R := \{(a, a)\}$ yields $(fa, fa) \in R$, as $(a, a) \in R$. So $fa = a$ for all a, hence $f = \lambda x.x$ by extensionality of functions.

(In an alternative language where types are cpo's there are two solutions. If $a \neq \bot$, we must take $R := \{(\bot, \bot), (a, a)\}$ and get $fa \in \{\bot, a\}$. Furthermore, for any $b\colon B$ we can get $(fa, fb) \in \{(\bot, \bot), (a, b)\}$. So in case $fa = \bot$ we have $fb = \bot$ for all b, hence $f = \lambda x.\bot$; and in case $fa = a$ we obtain $fb = b$ and hence $f = \lambda x.x$.)

Example D.2 Let * be a (postfix) functor, say of lists, so for $p\colon A \to A'$ we have $p^*\colon A^* \to A'^*$ and $(p^*) = (p)^*$. Let f be a function with the type of foldr, i.e.:

$$f\colon (\alpha \times \beta \to \beta) \times \beta \to (\alpha^* \to \beta)$$

Naturality of f on function-like relations says: if $p\colon A \to A'$, $q\colon B \to B'$ then

$$(f, f) \in (p \times q \to q) \times q \to (p^* \to q)$$

i.e. if $(c, c')\colon\in (p \times q \to q)$ and $(b, b')\colon\in (q)$ then $(f(c, b), f(c', b')) \in (p^* \to q)$. Using (D.8) this equivales: if

$$c \mathbin{\bar{\circ}} q = (p \times q) \mathbin{\bar{\circ}} c' : \;\; A \times B \to B'$$

then:

$$f(c, b) \mathbin{\bar{\circ}} q = p^* \mathbin{\bar{\circ}} f(c', qb) : \;\; A^* \to B'$$

(This is a generalization of the promotion-theorem for forward lists [52].) Notice that the result is independent of the definition of f.

In particular, we have for $\oplus\colon A' \times B' \to B$, as by (D.10) $((p\times q)\bar{\circ}\oplus,\ \oplus\bar{\circ}q) \in (p\times q \to q)$:

$$f((p \times q) \mathbin{\bar{\circ}} \oplus,\ b) \mathbin{\bar{\circ}} q = p^* \mathbin{\bar{\circ}} f(\oplus \mathbin{\bar{\circ}} q,\ qb) \tag{D.12}$$

Another instance yields, as $+\!\!< \mathbin{\bar{\circ}} \mathsf{foldr}(c', b') = (\mathsf{I} \times \mathsf{foldr}(c', b')) \mathbin{\bar{\circ}} c'$ and $\mathsf{foldr}(c', b')\ \mathsf{nil} = b'$:

$$f(+\!\!<, \mathsf{nil}) \mathbin{\bar{\circ}} \mathsf{foldr}(c', b') = f(c', b')$$

Example D.3 [1] Finally, we give an application using *ternary* relations. Let $({}^+)$ be a functor, say of non-empty lists, and $(/_B)$ a (polymorphic) mapping of operators $\oplus\colon B \times B \to B$ into $\oplus/\colon B^+ \to B$.

We will prove: if $f, g\colon A \to B$, and $\oplus\colon B \times B \to B$ is commutative and associative, then for $l\colon A^+$,

$$\oplus/(f^+ l) \;\oplus\; \oplus/(g^+ l) \;\;=\;\; \oplus/((f \oplus^A g)^+ l)$$

where $\oplus^A\colon (A \to B) \times (A \to B) \to (A \to B)$ is the lifted version of $\oplus$. We will regard $\oplus$ as a ternary relation $\oplus\colon\subseteq B \times B \times B$ so that:

$$(x, y, z) \in \oplus \quad := \quad x \oplus y = z$$

We derive:

$$\begin{array}{ll}
 & \forall l\colon A^+ \,.\, (\oplus/(f^+ l),\ \oplus/(g^+ l),\ \oplus/((f \oplus^A g)^+ l)) \in \oplus \\
\Leftarrow & \quad (f^+, g^+, (f \oplus^A g)^+) \in A^+ \to \oplus^+ \\
 & \quad \wedge\, (\oplus/, \oplus/, \oplus/) \in \oplus^+ \to \oplus \\
\Leftarrow & \quad (f, g, (f \oplus^A g)) \in A \to \oplus \qquad\qquad \{\text{naturality of } ({}^+),\ (/)\ \} \\
 & \quad \wedge\, (\oplus, \oplus, \oplus) \in \oplus \times \oplus \to \oplus \\
\Leftrightarrow & \forall x\colon A \,.\, fx \oplus gx = (f \oplus^A g)x \\
 & \wedge\, \forall (x_j, y_j, z_j)\colon\in \oplus \,.\, (x_0 \oplus x_1, y_0 \oplus y_1, z_0 \oplus z_1) \in \oplus \\
\Leftrightarrow & \mathsf{true} \\
 & \wedge\, \forall x_j, y_j \,.\, (x_0 \oplus x_1) \oplus (y_0 \oplus y_1) = (x_0 \oplus y_0) \oplus (x_1 \oplus y_1) \\
\Leftarrow & \quad \oplus \text{ is commutative and associative}
\end{array}$$

