Ordinals and Interactive Programs

Peter Hancock

Doctor of Philosophy
University of Edinburgh
2000

For Robin Gandy

Abstract

The work reported in this thesis arises from the old idea, going back to the origins of constructive logic, that a proof is fundamentally a kind of program.

If proofs can be considered as programs, then one might expect that proof theory should have much to contribute to the theory of programming. This has indeed turned out to be the case. Various technologies developed in proof theory are now widely used in computer science for formulating and investigating programming languages and logics connected with them.

Yet there is a vigorous and venerable part of proof theory which has so far had little impact in computer science, namely ordinal-theoretic proof theory. This focuses on proofs of wellfoundedness, usually expressed in the form of a schema of transfinite induction with respect to a representation of an initial segment of the countable ordinals. Proof theory of this kind is concerned with what it is that limits the capacity of a proof system to 'see into the transfinite'.

If proofs can be considered as programs, what kind of program is a proof of wellfoundedness? My hypothesis is that the limitations of a formal system for writing proofs of wellfoundedness reflect its limitations as a system in which to program strategies for defeating ones opponent in a certain kind of game. In recent computer science, games have proved invaluable as models for describing patterns of interaction between a system and its environment.

I cannot claim to have substantiated this hypothesis, but only to have taken a few steps in that direction. The work reported in the thesis lies in three areas.

First, I present a framework for dependently typed programming in the style advocated by Martin-Löf. The novelties here are connected with bringing the type-theoretic approach to programming that comes from the Curry-Howard correspondence closer to the calculational approach in the categorical tradition that comes from Lambek and Lawvere. A particular challenge is to find a smooth and practical way of encoding inductive and coinductive definitions.

Second, I have investigated a number of ways of modeling interactive systems and transition systems in a constructive context. The focus here is on models with a direct computational interpretation, that can actually be used in programming. The approach is inspired by a construction due to Petersson and Synek. It is shown how one may represent game-theoretic strategies of various kinds using these models.

Finally, I give a construction of provable ordinals within a Martin-Löf style type theory that has a type of natural numbers, and an external sequence of universes closed under generalised Cartesian products. The locus of the ideas for this construction lie more in conventional proof theory, and were the basis for a conjecture made by me almost thirty years ago in work that I then abandoned. What is new here is the concept of a 'lens'. This is a predicate transformer that has been implicit in the construction of proofs of wellfoundedness since Gentzen. I hope this concept may be of some use in an algebraic, systematic approach to setting lower bounds on the proof-theoretic strength of more extensive type theories.

Acknowledgements

Many thanks to my supervisor, Gordon Plotkin, for his time and trouble.

I acknowledge gratefully the support of an SERC Research Student-ship.

Many thanks to all at the Laboratory for the Foundations of Computer Science during my time there, for providing an extraordinarily stimulating and civilised environment. Particular thanks to my office-mates, Jim Laird and Conor McBride, for the pleasure of their company.

Many thanks too to all in the Programming Logic Group of in the Department of Computing Science at the University of Göteborg and Chalmers Institute of Technology for arranging for me to visit several times, for their charm and hospitality, and for many discussions on matters related to my work.

I owe a great deal to Anton Setzer and Thierry Coquand, for encouragement, inspiration, and thoughtful remarks. I am especially grateful to Anton for sharing some happy and occasionally hilarious hours coding enormous ordinals into unsuspecting computers.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Peter Hancock*)

Table of Contents

Chapter 1

Introduction

The work reported in this thesis arises from the old idea that a proof is fundamentally a kind of program. It is concerned with connections between the theory of programming languages, and proof theory, particularly ordinal analysis.

If proofs are programs, or can be considered as programs, then surely they constitute a very interesting kind of program, if only because programming of this kind has been practiced for over 2000 years, and has been intensively studied for more than 100. If programming is a dark street, then at any rate there are occasional street lights.

Among the most interesting kinds of proofs, from the perspective of comparing the strength of formal systems in which proofs might be written, are proofs of wellfoundedness. For general reasons, any formal system for writing proofs is strictly limited in its capacity to recognise the wellfoundedness of simple effective wellorderings between concrete data structures. These limits can be investigated and located in terms of ordinal arithmetic, as first demonstrated by Gentzen. This has been pursued extensively in the part of proof theory known as ordinal analysis [89], [85], hand in hand with developments in ordinal arithmetic necessary to analyse increasingly strong systems. If proofs are programs, what kind of programs are proofs of wellfoundedness? My hypothesis is that they are paradigms of terminating interactive programs. The limitations of a formal system for writing proofs of wellfoundedness reflect its limitations as a system in which to write a particular kind of terminating interactive program.

The pattern of alternating interaction between distinct components (master and slave, client and server, caller and called, environment and system) is very pervasive in practical programming and system design. There is considerable interest in knowing when and how such a sequence of interactions can be constrained to terminate[1]. For example, successful termination is a fundamental property of

[1]Of course, there is even more interest in knowing that termination will occur within a

transactions that interact with a number of resource managers. Thus the limitations of formal systems and the arithmetical description of those limitations have considerable interest if one wants to calculate and compare the competitive strength of systems for writing interactive programs which terminate.

Games serve as an apt metaphor for alternating interaction. In the games I have in mind, two players take turns, choosing successive moves from a set of legal moves that depends on the sequence played so far. The object is to 'beat' one's opponent by driving them inevitably into a position in which they have no legal move. A strategy which accomplishes this has essentially the same structure as a proof of wellfoundedness.

The idea of developing an arithmetic of competitive advantage for positions in terminating games has been extensively pursued by Conway and others [15], [41], [16]. Conway did not however consider the metamathematical question of how competitive strength is limited by the programming system in which the strategies are written, which seems to be a question more in the province of proof theory.

If proofs are programs, it does not follow that programs are proofs. If I understand him, that idea was advanced by Martin−Löf in [66]:

> *If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute (a difference Dijkstra has referred to as the difference between computer science and computing science), then it no longer seems possible to distinguish the discipline of programming from constructive mathematics.*

This seems to me to give insufficient weight to a key distinction between proofs and programs, namely that circularity, or running perpetually is a vice[2] in a proof, but a virtue in a program. A proof must be wellfounded in a sense which guarantees the termination of any investigation back into its construction, and rules out any kind of *petitio principii*.

In programming, we are sometimes very much concerned with systems which ideally would run 'forever', as for example when the program forms part of the control system for a machine, or a web site. In the terminology of games, the desirable outcomes are often those in which neither party loses. This raises the general question of extending the arithmetic point of view to games which need not terminate. Here the concerns are not so much with termination, wellfoundedness and initial algebras, as with non−termination, coinduction, and final coal-

reasonable time.

[2]As in: 'vicious' circle.

gebras. These have been more in the province of computer science, since the development by Park [83] and Milner [70] of the concept of bisimulation. There is a widespread appreciation of the value of coalgebraic notions when modeling computer systems mathematically, though little seems to be known about how our ability to program perpetual strategies is limited by the system we choose in which to program.

I am disappointed to say that I cannot claim to have substantiated my hypothesis, but only to have taken a few steps in that direction.

The first section of this introduction explains in more detail the background to the ideas above. The second gives an overview of the work reported in the body of the this thesis.

1.1 Background

This idea that a proof is fundamentally a kind of program has a long history. It is more or less explicit in the very origins of constructive logic and mathematics. An early expression is in Kolmogoroff's [55] explanations of the constructive interpretation of first order logic, according to which its statements express problems, and to assert a statement is to claim to know an effective solution to the problem expressed by the statement. If putting a solution into effect is thought of as running a program, then a proof of a statement is a program whose execution solves the problem expressed by the statement.

1.1.1 Some history

The constructive approach to logic arose against the background of the crisis in the foundations of mathematics which impelled the development of logic, type theory, and set theory at the end of the last century, and the beginning of this. The crisis was provoked by the need to improve upon the standards of rigour prevalent in 19$-$th century mathematics so as to deal with new notions of function, continuity, limit and infinity. The crisis grew in scope and urgency as the difficulty of obtaining a clear view of even the basis of elementary arithmetic became starkly apparent. Many of the questions raised by this crisis were philosophical in character, concerning for example the nature of the statements made in mathematics, what kind of meaning they possess, what it means to apply mathematics, and what sort of relation holds between a mathematical statement and a proof of it.

From a philosophical perspective, the appeal of the constructive interpretation of mathematical statements is that it connects proofs and propositions with human activity, specifically putting a program (of an idealised kind) into execution, or operation. There is therefore nothing mysterious about how mathematics can be applied, as this is so to speak its very nature. Neither is it mysterious what a proof has to do with what it proves, as the relation is the same between that between means and end. According to the more Platonistic view according to which mathematical statements are descriptions of a realm more or less imperfectly accessible to human beings, with no intrinsic application in human affairs, the properties of a proof in virtue of which it conjures up conviction in the proved statement are quite mysterious.

A resolution of the foundational crisis was eventually secured by axiomatisation of set theory, through a progression of *mathematical* moves by mathematicians such as Zermelo, Hausdorff, Fränkel, Skolem and von Neumann. Arguably this resolution was accomplished in spite of the philosophical preoccupations and initiatives of some of those who contributed to this resolution [54][3] With the formulation and general acceptance of Zermelo−Fränkel set theory, mathematics of the kind developed in the second half of the 19−th century had been set on as simple and rigorous a basis as could reasonably be desired, and could pass as it were from adolescent crisis into adulthood. This formalisation took place decades before the invention of the stored program digital computer, or the emergence of programming in the sense in which it is understood today. Such questions as to what extent mathematics could be rendered in a formal framework admitting a computational interpretation, and how much would be gained or lost by so doing were nevertheless raised, and addressed.

Hilbert regarded axiomatisation as an essential step for the resolution of foundational questions, but only the first. His central concern was the *consistency* of such an axiomatisation, proved mathematically, but on the basis of so called finitistic principles underlying combinatorial reasoning about concrete objects, such as natural numbers, and formulas and formal derivations [46]. Hilbert began to refer to his way of providing a foundation of mathematics as 'proof theory' (*Beweistheorie*), and so brought the term into currency, in connection with the mathematical investigation of (primarily) the consistency of formal systems adequate for the formalisation of significant parts of mathematics. He set about the creation of this new subject with his collaborators, principally Bernays and Ackermann. Hilbert was considerably exercised to refute the contrary views of

[3]It is perhaps the peculiar fate of philosophers that such an accomplishment as Frege's invention of predicate logic becomes 'what everybody always knew anyway' and merely the stage on which some more technical subject unfolds.

Brouwer, who considered formalisation irrelevant[4], and pursuit of a foundation for mathematics by metamathematics absurd. The following passage from Hilbert's polemical paper 'The Foundations of Mathematics' ([46]) illustrates this.

> *The formula game that Brouwer so deprecates has, besides its mathematical value, an important general philosophical significance. For this formula game is carried out according to certain definite rules, in which the technique of our thinking is expressed. These rules form a closed system that can be discovered and definitively stated. The fundamental idea of my proof theory is none other than to describe the activity of our understanding, to make a protocol of the rules according to which our thinking actually proceeds.*

Of course, Hilbert's hopes of a wholesale reduction of arithmetic, analysis and transfinite set theory to a finitistic basis were destroyed by Gödel's second incompleteness theorem. Moreover, Hilbert's conviction that mathematics could be expressed as a 'closed system' was revealed to be an illusion.

Gentzen's contributions to proof theory were profound, and are for the most part well known. The first of these was the formulation of systems of natural deduction for classical and intuitionistic first order logic, arguably taking more seriously than Hilbert the question of making 'a protocol of the rules according to which our thinking actually proceeds'. (Credit for this invention is due also to Jáskovski: see [87].) In a natural deduction system, the rules pertaining to each logical constant (*i.e.* connective or quantifier) are teased apart from each other (with respect to the axiom−based formulations of Frege and Hilbert). For each such constant, there are introduction rules, for introducing the constant as the outermost constant of the formula which is the conclusion of the rule, and elimination rules, for eliminating the operator from the 'main' premise of the rule. Gentzen did not consider reduction rules. The investigation of reduction rules in natural deduction systems was due to Prawitz [87]. This was a crucial step in establishing the correspondence between computation and the use of proofs as programs as something more than 'mere philosophy'.

Another major contribution of Gentzen's was the formulation (deriving from certain ideas of Paul Hertz) of an alternative presentation of the rules of classical and intuitionistic predicate logic, better suited to the investigation of systems of classical logic. This has become known as 'sequent calculus', or 'Gentzen systems'. A sequent is a pair (Γ, Δ) of sequences of formulas, commonly written $\Gamma \to \Delta$ or $\Gamma \vdash \Delta$, where the sequences Γ and Δ are the *antecedent* and *succedent* respectively. The conclusion and premises of a rule are not formulas, but sequents.

[4]Oddly, it was Brouwer who first (in his thesis) formulated the idea of metamathematics.

The rules are classified into logical rules pertaining to the various logical constants, and structural rules. The rules for each logical constant are divided into rules for introducing that constant into (a formula of) the antecedent and for introducing it to the succedent of the conclusion sequent. The structural rules (assumption, thinning, contraction, and interchange) are concerned with the bureaucracy of sequents themselves, while the remaining structural rule called 'cut' allows for the elimination from the conclusion of an entire formula which occurs in the antecedent of one of its premises, and from the succedent of the other. The content of Gentzen's *Hauptsatz* or cut−elimination theorem is that the rule of cut is an admissible rule of the system of rules without cut. From this, a host of corollaries flow, in virtue of the crucial subformula property of the (first order) system without cut, that in any derivation of a sequent $\Gamma \rightarrow \Delta$, the only formulas which occur are subformulas of Γ and Δ. (See for example Chapters 4 and 5 of [105].) The technology of natural deduction systems and sequent calculi has become pervasive throughout logic and computer science.

Yet another contribution of Gentzen's, of more direct relevance to the subject of this thesis, was to launch the whole subject of ordinal−theoretic proof theory, or 'ordinal analysis' [89], [85]. Unlike Gentzen's other contributions, to the best of my knowledge this subject has had little impact on computer science, and is perhaps considered to be at best a highly technical field within mathematical logic, very distant from the motivations and concerns from which it arose. Here the focus is on proofs of a particular kind of statement, namely statements of wellfoundedness. I shall usually reserve the term 'wellfoundedness' for the property of a binary relation that all points in the field of that relation are accessible. Accessibility of a point t is usually formulated schematically in the form of transfinite induction over the binary relation $\prec$, up to some point t in the field of $\prec$, with respect to a free schematic predicate P. One way to express this is as follows.

$$((\forall m)((\forall n)n \prec m \Rightarrow P(n)) \Rightarrow P(m)) \Rightarrow P(t)$$

By a proof of such a statement is meant a proof in the language extended with a new predicate constant P, using no special rules for P. Accessibility of a point is equivalent to wellfoundedness of the initial segment of the relation below that point in the transitive closure of the relation; so I may sometimes refer to the point itself as wellfounded. For general reasons, any sound formal system for writing mathematical proofs has a strictly limited capacity to recognise wellfoundedness, in the sense of permitting the construction of proofs of accessibility for naturally arising notations (such as Cantor normal form) for countable ordinals. Gentzen showed (in [35]) that the consistency of first order arithmetic followed by finitistic reasoning from the descending chain principle for the first ordinal closed under the operations of Cantor Normal form, namely Cantor's first 'epsilon' number ϵ_0. From this it followed (by Gödel's second incompleteness theorem) that the wellfoundedness of ϵ_0 cannot be proved in (classical or intuitionistic) first order number theory, so that this ordinal (whose importance had been noticed over half a century previously) represented a limit on the ability of such a system to 'see into the transfinite'. Not long afterwards (in [36]) Gentzen also established that this limit is best possible, in the sense that the wellfoundedness of notations in Cantor normal form can be systematically established with proofs expressible in first order arithmetic. (This is a modern treatment in chapter 10 of [105].) Since then, ordinal theoretic proof theory has been concerned with what it is about a system for writing mathematical proofs that determines its capacity for expressing proofs of wellfoundedness. Hand in hand with this has gone the invention of ordinal notation systems with which to describe ever more far−flung regions of the second number class in which the limits of more powerful systems are located. Dull would he be of soul perhaps, who did not find something marvelous (if difficult to pin down) about the connections between logic and arithmetic that have so far been traced. On the other hand, the question can be raised: what has this to do with programming, or computer science? The question is far from rhetorical, in view of the connection between proofs and programs in the appreciation of which Gentzen's formulation of natural deduction played a crucial rôle.

Gentzen's instigation of ordinal theoretic proof theory (in the setting of sequent calculi for first order arithmetic) was soon followed by another consistency proof for classical first order arithmetic given by Ackermann [2], by means of the descending chain principle for the same ordinal. Ackermann's consistency proof used a formulation of classical arithmetic exploiting Hilbert's so−called ε−symbol, or 'choice' operator $\varepsilon\ A = \varepsilon x : A(x)$, where $A(x)$ is a predicate expression with a distinguished argument indicated by the variable x that is bound by the higher−order operator ε. The term $\varepsilon\ A$ stands for an object of which the predicate holds if it holds for any object at all. This operator can take on the rôle played by the first order quantifiers, by means of the equivalences

$$\begin{array}{rcl} \forall x.A & \equiv & A(\varepsilon x : \neg A(x)) \\ \exists x.A & \equiv & A(\varepsilon x : A(x)) \end{array}$$

It turns out that in a given proof within classical first order arithmetic terms of ε−form can be systematically replaced (ultimately) by numerals in such a way as that what remains are true quantifier free statements (thus establishing consistency). Ordinals are used to establish that the process of replacement terminates. Ackermann's methods were squarely in the tradition of Hilbert's *Beweistheorie*. Ackermann's work was subsequently clarified by Tait ([103]) and developed by Kreisel ([56], [57]) into his *no−counterexample interpretation* of classical arithmetic. In this interpretation, roughly speaking, a proof of first order classical arithmetic is interpreted as a certain effective functional mapping finitely many first order numerical functions to numbers. (That is, the functional is of second order.) In a loose sense, this constitutes an interpretation of such a proof as a terminating program for responding to a stream of concrete data tokens with another such token, modulo some coding. This may also be thought of as a strategy (which can be implemented, though it may not be feasible to use the implementation) for winning a contest against someone who contends that the functional is undefined for a certain argument of which he is prepared to supply arbitrarily long initial segments. It may be worth noting that *all* proofs of classical arithmetic are interpreted in this way, not just proofs of wellfoundedness. Not only are the functionals continuous (considered as functions from Baire space to the discrete space of natural numbers), but they can be coded by wellfounded trees (the nodes being neighbourhoods throughout which the value of a given functional is not determined) whose depth is bounded by the ordinal ϵ_0.