[1] Suggested by Roland Backhouse

D.6 Dinatural transformations

Mac Lane [51] defines "dinatural transformations". A result by Backhouse [6, section 6] on a dinaturality property in relational calculus inspired us to the following derivation of dinaturality for polymorphic objects from general naturality. (Backhouse' theorem, second half, bears a relationship to our property (D.15), written as $T[p \parallel p] \cdot T[p \parallel \mathsf{I}] \subseteq T[\mathsf{I} \parallel p]$).)

Definition. A *difunctor* $T[\alpha \parallel \beta]$ is given by a type $T[\alpha \parallel \beta]$ and a term $T(x \parallel y)$ typed by

$$x{:}\, \alpha \to \alpha';\ y{:}\, \beta \to \beta' \vdash T(x \parallel y){:}\, T[\alpha' \parallel \beta] \to T[\alpha \parallel \beta'] \tag{D.13}$$

(note the contravariance in α) such that identity is respected.

Take care not to confuse, for $p{:}\, A \to A'$, the term $T(p \parallel p){:}\, T[A' \parallel A] \to T[A \parallel A']$ with the relation $T[p \parallel p]{:} \subseteq T[A \parallel A] \times T[A' \parallel A']$.

Normally, any type expression $T[\alpha]$ can be written as a difunctor such that $T[\alpha] = T[\alpha \parallel \alpha]$, by separating all covariant and contravariant type variable occurrences, as follows.

If $T[\alpha] = \alpha_i$, take $T[\alpha \parallel \beta] := \beta_i$ and $T(x \parallel y) := y_i$.

If $T[\alpha] = U[\alpha] \to V[\alpha]$, a difunctor is given by:

$$\begin{array}{rcl} T[\alpha \parallel \beta] & := & U[\beta \parallel \alpha] \to V[\alpha \parallel \beta] \\ T(x \parallel y) & := & U(y \parallel x) \circ\!\!\to V(x \parallel y) \\ \text{using } u \circ\!\!\to v & := & \lambda f.(u \mathbin{\bar\circ} f \mathbin{\bar\circ} v) \end{array}$$

Remark that, as a relation, $(u \circ\!\!\to v) = (u^{\cup}) \to (v)$ by (D.9).

If $T[\alpha] = \Theta U[\alpha]$, one needs a difunctorial type constructor $\Theta'(\alpha \parallel \beta)$ such that $\Theta U = \Theta'(U \parallel U)$. Normally, if Θ is already functorial one can take just $\Theta'(\alpha \parallel \beta) := \Theta\beta$, otherwise the language is required to contain such a constructor. Then one takes:

$$\begin{array}{rcl} T[\alpha \parallel \beta] & := & \Theta'(U[\beta \parallel \alpha] \parallel U[\alpha \parallel \beta]) \\ T(x \parallel y) & := & \Theta'(U[y \parallel x] \parallel U(x \parallel y)) \end{array}$$

Theorem D.4 (dinaturality) *When $\vdash t{:}\, T[\alpha]$ where T can be written as a difunctor, $T[\alpha] = T[\alpha \parallel \alpha]$, then for any $p{:}\, A \to A'$ one has:*

$$\vdash T(\mathsf{I} \parallel p)\, t_A = T(p \parallel \mathsf{I})\, t_{A'} \tag{D.14}$$

Proof. Naturality of $T(x \parallel y)$ as typed by (D.13) gives, using $(\mathsf{I}, p) \in \mathsf{I}_A \to p$ and $(p, \mathsf{I}) \in p \to \mathsf{I}_{A'}$:

$$(T(\mathsf{I} \parallel p),\, T(p \parallel \mathsf{I})) \ \in\ T[p \parallel p] \to T[\mathsf{I}_A \parallel \mathsf{I}_{A'}]\ . \tag{D.15}$$

Naturality of $t{:}\, T[\alpha \parallel \alpha]$ gives:

$$(t_A, t_{A'}) \in T[p \parallel p]\ .$$

Together this gives $(T(\mathsf{I} \parallel p)\, t_A,\, T(p \parallel \mathsf{I})\, t_{A'}) \in T[\mathsf{I}_A \parallel \mathsf{I}_{A'}]$. As T preserves identity relations, we obtain (D.14). ∎

Corollary D.5 *All polymorphic functions $f\colon U[\alpha] \to V[\alpha]$, where U and V can be written as difunctors, are* dinatural transformations. *That is to say, they satisfy for $p\colon A \to A'$:*

$$U(p \parallel \mathsf{I}) \mathbin{\bar{\circ}} f_A \mathbin{\bar{\circ}} V(\mathsf{I} \parallel p) = U(\mathsf{I} \parallel p) \mathbin{\bar{\circ}} f_{A'} \mathbin{\bar{\circ}} V(p \parallel \mathsf{I}) : \; U[A' \parallel A] \to V[A \parallel A']$$

Example D.4 Any function

$$f\colon (\alpha \times \beta \to \beta) \times \beta \to (\alpha^* \to \beta)$$

will satisfy for $p\colon A \to A'$, $q\colon B \to B'$:

$$((p \times q \multimapdotbothA \mathsf{I}) \times \mathsf{I}) \mathbin{\bar{\circ}} f_{AB} \mathbin{\bar{\circ}} (\mathsf{I}^* \multimapdotbothA q) = ((\mathsf{I} \times \mathsf{I} \multimapdotbothA q) \times q) \mathbin{\bar{\circ}} f_{A'B'} \mathbin{\bar{\circ}} (p^* \multimapdotbothA \mathsf{I})$$

Applied to some $(\oplus, b)\colon (A' \times B' \to B) \times B$, this is our result (D.12) in section D.5.

D.7 Second-order languages

We may use a second-order language, where, say, $\forall\alpha.T[\alpha]$ is a type with:

$$\begin{array}{rcl} x\colon S \vdash t\colon T[\alpha], \text{ where } \alpha \text{ does not occur free in } S & \Rightarrow & x\colon S \vdash t\colon \forall\alpha.T[\alpha] \\ x\colon S \vdash t\colon \forall\alpha.T[\alpha] & \Rightarrow & x\colon S \vdash t\colon T[U] \end{array}$$

The appropriate extension of relations is, if $R[Q]\colon\subseteq T[A] \times T[A']$ for any $Q\colon\subseteq A \times A'$ (that is closed under extensional equality):

$$\forall\rho.R[\rho] \;:=\; \{\, (t,t') \mid\colon \forall A, A'\colon \mathbf{Type}.\forall Q\colon\subseteq A \times A' \,.\, (t,t') \in R[Q]\}$$

As an application, we define type $\mathbb{N}$, $\mathbf{z}\colon \mathbb{N}$ and $\mathbf{s}\colon \mathbb{N} \to \mathbb{N}$ by:

$$\begin{array}{rcl} \mathbb{N} & := & \forall\alpha.((\alpha \to \alpha) \to (\alpha \to \alpha)) \\ \mathbf{z} & := & \lambda f.\lambda a.a \\ \mathbf{s} & := & \lambda m.\lambda f.\lambda a.f(mfa) \end{array}$$

We will prove Peano's induction-axiom.

Theorem D.6 *When for $P\colon\subseteq \mathbb{N}$ one has*

$$\mathbf{z} \in P \;\wedge\; \forall m\colon\in P :: \mathbf{s}m \in P$$

then $n \in P$ for all $n\colon \mathbb{N}$.

Proof. Remember that naturality of n says that for any types A, A', and relation $Q\colon\subseteq A \times A'$, we have $(n,n) \in (Q \to Q) \to (Q \to Q)$.
The proof is in two steps.

1. Take a *predicate-like* relation $Q := \{n\colon\in P :: (n,n)\,\}$. The assumptions say $(\mathbf{s},\mathbf{s}) \in (Q \to Q)$ and $(\mathbf{z},\mathbf{z}) \in Q$, hence by naturality of n we get $(n\mathbf{s}, n\mathbf{s}) \in (Q \to Q)$ and $(n\mathbf{sz}, n\mathbf{sz}) \in Q$, i.e. $n\mathbf{sz} \in P$.

2. What remains to prove is $n\mathbf{sz} = n$, i.e. for any type A, $f\colon A \to A$, $a\colon A$ we must prove $n\mathbf{sz}fa = nfa$.
Taking $Q := \{(m,x)\colon \mathbb{N} \times A \mid\colon mfa = x\}$, naturality guarantees $(n\mathbf{sz}, nfa) \in Q$ provided $(\mathbf{s}, f) \in (Q \to Q)$ and $(\mathbf{z}, a) \in Q$. But these properties hold by definition of $\mathbf{s}$ and $\mathbf{z}$. ■

D.8 Overloaded operators

We remarked that polymorphic functions may not use overloaded operators, like an effective equality-test $(==_A): A \times A \to \mathsf{bool}$ that is defined only for some types A. However, if we require types instantiated for type-variables to support certain operations, we can give similar requirements on relations. Such restrictions may be provided explicitly by a "type class" in the language Haskell [38].

Definition. Let z be the class of types α with associated operations $v_i: T_i[\alpha]$, and let A and A' be two "instances" of z with operations $t_i: T_i[A]$ and $t'_i: T_i[A']$. The same written in Haskell:

```
class z α where { v₁ ::  T₁[α] ;; ...;; vₙ ::  Tᵢ[α] }
instance z A where { v₁ = t₁ ;; ...;; vₙ = tₙ }
instance z A′ where { v₁ = t′₁ ;; ...;; vₙ = t′ₙ }
```

A relation $R: \subseteq A \times A'$ is said to *respect* class z, iff for each i, one has $(t_i, t'_i) \in T_i[R]$.