A better−known functional interpretation of proofs in first order intuitionistic arithmetic is Gödel's 'Dialectica interpretation' ([39]), in which functionals of all finite types over the natural numbers are used, closed under primitive recursion at all finite types not just second order functionals. Gödel's paper was published in 1958, but the interpretation was apparently known to him as early as 1941 (as reported by Feferman [32]). He described a system 'T' of formal rules for deriving typed equations, which are stipulated to be decidable. The computation of these functionals cannot be visualised as readily as in the second order case, so the ordinal theoretic content of Gödel's interpretation is not as accessible as with Kreisel's. On the other hand Gödel's interpretation is primarily for intuitionistic first order arithmetic, not directly for the system with classical logic. His interpretation diverges from the BHK interpretation of the intuitionistic connectives, but is much closer to it than Kreisel's, in that the type operations have roughly the same structure and complexity as the logical operations.

To analyse the ordinal theoretic strength of systems stronger than arithmetic, in particular to bound it from above, a crucial step was taken by Schütte [96], and independently by Lorenzen [60]. (The idea was anticipated by Novikov [81].)

Schütte and Lorenzen simplified the problem of cut−elimination for formal proofs involving induction principles by effectively representing such proofs by infinite wellfounded trees. Tait in [103] showed how one might use infinite *terms* to give an ordinal−theoretic analysis of the terms of Gödel's T. This is an early example of exploiting the (so−called) Curry Howard correspondence to transfer an idea from a system of rules for inferring statements (with reduction steps for eliminating cuts, to a system of rules for constructing functionals (with reduction steps for computing their values). Schütte's invention arose in the context of his analysis of systems based on the ramified fragment of Principia Mathematica, incorporating a certain *autonomy* principle providing for transfinite ramifications; similar studies were undertaken by Feferman. An improvement of of Tait's method was given by Martin−Löf in [64]. Another way of exploiting semi−formal systems (with infinitary rules) to analyse computations of Gödel's terms had also been demonstrated by Sanchis [95]. Sanchis' approach was developed further by Diller [25] and Howard in [50]. Some modern developments are by Schwichtenberg in [97].

It may be fair to say that the idea at work (subliminally) in the development of functional interpretations, and in the analysis of the computations of the functionals that arise is the idea that programs are implicit in proofs, but have to be extracted from them by some more or less ingenious meta−mathematical process, non−standard reinterpretation, or unwinding. The rather bold step (back) to the proposal (of Kolmogoroff's) that one could actually think of proofs as themselves (without any disembowelling) a kind of program, was taken by Howard, but arose from a *milieu* (of people in close contact) involving also Tait, Prawitz, Martin−Löf, Scott, Kreisel, Goodman and Gödel. (Gödel and Kreisel were very sceptical.) Howard in [49] gives a presentation of the proofs in intuitionistic arithmetic in natural deduction form as a system of typed terms, with computation rules for the terms in correspondence with normalisation rules for formal proofs. The implications of the Curry Howard analogy between program calculi and proof systems began to be appreciated and exploited in proof theory, for example by transferring and extending Tait's normalisation results for the terms of Gödel's T to systems of greater expressive and proof theoretic strength. The substance of the Curry−Howard correspondence had also been discovered in de Bruijn's Automath project, independently of the proof theoretic tradition (but not of Heyting's explanation of the intuitionistic meanings of the logical constants).

About the same time, the philosopher Michael Dummett was developing a view of direct semantics for mathematical statements that focussed on the rights and obligations of participants in the human activity of making assertions. These rights and obligations were explained in terms of putting into practice 'methods for obtaining' canonical proofs, where the notion of canonical proof is that in terms of which the meanings of the logical connectives and quantifiers are explained. Dummett's ideas invited comparison between his notion of canonical proof and the the technical notion of irreducible proof arising from proof theory.

In these circumstances Martin−Löf began to develop a series of systems of intuitionistic type theory, with no formal boundary between propositions and types. These were thought of as playing a foundational rôle for constructive, or computationally relevant mathematics analogous to that played by the Zermelo−Fränkel axiomatisation of set theory. In other words, the fundamental idea was that one should be able to understand these systems directly, roughly as indicated by Dummett, rather than indirectly by a mathematical semantics in the form of a mathematically defined interpretation function (whose definition would presuppose a direct understanding of the language used to express the definition). Martin−Löf's main technical invention, with respect to the state of development of type theory in the 60's, was of the notion of a *universe*, or a type of types permitting the definition of new types, particularly types of transfinite depth, as if they were data. His first system contained a universe designed so that with its use one could define a category of all categories and functors, in which the category is itself one of its own objects. This would indeed have been a foundational *coup*; but as is well known, Girard was able to point out that this universe was inconsistent, and the rules for it permitted the formation of expressions whose computation does not terminate. After this, Martin−Löf turned to the consideration of predicative universes, to which one could consistently adjoin schemata of recursion stipulating wellfoundedness of the natural extension of the subformula relation to types. These universe principles allowed for a much more practical expression of the autonomy principle compared to the formulations used by Schütte and Feferman in their analyses of rather artificial formal systems for predicative reasoning. The question arose of establishing the proof theoretic strength of these principles.

Up to the 60's, with some exceptions, the ordinal arithmetic on which the notation systems used in proof theoretic analysis had been based on Cantor normal form, (from the mid 19−th century), and Veblen's hierarchy of continuous and increasing functions over the ordinals in which each function enumerates the common fixed points of those that precede it (from the beginning of the 20−th century). To push further, notations for certain uncountable ordinals were introduced, using the idea due to Bachmann [7] of using higher (regular) number classes to index sequences of functions on lower number classes. This combination of ideas seemed to run out of steam in making use of an inaccessible number class [53]. A new idea was called for. Besides this, it was recognised that there were subtle properties of the notation systems actually used in ordinal theoretic proof theory that were frustratingly difficult to pin down mathematically. It is possible to cook up a well−ordering of order−type ω such that the consistency of first order arithmetic can be proved by adjoining the scheme of proof by induction over a notation system representing that ordering [89] The definition of the notation system is primitive recursive[5], but depends on an arithmetisation of the syntax of first order arithmetic, and is in that sense unnatural. There seemed to be more to notation systems than their order type, and this was not adequately captured by limiting their definition to be primitive recursive, or elementary in a complexity theoretic sense[6] In these circumstances the possibility of exploiting a type structure for functionals over the ordinals in order to develop new 'natural' notation systems was recognised by Feferman [30]. (These ideas had been anticipated a decade previously by Neumer.) He raised the question of exploiting a transfinite type structure. The question was pursued by Aczel [3], for two hierarchies of functionals, one based on the idea of $\omega-$iteration, and the other on 'the critical process' of adjoining to a notation system a new operator which enumerates its fixed points. Aczel showed that the ordinals obtained were not large, and in particular if one required that the use of transfinite types was autonomous, then his hierarchy of functionals based on $\omega-$iteration constituted a notation system for Veblen's ordinal Γ_0, used by Schütte and Feferman in their analysis of formal systems for predicativity. The other hierarchy gave $\phi_{\Gamma_{\Omega+1}}(0)$ Bachmann notation.

As Martin−Löf's first predicative type theory allowed for the formation of transfinite types in autonomous fashion, and the definition of functions of arbitrarily high type by primitive recursion (for example, the definition of $\omega-$iteration functionals), it was natural to conjecture that (something like) Aczel's weaker system of functionals could be defined in Martin−Löf's system, and conversely that any ordinal definable in Martin−Löf's system is bounded by an ordinal expressed using Aczel's $\omega-$iteration functionals. This became known as Hancock's conjecture, on the basis of some efforts I made to formulate a hierarchy like Aczel's

[5]In fact, Kalmar elementary

[6]This was stressed by Kreisel in a number of places. He had a 'jingle', stolen from a cigarette advertising campaign of the time, that "with proof theoretic ordinals it's not how large you make them, but how you make them large".

within Martin−Löf's theory, exploiting his rules for an external sequence of cumulative universe types. I also expected that by lifting suitable relations hereditarily to transfinite types (as with Plotkin's 'logical relations'), one would be able to bound the definable ordinals by notations for ordinals below Γ_0. This conjecture was established by Aczel (in the fragment with one universe), and Feferman (for the sequence of universes) by relating Martin−Löf's system to certain systems known as $\widehat{\mathrm{ID}_{\mathrm{n}}}^{c}$. These are systems based on classical first order arithmetic, which iterate finitely the principle that a positive inductive definition has a fixed point (but not necessarily a *least* such fixed point). The highly indirect and ingenious nature of the proof theoretic reductions employed by Aczel and Feferman meant that it remained for practical purposes obscure how one could exploit the universe constructions available in Martin−Löf type theory to exhaust its capacity to recognise wellfoundedness. In the fourth chapter of this thesis, I have remedied this to some extent, by showing how to define ordinals up to Γ_0 directly in type theory; what is still missing is a simple argument establishing a bound for the definable ordinals.

Since the 70's, the proof theoretical analysis of Martin−Löf type theory has been pushed considerably further in various publications by Setzer [98], [100] [99], Rathjen [90] [88], Palmgren [82], and Griffor [91]. For the most part, this has consisted in the treatment of stronger forms of universe construction than those originally introduced by Martin−Löf. Roughly speaking, the methods employed use mutual interpretations of type theories in (classical) systems of Kripke−Platek set theory extended by axioms asserting the existence of large cardinals. The systems of Kripke−Platek set theory are analysed by other means, using sophisticated developments in the proof theory of infinitary systems due to Bücholtz and Pohlers (local predicativity), and in the principles underlying the construction of ordinal notation systems.

From the beginning, Martin−Löf stressed that his type theory could be considered to be an implementable programming language, with the property that the computation of the value of a well−typed expression must terminate. For example, if one was able to establish by a constructive proof carried out within such a type theory that such and such a family of differential equations have solutions, then one could use the proof to compute arbitrarily precise rational approximations to the value of a solution for a particular argument given arbitrarily precise rational approximations to that argument. The whole of [66] is devoted to an explanation of how type systems of the kind he had developed primarily with the aim of providing a foundational framework for constructive mathematics could be considered as type systems for programming languages. The view of type systems advanced in this paper is radically different from those prevailing in computer science. The prevailing view is perhaps that the purpose of a type system is to prevent programs ‘going wrong’ [12]. In contrast, Martin−Löf’s type systems were designed so that (not only should programs not ‘go wrong’, but also) any computation evoked by a well−typed program should terminate, or in other words to ensure that programs should ‘go right’.

1.1.2 The notion of program

If proofs are programs, than at least on a superficial level they do not much resemble the kind of programs that one encounters in the computer industry. Over 20 years ago, when I was preoccupied with philosophical questions about mathematics and logic, the idea that proofs were intrinsically programs struck me as an important insight, an impression which I have never succeeded in shaking off. For a long time, employed in the computer industry as a programmer, I have tried to clarify to myself what it actually amounts to, and to square it with and apply it to what I have learnt about programming. In retrospect it is more than faintly ridiculous that I should have found the comparison of a proof with a program enlightening, having had even less experience of actual programming than of mathematics.

In some cases the programs I was employed to write formed part of the control system for complex machinery. I was lucky to work sometimes with engineers from fields such as hydraulics, metallurgy, and electronics. I began to appreciate how little I knew about Fourier analysis, differential equations, discrete sampling, and many other fields of mathematics extensively used in engineering.

It was a particular pleasure in my career as a programmer, even a relief to collaborate with engineers from other fields than software, because there appeared to be a solid foundation for what these other engineers said and did, despite the fact that they hotly denied the existence or relevance to them of any such foundation. The effect of having this foundation was that they had available to them a variety of well−understood ways of describing and modeling the things they built, and that the models supported useful calculational techniques. My impression was always that if I did not understand something they said, and it mattered enough, then in principle it would be possible to tell what they were talking about, what they were saying about it, and why they were saying it. In contrast, a foundation of this kind seemed much less explicit in software engineering. My first impressions of computer science were of a Darwinian jungle of competing theories, compared to the elegant flower beds of engineering mathematics. The difficulties software engineers face in communicating with each other, or in representing even to themselves what they are doing are profound. Yet what they do is surely programming, the activity I had thought such a firm bed−rock for the very meaning of mathematical statements.

As a result of my experiences as a programmer, I am now less interested in answers to philosophical questions, but more in what basis there is for thinking about programs and programming, in a sense of 'program' intelligible to programmers. I hope to have made it obvious why I have inevitably asked myself how matters look from the perspective that proofs are a kind of program. There is a certain dilemma: on the one hand the idea that a proof is a program is in danger of collapsing into platitude, and on the other hand from a programming perspective it can seem a paradox whose absurdity is close to grotesque. How could one write a program to control a large piece of complicated and dangerous machinery, for example the aileron system of an aircraft, as a constructive proof? What might the theorem say? How should one understand the notion of 'program' if this is even to make sense?

A program is a guide to action. Two things go on when we follow a program. The first is to figure out from the program what to do next, and the second is actually to perform that action when we have found out what it is. These are different 'phases' in the execution of a program. If we think of a machine language program, the two phases are well known as the different parts of the 'fetch−execute' cycle of the instruction processor. In this case, the 'fetch' phase is almost trivial[7], in that it generally consists of little more than reading a memory cell into an internal register where it will be decoded into micro−instructions to be issued to different parts of the machine. If we think instead of a functional program, for example a program written in Haskell, the fetch phase is usually implemented using some form of graph reduction, and is highly non−trivial. It may not even terminate, in which case programmers describe the execution of the program as 'hung'. With functional programs it is more difficult to understand what the instructions are which are the result of the internal computations of a functional program. For several decades this has been a deep conceptual problem affecting the very notion of a functional program, and what it means to run or execute a program. Indeed, if we take seriously Wadler's witty comparison in [111] of the problem of 'how functional programs can affect the world' with Descartes's

[7]It is not at all trivial at the level of the micro−architecture of modern instruction processors, which may have several arithmetic units, register banks, and instruction streams.

problem of how incorporeal minds can interact with physical bodies, it is possible to view the problem as a modern manifestation of one of the deepest puzzles in Western philosophy. For that matter, Wadler might as well have compared the problem to the still more ancient problem of understanding how it is that mathematics can be applied to the physical world, or as a guide in human affairs.

If one wants a very hum−drum example of the application of mathematics as a guide to action in human affairs, as familiar to school−children as to professional engineers, one need look no further than the practice of paying for goods in shops. The mathematical objects we apply in such a situation are primarily the natural numbers. We make a certain computation – five packets of Smarties at £3.84 a packet makes so much – and we use the result of that computation (a natural number in canonical form) as a guide or template in handing over coins and bank notes in exchange for the goods. Similarly, the shopkeeper makes a computation and uses its result as a guide for handing back change, if any is due. In a certain sense, the natural numbers are being used as programs. The canonical form '$S(\ldots)$' is interpreted as an instruction to hand over another coin; the canonical form '0' is interpreted as successful completion of the execution of the program. The canonical forms are interpreted as instructions to perform some real, extra−mathematical action. Computation is necessary to figure out or 'fetch' the next instruction; the performance of that instruction (the whole goal of the computation) is something lying outside mathematics itself.

In a certain sense, computation is what a (computer running a) program is doing when we do not care what it is doing, beyond that it should hurry up and finish. (Of course, we care very much that it should get the right answer.) The result of a computation tells us the canonical form of a mathematical value; we then interpret this canonical form imperatively, as a command or stimulus with which to initiate an interaction. One can compare these two phases with the 'fetch' and 'execute' phases of an instruction processor or CPU cycle. The next instruction is *fetched* by computation, in the sense of calculation, evaluation, or reduction to canonical form. When the canonical form has been obtained, it is then *executed* or performed. Here the instruction is the canonical form, which is general instantiated with certain subcomponents. These subcomponents, together with data supplied externally through the execution of the instruction determine a residual program which is the starting point of the next machine cycle. (In the case that a subcomponent has the type of a function, the external data is used to form the argument of this subcomponent.) It is the sequence of actions and interactions that we directly care about, for example the commands sent to the actuators controlling the ailerons of an aircraft and the data read from accelerometers and other such sensors.

Within the last decade, through the work of Moggi [72], Wadler [109] [112] [111], which has resulted in the design of the input−output system of Haskell and other pure functional languages, a clearer view has emerged of what it means to run a functional program (in a sense going beyond mere computation to include interaction with the physical universe), and by extension, to run a mathematical proof. If we are to look for a significant application of proof theory to programming, then we will hardly find it by concentrating on the proofs of Π^0_1 statements (as for example in [97]). At best, these proofs can serve only as models of 'batch' programs, in which all interaction takes place before computation is initiated and after it has ceased.

There is still a distinction to be drawn between the notion of program in functional programming and that which underlies the paradigm of proofs as programs. The type systems that have been devised for functional programming, such as the Hindley−Milner polymorphic type system ([71]) provide for a programming language which is Turing−complete, in the sense that any partial recursive function can be defined by an expression of the language. From this it inevitably follows that there are properly typed expressions whose computation does not terminate. On the other hand, if we consider a type system for functional programming of the kind advocated by Martin−Löf [66], it follows from the normalisation theorem for such a type system that the computation of any properly typed expression must terminate. From this it inevitably follows that any programming language with such a type system is not Turing complete. There is a certain dilemma in this situation, which I have referred to at the end of the previous section.

It is not easy to get a clear view of the issues involved in this dilemma. On the one hand, a type system for which we can exhibit computable numerical functions that cannot be typed within that system certainly exhibits a kind of incompleteness; it has 'false negatives'. It will always be possible to devise stronger and stronger type systems that allow more and more acceptable programs to be typed. (The work of Setzer, Palmgren, Rathjen and Dybjer on the formulation of strong universe principles can be understood in this way.) On the other hand, a type system for which we can exhibit well−typed expressions whose computation hangs (without actually 'going wrong' in Milner's sense) certainly exhibits a kind of overcompleteness; it has 'false positives'. One thing is however reasonably clear: an ability to write within a certain programming language programs which hang has nothing to do with being able to write programs which run 'forever'.

On the contrary: a program which has hung is a program which has ceased to run. It is precisely because we are interested in programs which run forever[8], in the sense of continuing to interact with their environment, that we should insist on the termination of each computation in the infinite series involved in that interaction.

1.2 An overview of the contents of the thesis

My work is described in three chapters, which form the body of the thesis. They are sandwiched between this introductory chapter, and a concluding chapter which sums up what has been done, and what perhaps should have been done, and remains to do.

1.2.1 Framework

In the second chapter I define the setting in which I work, which is essentially the framework for dependently typed programming in the style advocated by Martin−Löf. The novelties here are connected with bringing the type−theoretic approach to programming that comes from the Curry−Howard correspondence closer to the calculational approach in the categorical tradition that comes from Lambek and Lawvere. A particular challenge here that has proved beyond me is to find a smooth and practical way of representing sets, predicates and families defined inductively.

1.2.2 Modeling interaction

In the third chapter I have investigated a number of ways of modeling interactive systems and transition systems in a constructive context. The focus here is on models with a direct computational interpretation, that can actually be used in programming. The approach is inspired by a construction due to Petersson and Synek. It is shown how one may represent game−theoretic strategies of various kinds using these models.

[8] Perhaps there are, exceptionally, some situations in which we simply want a program to hang. Most computers contain a STOP or HALT instruction in their instruction sets, the purpose of which is simply to put a processor into a 'coma' (consuming no bus cycles, *etc.*) until it is reset.