For example, consider the class `Eq` of types `a` with equality-test:

```
class Eq a where (==) ::  a -> a -> Bool
```

Relation R respects `Eq` iff for all $(x, x'): \in R$, $(y, y'): \in R$ one has $(x == y) = (x' == y')$: `Bool`. Note that not all relations have this property, hence $(==)$ is not natural. But one can prove the following variant:

Restricted naturality. If expression s has type $S[\alpha]$ for any instance α of class z as above, which is expressed in Haskell by

```
s ::  z α => S[α]
```

then for any relation $R: \subseteq A \times A'$ that respects z we have that $(s, s) \in S[R]$.

Acknowledgement. Thanks are due to Roland Backhouse and Wim Hesselink for many comments that greatly improved upon our presentation.

Index

Bibliography

[1] C. J. Aarts, R. C. Backhouse, P. Hoogendijk, T. S. Voermans, and J. van der Woude, *A Relational Theory of Datatypes*. Available via anonymous ftp from `ftp.win.tue.nl` in directory `pub/math.prog.construction`, Eindhoven University of Technology 1992. 12

[2] H. Abrahamson and V. Dahl, *Logic Grammars*. Springer-Verlag 1989. 19

[3] Peter Aczel, *An introduction to Inductive Definitions*. In: **Handbook of Mathematical Logic** (ed. Jon Barwise), North Holland 1977, pp. 739–782. 44, 45, 47

[4] Peter Aczel, *Non-well-founded Sets*. CSLI Lecture Notes no. 14, Stanford 1988. 93, 125

[5] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman, *Do-it-yourself type theory*. **Formal Aspects of Computing 1** (1989), pp. 19–84.

[6] R.C. Backhouse, *On a Relation on Functions*. In: **Beauty Is Our Business—A Birthday Salute to Edsger W. Dijkstra** (ed. W.H.J. Feijen e.a.), Springer Verlag 1990, pp. 7–18. 153

[7] R.C. Backhouse, P.J. de Bruin, G.R. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude, *Relational catamorphisms*. In: **Constructing Programs From Specifications**, North Holland 1991, pp. 287–318.

[8] Roland Backhouse and Henk Doornbos, *Mathematical Induction Made Calculational*. CS-report 94-16, Eindhoven University of Technology 1994. 12

[9] E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott, *Functorial Polymorphism*. **Theoretical Computer Science 70** (1990), pp. 35–64. 145

[10] Erik Barendsen and Marc Bezem, *Bar Recursion versus Polymorphism*. Technical Report 81, Utrecht Research Institute for Philosophy, Utrecht University 1991. 112

[11] N.G. de Bruijn, *A survey of the project AUTOMATH*. In: **To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism** (ed. Seldin and Hindley), Academic Press 1980, pp. 579–607. 11

[12] N.G. de Bruijn, *The Mathematical Vernacular, a language for mathematics with typed sets*. In: **Workshop on Programming Logic**, Marstrand Sweden 1987. 11

[13] P.J. de Bruin, *Towards decidable Constructive Type Theories as practical descriptive and programming languages*. Master's thesis, report 87-6, Dept. of Informatics, University of Nijmegen 1987.

[14] P.J. de Bruin, *Naturalness of Polymorphism*. Report CS8916, Dept. of Mathematics and Computing Science, University of Groningen 1989. 144

[15] P.J. de Bruin, *Proof elimination in Type Theory*. Report CS9202, Dept. of Mathematics and Computing Science, University of Groningen 1992. 135

[16] R. Burstall and B. Lampson, *A kernel language for abstract data types and modules.* In: **Semantics of Data Types 1984**, LNCS 173, pp. 1–50. 30

[17] A. Carboni, P.J. Freyd, and A. Scedrov, *A Categorical Approach to Realizability and Polymorphic Types.* In: **Mathematical Foundations of Programming Language Semantics 1987**, LNCS 298, pp. 23–42. 145

[18] R.L. Constable e.a., *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall 1986. 11, 72, 102, 139

[19] Robert L. Constable, *Type Theory as a Foundation for Computer Science.* In: **Theoretical Aspects of Computer Science 1991**, LNCS 526, pp. 226–243. 137

[20] R.L. Constable and N.P. Mendler, *Recursive Definitions in Type Theory.* In: **Logics of Programs 1985**, LNCS 193, pp. 61–78.