1.2.3 Lenses

Finally, I give a construction of provable ordinals within a Martin−Löf style type theory that has a type of natural numbers, and an external sequence of universes closed under generalised Cartesian products. The general question is how closure properties of a type system (*i.e.* the type constructors under which the collection is closed) bear on closure properties of the programs which are well typed according to that system.

The locus of the ideas for this construction lie more in conventional proof theory, and were the basis for a conjecture made by me almost thirty years ago in abandoned work on 'Hancock's conjecture' [65], [31]. What is new here is the concept of a 'lens' that plays a key rôle in organising the mass of detail in which I had formerly become lost. A lens is a predicate transformer that has been implicit in the construction of proofs of wellfoundedness since Gentzen. I hope this concept may be of some use in developing a systematic algebraic approach to setting lower bounds on the proof−theoretic strength of more extensive type theories. By means of such devices, we can to a certain extent obtain the effect of structural recursion principles on an initial algebra for a functor B by using instead an operation on algebras for B that exploits closure properties of the type structure.

The notion of lens is implicit in many proofs of wellfoundedness, since Gentzen. In a certain sense, the difficulty of arriving at it has been the difficulty of realising what is under one's nose. The formulation of this concept is in part due to Thierry Coquand and Anton Setzer, who asked me what was behind my conjecture. I am very grateful to them for their interest, patience, penetrating comments, and encouragement.

1.2.4 Appendix: combinatorial completeness of the arithmetic combinators

Many people who have thought about *why* the proof theoretical strength of Gödel's T[9] is no greater than the ordinal ϵ_0 have privately expressed some dissatisfaction with the gruesomely syntactical nature of the various proofs of it that are available[10]. Somehow it is 'written all over' the system, and this is not brought out by these proofs; there ought to be an extremely simple, direct proof

[9] Gödel's T is, roughly speaking, 'nothing but' the simply typed $\lambda-$calculus with a combinator for $\omega-$recursion.

[10] Of course, many others are (sensibly) perfectly satisfied with one or more of the proofs that have been given.

of it, though one which no doubt will need a very deep insight into the problem to arrive at it. Like many others, at one time I had the perhaps Quixotic ambition to discover this supposed proof. (The hope was that the proof technique would extend immediately from Gödel's T to allow one to 'read off' upper bounds for the provable ordinals of other type theories. As a matter of empirical fact 'reading off' ordinals is what experienced proof theorists actually seem able to do!)

The ordinal ϵ_0 is the least ordinal which contains 0, ω and is closed under the binary operations of addition, multiplication, and exponentiation. It therefore came as an unwelcome surprise to notice that these arithmetical operations are combinatorially complete, both for the untyped and the simply typed λ calculus. In essence, this had been noticed several times before (by Fitch and Stenlund, for example), although their definitions of the combinators are in my opinion not quite correct. I have included an appendix on this matter. It is not easy to resist the temptation to think that there are connections with certain 'arithmetical' lenses that are hinted at in the third and fourth chapters of this thesis.

1.3 Misgivings

The subject of ordinal theoretic proof theory is quite old. It was conceived in the grip of a philosophy of mathematics, namely Hilbert's finitism, which has the rare distinction that it is universally acknowledged to be mistaken. The conditions of stormy foundational crisis in which the subject arose subsided quite rapidly into what appears to be utter calm and complacency. The very first paper in the subject to be published was written to side−step (unfounded) misgivings the referees had about an earlier draft that established the same result by different methods, not employing ordinal arithmetic. From these inauspicious beginnings, the subject rapidly developed into a highly specialised and almost impenetrably technical field of mathematical logic. The most basic notions of the subject, such as the very notion of 'provable ordinal' have for over 40 years proved exasperatingly difficult to pin down mathematically, and resistant to conceptual analysis. At the risk of seeming impudent, one can ask what the subject is really about, and where precisely is its interest? My personal opinion is that ordinal theoretic proof theory is in need of rehabilitation. A question mark hangs over the subject. Its interest needs to be rediscovered and explained in a new way. My personal inclination is to suppose that there *is* such an explanation, but that it will not be easy to hit this nail squarely on its head.

The question is not one of finding 'applications' in other, equally recondite fields of mathematical logic, or for that matter mathematics or computer science. It is reasonable to suppose that any subject worth the expenditure of human intellect should ultimately bear fruit in terms of practical affairs. It is unreasonable to suppose that it should be easy to foresee this practical outcome, or predict what form it will take. A case in point may be the investigation of the distribution of prime numbers that has exercised number theorists including some of the most celebrated mathematicians in all history for over three centuries. It could hardly have been foreseen that these investigations should have prepared the ground for public−key encryption, a major technology underlying the security of internet commerce, and considered to be a munition by the government of the United States. It would surely be foolish to imagine that this practical outcome should all along have been 'the' justification for the effort pure mathematicians have spent on understanding the nature and distribution of the prime numbers. On the other hand, it would just as surely be indefensible to justify this expenditure of effort on the grounds that it was an agreeable entertainment for those involved.

It is the sheer intractability of the problem of factorising large numbers which has turned out to be the key to this exploitation of work in pure number theory. Perhaps there is more than a distant analogy here with the situation in ordinal theoretic proof theory.

The question I have asked myself is as follows. If proofs can be considered as programs, what kind of program is a proof of wellfoundedness? My hypothesis is that the limitations of a formal system for writing proofs of wellfoundedness reflect its limitations as a system in which to write a particular kind of interactive program, namely a strategy to 'defeat', or find a 'bug' in a system program that is intended to run forever. In recent computer science, the game−theoretic metaphor has proved invaluable as an inspiration for constructing models that help us to describe the patterns of interaction between a system and its environment. In game theoretic terms, the limitations of a formal system for writing proofs of wellfoundedness reflect its limitations as a system in which to program strategies for defeating ones opponent in a certain kind of game. I cannot claim to have substantiated this hypothesis, but only to have taken a few preparatory steps in that direction. My suspicion is that there are practical applications to be discovered of the notion of proof theoretic strength for programming languages. On the one hand there is the language in which one writes the program for a system that is intended to provide certain services (practically) forever, and on the other there is the language in which one writes a program that forms part of the environment of such a system, and runs in 'contest' with it to obtain the services that it provides.

Chapter 2

Framework, Types, Sets

Later chapters are concerned with transition structures and interactive structures on a set, ordinal notation systems, and proofs of accessibility of those notations in a particular type theory. These are ordinary mathematical notions of no great sophistication. However, I have aimed at constructions with a clear computational, and ultimately practical meaning. In pursuit of that, I have used on the meta level principles of more or less the same kind as are formalised in the type theory being considered. From a mathematical point of view this may be eccentric, compared with using set−theoretic abstractions. It seems quite natural from a programming point of view.

In this chapter I set out, as far as I have been able to, the principles I use on the meta level. The presentation builds on work by Martin−Löf on the syntax and semantics of his type theory, presented at various occasions in the last 10 years, and circulating in the form of lecture notes. This material includes among other things a 'variable free' presentation of the logical framework in which Martin−Löf style type systems are expressed. In essentials, I have adopted it here. Another source is Hofmann's survey article [48].

What is a logical framework? One can try to answer this general question in two ways

- by saying what a framework looks like, or its formal representation.
- by saying what a framework is for, whatever it looks like and however it is represented.

In this chapter I am concerned only with a framework for type systems belonging to a certain limited family, rather than logics in general. Some of the limitations are that I have made no attempt to deal with issues of linearity, or modal operators.

Formally, a logical framework looks like a finitely typed dependent type theory over a family of ground types, one of which (written here Set) is a type of indices A for the other ground types El A, and serves as the type of 'small' types of a type theory represented in the framework. As just indicated, the function of a logical framework is to provide a single setting in which a range of type theories may be represented, taking care once and for all of mechanisms pertaining to schemata. A type theory is represented within the logical framework by a collection of typed constants which impose closure conditions on the index type Set and the indexed types El A, together with equations between certain expressions constructed from those constants.

What is novel may be that I have described a version of the logical framework with co−products (which is a great convenience), and moreover one in which the judgemental or computational equality is rather extensional, yet still calculation−oriented. I do not claim that these ideas are of great originality, but only that they deserved some effort to write them down, and hopefully the effort of reading them.

The description can be divided 'horizontally' into three levels of generality.

- At the most general level is a categorical part, which concerns the most general features of any notion of context dependent type (or object), and construction of a value of such a type. The categorical framework lets us represent expressions of the type system (the next level down) as composed from certain primitive linguistic forms, and provides the general machinery of composition.

- Then there is a logical or type−theoretical part, which concerns basic type constructions corresponding to conjunctive, disjunctive, and schematic form. The logical part is essentially a dependent type theory closed under products and sums of families of types, under binary sums, over a family of 'small' ground types, or *sets*.

- Then there is a set−theoretical part. This is essentially a collection of closure properties on the ground family of sets, formulated as typing judgments and recursive equations, which I call a set theory. This places limits on the forms of recursive equation which we suppose to have solutions, and extreme solutions. (To avoid confusion, I call the normal kind of set theory expressed in predicate calculus 'ZF−style' set theory.)

2.1 Framework level

The logical framework is a system of judgments and inference rules appropriate to a type theory. There are four basic judgment forms:

$$\begin{array}{ll} A : \mathsf{Type}\ \Gamma & A = B : \mathsf{Type}\ \Gamma \\ a : \Gamma \to A & a = b : \Gamma \to A \end{array}$$

as well as three auxiliary judgment forms:

$$\Gamma : \mathsf{Context} \qquad \gamma : \Delta \to \Gamma \qquad \gamma = \gamma' : \Delta \to \Gamma$$

The auxiliary judgment forms may be taken as basic, and the basic ones as auxiliary. This may be preferred because the three auxiliary judgment forms correspond directly to the categorical notions of object, morphism, and equality between morphisms, or merely because it is formally neat.

One can think of the logical framework as the (generalised algebraic [13]) theory of a categorical structure in which the key component is a category of contexts. The intuition for a context is that it is a 'coordinate space' in which the points are ordered sequences or vectors of values. A morphism between contexts is a translation of points which defines each coordinate in the output vector in terms of the coordinates of the input vector.

The extra structure with which the category of contexts is equipped is a contravariant functor to a category which is 'universal' in a certain sense. There are various choices for this universal category. (There is a brief survey in section 3.2 of Hofmann's article [48]). The essential thing is that the objects of this universal category should be 'worlds' rich enough to model both the types available in a context, and the values inhabiting those types. The contexts should be closed under extension of a context Γ by declaring a new coordinate position, whose values are confined to a type A available in Γ. The new context is written (Γ, A); this operation is a form of dependent or cumulative conjunction. Moreover, there should be an empty context $()$; this is a form of empty conjunction.

One suggestion for a suitable universal category comes from Peter Dybjer ([27]), namely the category of families of sets.

- An 'object' or world in the category of families of sets is a small collection, or set−indexed family of sets, having the form (I, J) where $I : \mathit{Set}$ and $J : I \to \mathit{Set}$. An element i of the family's index set I models a type in that world, and an element of the set $J(i)$ models a value of type i in that world.

- A 'morphism' or translation from (I, J) to (I', J') is a pair (f, g) with $f : I \to I'$, and $g : (\prod i : I)J(i) \to J'(f(i))$. That is to say, f is a function between the index sets of the families, and g is a function which when applied to an index i from the first family, and an element of the set $J(i)$ so indexed, gives an element of the set $J(f(i))$ indexed by the image of i under f.

There are however severe technical problems in really making this structure into a category when using type theoretical abstractions at the meta−level (to the level of detail required for a machine−checked construction). The problem is that there is no completely satisfactory mathematical notion of equality between the morphisms, as is required for a category. Instead, as stressed by Dybjer, we have to consider not just sets, but *setoids*, or sets endowed with partial equivalence relations that are respected by morphisms between them. The objects of the universal category are then not merely discretely indexed families of such setoids, but ones such that equal indices determine isomorphic setoids in a coherent fashion.

Other choices for a universal category are possible which may be simpler (and less sensitive to the subtle issues peculiar to type theory surrounding the notion of equality in its various manifestations). In particular, instead of the category of families[1] $(\sum I : Set)I \to Set$, we may take (to a first approximation)

$$(\sum I : Pow(S))(\prod s : S)I(s) \to Pow(S),$$

where S is a set (with equality) of 'realisors' (such as a combinatory algebra) and $Pow(S)$ is the type of predicates $S \to Set$ over S. This is to replace Set − the types of 'sets in thin air' (and no obvious equality or morphisms) by $Pow(S)$ − the type of equality respecting predicates over S ('solid ground'), with inclusion morphisms and extensional equality. Now a world is represented by two things: first a predicate I of an arbitrary realisor (the predicate's interpretation being that the realisor represents a type), and second a function which maps a realisor s together with a proof that it satisfies I to a predicate of an arbitrary realisor (the predicate's interpretation being that the realisor represents a value of the type represented by s). Constructively, that function has really two arguments: the realisor, and the proof that it satisfies I, and the value of the function will depend on this proof. However the predicates arising from any pair of proofs should be extensionally equal. Coquand's model in [18] can be viewed in this way.

[1] Here and below, the scope of quantifiers should be taken as extending as far as possible to the right.

These suggestions for a 'universal' category determine *metamathematical* models, in which the judgments are interpreted as propositions about a mathematical structure. These may for example be expressed in the language of ZF−style set theory, serving as a 'neutral' *lingua franca*. Besides models of this kind, there are models in which the undefined constants and primitive rules of one system are translated into the expressions and derived rules of a metalanguage, which may have type theoretical judgement forms, rather than truth of a proposition, as with ZF set theory. The target for the translation may be a type theory that one actually uses and directly understands. Of course, neither kind of metamathematical model can serve to bring about such a direct understanding, but on the contrary presupposes such a directly understood metalanguage.

2.1.1 Syntax

The constants and operators of the framework are as follows. I will make heavy and reckless use of overloading, in the interests of readability[2].

- The binary operator $\_\circ\_$ is used for composition of morphisms as in $\gamma \circ \delta$, as well as for the translation of types and terms between contexts, as in $A \circ \gamma$ and $a \circ \gamma$.
- The constant 1 is used for the unit of composition, as well as the identity translation of types and terms.
- The binary operator $(\_,\_)$, pronounced 'snoc', is used both for extending a context Γ by a declaration, as in (Γ, A), and for extending a morphism by a new term as in (γ, a). I take it to be left associative in the sense that $(\gamma, a, b) = ((\gamma, a), b)$. I also write $(a_1, \ldots, a_k)$ instead of $((), a_1, \ldots, a_k)$.
- The 'nil' constant $()$ is used both for empty contexts and for empty morphisms.
- The constant p is the projection morphism from an extended context (Γ, A) to the interior context Γ. The constant q selects the outermost value a from a morphism (γ, a). (The rules for p and q are given on page 36.)

A more scrupulous notation probably requires heavy use of subscripts, as indicated in the following notation.

[2]Type theory seems to be peculiarly cursed in that it in practice very difficult to devise notations that are at the same time readable, and logically impeccable. A presentation of type theory in a framework based on a name−free substitution calculus encounters this problem in spades.

$$\begin{array}{ll}
(),\ (\Gamma, A) & \text{contexts} \\
()_\Gamma,\ (\gamma, a)_{(\Gamma,A)} & \text{substitutions} \\
\theta \circ_\Gamma \gamma,\ 1_\Gamma & \text{categorical structure} \\
A \circ_\Gamma \gamma,\ a \circ_\Gamma \gamma, & \text{functorial structure}
\end{array}$$

In practice, this obscures things.

The expressions of the framework are built up from expressions of the type theory (which occur in A and a position) using these ingredients. The expressions built up in this way are constituents of judgments. We have no way yet of making A's and a's.

As mentioned already, the judgments have the following six forms.

$$\begin{array}{ccc}
 & \Gamma : \mathsf{Context} & \\
 & \gamma : \Delta \to \Gamma & \\
A : \mathsf{Type}\,\Gamma & & A = B : \mathsf{Type}\,\Gamma \\
a : \Gamma \to A & & a = b : \Gamma \to A
\end{array}$$

To save space, where several judgements occur together which are the same in the part beyond the ':', I shall collapse them into one[3]. Thus $\gamma, \gamma' : \Delta \to \Gamma$ is short for the two judgements $\gamma : \Delta \to \Gamma$ and $\gamma' : \Delta \to \Gamma$; likewise, $a, b, c : \Gamma \to A$ is short for three judgements.

The framework itself gives only general structural features of any theory of context dependent types, and elements of those types.

2.1.1.1 Conventions

I have used an abbreviatory device in presenting rules for inferring equations, with the intention of bringing into focus their real function, rather than just saving space. This lets me write rules which say that in a certain context, if *these* things are equal, then so are *those*. Such a rule states a criterion for equality between an arbitrary morphism and a morphism of a specified form. To express this, use an auxiliary 'biconditional' judgment.

$$\begin{array}{l}
\dfrac{\text{context}}{\text{comparison}} \\
\Leftrightarrow \left\{ \begin{array}{l} \text{criteria} \\ \ldots \end{array} \right.
\end{array}$$

where the comparison and criteria consist of one or more equations, understood conjunctively. This figure abbreviates the left−to−right rule

$$\frac{\begin{array}{l}\text{context,} \\ \text{criteria}\end{array}}{\text{comparison}}$$

[3] A similar convention is used in many functional programming languages.

as well as the several right−to−left rules

$$\frac{\text{context, comparison}}{\text{criterion}}$$

2.1.2 Rules

The rules of the framework divide into those concerning the categorical structure, the functorial structure, the terminal (*i.e.* final) structure, and the comprehension structure.

2.1.2.1 Categorical structure

The following rules stipulate that the contexts form a category.

- That morphism equality is an equivalence relation.

$$\frac{\Gamma, \Delta : \mathsf{Context}, \quad \gamma : \Delta \to \Gamma}{\gamma = \gamma : \Delta \to \Gamma}$$

$$\frac{\Gamma, \Delta : \mathsf{Context}, \quad \gamma, \gamma' : \Delta \to \Gamma, \quad \gamma = \gamma' : \Delta \to \Gamma}{\gamma' = \gamma : \Delta \to \Gamma} \qquad \frac{\Gamma, \Delta : \mathsf{Context}, \quad \gamma, \gamma', \gamma'' : \Delta \to \Gamma, \quad \gamma = \gamma' : \Delta \to \Gamma, \quad \gamma' = \gamma'' : \Delta \to \Gamma}{\gamma = \gamma'' : \Delta \to \Gamma}$$

- Composition of morphisms, compatibility with equality.

$$\frac{\Gamma, \Delta, \Theta : \mathsf{Context}, \quad \gamma : \Delta \to \Gamma, \quad \delta : \Theta \to \Delta}{\gamma \circ \delta : \Theta \to \Gamma} \qquad \frac{\Gamma, \Delta, \Theta : \mathsf{Context}, \quad \gamma, \gamma' : \Delta \to \Gamma, \quad \delta, \delta' : \Theta \to \Delta, \quad \gamma = \gamma' : \Delta \to \Gamma, \quad \delta = \delta' : \Theta \to \Delta,}{\gamma \circ \delta = \gamma' \circ \delta' : \Theta \to \Gamma}$$

- Associativity of composition.

$$\frac{\Gamma, \Delta, \Theta, \Lambda : \mathsf{Context} \quad \gamma : \Delta \to \Gamma, \quad \delta : \Theta \to \Delta, \quad \theta : \Lambda \to \Theta}{\gamma \circ (\delta \circ \theta) = (\gamma \circ \delta) \circ \theta : \Lambda \to \Gamma}$$

- The identity morphism.

$$\frac{\Gamma : \mathsf{Context}}{1 : \Gamma \to \Gamma}$$

- That identity is a unit for composition.

$$\frac{\Gamma, \Delta : \mathsf{Context}, \quad \gamma : \Delta \to \Gamma}{\gamma \circ 1 = \gamma : \Delta \to \Gamma} \qquad \frac{\Gamma, \Delta : \mathsf{Context}, \quad \gamma : \Delta \to \Gamma}{1 \circ \gamma = \gamma : \Delta \to \Gamma}$$

2.1.2.2 Functorial structure

The following rules require that we have a contravariant functor from the category of contexts into the category of families of sets.