[21] Th. Coquand and G. Huet, *A Theory of Constructions.* In: **Semantics of Data Types** (ed. G. Kahn e.a.), Sophia Antipolis 1985. 12, 15, 107, 135

[22] Thierry Coquand and Christine Paulin, *Inductively defined types.* In: **COLOG-88**, LNCS 417, pp. 50–66, and **Workshop on Programming Logic 1989**, report 54, Programming Methodology Group, Göteborg, pp. 191–208. 72, 78, 108

[23] Thierry Coquand, *Pattern matching with dependent types.* In: **Proceedings of the 1992 Workshop on Types for Proofs and Programs**, Göteborg 1992. 12

[24] D. DeGroot and G. Lindstrom (ed.), *Logic Programming: Functions, Relations, and Equations*, Prentice Hall 1986. 115, 161

[25] Peter Dybjer and Herbert Sander, *A Functional Programming Approach to the Specification and Verification of Concurrent Systems.* In: **Workshop on Specification and Verification of Concurrent Systems**, Stirling 1988, and **Formal Aspects of Computing 1** (1989), pp. 303–319. 88

[26] Peter Dybjer, *An inversion principle for Martin-Löf's type theory.* In: **Workshop on Programming Logic 1989**, report 54, Programming Methodology Group, Göteborg, pp. 177–190.

[27] Roy Dyckhoff, *Category Theory as an extension of Martin-Löf Type Theory.* Report CS/85/3, Dept. of Computational Science, University of St. Andrews 1985.

[28] H.-D. Ehrich, *Specifying algebraic data types by domain equations.* In: **Foundations of Computation Theory 1981**, LNCS 117, pp. 120–129.

[29] Maarten M. Fokkinga and Erik Meijer, *Program Calculation Properties of Continuous Algebras.* Report CS-R9104, CWI Amsterdam 1991.

[30] Maarten M. Fokkinga, *Law and Order in Algorithmics.* Ph.D. Thesis, Twente University of Technology 1992. 52, 57, 59, 75

[31] G. Frege, *Grundgesetze der Arithmetik* (vol. 1), Jena 1893. 136

[32] P.J. Freyd, J.Y. Girard, A. Scedrov, and P.J. Scott, *Semantic Parametricity in Polymorphic Lambda Calculus.* In: **Logic in Computing Science 1988**, IEEE, pp. 274–279. 145

[33] A.J.M. van Gasteren, *On the shape of mathematical arguments.* Ph.D. thesis, Eindhoven 1988, and LNCS 445 (1990).

[34] J.A. Goguen and J. Meseguer, *Eqlog: equality, types, and generic modules for logic programming.* In [24], pp. 295–363.

[35] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF*. LNCS 78 (1979). 11, 101

[36] John W. Gray, *A Categorical Treatment of Polymorphic Operations*. In: **Mathematical Foundations of Programming Language Semantics 1987**, LNCS 298, pp. 2–22. 145

[37] Tatsuya Hagino, *A Typed Lambda Calculus with Categorical Type Constructors*. In: **Category Theory and Computer Science 1987**, LNCS 283, pp. 140–157. 50, 54, 146

[38] P. Hudak and P. Wadler, editors, *Report on the Functional Programming Language Haskell*. Technical Report, Yale University and University of Glasgow, Dept. of Computer Science, December 1988. 155

[39] Martin C. Henson and Raymond Turner, *A Constructive Set Theory for Program Development*. In: **8th Conf. on Foundations of Software Technology and Theoretical Computer Science**, LNCS 338 (1988), pp. 329–347. 109, 110

[40] Martin C. Henson, *Program Development in the Constructive Set Theory TK*. **Formal Aspects of Computing 1** (1989), pp. 173–192. 110

[41] C.A.R. Hoare, *Communicating Sequential Processes*. **Communications of the ACM 21** (1978), pp. 666–677. 88

[42] G. Huet and G. Plotkin (eds.), *Logical Frameworks*. Cambridge 1991. 116

[43] Bart Jacobs, *The Inconsistency of Higher Order Extensions of Martin-Löf's Type Theory*. **Journ. Philosophical Logic 18** (1988), pp. 399–422. 136

[44] Bart Jacobs, *Categorical Type Theory*. Ph.D. Thesis, University of Nijmegen, 1991. 10

[45] Robert Kerkhoff, *Eine Konstruktion absolut freier Algebren*. **Mathematische Annalen 158** (1969), pp. 109–112. 92, 94, 142

[46] J. Lambek and P.J. Scott, *Introduction to higher order categorical logic*. Cambridge 1986. 13, 61, 115

[47] Leslie Lamport, *How to Write a Proof*. SRC report 94, DEC Systems Research Center 1993. 116

[48] Zhaohui Luo, *ECC, an Extended Calculus of Constructions*. In: **Logic in Computer Science 1989**, IEEE, pp. 386–395. 107, 126, 136

[49] QingMing Ma and John C. Reynolds, *Types, Abstraction, and Parametric Polymorphism, Part 2*. In: **Mathematical Foundations of Programming Semantics 1991**, LNCS 598, pp. 1–40.