- That equality between the types in a context is an equivalence relation. (Omitted, as are certain other rules it would be pointless to write out.)

- That equal types in a context have the same values.

$$\frac{\Gamma : \mathsf{Context}, \quad A : \mathsf{Type}\,\Gamma, \quad A' = A : \mathsf{Type}\,\Gamma, \quad a : \Gamma \to A}{a : \Gamma \to A'}$$

- That equality on values is an equivalence relation. (Omitted.)

- That equal types put the same equivalence relation on values.

$$\frac{\Gamma : \mathsf{Context}, \quad A : \mathsf{Type}\,\Gamma, \quad A' = A : \mathsf{Type}\,\Gamma, \quad a = a' : \Gamma \to A}{a = a' : \Gamma \to A'}$$

- Translation of types by morphisms, and its compatibility with equality.

$$\frac{\Gamma, \Delta : \mathsf{Context}, \quad \gamma : \Delta \to \Gamma, \quad A : \mathsf{Type}\,\Gamma}{A \circ \gamma : \mathsf{Type}\,\Delta} \qquad \frac{\Gamma, \Delta : \mathsf{Context}, \quad \gamma = \gamma' : \Delta \to \Gamma, \quad A = A' : \mathsf{Type}\,\Gamma}{A \circ \gamma = A' \circ \gamma' : \mathsf{Type}\,\Delta}$$

- Translation of types by the identity morphism.

$$\frac{\Gamma : \mathsf{Context}, \quad A : \mathsf{Type}\,\Gamma}{A \circ 1 = A : \mathsf{Type}\,\Gamma}$$

- Translation of types by composite morphisms.

$$\frac{\Gamma, \Delta, \Theta : \mathsf{Context}, \quad A : \mathsf{Type}\,\Gamma, \quad \gamma : \Delta \to \Gamma, \quad \delta : \Theta \to \Delta}{A \circ (\gamma \circ \delta) = (A \circ \gamma) \circ \delta : \mathsf{Type}\,\Theta}$$

- Translation of values, compatibility with equality.

$$\frac{\Gamma, \Delta : \mathsf{Context}, \quad \gamma : \Delta \to \Gamma, \quad A : \mathsf{Type}\,\Gamma, \quad a : \Gamma \to A}{a \circ \gamma : \Delta \to A \circ \gamma} \qquad \frac{\Gamma, \Delta : \mathsf{Context}, \quad \gamma = \gamma' : \Delta \to \Gamma, \quad A : \mathsf{Type}\,\Gamma, \quad a = a' : \Gamma \to A}{a \circ \gamma = a' \circ \gamma' : \Delta \to A \circ \gamma}$$

- Translation of values by the identity morphism.

$$\frac{\Gamma : \mathsf{Context}, \quad A : \mathsf{Type}\,\Gamma, \quad a : \Gamma \to A}{a \circ 1 = a : \Gamma \to A}$$

- Translation of values by composite morphisms.

$$\frac{\Gamma, \Delta, \Theta : \mathsf{Context}, \quad A : \mathsf{Type}\,\Gamma, \quad a : \Gamma \to A, \quad \gamma : \Delta \to \Gamma, \quad \delta : \Theta \to \Delta}{a \circ (\gamma \circ \delta) = (a \circ \gamma) \circ \delta : \Theta \to (A \circ \gamma) \circ \delta}$$

2.1.2.3 Terminal structure

The following rules require that the category of contexts should have a terminal object, to which any other context can be vacuously mapped.

- The terminal context.

$$() : \mathsf{Context}$$

- The terminal morphism.

$$\frac{\Gamma : \mathsf{Context}}{() : \Gamma \to ()}$$

- Uniqueness of the terminal morphism.

$$\frac{\begin{array}{l}\Gamma : \mathsf{Context},\\ \theta : \Gamma \to ()\end{array}}{() = \theta : \Gamma \to ()}$$

2.1.2.4 Comprehension structure

The following rules require that the category of contexts should support an operation of extension of a context by a new declaration. In categorical terms, the extension is a form of product.

This operation is sometimes called comprehension. The word 'comprehension' is appropriate because of the analogy between on the one hand contexts and sets, and on the other between types over those contexts and predicates over those sets. Zermelo's separation or *Aussonderung* axiom, better and less pedantically known as the comprehension axiom requires that sets are closed under restriction by predicates. Analogously, here we require that contexts are closed under extension by a declaration.

The rules that follow share the following global context.

$$\begin{array}{l}\Gamma : \mathsf{Context},\\ A : \mathsf{Type}\,\Gamma\end{array}$$

This context is a tacit prefix of each rule.

The rules require that there be a context $\Delta = (\Gamma, A)$, a morphism $\gamma = \mathsf{p} : \Delta \to \Gamma$ and a value $a = \mathsf{q} : \Delta \to A \circ \gamma$, where (Γ, A) is terminal among such contexts.

- Extension of a context by a type in that context.

  $$(\Gamma, A) : \mathsf{Context}$$

  The context (Γ, A) is called an *extended context*, with *interior* context Γ, and *outermost type* A.

- Extension of a morphism to one into an extended context, by a value in the extension type.

  $$\frac{\begin{array}{l}\Delta : \mathsf{Context},\\ \gamma : \Delta \to \Gamma,\\ a : \Delta \to A \circ \gamma\end{array}}{(\gamma, a) : \Delta \to (\Gamma, A)}$$

The morphism (γ, a) is called an *extended morphism*, with *interior* morphism γ, and *outermost value* a.

- Compatibility of equality with morphism extension.

$$\frac{\begin{array}{l}\Delta : \mathsf{Context},\\ \gamma = \gamma' : \Delta \to \Gamma,\\ a = a' : \Delta \to A \circ \gamma\end{array}}{(\gamma, a) = (\gamma', a') : \Delta \to (\Gamma, A)}$$

- The projection morphism from an extended context (Γ, A) to its interior Γ.

$$\mathsf{p} : (\Gamma, A) \to \Gamma$$

- The outermost value of an extended context.

$$\mathsf{q} : (\Gamma, A) \to A \circ \mathsf{p}$$

- Criterion for equality with an extended morphism.

$$\frac{\begin{array}{l}\Delta : \mathsf{Context},\\ \gamma : \Delta \to \Gamma,\\ a : \Delta \to A \circ \gamma,\\ \theta : \Delta \to (\Gamma, A)\end{array}}{\begin{array}{ll} & (\gamma, a) = \theta \;\;:\;\; \Delta \to (\Gamma, A) \\ \Leftrightarrow & \begin{cases} \mathsf{p} \circ \theta = \gamma \;\;:\;\; \Delta \to \Gamma \\ \mathsf{q} \circ \theta = a \;\;:\;\; \Delta \to A \circ \gamma \end{cases}\end{array}}$$

Note that *a priori*, the values of $\mathsf{q} \circ \theta$ have type $(A \circ \mathsf{p}) \circ \theta$, which equals $A \circ (\mathsf{p} \circ \theta)$ (by the functorial structure). So the second equation in the block above type−checks, but only because of the first equation.

2.1.2.5 abbreviations

A 'name−free' presentation of the syntax helps to shed light on some quite intricate aspects of its functioning, and serves as a paradigm for implementation. However it is not really usable as a medium of communication between human beings. To that end, I introduce now a number of abbreviations and notational devices.

First we introduce notation for extending a morphism to a more specific context by leaving the new type undisturbed. (My use of $^+$ clashes with Hofmann's in [48].)

$$\gamma^+ \;\; = \;\; (\gamma \circ \mathsf{p}, \mathsf{q}) \qquad \text{– \textit{'burying' a morphism}}$$

We have:

$$\frac{\begin{array}{c}\Gamma : \mathsf{Context},\\ A : \mathsf{Type}\ \Gamma,\\ \Delta : \mathsf{Context},\\ \gamma : \Delta \to \Gamma\end{array}}{\gamma^+ : (\Delta, A \circ \gamma) \to (\Gamma, A)}$$

Next I allow morphism expressions which do not begin with a unit $()$, and so are not of the form $((), a, b, \ldots)$ or $(1, a, b, \ldots)$. The morphism $\overline{(a, b, \ldots)}$ is a shorter way of writing $(1, a, b, \ldots)$. Morphisms of this form are sections for the retractions p and their composites. (My use of over−lining is consistent with Hofmann's in [48].)

$$\begin{array}{lcll}\overline{a} & = & (1, a) & \text{– } \textit{sections of } \mathsf{p}\\ \overline{a, b} & = & (\overline{a}, b) & \\ a(b) & = & a \circ \overline{b} & \text{– } \textit{preponent section}\end{array}$$

We have:

$$\frac{\begin{array}{c}\Gamma : \mathsf{Context},\\ A : \mathsf{Type}\ \Gamma,\\ a : \Gamma \to A\end{array}}{\overline{a} : \Gamma \to (\Gamma, A)}$$

Finally, as a concession to human comprehensibility, I allow the use of names to indicate positions within an input context rather than what amounts to a form of de−Bruijn index. So I shall label positions in an input context with names, as in $(\Gamma, x : A)$, and use these names x in types and terms rather than terms of the form $q \circ p^n$. I have tried to ensure that in every case, one can 'compile away' names into the p, q notation.

Note that as part of this we have already one form of application $f(a, \ldots, b)$ in which f is acted upon by by the morphism $(1, a, \ldots, b)$ – this is before we have introduced any form of function type. The notion of function which is involved here is sometimes called the 'old−fashioned' notion of function. (One can think of $f(a)$ as postfix application for the action of the morphism (a) upon f.)

The effect of the 'sugar' introduced above is that one can begin to use standard notations in the presentation of contexts, morphisms, terms and type expressions.

2.2 Type theory level

In the last section I set out a categorical framework, consisting of rules for a category of contexts closed under a certain form of limit. In this section I set out a type theory in that framework. The types are (arguably) 'logical' types[4],

[4] It might be better to say 'grammatical', and reserve 'logical' for the reflections in Set of the grammatical constructions.

which are used to classify syntactical structures of a given object language. These structures are schemata, tuples, alternatives, and other syntactic stuff.

The type theory is in essence the type theory or logical framework in which Martin−Löf style constructive set theory is (if I understand it) 'officially' presented; the novelties here are as follows.

- I have included besides $\Pi-$types also forms of coproduct type, with equational rules that require the relevant universal arrows to be unique. The coproducts are useful for dealing with 'blocks' of context, and classifications of different kinds.

- I have tried to disentangle the $\Pi-$type from the other type forms, so that one can consider a type theory having only coproducts. This involved teasing apart the notions of composition, application, and variable binding; these are often run together.

- The type theory is extensional, in that I require (as in category theory) a *unique* universal morphism for each construction. So not only are there '$\eta-$rules', but also (in proof theoretical terminology), rules of permutative conversion.

The only justification I have of the extensionality is rhetorical. Type theory should be a labour saving device. The easier it is to remember the rules, and the more equations available for calculational reasoning, the better.

Engineering seems to demand the development of 'calculi' with a prosthetic function, in which symbolic calculation on the basis of a stock of memorable algebraic laws helps to extend one's intellectual reach. Sometimes these are used with a certain optimism about how far the calculus is applicable, or how tricky its foundation or implementation turns out to be.

2.2.1 Syntax

The syntax of the type theory is follows.

Its expressions are built up using certain constants and operators on morphisms, using composition and extension to build up morphisms from $()$ and 1 and previously built expressions. I shall call these *logical* constants, connectives, and quantifiers.

In theory, this is all one needs. For practical use by human beings it is often humane and sometimes essential to employ names for free and bound variable, with the usual conventions of 'punctual' notation ([24]). So we suppose that expressions are built up also from mnemonic names (identifiers) which refer to defined or merely declared values. There should an unlimited supply of new names. These names are used to show where in the body of a quantifier (if anywhere) the quantified argument occurs.

The forms of expression are listed below.

- Function types. $\prod AB$, λb, $f(\_)$. (The latter is a postfix operator.)

  I abbreviate $f(\_) \circ \overline{a}$ by $f(a)$, and $f(a)(b)$ by $f(a, b)$ in the usual way.

  I use (right associative) arrow notation $A \to B$ to abbreviate the function type $\prod A(B \circ \mathsf{p})$. (The arrow is also a punctuation mark in judgments.)

  In concrete terms, B and b may be taken to be open terms in which each occurrence of a variable has 'de Bruijn' form $\mathsf{q} \circ \mathsf{p} \circ \ldots \circ \mathsf{p}$. Operationally, p pops a stack of arguments, and q takes the top.

  By $B[x]$ I mean the expression B rewritten so that references to the topmost value in B's input context (*i.e.* qs) have been replaced by references to the value x, and all other context references have been been shorn of one $\_ \circ \mathsf{p}$.

  - $\prod AB$ is the type of abstractions f such that $f(x)$ has type $B[x]$ when $x : A$.
  - λb is the abstraction whose value for argument $x : A$ is $b[x]$.
  - $f(a)$ is the instance of the abstraction f for argument a.

- Empty type. $\{\,\}$, $[\,]$.

  - $\{\,\}$ is the empty type.
  - $[\,]$ is the empty map out of it.

- Singleton type. $\{*\}$, $*$, $[c]$.

  - $\{*\}$ is the standard singleton
  - $*$ is the constant map into it.
  - the $[c]$ construction allows a function to be defined on the whole of $\{*\}$ by specifying its value merely at $*$.

- Disjunction types. $A + B$, i, j, $[a, b]$.

  - The disjoint union of A and B.
  - i and j are the two injections.

  - the $[a, b]$ construction allows a function to be defined on a disjunction type by specifying its value separately in each disjunct. (Definition by cases.)

- Ordered pair types. $\sum AB$, pair, Split[c].
  - $\sum AB$ is the type of ordered pairs (x, y) such that $x : A$ and $y : B[x]$.
  - pair is the form of such ordered pairs.
  - the Split[c] construction gives a way of splitting an argument which is a pair into its components.

- Ground types. $\mathrm{Set}, \mathrm{El}$.

2.2.2 Rules

The rules of the type theory are as follows, organised by type form.

2.2.2.1 Empty type

The following rules are all in the global context

$$\Gamma : \mathsf{Context}$$

- Formation.

$$\{\,\} : \mathsf{Type}\,\Gamma \qquad\qquad \frac{\Delta : \mathsf{Context}, \quad \gamma : \Delta \to \Gamma}{\{\,\} \circ \gamma = \{\,\} : \mathsf{Type}\,\Delta}$$

- The empty function, or *ex falso quodlibet.*

$$\frac{C : \mathsf{Type}(\Gamma, \{\,\})}{[\,] : (\Gamma, \{\,\}) \to C}$$

- Equality. This rule allows us to conclude that any function defined on the empty type is equal to the empty function.

$$\frac{C : \mathsf{Type}(\Gamma, \{\,\}), \quad \theta : (\Gamma, \{\,\}) \to C}{[\,] = \theta : (\Gamma, \{\,\}) \to C}$$

2.2.2.2 Singleton type

The following rules are all in the global context

$$\Gamma : \mathsf{Context}$$

- Formation.

$$\{*\} : \mathsf{Type}\,\Gamma \qquad\qquad \frac{\Delta : \mathsf{Context},\quad \gamma : \Delta \to \Gamma}{\{*\} \circ \gamma = \{*\} : \mathsf{Type}\,\Gamma}$$

- Singleton element.

$$* : \Gamma \to \{*\} \qquad\qquad \frac{\Delta : \mathsf{Context},\quad \gamma : \Delta \to \Gamma}{* \circ \gamma = * : \Delta \to \{*\}}$$

- Singleton function, given by the value. This rule allows one to assume without loss of generality that an arbitrary object of a singleton type is in fact its solitary denizen.

$$\frac{C : \mathsf{Type}(\Gamma, \{*\}),\quad c : \Gamma \to C \circ \overline{*}}{[c] : (\Gamma, \{*\}) \to C}$$

- Criterion for equality with a morphism of form $[c]$.

$$\frac{C : \mathsf{Type}(\Gamma, \{*\}),\quad c : \Gamma \to C \circ \overline{*},\quad \theta : (\Gamma, \{*\}) \to C}{\begin{array}{rrcl} & [c] = \theta & : & (\Gamma, \{*\}) \to C \\ \Leftrightarrow & \theta \circ \overline{*} = c & : & \Gamma \to C \circ \overline{*} \end{array}}$$

2.2.2.3 Disjoint union types

The following rules are all in the global context

$$\Gamma : \mathsf{Context},\quad A, B : \mathsf{Type}\,\Gamma$$

- Formation.

$$A + B : \mathsf{Type}\,\Gamma$$

$$\frac{\Delta : \mathsf{Context},\quad \gamma : \Delta \to \Gamma}{(A + B) \circ \gamma = (A \circ \gamma) + (B \circ \gamma) : \mathsf{Type}\,\Delta}$$

- Injection.

$$\mathrm{i} : (\Gamma, A) \to (A+B) \circ \mathsf{p}$$

$$\mathrm{j} : (\Gamma, B) \to (A+B) \circ \mathsf{p}$$

- Binary case split, or case form. This rule allows one to define a function with a disjoint union argument by defining instead two functions separately, one for arguments of i form, and the other for j.

$$\frac{\begin{array}{l} C : \mathsf{Type}(\Gamma, A+B), \\ a : (\Gamma, A) \to C \circ (\mathsf{p}, \mathrm{i}), \\ b : (\Gamma, B) \to C \circ (\mathsf{p}, \mathrm{j}) \end{array}}{[a, b] : (\Gamma, A+B) \to C}$$

- Criterion for equality with morphisms of case form. This rule allows us to express any morphism whose outermost argument has a disjoint union type in case form.

$$\frac{\begin{array}{l} C : \mathsf{Type}(\Gamma, A+B), \\ a : (\Gamma, A) \to C \circ (\mathsf{p}, \mathrm{i}), \\ b : (\Gamma, B) \to C \circ (\mathsf{p}, \mathrm{j}) \\ \theta : (\Gamma, A+B) \to C \end{array}}{\begin{array}{ll} & [a, b] = \theta : (\Gamma, A+B) \to C \\ \Leftrightarrow & \left\{ \begin{array}{lll} \theta \circ (\mathsf{p}, \mathrm{i}) = a & : & (\Gamma, A) \to C \circ (\mathsf{p}, \mathrm{i}) \\ \theta \circ (\mathsf{p}, \mathrm{j}) = b & : & (\Gamma, B) \to C \circ (\mathsf{p}, \mathrm{j}) \end{array} \right. \end{array}}$$

2.2.2.4 Pair types

Contexts are closed under extension by fresh declarations. With pair types, the types in a given context are closed under a similar kind of comprehension.