[50] Lena Magnuson and Bengt Nordström, *The ALF Proof Editor and its Proof Engine*. In: **Types for Proofs and Programs** (Nijmegen 1993), LNCS 806, pp. 213–237. 12, 114

[51] S. Mac Lane, *Categories for the working mathematician*. Graduate Texts in Mathematics 5, Springer-Verlag 1971. 145, 153

[52] Grant Malcolm, *Algebraic Data Types and Program Transformation*. Ph.D. Thesis, University of Groningen 1990. 42, 75, 89, 152

[53] Jan Małuszyński, *Attribute Grammars and Logic Programs: A Comparison of Concepts*. In: **Attribute Grammars, Applications and Systems**, Prague 1991, LNCS 545, pp. 330–357. 19

[54] Ernest G. Manes, *Algebraic Theories*. Graduate Texts in Mathematics 26, Springer-Verlag 1976. 57, 61, 65, 92, 120

[55] P. Martin-Löf, *Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions.* In: **Second Scandinavian Logic Symposium** (ed. J.E. Fenstad), North-Holland 1971, pp. 179–216. 42, 45

[56] P. Martin-Löf, *Constructive Mathematics and Computer Programming.* In: **Logic, Methodology, and Philosophy of Science VI**, 1979 (ed. L.J. Cohen e.a.), North-Holland 1982, pp. 153–175. 11, 14, 68, 135

[57] Lambert Meertens, *Constructing a calculus of programs.* In: **Mathematics of Program Construction 1989** (ed. J.L.A. van de Snepscheut), LNCS 375, pp. 66–90. 54, 74

[58] L.G.L.T. Meertens, *Paramorphisms.* **Formal Aspects of Computing 4** (1992), pp. 413–424. 74

[59] N.P. Mendler, *Inductive Definition in Type Theory.* Ph.D. Thesis, Cornell University 1987. 14, 78

[60] N.P. Mendler, *Recursive Types and Type Constraints in Second-Order Lambda Calculus.* In: **Logic in Computer Science 1987**, IEEE, pp. 30–36. 79

[61] N.P. Mendler, *Predicative Type Universes and Primitive Recursion.* In: **Logic in Computer Science 1991**, IEEE, pp. 173–185. 110

[62] J.C. Mitchell and A.R. Meyer, *Second-order logical relations.* In: **Logics of Programs 1985**, LNCS 193, pp. 225–236. 145, 148

[63] J.D. Monk, *Introduction to Set Theory.* McGraw-Hill 1969. 120, 122

[64] Yiannis N. Moschovakis, *Elementary Induction on Abstract Structures.* Studies in Logic and the Foundation of Mathematics, North Holland 1974.

[65] P. Odifreddi, *Classical Recursion Theory.* Studies in Logic and the Foundation of Mathematics, North Holland 1989. 14

[66] Christian-Emil Ore, *The Extended Calculus of Constructions (ECC) with Inductive Types.* **Information and Computation 99** (1992), pp. 231-264. 107, 108

[67] Ross Paterson, *Reasoning about Functional Programs.* Ph.D. thesis, University of Queensland 1987. 100

[68] Christine Paulin-Mohring, *Inductive Definitions in the system Coq — Rules and Properties.* In: **Typed Lambda Calculi and Applications** (Utrecht 1993), LNCS 664, pp. 328–345. 12, 72

[69] Lawrence C. Paulson, *Constructing Recursion Operators in Intuitionistic Type Theory.* Cambridge 1984. 47

[70] Lawrence C. Paulson, *Logic and computation — Interactive proof with Cambridge LCF.* Cambridge 1987. 11, 101

[71] Duško Pavlović, *Constructions and Predicates.* In: **Category Theory and Computer Science 1991**, LNCS 530, pp. 173–196. 137

[72] Kent Petersson and Dan Synek, *A Set Constructor for Inductive Sets in Martin-Löf's Type Theory.* In: **Workshop on Programming Logic 1989**, report 54, Programming Methodoly Group, Göteborg, pp. 162–175. 69

[73] F. Pfenning and Ch. Paulin-Mohring, *Inductively Defined Types in the Calculus of Constructions.* In: **Mathematical Foundations of Programming Semantics 1989**, LNCS 442, pp. 209–228. 12, 108

[74] G.D. Plotkin, *Lambda-definability in the full type hierarchy.* In: **To H.B. Curry: Essays on combinatory logic, lambda calculus, and formalism** (ed. Seldin and Hindley), Academic Press, New York 1980, pp. 363–373. 145

[75] J.C. Reynolds, *Types, abstraction, and parametric polymorphism.* In: **Information Processing 1983** (ed. R.E.A.Mason), North-Holland, Amsterdam, pp. 513–523. 145, 148

[76] David E. Rydeheard, *Functors and Natural Transformations.* In: **Category Theory and Computer Programming 1985**, LNCS 240, pp. 43–57. 50, 144

[77] D.S. Scott, *Domains for denotational semantics.* In: **Automata, Languages and Programming 1982** (ed. M.Nielsen, E.M.Schmidt), LNCS 140, pp. 577–613. 98

[78] M. Sintzoff, M. Weber, Ph. de Groote, J. Cazin, *Definition 1.1 of the generic development language Deva.* ToolUse-project, Research report, December 1991, Unité d'Informatique, Université Catholique de Louvain, Belgium. 12

[79] M.B. Smyth and G.D. Plotkin, *The Category-theoretic Solution of Recursive Domain Equations.* **Siam Journal of Computing 11** (1982), pp. 761–783. 98

[80] C. Spector, *Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles formulated in current intuitionistic mathematics.* In: **Proc. Symp. in Pure Mathematics V** (ed. J.C.E.Dekker), AMS, Providence 1962, pp. 1–27. 112

[81] A.S. Troelstra and D. van Dalen, *Constructivism in Mathematics.* Studies in Logic and the Foundation of Mathematics 123 and 125, North-Holland 1988.