The following rules are all in the global context

$$\begin{array}{l} \Gamma : \mathsf{Context}, \\ A : \mathsf{Type}\,\Gamma, \\ B : \mathsf{Type}(\Gamma, A) \end{array}$$

- Type formation.

$$\textstyle\sum AB : \mathsf{Type}\,\Gamma$$

$$\frac{\begin{array}{l} \Delta : \mathsf{Context}, \\ \gamma : \Delta \to \Gamma \end{array}}{(\sum AB) \circ \gamma = \sum (A \circ \gamma)(B \circ \gamma^+) : \mathsf{Type}\,\Delta}$$

- Pairing.

$$\mathrm{pair} : (\Gamma, A, B) \to (\textstyle\sum AB) \circ \mathsf{p}^2$$

- Splitting an argument of pair type. This rule allows one to define a function with a pair type argument by defining instead a a function with two arguments corresponding to the components of the pair.

$$\frac{\begin{array}{l} C : \mathsf{Type}(\Gamma, \sum AB), \\ c : (\Gamma, A, B) \to C \circ (\mathsf{p}^2, \text{pair}) \end{array}}{\text{Split}[c] : (\Gamma, \sum AB) \to C}$$

- Criterion for equality with morphisms of Split[_] form. This rule allows us to bring any morphism whose outermost argument has pair type into Split[_] form.

$$\frac{\begin{array}{l} C : \mathsf{Type}(\Gamma, \sum AB), \\ c : (\Gamma, A, B) \to C \circ (\mathsf{p}^2, \text{pair}), \\ \theta : (\Gamma, \sum AB) \to C \end{array}}{\begin{array}{rlcl} & \text{Split}[c] = \theta & : & (\Gamma, \sum AB) \to C \\ \Leftrightarrow & \theta \circ (\mathsf{p}^2, \text{pair}) = c & : & (\Gamma, A, B) \to C \circ (\mathsf{p}^2, \text{pair}) \end{array}}$$

2.2.2.5 Alternative : disjunctive normal form

The types $\{\}$, $\{*\}$, connectives $+$, and quantifiers Σ introduced so far can be obtained as instances of a certain schema, that I shall call disjunctive normal form, which is indispensable in practice. This schema allows us to introduce in a given context Γ a new type which is the disjoint union of a finite family $\{ \Delta_i \mid i = 1, \ldots, n \}$ of *context blocks*, or *extensions* of Γ. Such an extension Δ_i may be empty, or start with a type $A : \mathsf{Type}\ \Gamma$ in the context Γ, and continue with an extension of the context (Γ, A).

It is almost essential in practice to allow the branches of a disjunction to be labelled using 'meaningful names' with a mnemonic rôle. (For example, for one might use *nil* and *cons* as labels). Each branch of a disjunction is in general a dependent (*i.e.* order sensitive) conjunction, and where conjunctions are non−empty (as with *cons*), one sometimes wants to refer to the components by names (as with _.*head* and _.*tail*).

The use of such a mechanism raises some difficult issues of specification (for example, to do with equality), over which I shall skate. What matters is that in the end we can 'compile' an arbitrary instance of the schema using only the particular instances $\{\}$, $\{*\}$, $+$, and Σ. Of course there may be more than one way of compiling a multi−way disjunction 'into binary', and other headaches of implementation.

The notation I shall use is as follows:

- Disjunction, or data type.

$$\begin{aligned} \textbf{data}\ \ c_1 &: \Delta_1 \\ &\dots \\ c_n &: \Delta_n \end{aligned}$$

  Here $c_1, \dots, c_n$ stand for branch labels, which should all be distinct. The labels will be called *constructors*.

- Case expressions, for analysis of data types. We can ask a 'datum' (value of a data type) about its constructor form, and its parts.

$$\begin{array}{l} \textbf{case}\ d\ \textbf{of} \\ \quad c_1 \mapsto e_1 \\ \quad \dots \\ \quad c_n \mapsto e_n \end{array}$$

  The case expression compiles into various kinds of split.

To formulate the rules for disjunctive normal form, one would introduce an auxiliary judgment, written as follows.

$$\Delta : \mathsf{Context}\,\Gamma$$

The main rules peculiar to this judgement form are as follows.

$$() : \mathsf{Context}\,\Gamma$$

$$\frac{\begin{array}{l}\Gamma : \mathsf{Context} \\ A : \mathsf{Type}\,\Gamma \\ \Delta : \mathsf{Context}\,(\Gamma, A)\end{array}}{(A, \Delta) : \mathsf{Context}\,\Gamma}$$

Note that the notation $(\dots, \dots)$ is being used here in a new way; the first argument place is for a type, and the second for a context.

One would then introduce the semicolon ';' as a concatenation operator, so that

$$\frac{\begin{array}{l}\Gamma : \mathsf{Context} \\ \Delta : \mathsf{Context}\,\Gamma\end{array}}{\Gamma; \Delta : \mathsf{Context}}$$

However to do this properly one must introduce equality between contexts, and a plethora of associated rules; so I content myself with this indication.

If D stands for the typical data type given above, then the formation rule is as follows.

$$\frac{\begin{array}{l}\Gamma; \Delta_1 : \mathsf{Context} \\ \dots \\ \Gamma; \Delta_n : \mathsf{Context}\end{array}}{D : \mathsf{Type}\,\Gamma}$$

The typing rules for data constructors are as follows.

$$\begin{array}{l}c_1 : (\Gamma; \Delta_1) \to D \circ p^{d_1} \\ \dots \\ c_n : (\Gamma; \Delta_n) \to D \circ p^{d_n}\end{array}$$

If additionally C stands for the typical case expression given above, then the typing rule for case expressions is as follows.

$$\frac{\begin{array}{l}e_1 : (\Gamma; \Delta_1) \to E \circ (p^{d_1}, c_1) \\ \dots \\ e_i : (\Gamma; \Delta_n) \to E \circ (p^{d_n}, c_n)\end{array}}{C : (\Gamma, D) \to E}$$

2.2.2.6 Function types

The function type rules are all in the global context

$$\begin{array}{l}\Gamma : \mathsf{Context}, \\ A : \mathsf{Type}\,\Gamma, \\ B : \mathsf{Type}(\Gamma, A)\end{array}$$

- Formation of function types.

$$\prod AB : \mathsf{Type}\,\Gamma$$

$$\frac{\begin{array}{c}\Delta : \mathsf{Context}, \\ \gamma : \Delta \to \Gamma\end{array}}{(\prod AB) \circ \gamma = \prod (A \circ \gamma)(B \circ \gamma^+) : \mathsf{Type}\,\Delta}$$

- Formation of functions.

$$\frac{b : (\Gamma, A) \to B}{\lambda\, b : \Gamma \to \prod AB}$$

- Application.

$$\frac{f : \Gamma \to \prod AB}{f(\_) : (\Gamma, A) \to B}$$

- Criterion for equality with a morphism whose leftmost component is of function form.

$$\frac{\begin{array}{l}\Delta : \mathsf{Context},\\ \gamma : \Delta \to \Gamma,\\ b : (\Gamma, A) \to B,\\ \theta : \Delta \to (\prod AB) \circ \gamma\end{array}}{\begin{array}{rlcl} & (\lambda\, b) \circ \gamma = \theta & : & \Delta \to (\prod AB) \circ \gamma \\ \Leftrightarrow & \theta(\_) = b \circ \gamma^{+} & : & (\Delta, A \circ \gamma) \to B \circ \gamma^{+}\end{array}}$$

2.2.2.7 Ground types

The following rules share the following global context.

$$\Gamma : \mathsf{Context}$$

They are rules of type formation, and equality between types.

$$\mathrm{Set} : \mathsf{Type}\,\Gamma \qquad\qquad \frac{\begin{array}{l}\Delta : \mathsf{Context},\\ \gamma : \Delta \to \Gamma\end{array}}{\mathrm{Set} \circ \gamma = \mathrm{Set} : \mathsf{Type}\,\Delta}$$

$$\mathrm{El} : \mathsf{Type}(\Gamma, \mathrm{Set}) \qquad\qquad \frac{\begin{array}{l}\Delta : \mathsf{Context},\\ \gamma : \Delta \to \Gamma\end{array}}{\mathrm{El} \circ \gamma^{+} = \mathrm{El} : \mathsf{Type}(\Delta, \mathrm{Set})}$$

I call use the words ‘small’ and ‘large’ in connection with ground types so that Set is the only *large* ground types, and all other ground types (namely those having a form $\mathrm{El} \circ (\gamma, a)$, $\{\,\}$, or $\{*\}$) are *small*. Informally, I call a type of some form large if any of its components is large, and a function large if its type of values is large.

The following rule rule is worth noting.

$$\frac{a : \Gamma \to \mathrm{Set}}{\mathrm{El} \circ \overline{a} : \mathsf{Type}\,\Gamma}$$

In general, I shall allow myself to write $\mathrm{El}\ a \circ \gamma$ instead of $\mathrm{El} \circ (\gamma, a)$, and $\mathrm{El}\ a$ instead of $\mathrm{El} \circ \overline{a}$.

2.3 Set theory level

So far, with an empty family of sets, the type system is empty. There is only a general framework in which to lay out particular *set−theories*.

The description takes the form of closure conditions on $(\mathrm{Set}, \mathrm{El})$, expressed by typing judgments and equations. The closure conditions are of three broad kinds.

- Closure under reflections of the logical operations $(\{\,\}, \{*\}, +, \sum, \prod)$ of the framework type system.

- Closure under principles of inductive definition of prescribed form, or more generally principles guaranteeing the existence of extreme solutions to equations of various kinds.

- Closure under reflections of $(\mathrm{Set}, \mathrm{El})$, with its closure properties as universe sets.

2.3.1 Closure under reflections of logical operations

Here we say what it means that $(\mathrm{Set}, \mathrm{El})$ is closed under reflections of the logical operations. For each operation, there is a constant with which sets are formed, and an equation between the type of elements of such a set and a type formed with the operation. The equations for El are recursive.

I have taken an approach which is discussed by Hofmann ([48], sec. 2.1.6). Where an operation on types (such as the function type operation) is to be reflected by a corresponding form of sets (such as sets of functions), I have required that the type of elements of a set of the form in question is actually definitionally equal to the reflected type, rather than just behaving like it. Hofmann points out that this has consequences for the proof of metatheoretical results such as subject reduction. Moreover, he draws attention to the fact that some standard models of impredicative type theories (Such as Coq) do not validate these equations, which hold only up to an isomorphism. Interestingly, the situation seems to be different when the type of sets is predicative, as is the case below. This is a relief, because a few definitional equations can then replace a plethora of rules.

For the moment, I shall use the following as symbols for the logical constants.

$$c_{\{\,\}},\, c_{\{*\}},\, c_{+},\, c_{\Sigma},\, c_{\Pi}$$

Instead of $\_ : () \to A$, I shall write $\_ \in A$, where the blank $\_$ may be filled by an equation or by a term. If $a \in A$, I shall say that a is an element of A. (The terminology is in accordance with the category−theoretic idea that an element of an object A is a morphism into A from the terminal context ().) Instead of $\mathsf{Type}\ ()$, I shall write simply Type.

- The empty set

$$
\begin{aligned}
&c_{\{\}} \in \mathrm{Set} \\
&\mathrm{El}\ c_{\{\}} = \{\,\} : \mathsf{Type} \\
&d_{\{\}} \in (\,\textstyle\prod C : \{\,\} \to \mathrm{Set}, z : \{\,\}\,)\ \mathrm{El}\ C(z)
\end{aligned}
$$

- The singleton

$$
\begin{aligned}
&c_{\{*\}} \in \mathrm{Set} \\
&\mathrm{El}\ c_{\{*\}} = \{*\} : \mathsf{Type} \\
&d_{\{*\}} \in (\,\textstyle\prod C : \{*\} \to \mathrm{Set}, c : \mathrm{El}\ C(*), z : \{*\}\,)\ \mathrm{El}\ C(z) \\
&d_{\{*\}}(C, c, *) = c : (C : \{*\} \to \mathrm{Set}, c : \mathrm{El}\ C(*)) \to \mathrm{El}\ C(*)
\end{aligned}
$$

- Binary disjoint union

$$
\begin{aligned}
&c_{+} \in \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set} \\
&\mathrm{El}\ c_{+}(A, B) = \mathrm{El}\ A + \mathrm{El}\ B : \mathsf{Type}(A : \mathrm{Set}, B : \mathrm{Set}) \\
&d_{+} \in (\textstyle\prod A : \mathrm{Set}, B : \mathrm{Set}, \\
&\qquad\quad C : (\mathrm{El}\ A + \mathrm{El}\ B) \to \mathrm{Set}, \\
&\qquad\quad c_{\mathrm{i}} : (\,\textstyle\prod x : A\,)\ \mathrm{El}\ C(\mathrm{i}\ x), \\
&\qquad\quad c_{\mathrm{j}} : (\,\textstyle\prod y : B\,)\ \mathrm{El}\ C(\mathrm{j}\ y), \\
&\qquad\quad z : \mathrm{El}\ A + \mathrm{El}\ B) \\
&\qquad \to\ \mathrm{El}\ C(z) \\
&d_{+}(A, B, C, c_{\mathrm{i}}, c_{\mathrm{j}}, \mathrm{i}\ a) = c_{\mathrm{i}} : (A : \mathrm{Set}, B : \mathrm{Set}, \\
&\qquad\qquad C : (\mathrm{El}\ A + \mathrm{El}\ B) \to \mathrm{Set}, \\
&\qquad\qquad c_{\mathrm{i}} : (\,\textstyle\prod x : \mathrm{El}\ A\,)\ \mathrm{El}\ C(\mathrm{i}\ x), \\
&\qquad\qquad c_{\mathrm{j}} : (\,\textstyle\prod y : \mathrm{El}\ B\,)\ \mathrm{El}\ C(\mathrm{j}\ y), \\
&\qquad\qquad a : \mathrm{El}\ A) \to \mathrm{El}\ C(\mathrm{i}\ a) \\
&d_{+}(A, B, C, c_{\mathrm{i}}, c_{\mathrm{j}}, \mathrm{j}\ b) = c_{\mathrm{j}} : (A : \mathrm{Set}, B : \mathrm{Set}, \\
&\qquad\qquad C : (\mathrm{El}\ A + \mathrm{El}\ B) \to \mathrm{Set}, \\
&\qquad\qquad c_{\mathrm{i}} : (\,\textstyle\prod x : \mathrm{El}\ A\,)\ \mathrm{El}\ C(\mathrm{i}\ x), \\
&\qquad\qquad c_{\mathrm{j}} : (\,\textstyle\prod y : \mathrm{El}\ B\,)\ \mathrm{El}\ C(\mathrm{j}\ y), \\
&\qquad\qquad a : \mathrm{El}\ A) \to \mathrm{El}\ C(\mathrm{j}\ b)
\end{aligned}
$$

- Disjoint union of a family

$$c_\Sigma \in (\prod A : \mathrm{Set}\,)\,(A \to \mathrm{Set}) \to \mathrm{Set}$$

$$\mathrm{El}\ c_\Sigma(A,B) = (\sum x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x) : \mathsf{Type}(A : \mathrm{Set}, B : \mathrm{El}\ A \to \mathrm{Set})$$

$$\begin{aligned}
d_\Sigma \in (\textstyle\prod A &: \mathrm{Set}, B : \mathrm{El}\ A \to \mathrm{Set},\\
&C : ((\textstyle\sum x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x)) \to \mathrm{Set},\\
&c : (\textstyle\prod x : \mathrm{El}\ A, y : \mathrm{El}\ B(x)\,)\ \mathrm{El}\ C \circ \mathrm{pair} \circ \overline{x, y},\\
&z : (\textstyle\sum x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x))\\
\to\ &\mathrm{El}\ C(z)\\
d_\Sigma(A, B, &C, c, \mathrm{pair}\ x\ y)\\
= c(x,y) &: (A : \mathrm{Set}, B : \mathrm{El}\ A \to \mathrm{Set},\\
&C : ((\textstyle\sum x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x)) \to \mathrm{Set},\\
&c : (\textstyle\prod x : \mathrm{El}\ A, y : \mathrm{El}\ B(x)\,)\ \mathrm{El}\ C \circ \mathrm{pair} \circ \overline{x, y},\\
&x : \mathrm{El}\ A, y : \mathrm{El}\ B(x)) \to \mathrm{El}\ C \circ \mathrm{pair} \circ \overline{x, y}
\end{aligned}$$

- Cartesian product of a family

$$c_\Pi \in (\prod A : \mathrm{Set}\,)\,(A \to \mathrm{Set}) \to \mathrm{Set}$$

$$\mathrm{El}\ c_\Pi(A,B) = (\prod x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x) : \mathsf{Type}(A : \mathrm{Set}, B : \mathrm{El}\ A \to \mathrm{Set})$$

$$\begin{aligned}
d_\Pi \in (\textstyle\prod A &: \mathrm{Set}, B : \mathrm{El}\ A \to \mathrm{Set},\\
&C : ((\textstyle\prod x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x)) \to \mathrm{Set},\\
&c : (\textstyle\prod b : (\textstyle\prod x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x)\,)\ \mathrm{El}\ C(\lambda\, b),\\
&z : (\textstyle\prod x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x))\\
\to\ &\mathrm{El}\ C(z)\\
d_\Pi(A, B, C, c, \lambda\, b) = c(b) &: (A : \mathrm{Set}, B : \mathrm{El}\ A \to \mathrm{Set},\\
&C : ((\textstyle\prod x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x)) \to \mathrm{Set},\\
&c : (\textstyle\prod b : (\textstyle\prod x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x)\,)\ \mathrm{El}\ C(\lambda\, b),\\
&b : (\textstyle\prod x : \mathrm{El}\ A\,)\ \mathrm{El}\ B(x)) \to \mathrm{El}\ C(\lambda\, b)
\end{aligned}$$

It is perhaps worth remarking that d_Π is the socalled *funsplit* constant [80][page 55] which exploits the existence of higher type (*i.e.* nested) function types in the logical framework. At the level of *types*, the function space construction has (roughly speaking) the flavour of a categorical *limit*, whereas at the level of *sets*, it has the flavour of a *colimit*, just as with all the other set constructors. An analogous phenomenon appears at the level of types, where the singleton type $\{*\}$ is defined by initiality, rather than as a terminal object.

2.3.2 Inductively defined sets

So far we have typing axioms and equations for the constants that reflect the logical operations. Now we give axioms and equations for inductively defined sets.

The next chapter uses inductive definitions heavily − and not just of sets, but predicates (as least fixed points of monotone predicate transformers), families of sets (as least fixed points of certain operators on these), and sets together with some other structure. I should like to have been able to give a closed form presentation of a principle of inductive definition sufficiently capacious to include those which I consider to be justified, and that I am accustomed to use. This has proved however to be far too ambitious. The problem lies at the frontiers of research in type theory.

A schematic description of a sufficiently extensive class of inductive definitions within a type theory *à la* Martin−Löf has been given by Peter Dybjer [28]. It would be desirable to have instead a closed−form description, exploiting the machinery available in the logical framework to deal with schemata. Together with Anton Setzer, Peter Dybjer have recently [29] written such a description, though it is still in a preliminary form, and far from easy to understand. The paper [29] also sketches a direct interpretation of the principle they describe in (classical) Zermelo−Fränkel extended by an axiom asserting the existence of a Mahlo cardinal. It would be desirable also to have a description of the Setzer−Dybjer principle from a categorical perspective, which seems challenging. Roughly speaking, their principle asserts the existence of weakly initial algebras for an inductively defined family of endofunctors on certain 'slice' categories in which the objects are pairs, of the form (I, f) where I is a (small) set, and f is a morphism from I to a fixed (large) type.

Here I will content myself with showing an approach to formulating the rules for two 'standard' inductively defined sets that may have some novelty. They are based on the use of pre fixed points for predicate transformers over a datatype that is defined inductively. It may be that this approach is some help in simplifying the description of a broad class of inductively defined datatypes.

2.3.2.1 Natural numbers

The formation and introduction rules for the set $\mathbb{N}$ of natural numbers are expressed as follows.

$$\begin{aligned}&c_{\mathbb{N}} : \mathrm{Set}\\ &c_{\mathbb{N}} = c_{+}(c_{\{*\}}, c_{\mathbb{N}}) : \mathrm{Set}\end{aligned}$$

Let $\mathbb{N} = \mathrm{El} \circ \overline{c_{\mathbb{N}}}$. It follows that $\mathbb{N} = \{*\} + \mathbb{N}$.

We take an approach to recursion on the structure of natural numbers which uses a predicate transformer. (By a predicate over a type A I mean an element of the type $A \to \mathrm{Set}$, and by a predicate transformer over A I mean an element of the type $(A \to \text{Set}) \to A \to \text{Set}$.) I write the predicate transformer with a superscript $-$, as in C^-.

$$\begin{array}{l} \_^- : Pow(\mathbb{N}) \to Pow(\mathbb{N}) \\ C^-(n : \mathbb{N}) = \textbf{case}\ n\ \textbf{of}\ \ \begin{array}[t]{l} \mathrm{i} * \to c_{\{*\}} \\ \mathrm{j}\ p \to C(p) \end{array} \end{array}$$

A predicate $C : Pow(\mathbb{N})$ is *progressive* if $(\prod n : \mathbb{N})\ C^-(n) \to C(n)$, which can be written more compactly as $C^- \subseteq C$. An element of $\mathbb{N}$ is *accessible* if (to put it impredicatively) it is in the intersection of all progressive predicates. The principle of recursion on the structure of natural numbers requires that all elements of $\mathbb{N}$ are accessible.

$$d_{\mathbb{N}} \in (\prod C : Pow(\mathbb{N}), c : C^- \subseteq C, n : \mathbb{N})\ C(n)$$

$$\begin{array}{rcl} d_{\mathbb{N}}(C, c, n) & = & \textbf{let}\ \ x = \textbf{case}\ n\ \textbf{of}\ \ \begin{array}[t]{l} \mathrm{i} * \to * \\ \mathrm{j}\ p \to d_{\mathbb{N}}(C, c, p) \end{array} \\ & & \textbf{in}\ \ \ c(n, x) \\ & : & (C : Pow(\mathbb{N}), c : C^- \subseteq C, n : \mathbb{N}) \to \mathrm{El}\ C(n) \end{array}$$

2.3.2.2 W types.

The same approach as taken in the last subsection for the set $\mathbb{N}$ of natural numbers can be illustrated also for the so−called ‘well−ordering’ type constructor introduced by Martin−Löf in [67], and described also in [80][pages 109–114].

Given a family $(A, B) : (\sum A : Set)\ A \to \text{Set}$, intuitively the set $W = W(A, B)$ is the least solution of

$$\begin{array}{l} W : \text{Set} \\ W = (\sum a : A)\ B(a) \to W \end{array}$$

The formation and introduction rules for W are expressed as follows.

$$c_W \in (\prod A : \text{Set})\ (\mathrm{El}\ A \to \text{Set}) \to \text{Set}$$

$$\begin{array}{l} \mathrm{El}\ c_W(A, B) = (\sum a : \mathrm{El}\ A)\ \mathrm{El}\ B(a) \to \mathrm{El}\ c_W(A, B) \\ \qquad\qquad : \ \mathsf{Type}(A : \text{Set}, B : \mathrm{El}\ A \to \text{Set}) \end{array}$$

Let $W(A, B) = \mathrm{El} \circ \overline{c_W(A, B)}$. It follows that

$$W(A, B) = (\sum a : \mathrm{El}\ A)\ \mathrm{El}\ B(a) \to W(A, B)$$

Again, we use a predicate transformer over $W(A, B)$, written with a superscript $-$, as in C^-. For simplicity, I omit the global parameters A and B.

$$\begin{array}{l} \_^- : Pow(W(A, B)) \to Pow(W(A, B)) \\ C^-(z : c_W(A, B)) = \textbf{case}\ z\ \textbf{of}\ \ \mathrm{pair}\ a\ f \to (\prod b : B(a))\ C(f(b)) \end{array}$$

A predicate $C : Pow(W(A, B))$ is *progressive* if $(\prod p : W(A, B))\ C^{-}(p) \to C(p)$, which can be written more compactly as $C^{-} \subseteq C$. An element of $W(A, B)$ is *accessible* if (to put it impredicatively) it is in the intersection of all progressive predicates. The recursion principle requires that all elements of $W(A, B)$ are accessible, and expresses the 'leastness' of $W(A, B)$.

$$
\begin{array}{l}
d_W \in (\prod A : \mathrm{Set}, B : Pow(\mathrm{El}\ A) \\
\qquad\quad C : Pow(W(A,B)), c : C^{-} \subseteq C \\
\qquad\quad p : W(A,B)) \\
\qquad \to\ C(p)
\end{array}
$$

$$
\begin{array}{rcl}
d_W(A, B, C, c, p) & = & \textbf{let}\ \ x = \textbf{case}\ p\ \textbf{of}\ \ \mathrm{pair}\ a\ f \to \lambda b.\ d_W(A, B, C, c, f(b)) \\
 & & \textbf{in}\ \ \ c(p, x) \\
 & : & (A : \mathrm{Set}, B : Pow(\mathrm{El}\ A), \\
 & & \ \ C : Pow(W(A,B)), c : C^{-} \subseteq C \\
 & & \ \ p : W(A,B)) \to \mathrm{El}\ C(p)
\end{array}
$$

2.3.3 Universes

By a universe I mean a set−indexed family of sets closed under some sets and set−forming operators. The notion has a certain arbitrariness as it depends on a particular collection of sets and set−forming operators − *e.g.* 0, 1, +, and Σ. There is an excellent survey of different notions of universe in [82]. In this subsection, after some introductory remarks, I give constants and equations for a family of (cumulative) universes closed under $\mathbb{N}$, $\prod$ and $\sum$.

One purpose fulfilled by the introduction of a universe is to permit the definition of families of sets by recursion, by providing a set or family of sets for that recursion to 'happen into'. For example, one may wish (as in the construction of Scott's D^{∞} model of the untyped λ−calculus), to define a set:

$$
\begin{array}{lrcl}
(\prod n : \mathbb{N})\ F(n) & & & \\
\textbf{where} & F(0) & = & \mathbb{N} \\
 & F(S\ n) & = & F(n) \to F(n)
\end{array}
$$

A definition by recursion on $\mathbb{N}$ requires a *set* (or set valued family indexed by $\mathbb{N}$) in which the values of the function lie. To formalise the definition above we need a family of sets which (at least) contains the set $\mathbb{N}$ and is closed under the unary operator $X \mapsto X \to X$. The construction of provable ordinals given in the fourth chapter is riddled with definitions of this kind.

Another example of the definition of set−valued function by recursion into a universe is afforded by the definition of the $\mathbb{N}$−indexed family of 'standard' finite sets, which can be be written as follows.

$$\begin{array}{lcl} \mathbb{N}(0) & = & \{\,\} \\ \mathbb{N}(S\,n) & = & \mathbb{N}(n) + 1 \end{array}$$

Here one requires a universe which contains the empty set, and is closed under the unary operator $X \mapsto X + 1$. (If one does not make use of such a universe, then as observed by Altenkirch [5] some form of inductively defined identity−type seems required to formalise the construction of this family. As discussed in section 3.1.1 on page 59 there are is something problematic about a notion of inductive definition in which two variables may in some contexts be constrained to be equal. It is curious that by exploiting universes one can sometimes, as in the case of the family of finite sets, obtain the effect of problematic forms of inductive definition.)

A universe is specified merely by closure properties. It is not required that a universe is the *least* family of sets closed under some given constructors, although for some purposes this may be appropriate. (For example, one may wish to introduce a universe of 'standard' countable sets, as the least family containing the (entire) family of finite sets, closed under the restricted quantifiers $(\prod n : \mathbb{N})\ \ldots$ and $(\sum n : \mathbb{N})\ \ldots$.)

In giving examples it soon becomes necessary to use the '**data**' form of co-product constructor with corresponding '`case`' expressions. This allows one to use names for constructors more meaningful than composites of $\mathrm{i}\_$, $\mathrm{j}\_$ and the like.

In the remainder of this chapter we introduce constants and equations for a variety of universes. First c_{U_0}, c_{T_0} for an empty universe, and constants for some other 'microscopic' universes; next c_U, c_T for a 'next universe' operator; finally c_V, c_S for a 'superuniverse' closed not only under the ordinary constructions but also under the next universe operator. Having such a superuniverse allows one to define (internally) a sequence of cumulative universes by recursion on the structure of of natural numbers.

In the following rules, the equations specifying the decoding functions (c_{T_0}, c_T, c_S) have been stipulated to hold 'at the level of types'. Regrettably, I have not given sufficient thought to the question of whether they should have been stipulated at the level of sets.

- The empty universe.

$$\begin{array}{l} c_{U_0} : \mathrm{Set} \\ c_{T_0} : \mathrm{El}\ c_U 0 \to \mathrm{Set} \end{array}$$

These constants can be introduced by definition; that is to say that the rules are redundant.

$$\begin{aligned} c_{U_0} &= c_{\{\ \}} \\ c_{T_0} &= [] = \lambda x.\, \mathbf{case}\ x\ \mathbf{of}\quad - \end{aligned}$$

The following type judgements then hold.

$$\begin{aligned} &\mathrm{El}\ c_{U_0} = \{\ \} : \mathsf{Type} \\ &\mathrm{El}\ c_{T_0}(c) : \mathsf{Type}(c : \{\ \}) \end{aligned}$$

Although the rules are redundant, (c_{U_0}, c_{T_0}) is an (extreme) example of a universe, which is a set reflecting certain closure properties of the "ambient" universe of sets. The notion empty universe raises some rather delicate issues as to what exactly it is closed under. It is for example closed under sum types, the quantifiers Π and Σ, and apparently any set forming operation which is not a plain constant.

- The universe containing only the empty set.

$$\begin{aligned} &c_{U_1} : \mathrm{Set} \\ &c_{T_1} : \mathrm{El}\ c_{U_1} \to \mathrm{Set} \end{aligned}$$

These constants can be introduced by definition.

$$\begin{aligned} c_{U_1} &= c_{\{*\}} \\ c_{T_1} &= [c_{\{\ \}}] = \lambda x.\, \mathbf{case}\ x\ \mathbf{of}\ \ * \to c_{\{\ \}} \end{aligned}$$

The following type judgements then hold.

$$\begin{aligned} &\mathrm{El}\ c_{U_1} = \{*\} : \mathsf{Type} \\ &\mathrm{El}\ c_{T_1}(c) = \{\ \} : \mathsf{Type}(c : \{*\}) \end{aligned}$$

- The Boolean universe, containing just the empty set and the singleton set.

$$\begin{aligned} &c_{U_2} : \mathrm{Set}, \\ &c_{T_2} : \mathrm{El}\ c_{U_2} \to \mathrm{Set}, \end{aligned}$$

We can define this universe as follows.

$$\begin{aligned} c_{U_2} &= c_{+}(c_{\{*\}}, c_{\{*\}}) \\ c_{T_2}(c) &= \mathbf{case}\ c\ \mathbf{of}\ \ \mathrm{i}\,* \to c_{\{\ \}} \\ &\qquad\qquad\quad\ \ \mathrm{j}\,* \to c_{\{*\}} \end{aligned}$$

With the following definitions

$$\begin{aligned} &\mathrm{tt}, \mathrm{ff} : \mathrm{El}\ c_{U_2} \\ &\mathrm{tt} = \mathrm{j}\,* \\ &\mathrm{ff} = \mathrm{i}\,* \end{aligned}$$

the following type judgements then hold.

$$\begin{aligned}
&\text{El } c_{U_2} = \{*\} + \{*\} \\
&\text{El } c_{T_2}(\text{tt}) = \{*\} : \mathsf{Type} \\
&\text{El } c_{T_2}(\text{ff}) = \{\ \} : \mathsf{Type}
\end{aligned}$$

- The next universe operator. This operator maps a family of sets to a more extensive family which contains it and is also closed under 'the usual' set−forming operations.

$$\begin{aligned}
&c_U : (\textstyle\prod A : \text{Set}\,)\ (\text{El } A \to \text{Set}) \to \text{Set} \\
&c_T : (\textstyle\prod A : \text{Set}, B : \text{El } A \to \text{Set}\,)\ \text{El } c_U(A, B) \to \text{Set}
\end{aligned}$$

In the following I shall replace Elc_U by U, Elc_T by T and so on. Moreover, I omit the parameter (A, B) from all occurrences of U and T.

$$\begin{array}{lll}
U = \textbf{data} & \hat{u} & : () \\
& \hat{t}\ a & : (a : \text{El } A) \\
& \hat{+}\ a\ b & : (a : U, b : U) \\
& \hat{n}_0 & : () \\
& \hat{\sigma}\ a\ b & : (a : U, b : T(a) \to U) \\
& \hat{n}_1 & : () \\
& \hat{\pi}\ a\ b & : (a : U, b : T(a) \to U) \\
& \hat{n} & : () \\
\quad : \mathsf{Type} & &
\end{array}$$

$$\begin{array}{lll}
T(c) = \textbf{case } c \textbf{ of} & \hat{u} & \to \text{El } A \\
& \hat{t}\ a & \to \text{El } B(a) \\
& \hat{+}\ a\ b & \to T(a) + T(b) \\
& \hat{n}_0 & \to \{\ \} \\
& \hat{\sigma}\ a\ b & \to (\sum x : T(a)\,)\ T(b(x)) \\
& \hat{n}_1 & \to \{*\} \\
& \hat{\pi}\ a\ b & \to (\prod x : T(a)\,)\ T(b(x)) \\
& \hat{n} & \to \mathbb{N} \\
\quad : \mathsf{Type}(c : U) & &
\end{array}$$

- The super universe [82]. The super−universe is the least family of sets closed under the next universe operator (as well as all the standard type forming operations, which for simplicity I omit).

$$\begin{aligned}
&c_V : \text{Set} \\
&c_S : \text{El } c_V \to \text{Set}
\end{aligned}$$

In the following I shall replace Elc_U by U, Elc_T by T and so on. Recall that c_U and c_T are the set forming constants for the next universe operator.

$$
\begin{array}{l}
V = \ \textbf{data} \ \begin{array}[t]{ll} \hat{u}\, a\, b & : (a : V, b : S(a) \to V) \\ \hat{t}\, a\, b\, c & : (a : V, b : S(a) \to V, c : U(S(a), S \circ b)) \end{array} \\
\qquad : \mathsf{Type} \\
S(c) = \ \textbf{case} \ c \ \textbf{of} \ \begin{array}[t]{ll} \hat{u}\ a\ b & \to U(S(a), S \circ b) \\ \hat{t}\ a\ b\ c & \to T(S(a), S \circ b, c) \end{array} \\
\qquad\quad : \mathsf{Type}(c : V)
\end{array}
$$

There is a small abuse of the composition operator above. $S \circ b$ is used as a suggestive notation for $\lambda x.\, S(b(x))$.

- A vista of universes. The superuniverse construction can be elaborated upon so as to obtain an operator on families, which assigns to a family the next more extensive family which is closed under the next universe operator. This generalisation can be taken in a number of directions.

  In one direction, one can iterate the step from the next−universe operator to the next−super−universe operator. As remarked by Peter Dybjer, this step has close similarities with a key part of the ordinal−theoretic Veblen derivative operator, namely the construction of the next infinite ordinal above a given ordinal which is closed under a given normal function.

  In another direction, one can add a certain fixed−point operator to the operations under which a universe is closed, which abstracts away from the operation under which the universe must be closed. Pursuing this direction we arrive at certain forms of Mahlo universe, due to Setzer [100]. The Mahlo universe is interesting because all the universes it contains internally admit a principle of structural recursion, although the Mahlo universe itself admits no such scheme. It is only a fixed point, not one which is in any sense ‘least’.

Chapter 3

Transition systems, interactive systems

The previous chapter set out a framework for constructive set theories, and in that framework described a particular limited set theory. The chapter after this one exploits the closure properties of the sets in this theory to determine a lower bound on its proof theoretical strength.

This chapter concerns two kinds of systems which are pervasive not only in computer science, but also in nature, namely transition systems and interactive systems. Ordinal notation systems are a particular kind of transition system. Interactive systems are useful for modelling interactive systems typified by 2−person games of a certain kind. I describe an approach to representing these with data structures in dependent type theory, so that one may program with them.

Some aspects of my approach may be novel. For example, a transition system is usually taken to be simply a binary relation on a set, and modelled with a binary propositional function. It appears though that there is often more than one way to skin a cat, particularly in type theory. Even such an apparently straightforward notion as a binary relation is susceptible to differing interpretations, and admits at least one representation subtly different from 'the obvious' one. In a sense, this representation is more computational, and less propositional. There is perhaps at least a moral here, that in approaching a problem in type theory one must start thinking right from the beginning. From one point of view this is sad, but one may also look on it more positively as an opportunity.

The bifurcation between the computational and propositional representation of relations can be traced back to the questions of representing 'subsets' in type theory. There are really two notions: family (given parametrically, by an exhaustive enumeration), and predicate (given by a propositional function). The relationship between these different notions is not at all straightforward; it is intimately connected with the extraordinarily vexatious issue of understanding 'equality' in type theory[1]. From the beginning, it has been clear that one must distinguish between equational *judgements* and *proposition*; but this is only to double the problems that arise. The course I have adopted here is simply to abstain from appeal to any general notion of propositional equality, or form of the identity type, whether extensional or intensional. The first two sections of this chapter discuss the notions of 'subset' in type theory from this perspective, and the repercussions of taking this abstentionist point of view.