[82] David Turner, *A New Formulation of Constructive Type Theory.* In: **Workshop on Programming Logic 1989**, report 54, Programming Methodology Group, Göteborg, pp. 258–294.

[83] Phil Wadler, *Theorems for free!.* In: **Functional Programming Languages and Computer Architecture 1989** (London), ACM Press, pp. 347–359. 144, 147, 148

[84] Matthias Weber, *Formalization of the Bird-Meertens Algorithmic Calculus in the Deva Meta-Calculus.* In: **Programming Concepts and Methods** (ed. Broy and Jones), North Holland 1990, pp. 201–232. 12

[85] M. Weber, M. Simons, C. Lafontaine, *The Generic Development Language DEVA.* LNCS 738 (1993). 12, 27, 30

[86] A. van Wijngaarden e.a., *Revised Report of the Algorithmic Language Algol 68.* Springer Verlag 1976. 19

[87] Martin Wirsing, *Algebraic Specification.* In: **Handbook of Theoretical Computer Science** (ed. J.van Leeuwen), Elsevier 1990, pp. 675–788. 63, 90

Note. 'LNCS' refers to the series "Lecture Notes in Computer Science", Springer-Verlag, Berlin.

Inductieve Typen in Constructieve Talen

Samenvatting

Deze dissertatie gaat over constructieve talen: talen om wiskundige constructies formeel in uit te drukken. Het begrip *constructie* omvat niet alleen berekeningen, zoals die in een programmeertaal kunnen worden uitgedrukt, maar ook beweringen en bewijzen, zoals die in een wiskundige logica kunnen worden uitgedrukt, en in het bijzonder de constructie van gestructureerde wiskundige objecten zoals rijtjes en bomen. *Typen* kan men zich voorstellen als klassen van zulke objecten, en *inductieve typen* zijn typen waarvan de objecten gegenereerd worden door productieregels.

Het doel van deze dissertatie is tweeledig. Ten eerste ben ik op zoek naar talen waarin de wiskundige zijn inspiraties goed gestructureerd, correct, en toch zo vrij mogelijk kan uitdrukken. Ten tweede wil ik de uiteenlopende benaderingen van inductieve typen in één kader samenbrengen, zodat men kan zien hoe de diverse constructie- en afleidingsregels uit een enkel basis-idee voortvloeien en ook hoe deze regels eventueel gegeneraliseerd kunnen worden. Als basis-idee gebruik ik het begrip *initiële algebra* uit de categorieëntheorie.

Mijn onderzoek naar wiskundige talen heeft niet tot een afgerond voorstel geleid. De huidige presentatie beperkt zich tot algemene overwegingen en een deels formele, deels informele beschrijving van een taal, ADAM. Deze dient vervolgens als medium voor de studie van inductieve typen, die het hoofdbestanddeel van de dissertatie vormt.

De opzet van ADAM is als volgt. Om de geldigheid van de in de taal geformuleerde argumenten te garanderen, behoeft deze een degelijke grondslag. Hiervoor stel ik een constructieve type-theorie ATT samen, een combinatie van de "Intuitionistic Theory of Types" van P. Martin-Löf en de "Calculus of Constructions" van Th. Coquand. Teneinde alle wiskundige redeneervormen te kunnen omvatten, voeg ik de iota- of descriptie-operator van Frege toe. Het is niet noodzakelijk om inductieve typen als basisprincipe op te nemen; natuurlijke getallen volstaan om deze te construeren.

Op deze grondslag bouw ik vervolgens de taal ADAM door te bezien hoe we constructies en bewijzen die ik tegenkwam of zelf opstelde zo natuurlijk mogelijk maar wel volgens de regels van typetheorie kon opschrijven. De formele definitie van ADAM, voor zover beschikbaar, en haar semantiek in termen van de onderliggende type-theorie worden gelijktijdig gegeven door een twee-niveau-grammatica. Dit maakt het in beginsel mogelijk de taal naar behoefte met behoud van geldigheid uit te breiden met notaties of deeltalen voor speciale toepassingen, zoals programmacorrectheid. De voorgestelde notaties dienen dan ook niet als onaantastbaar te worden beschouwd. Het enige kenmerkende taalelement is wellicht de notatie voor (en het consistente gebruik van) *families* van objecten.