There are two main sections in this chapter. The first 3.2 is concerned with transition systems. It gives the definition, the definitions of some related notions, and illustrates their application with a number of examples and constructions. The other 3.3 is concerned with interactive systems, which are somewhat more complex than transition systems. Again, I give the definition, that of some related notions that pertain to interactive programming (and strategies for games), and present some illustrative examples and constructions.

The structure I shall use to represent interactive systems seems to have appeared first in the work of Petersson and Synek [84], also described in chapter 16 of [80]. The structure I shall use to represent transition systems is a formal simplification of their notion. It is natural to wonder whether an endless series of other variants may not be wrung from it. As far as I have been able to see, this is (fortunately) not the case. There is indeed a yet further simplification one can make, obtaining a vestigial notion of dynamical system, and so a 'trinity' of structures; but in a sense, that's all. The question appears to be connected with certain investigations in predicate transformer semantics by Gardiner, Martin and de Moore [34]. There is some discussion of this in the final section 3.4 of this chapter.

These matters dealt with in this chapter lead towards the concluding chapter, which contains my suggestions of how the study of proofs of wellfoundedness initiated by Gentzen and developed in ordinal−theoretic proof theory may be relevant to issues in computer science.

[1] For that matter, in logic generally. It is interesting to note that the symbol = for equality was introduced as late as 1557 (by Robert Recorde, choosing the sign = 'bicause noe 2. thynges can be moare equalle' than such parallel lines). The nature of and rôle of equality among the logical constants and quantifiers has been hotly debated in philosophical logic for well over a century. The problems have only been exacerbated with the development of constructive logic, which has raised in a new form quite old questions pertaining to 'intensional' equality.

3.1 Predicates versus families

There are at least two ways in which we are given subsets of a set S: using a *predicate* over S, (*e.g.* $\{\ n \in \mathbb{N} \mid \forall m, n \in \mathbb{N}\ .\ m \times n = k \Rightarrow m = 1 \vee n = 1\ \}$), or using a *function* into S (*e.g.* $\{\ (t^2, 2 \times t) \mid t \in \mathcal{R}\ \}$). In Zermelo−Fränkel set theory, it is the separation axiom which guarantees us subsets of the first kind, so it is natural to refer to such subsets as separative. The axioms that guarantee the existence of sets whose elements are the values attained by a function are more various, but a typical axiom with this rôle is the replacement axiom. The general forms are as follows.

$$\begin{array}{lll} \textit{separative} & \{\ s : S \mid P(s)\ \} & \text{where } P : S \to \mathrm{Set} \\ \textit{parametric} & \{\ p(i) \mid i : I\ \} & \text{where } (I, p) : (\sum I : \mathrm{Set}\,)\ I \to S \end{array}$$

In a set expression of *separative* form, we pick out the elements of the set from a given set S with a predicate P, or propositional (*i.e.* proposition valued) function, defined on S. According to the identification of propositions with sets that is made in set theories in Martin−Löf's style, a predicate defined on S is a function defined on S whose values are sets. I shall usually refer to a subset given in separative form as a predicate.

In a set expression of *parametric* form, we have a function p defined on an *index* set I, which gives a 'generic' element $p(i)$ of the set expressed in terms of the parameter $i : I$.

These two forms of set expression correspond to the covariant (parametric) and contravariant (separative) powerset functors on the category of sets, which may be written as follows:

$$\begin{array}{lll} \mathit{Fam}(X) & = & (\sum I : \mathrm{Set}\,)\ I \to X \\ \mathit{Fam}(f : X \to Y) & = & \lambda(I, p).\,(I, f \circ p) : \mathit{Fam}(X) \to \mathit{Fam}(Y) \\ \mathit{Pow}(X) & = & X \to \mathrm{Set} \\ \mathit{Pow}(f : X \to Y) & = & \lambda P.\, P \circ f : \mathit{Pow}(Y) \to \mathit{Pow}(X) \end{array}$$

Let Type denote the (large) category of sets and functions between them. In accordance with the analysis of these notions in the predicative framework of the previous chapter, I shall suppose that $\mathit{Fam}()$ and $\mathit{Pow}()$ are in fact functors on the category of types and functions.

$$\mathit{Fam}(\_), \mathit{Pow}(\_) \quad : \quad \mathsf{Type} \to \mathsf{Type}$$

3.1.1 Equality

The relation between the two notions of subset is rather subtle in type theory. As demonstrated in this section, the relation hinges on questions connected with equality, and in particular propositional equality, most particularly *identity*, the 'least reflexive relation'. As evidence that this topic is difficult, one can point to the several formulations of rules for identity that have appeared (or never got that far) in the development of Martin–Löf's systems. Furthermore, there are the surprising results of Hofmann ([47]) and Streicher ([102]) about identity between proofs of identity propositions.

I wish to abstain from assuming that there is for an arbitrary set S a distinguished equivalence relation $Id(S, \_, \_)$ which is included in any other. In other words, I wish to avoid use of any form of schematic identity type, whether intensional or extensional. Of course, this is a nuisance. What it means in practice is that many if not most of the 'sets' which we refer to in informal reasoning must be rendered as 'setoids' (that is, sets with an equivalence relation) when that reasoning is formalised. We must check that these equivalence relations are compatible with the operations and predicates of interest. For example, this is the case when defining a category; the homsets must be equipped with equivalence relations compatible with composition. For some purposes it may even be necessary to equip the objects with an equivalence relation. A considerable part of the charm of a set–theoretic framework is that 'one knows what one means by equality'.

There are equivalence relations which percolate through most of the usual type forming operators. For example, in a set theory which includes the set of natural numbers and the universe $(c_{U_2}, \lambda b : c_{U_2}.\, T_{c_{U_2}}(b) : \mathrm{Set})$ of Boolean truth–values we can (as is well known) define a boolean valued equality between natural numbers by recursion on their structure.

$$
\begin{aligned}
&eq : \mathbb{N} \to \mathbb{N} \to c_{U_2} \\
&eq(0, 0) = 1 \\
&eq(0, n+1) = 0 \\
&eq(m+1, 0) = 0 \\
&eq(m+1, n+1) = eq(m, n)
\end{aligned}
$$

By pre–composing this function to $T_{c_{U_2}}$ we obtain a set valued equivalence relation which is included in any reflexive relation on $\mathbb{N}$. It is clear that this way of obtaining an equality relation can be extended in a natural way to concrete data structures in general, and from there to extensional equality between functions. Yet these 'percolated' equivalence relations are often finer than the equalities of mathematical interest in a particular case, for example the relation of equiconvergence between Cauchy sequences of rationals, or the equivalence relation between pairs of natural numbers in the usual construction of the integers from the rational numbers. Moreover, this percolation does not amount to a distinguished equality relation applicable uniformly throughout the type of sets; the type of sets is not given by an exhaustive specification of the possible constructors for sets.

I have mentioned that the relation $I(S, \_, \_)$ is usually characterised as the least reflexive relation on the set S. This suggests that one should think of it as inductively defined, as for example in [80], page 61. For present purposes, it makes little difference if one thinks of (say) the first argument of the relation as a parameter s in the inductive definition of a 'singleton' predicate $I(S, s, \_)$. The problem is that it is far from easy to arrive at a general conception of inductive definition encompassing the identity type. Such an inductive definition, if that is what it is, differs in a crucial way from inductive definitions of datatypes such as the natural numbers, or of predicates and relations such as accessibility with respect to a binary relation. (There are several examples throughout this chapter.) In a certain sense, the definition is not *linear*. In the case of the identity relation, the specification of the constructors by means of which one may form elements of the set $I(S, s, s')$ depends on whether the parameters s and s' are equal, or the same. It seems this equality between parameters has to be interpreted as judgemental equality; after all, we are in the course of attempting to define a general propositional equality. However it is far from clear whether these equational constraints, which occur negatively in the definition of $I(S, s, s')$ have to be allowed as part of a context, or accounted for in some other way. What is clear is that if $I(S, s, s')$ is to be regarded as inductively defined, then there is something new in this conception of inductive definitions.

In this connection, it may be worth mentioning how inductive definitions are dealt with in the experimental system 'half'[2] developed in Chalmers as a framework for checking constructive proofs. To introduce an inductively defined datatype such as that of the natural numbers $\mathbb{N}$, one writes a recursive equation for the constant $\mathbb{N}$, as follows.

$$\mathbb{N} = \mathbf{data}\ \{0, S(p : \mathbb{N})\} : \mathrm{Set}$$

To introduce a recursor constant for this datatype, one writes another recursive

[2]The constructions of provable ordinals reported in the next chapter were carried out and checked with the aid of this system.
The 'half' system is unsupported by anyone at Chalmers. It has been superseded by 'Agda', currently maintained by Catarina Coquand, and accessible at `http://www.cs.chalmers.se/~catarina/agda`.

equation, as follows.

$$\begin{aligned} R(A : Pow(\mathbb{N}), a : A(0), b : (\textstyle\prod n : \mathbb{N}, \_ : A(n)\,)\ A(S(n)), n : \mathbb{N}) \\ = \textbf{case}\ n\ \textbf{of}\ \ 0 \quad\ \ &\to a \\ S(p) &\to b(n, R(A, a, b, n)) \\ : A(n) \end{aligned}$$

There is nothing in this system to distinguish a recursive equation which introduces an inductively defined set (like $\mathbb{N}$) from one which introduces a function defined by recursion on the construction of elements of that set (like R). It just happens that in the former case, the type of the equation is Set, or more generally a Set−valued function, and the right hand side is in most cases a disjoint union of some kind (signified by the **data** construction, with constructors 0 and $S(\_)$). It just happens that in the latter case, the right hand side has usually the form of a **case** construction. The system does not 'police' any particular scheme of correct inductive definition (such as strict positivity, or even monotonicity). However, as with any other definition, the left hand side of an inductive definition must be *linear*, in that no parameter of the definition may be repeated. So if one tries to define an identity predicate by writing a definition of the form

$$I(S : \mathrm{Set}, s : S, s : S) = \textbf{data}\ \ldots$$

the system complains of an error. This is exactly what one expects if one is accustomed to the treatment of definitions in most[3] functional programming languages.

As far as I am aware, the issue to which I am referring has not been explicitly discussed in the literature of inductive definitions in type theory. (In essence, the issue is not peculiar to type theory; it arises already in a presentation of inductive definitions in the language of first order logic, as in [63].) Because of these misgivings, I am particularly interested in constructions which can be defined with no reference to general propositional equality on a set. Of course, among these one may be particularly interested in constructions of *particular* equivalence relations.

The rest of this section demonstrates three points of contrast between predicates and parametric subsets.

[3]The functional programming language 'Miranda' is an exception. In Miranda the left hand side of a definition need not be linear in its parameters. One can regard this as a form of 'sugar', whose expansion involves the explicit introduction of conditional expressions that depend on an appropriate, 'percolated' boolean−valued equality function.

3.1.2 Separative is 'more than' parametric.

If we are given a subset of a (small) set S in separative form, $\{ s : S \mid P(s) \}$, we may form the set $(\sum s : S)\ P(s)$ of pairs (s, p) with $s : S$, $p : P(s)$. This set of pairs can serve as the index set for a subset of S in parametric form, as follows.

$$\{ p_0 \mid p : (\sum s : S)\ P(s) \}.$$

This parametrically given subset is intuitively equivalent to the original separative subset in the sense that every $s \in S$ for which $P(s)$ holds, that is for which we have a proof $p : P(s)$, is obtained ('once' per proof of $P(s)$) as a value of the projection function. Conversely, every value of the projection function $\__0$ satisfies P by construction. However, to actually *say* that an arbitrarily given $s : S$ 'is' a value of the projection function presupposes that we are given an equality on S. Even if we had such an equality, the equivalence between the two representations holds only if S is a set, as the index of a parametric subset has to be small.

3.1.3 Predicates enjoy more closure properties

Both kinds of subset are closed under unions (of small families).

$$\begin{array}{lcl} \bigcup_{i:I} P(i) & \triangleq & \lambda s.\, (\sum i : I)\ P(i, s) \\ \bigcup_{i:I}(J(i), \lambda j.\, p(i, j)) & \triangleq & ((\sum i : I)\ J(i), \lambda(i, j).\, p(i, j)) \end{array}$$

In particular, the notion of an empty subset of a set can be expressed both in separative and parametric form.

Predicates are closed under intersections of small families.

$$\bigcap_{i:I} P(i) \quad = \quad \lambda s.\, (\prod i : I)\ P(i, s)$$

Indeed, the predicates over a set (or for that matter a type) form a Heyting algebra. On the other hand, to form the intersection of even two parametrically given subsets we need to refer to an equality relation.

$$\{ f(i) \mid i : I \} \cap \{ g(j) \mid i : J \} = \{ f(k_0) \mid k : (\sum (i, j) : I \times J)\ f(i) = g(j) \}$$

Only in the degenerate case of an empty intersection can we express the notion of the full subset in parametric form, and then only when the underlying set is small.

3.1.4 Inclusion and overlap

We can say that a family is included in a predicate, or intersects with (*i.e.* has nonempty intersection with) a predicate.

$$\begin{array}{lll} \{\ p(i) \mid i : I\ \} \subseteq \{\ s : S \mid P(s)\ \} & \triangleq & (\prod i : I\ )\ P(p(i)) \\ \{\,\} \neq \{\ p(i) \mid i : I\ \} \cap \{\ s : S \mid P(s)\ \} & \triangleq & (\sum i : I\ )\ P(p(i)) \end{array}$$

However, to express inclusion in a parametric subset, or that two parametric subsets overlap requires equality on the underlying set.

3.2 Transition systems

A binary relation R on a set S assigns to each element $s \in S$ two subsets of S; namely the set of elements of S to which it is related, and the set of elements which are related to it. Of course, it is enough to have just one of these subsets, so long as we know which 'direction' of the relation it is for. We may therefore think of a relation simply as a function from elements of S to subsets of S. There are two ways of modelling subsets, as described above. So for each of the approaches to modelling subsets of a set S explained above, there is an approach to modelling binary relations on S. The following equations show the typical forms of the two kinds of relation. (Thus in the second equation, the relation is represented as a set−valued function I together with a function $d : (\prod s : S\ )\ I(s) \to S$.

$$\begin{array}{ll} R = \lambda s, s' : S.\, R(s, s') & : S \to \mathit{Pow}(S) \\ R = \lambda s.\,(I(s), \lambda i : I(s).\, d(s,i)) & : S \to \mathit{Fam}(S) \end{array}$$

The first of these is entirely familiar. The second does not seem to have received much attention among type theorists as a possible analysis of the notion of binary relation[4]. Note however that a transition system can be regarded as a coalgebra for the functor $\mathit{Fam}(\_)$ on the category of types, and in that guise it is unlikely that these have escaped the attention of category theorists.

The general form of an object of type $S \to \mathit{Fam}(S)$ is $\lambda x : S.\,(T(s), \lambda t : T(s).\, d(s,t))$ where

$$\begin{array}{lll} T & : S \to \mathrm{Set} & \text{– \textit{transition predicate}} \\ d & : (\prod s : S\ )\ T(s) \to S & \text{– \textit{transition function}} \end{array}$$

[4]The usual representation of a directed, binary−branching graph in LISP storage cells can however be seen as of this kind. A directed graph is a binary relation between vertices. When a graph is represented by a heap, the set S consists of heap cell addresses. The content of the heap cells determine for each cell s whether it is a '*nil*' cell (related to no other cells), or a '*cons*' cell, with two fields '*car*' and '*cdr*'. In the case of a nil cell, the field set $I(s)$ is empty, whereas in the case of a *cons* call it is the two−element set $\{\, car, cdr\,\}$, and the heap contents determine for each field label $i : \{\, car, cdr\,\}$ the cell $d(s,i)$ to which the *cons* cell is related by that field.

I shall frequently write the transition function $d(s,t)$ using the less noisy notation $s[t]$, and where necessary use decorations such as $s[t]'$, $s[t]^*$ *etc.* to distinguish different transition functions. (This notation also happens to be one widely used in connection with ordinal notation systems.) I call such a structure a *transition structure* on S, and a transition system to be a set S together with a transition structure on S. I shall often refer to the elements of the set S as states. One can think of the set $T(s)$ as the set of transitions from the state s, and $d(s,t)$ as the state which is the destination of the transition t. (It is also possible to turn things round, and think of $T(s)$ as the set of transitions with *destination* s, and of $d(s,t)$ as the *origin* of transition t.)

Note that the type $S \to \mathit{Fam}(S)$ is a proper type, as $\mathit{Fam}(\_)$ contains an existential quantification over Set. Given a family of sets $(A,B) : \mathit{Fam}(\mathrm{Set})$ (for example the family of finite sets), we can relativise the definition of 'transition structure' so that the sets of transitions are restricted to be values of B, and so obtain a *set* of transition structures.

$$\mathit{fam}_{(A,B)}(S : \mathrm{Set}) \;\triangleq\; (\textstyle\sum a : A\,)\; B(a) \to S$$

It is occasionally useful to generalise the notion of transition structure slightly to a function $A_1 \to \mathit{Fam}(A_2)$, where A_1 and A_2 need not be the same. In categorical terms, such a function is a morphism in the Kleisli category associated with the endofunctor $\mathit{Fam}(\_)$ on the large category Type. I shall call such a function a transition structure *from* A_1 to A_2.

If a is related to b, then there is always some reason for it, or in other words, a proof of the proposition that the relation holds. If we picture the relation as a directed graph, then the arrows in the graph are really bunches of labelled arrows, where the labels are the various reasons why the source and destination nodes are related. A more intensional way to model a relation is as a function which assigns to a node an indexed family, consisting of an index set and a node−valued function defined on it, giving the immediate predecessors of the node in parametric form. One can think of the indices as 'transitions', and so it is natural to call this a transition system.

3.2.1 Predicate transformers induced by a transition structure

Let $\lambda x.\,(T(s), \lambda t.\, s[t])$ be a transition structure from S_1 to S_2. The transition structure gives rise to two predicate transformers (functions between predicates) as follows.

$$\begin{array}{lcl} (\lambda P.{}^{\bullet}P), (\lambda P.{}^{\circ}P) : Pow(S_2) \rightarrow Pow(S_1) \\ {}^{\bullet}P(s) & \triangleq & (\prod t : T(s)\,)\ P(s[t]) \\ {}^{\circ}P(s) & \triangleq & (\sum t : T(s)\,)\ P(s[t]) \end{array}$$

If Φ is the transition structure, the predicate transformers will be written $\Phi^{\bullet}\_$ and $\Phi^{\circ}\_$. (When the transition structure is implicit, I prefer to write just ${}^{\bullet}\_$, ${}^{\circ}\_$.)