Als voorbereiding op inductieve typen geef ik eerst de klassieke benaderingen van inductieve definities weer, waarna ik de benodigde machinerie in ADAM introduceer – de beginselen van categorieëntheorie en algebra.

De kern van de verhandeling wordt gevormd door de beschrijving en rechtvaardiging van inductieve typen als initiële algebra's. Eerst beschouw ik op abstract niveau de diverse manieren waarop inductieve typen gespecificeerd kunnen worden en hoe deze specificaties (in de vorm van een polynomiale functor) een algebra-signatuur bepalen, eventueel met gelijkheden. Vervolgens analyseer en generaliseer ik de manieren waarop recursieve functies op een inductief type gedefinieerd kunnen worden. Dan bezie ik in hoeverre deze constructieprincipes gedualiseerd kunnen worden tot co-inductieve typen, ofwel finale co-algebra's. Ten slotte construeer ik, uitgaande van hetzij elementaire verzamelingenleer of typetheorie, daadwerkelijk initiële algebra's en finale co-algebra's voor een willekeurige polynomiale functor, en bewijs daarmee de relatieve consistentie van alle beschreven constructieprincipes ten opzichte van ADAM's typetheorie ATT.

Het voorgaande wordt aangevuld met de behandeling van enkele aan inductieve typen verwante onderwerpen. Ten eerste zijn dat recursieve datatypen met partiële objecten, zoals die in programmeertalen voorkomen waarbij men rekening moet houden met mogelijk niet-terminerende programmadelen. Ik vat de benodigde domeintheorie samen, en construeer zulke domeinen in ADAM uitgaande van finale co-algebra's. Verder bespreek ik kort inductieve typen in impredicatieve talen, typen als verzameling van type-vrije objecten, en het principe van bar-recursie, en doe ik een suggestie voor de inductieve definitie van nieuwe type-universa binnen een typetheorie. Ten slotte geef ik enkele verdere overwegingen over wiskundige taal en bewijsnotatie, en vat de benaderingen van inductieve typen samen.

De appendices bevatten de basisprincipes van de verzamelingenleer en van ATT, de benodigde toevoeging van de iota-operator ofwel bewijs-eliminatie aan typetheorie, en een studie naar uniformiteits-eigenschappen (*natuurlijkheid*) van polymorfe objecten, die ik in enkele gevallen nodig heb.

Omslag-diagram

Aanschouw het wiskundig universum,
zich ontwikkelend van oorspronkelijke eenheid
tot categorische dualiteit.

De centrale straal bevat
het initiële en het finale type,
samen met de overige platte eindige typen.

Zij worden geflankeerd door de duale principes
van gegeneraliseerde som en product
en van initiële en finale dekpunt-constructie.

STELLINGEN

behorende bij het proefschrift

Inductive Types in Constructive Languages

1. Vers 18:62 van de Bhagavad Gita geeft een zeer goede verklaring van het woord *Islam* (overgave), wat verwant is met *Salam* (vrede):

   Zoek dan uw toevlucht in Hem en geef uzelf met geheel uw hart aan Hem over, dan zult ge door Zijn genade tot de Opperste vrede komen en het Eeuwig Tehuis bereiken. [Vertaling: Stichting school voor filosofie / Amsterdam]

2. Het wrede schijn-oordeel van koning Salomo [1 Koningen 3:25],

   "Snijdt het levende kind in tweeën en geeft de helft aan de ene en de helft aan de andere vrouw", [Vertaling: NBG 1951]

   geeft de sleutel tot een rechtvaardig printer-toewijzingsalgoritme: versnipper de beschikbare afdrukregels in schijn gelijkmatig over alle afdrukopdrachten, rekening houdend met hun tijdstip van indienen, maar wijs de printer dan toe aan de opdracht die het eerst voltooid zou zijn.

3. De afbeelding op de omslag van dit proefschrift symboliseert het wiskundig universum.

4. De uiteindelijke betekenis van de informatietechnologie ligt niet in de producten die zij levert maar in de denkwijze die zij ons leert.

5. Poëtische zowel als mathematische inspiratie vindt het best uitdrukking in een taal die weinig beperkingen oplegt.

6. Als men in een constructieve taal wederzijds inductieve typen opneemt, dient men te letten op het onderscheid tussen de verschillende vormen van toegestane algebra-specificatie [dit proefschrift, sectie 5.2]. Evenzo dient men bij het definiëren van een recursieprincipe over wederzijds inductieve typen te letten op het onderscheid tussen "standaard recursie" [regel (6.2)] en "liberale recursie" [regel (6.11)].

7. De jaarlijks toenemende watervloed vervult deze functies: hij leert ons saamhorigheid, offervaardigheid en ongehechtheid, en hij bereidt ons zachtjes voor op de mogelijkheid van ingrijpende veranderingen in onze wereldordening.

8. Muziek draagt de essentie van het leven over aan de ziel die luistert.

9. De wereld beweegt zich onweerstaanbaar naar de heelheid.

Peter J. de Bruin