The predicate ${}^{\bullet}P$ holds of a state s if *all* transitions from the state s lead to a state satisfying P, and the predicate ${}^{\circ}P$ holds of a state s if there is *some* transition from the state s that leads to a state satisfying P. For example, the predicate ${}^{\bullet}\mathit{False}$ means that no transitions lead from the argument state, while ${}^{\circ}\mathit{True}$ means that there is at least one transition which leads from the argument state.

Note that ${}^{\bullet}$ commutes with taking the intersection of a (small−) indexed family of predicates, in the sense that ${}^{\bullet}(\cap_{i:I} X_i) = \cap_{i:I}({}^{\bullet}X_i)$. It follows that ${}^{\bullet}$ is monotone with respect to the ordering of predicates by inclusion. Similarly, ${}^{\circ}$ commutes with taking the union of a family of predicates: ${}^{\circ}(\cup_{i:I} X_i)$ equals $\cup_{i:I}({}^{\circ}X_i)$, and it follows that the operator is monotone with respect to inclusion. There is in general a considerable duality between ${}^{\bullet}$ and ${}^{\circ}$. By the de Morgan rules of classical logic, we have that $\complement({}^{\circ}X) = {}^{\bullet}(\complement X)$ (where $\complement\_$ denotes the complementation operator).

In the case when there is just one state set S, we call a predicate *progressive* if ${}^{\bullet}P \subseteq P$. A proof of progressivity is in essence an algebra for ${}^{\bullet}$, considered as a functor on the category of predicates over S with inclusion proofs as morphisms, with respect to the trivial equality between proofs (which identifies all proofs of the same proposition).

An *accessible* state is one in the intersection of all progressive predicates.

$$\begin{array}{lcl} \mathit{Acc} & : & Pow(S) \\ \mathit{Acc} & \triangleq & \bigcap\{\ X : Pow(S) \mid \mathit{Prog}(X)\ \} \end{array}$$

The definition of *Acc* can be expressed using one level of second order universal quantification over predicates. (This is not predicative, because the domain of the quantifier includes the predicate being defined). Unpacked, it reads:

$$\mathit{Acc} \ \triangleq\ \{\ s : S \mid (\prod X : Pow(S)\,)\ \mathit{Prog}(X) \rightarrow X(s)\ \}$$

We shall be concerned in the next chapter with the extent to which proofs of accessibility can be be constructed without essential use of second order quantification.

A transition system is *wellfounded* if $S \subseteq \mathit{Acc}$, or in words every state satisfies every progressive predicate.

3.2.2 Sequential composition of transition systems

Suppose we are given two transition structures Φ_1 and Φ_2 with the following types.

$$\begin{array}{l}\Phi_1 = \lambda a.\,(B_1(a), \lambda b_1.\, a[b_1]) : A_1 \to \mathit{Fam}(A_2) \\ \Phi_2 = \lambda a.\,(B_2(a), \lambda b_2.\, a[b_2]) : A_2 \to \mathit{Fam}(A_3)\end{array}$$

We define another

$$\Phi = \lambda a.\,(B(a), \lambda b.\, a[b]) : A_1 \to \mathit{Fam}(A_3)$$

where

$$\begin{array}{lcl}B(a) & = & (\sum b : B_1(a)\,)\ B_2(a[b]) \\ a[(b_1, b_2)] & = & a[b_1][b_2]\end{array}$$

Then we have

$$\begin{array}{lcll}\Phi^{\circ} & = & \Phi_1^{\circ} \circ \Phi_2^{\circ} & : \ \mathit{Pow}(A_3) \to \mathit{Pow}(A_1) \\ \Phi^{\bullet} & = & \Phi_1^{\bullet} \circ \Phi_2^{\bullet} & : \ \mathit{Pow}(A_3) \to \mathit{Pow}(A_1)\end{array}$$

We write $\Phi_1 \mathbin{\hat{\circ}} \Phi_2$ for Φ, and refer to $\_ \mathbin{\hat{\circ}} \_$ as *sequential composition*. If you think of Φ_1 as a relation $R_1 \subseteq A_1 \times A_2$, and Φ_2 as a relation $R_2 \subseteq A_2 \times A_3$, then $\Phi_1 \mathbin{\hat{\circ}} \Phi_2$ represents the relational composition $(R_1; R_2) \subseteq A_1 \times A_3$.

When there is just one state space (*i.e.* $A_1 = A_2 = A_3$), the following transition structure **skip** serves as a unit for sequential composition.

$$\begin{array}{lcl}B(a) & = & N_1 \\ a[b] & = & a\end{array}$$

The choice of an index set (here N_1) is to some extent arbitrary (as long at it is always small and inhabited), and could even depend on the state. The transition structure **skip** represents the identity relation.

3.2.3 Union of transition systems

Suppose we are given two transition structures Φ_1 and Φ_2, both from a set A_1 to a set A_2.

$$\begin{array}{l}\Phi_1 = \lambda a.\,(B_1(a), \lambda b_1.\, a[b_1]) : A_1 \to \mathit{Fam}(A_2) \\ \Phi_2 = \lambda a.\,(B_2(a), \lambda b_2.\, a[b_2]) : A_1 \to \mathit{Fam}(A_2)\end{array}$$

we define another such transition system

$$\Phi = \lambda a.\,(B(a), \lambda b.\, a[b]) : A_1 \to \mathit{Fam}(A_2)$$

where

$$\begin{array}{lcl}B(a) & = & B_1(a) + B_2(a), \\ a[\mathrm{i}\ b_1] & = & a[b_1], \\ a[\mathrm{j}\ b_2] & = & a[b_2]\end{array}$$

Then we have

$$\Phi^\circ(P) = \Phi_1^\circ(P) \cup \Phi_2^\circ(P)$$

We write $\Phi_1 \cup \Phi_2$ for Φ, and call it the *union* of Φ_1 and Φ_2. The interpretation of $\Phi_1 \cup \Phi_2$ is that the transitions from a state are put together. If you are choosing successive transitions, at any point you have available to you transitions from *both* structures.

For a zero, the transition structure 0 on a state space A can be defined by

$$\begin{array}{lcl} B(a) & = & \{\,\} \\ a[b] & = & \textbf{case } a \textbf{ of } \quad - \end{array}$$

3.2.4 The transitive and reflexive closure of a transition system

If $\Phi = \lambda a.\,(B(a), \lambda b.\, a[b]) : A \to \mathit{Fam}(A)$ is a transition structure on a set A, then we may form its transitive and reflexive closure Φ^* , which is intuitively the least solution of the equation $\Psi = \textbf{skip} \cup (\Phi \mathbin{\hat{;}} \Psi)$. This is only an 'intuition', because there is no obvious way to define an order on $A \to \mathit{Fam}(A)$ unless we are given an equality relation (or at least an order) on A. Nevertheless the intuition is sound, and we may define

$$\Phi^* = \lambda a.\,(B^*(a), \lambda bs.\, a[bs]^*) : A \to \mathit{Fam}(A)$$

where

$$\begin{array}{lcl} B^*(a) & = & \{0\} + (\sum b : B(a))\; B^*(a[b]) \\ a[bs]^* & = & \textbf{case } bs \textbf{ of } \; \begin{array}{l} 0 \qquad\quad \to a \\ (b, bs') \to a[b][bs']^* \end{array} \end{array}$$

The recursive equation specifying B^* should be understood as defining B^* to be it's *least* solution.

The transitions in $B^*(a)$ are finite lists $bs = (b_0, b_1, \ldots, b_k)$ of the transitions from Φ, where $b_0 : B(a)$, $b_1 : B(a[b_0])$, $\ldots$, $b_k : B(a[b_0][b_1]\ldots[b_k])$. The state $a[bs]$ is the state we finally land in after taking the successive transitions which form bs. There is a natural concatenation operation, which is a proof of the transitivity of Φ^*.

$$\begin{array}{lcl} \mathit{concat} & : & (\prod a : A, bs : B^*(a))\; B^*(a[bs]^*) \to B(a) \\ \mathit{concat}(a, bs) & \triangleq & \textbf{case } bs \textbf{ of} \\ & & \quad \begin{array}{lcl} \mathrm{i}\,b & \mapsto & \lambda bs'.\, \mathrm{j}(b, bs') \\ \mathrm{j}(b, bs') & \mapsto & \lambda bs''.\, \mathrm{j}(b, \mathit{concat}(a[b], bs', bs'')) \end{array} \end{array}$$

The transitive closure of Φ (written Φ^+) may then be defined as $\Phi \mathbin{\hat{;}} \Phi^*$.

The transitive closure construction preserves wellfoundedness. To show this, we need to show that for an arbitrary state a and predicate X, if X is progressive with respect to (the composite transitions of) Φ^+ then it is progressive already with respect to (the atomic transitions of) Φ. This is obvious because the atomic transitions are *among* the composite transitions of Φ^+.

The reader may have already become a little exasperated that I have not actually defined what it means for a transition structure to be transitive. I have not done so because such a definition would require an equality relation on the underlying state, and would depend ultimately on being able to express with a proposition that one family is included in another. Although we cannot *say* that a transition structure is transitive, it may be perhaps slightly surprising that we can nevertheless express quantification over transitive transition structures. Instead of saying for example 'If Φ is a transitive transition structure on A, then $P(\Phi)$', we can say 'If Φ is *any* transition structure on A, then $P(\Phi^+)$'.

3.2.5 Segments of a transition structure

Suppose that $\lambda a.\,(B(a), \lambda b.\, a[b]) : A \to \mathit{Fam}(A)$ is a transition structure on a set A. Given an element a of A we can define an transition structure $\lambda s.\,(C(s), \lambda c.\, s[c])$ on $B^+(a)$, using concatenation:

$$\begin{array}{rcl} C(bs) & = & B^+(a[bs]) \\ bs[bs'] & = & \mathit{concat}(a, bs, bs') \end{array}$$

This transition structure is called the *segment* of A up to a, and written $\mathit{Seg}_A(a)$. As a varies through A, we get a family of transition structures.

3.2.6 Relation transformers induced by a transition structure

Suppose we are given two transition structures Φ_1 and Φ_2 with the following types.

$$\begin{array}{l} \Phi_1 = \lambda a.\,(B_1(a), \lambda b_1.\, a[b_1]) : A_1 \to \mathit{Fam}(A_1) \\ \Phi_2 = \lambda a.\,(B_2(a), \lambda b_2.\, a[b_2]) : A_2 \to \mathit{Fam}(A_2) \end{array}$$

The two predicate transformers ${\Phi_1}^\bullet$ and ${\Phi_2}^\circ$ can be made to act jointly on the argument places of a binary relation $R : A_1 \to \mathit{Pow}(A_2)$ in essentially two interesting ways (modulo transposition of the relation).

Given a binary relation $R : A_1 \to \mathit{Pow}(A_2)$ we define $(\Phi_1, \Phi_2)^{\bullet\circ} R : A_1 \to \mathit{Pow}(A_2)$ as follows.

$$\begin{array}{lcl} (\Phi_1, \Phi_2)^{\bullet\circ} R & : & A_1 \to \mathit{Pow}(A_2) \\ ((\Phi_1, \Phi_2)^{\bullet\circ} R)(a_1, a_2) & = & (\prod b_1 : B_1(a_1)\,)\,(\sum b_2 : B_2(a_2)\,)\; R(a_1[b_1], a_2[b_2]) \end{array}$$

Similarly, we define $(\Phi_1, \Phi_2)^{\circ\bullet} R : A_1 \to Pow(A_2)$ as follows.

$$
\begin{array}{lcl}
(\Phi_1, \Phi_2)^{\circ\bullet} R & : & A_1 \to Pow(A_2) \\
((\Phi_1, \Phi_2)^{\circ\bullet} R)(a_1, a_2) & = & (\sum b_1 : B_1(a_1))\,(\prod b_2 : B_2(a_2))\; R(a_1[b_1], a_2[b_2])
\end{array}
$$

Now a *simulation* of Φ_1 by Φ_2 is a relation $R : A_1 \to Pow(A_2)$ such that a transition according to Φ_1 can be matched by a transition according to Φ_2 that preserves R, in the sense that $R \subseteq (\Phi_1, \Phi_2)^{\bullet\circ} R$. For example, the empty relation is obviously a simulation; if R and S are simulations, then so is $R \cup S$; and in general simulations are closed under unions of (small) indexed families. It is also easily seen that simulations are closed under relational composition, in the sense that if R is a simulation of Φ by Ψ, and S is a simulation of Ψ by Θ, then $(R; S)$ is a simulation of Φ by Θ.

Clearly the notion of a transition system is highly intensional – there can be lots of representational redundancy. For example, here are two representations of the $\geq$ relation between natural numbers.

$$
\begin{array}{lcl}
B_1(n) & = & \mathbb{N}, \\
n[m]_1 & = & n + m
\end{array}
$$

Here is a different transition structure that represents extensionally the same relation.

$$
\begin{array}{lcl}
B_2(n) & = & \mathbb{N}, \\
n[m]_2 & = & n + \lfloor m \div 19 \rfloor
\end{array}
$$

In the second transition system there are 19 different transitions for each of those in the first system, and in a sense the identity of these transitions is irrelevant. Though they differ as families of sets, the two transition systems are mutually similar.

A simulation relation is insensitive to the representation, as it does not distinguish between a state and its family of destination states.

Where there is something called a structure, it is natural to look for something to call a morphism between such structures. If one considers a transition system to be a coalgebra for the functor $Fam(\_)$, it is tempting to take a morphism between transition systems to be simply a coalgebra morphism. However this definition presupposes that we are given an equality relation between states. It is clear that we are far short of having a category[5] of transition systems. One reason to be interested in the notion of a simulation is that it gives us a way of comparing the structure of transition systems, yet its definition does not presuppose an equality relation between states. If there is a simulation from (A, Φ) to (B, Ψ) which is total, then in some sense the structure of transitions in (A, Φ) is preserved in that

[5]or even a '2−category'.

of (B, Ψ). It may be interesting (though I have not done so) to investigate the (large) relation between two transition systems defined by the presence of a total simulation from the first to the second.

3.2.7 Successor of a transition system

Given a transition system $(A, \lambda a.\, (B(a), \lambda b.\, a[b]))$ we can construct from it another transition system $(A', \lambda a.\, (B'(a), \lambda b.\, a[b]'))$ $A^\top$ which has a new element 'at the top', with a transition from it to each of the states in A. In a sense, this is analogous to the 'von Neumann' successor of A.

$$\begin{array}{lcl} A' & = & 1 + A \\ B'(a) & = & \textbf{case } a \textbf{ of} \\ & & \quad \mathrm{i}\ 0 \mapsto A \\ & & \quad \mathrm{j}\ a \mapsto B(a) \\ a[b]' & = & \textbf{case } a \textbf{ of} \\ & & \quad \mathrm{i}\ 0 \mapsto \mathrm{j}\ a \\ & & \quad \mathrm{j}\ b \mapsto \mathrm{j}\ a[b] \end{array}$$

There is an analogous construction which adds a new element 'at the bottom'.

$$\begin{array}{lcl} A' & = & 1 + A \\ B'(a) & = & \textbf{case } a \textbf{ of} \\ & & \quad \mathrm{i}\ 0 \mapsto \{\,\} \\ & & \quad \mathrm{j}\ a \mapsto 1 + B(a) \\ a[b]' & = & \textbf{case } a \textbf{ of} \\ & & \quad \mathrm{i}\ 0 \mapsto \textbf{case } b \textbf{ of} \quad - \\ & & \quad \mathrm{j}\ b \mapsto \textbf{case } b \textbf{ of} \ \ \mathrm{i}\ 0 \rightarrow \mathrm{i}\ 0 \\ & & \qquad\qquad\qquad\qquad\ \ \mathrm{j}\ b \rightarrow \mathrm{j}\ a[b] \end{array}$$

3.2.8 Ordered addition of two transition systems

Given two transition systems

$$\begin{array}{l} A_1 = (A_1, \lambda a.\, (B_1(a), \lambda b.\, a[b]_1)) \\ A_2 = (A_2, \lambda a.\, (B_2(a), \lambda b.\, a[b]_2)) \end{array}$$

we define an ordered binary sum $A_1 + A_2 = (A, \lambda a.\, (B(a), \lambda b.\, a[b]))$, corresponding to Cantor's definition of the ordered addition of two linear orders.

$$\begin{array}{lcl} A & = & A_1 + A_2 \\ B(a) & = & \textbf{case } a \textbf{ of} \\ & & \quad \mathrm{i}\ a_1 \mapsto B_1(a) \\ & & \quad \mathrm{j}\ a_2 \mapsto A_1 + B_2(a) \\ a[b] & = & \textbf{case } a \textbf{ of} \\ & & \quad \mathrm{i}\ a_1 \mapsto \mathrm{i}\ a_1[b]_1 \\ & & \quad \mathrm{j}\ a_2 \mapsto \textbf{case } b \textbf{ of} \\ & & \qquad\qquad \mathrm{i}\ a_1 \mapsto \mathrm{i}\ a_1 \\ & & \qquad\qquad \mathrm{j}\ b_2 \mapsto \mathrm{j}\ a_2[b_2] \end{array}$$

In this construction we put B 'above' A. A transition from an element of A is the same as a transition in A, while we add to the transitions from an element of B several new transitions, one for each element of A. Intuitively, it is clear that this construction preserves transitivity.

The addition construction preserves wellfoundedness. This means that if we are given a predicate X on $A+B$ which is progressive with respect to the transitions defined above, then X holds throughout $A+B$. So let X be such a predicate. Define from it the following predicates:

$$\begin{array}{lcl} X_1 & : & Pow(A_1) \\ X_2 & : & Pow(A_2) \\ X_1(a_1) & = & X(\mathrm{i}\ a_1) \\ X_2(a_2) & = & X(\mathrm{j}\ a_2) \times (\prod a : A_1\ )\ X_1(a) \end{array}$$

It can be checked that X_0 and X_1 are progressive in A_1 and A_2 respectively. If A_1 and A_2 are wellfounded, then this implies that X_0 holds throughout A_1, and X_1 throughout A_2. So X holds throughout $A_1 + A_2$.

3.2.9 Ordered sum of a family of transition systems

Suppose we are given a transition system[6]

$$X = (X_A, \lambda xa.\,(X_B(xa), \lambda xb.\,X_c(xa, xb)))$$

and for each $xa : X_A$ a transition system

$$Y(xa) = (\,Y_A(xa), \lambda ya.\,(\,Y_B(xa, ya), \lambda yb.\ Y_c(xa, ya, yb)))$$

(For example, the family may be constant, or be the family of segments of a given transition system.) Then there is another transition system

$$\begin{array}{l} Z = (\sum xa : X\ )\ Y(xa) \\ = (Z_A, \lambda za.\,(Z_B(za), \lambda zb.\,Z_c(za, zb))) \end{array}$$

which is the ordered sum of that family, the order being given by the transition system X.

$$\begin{array}{lcl} Z_A & = & (\sum xa : X_A\ )\ Y_A(xa) \\ Z_B(xa, ya) & = & Y_B(xa, ya)\ +\ (\sum xb : X_B(xa)\,)\ Y_A(X_c(xa, xb)) \\ Z_c((xa, ya), zb) & = & \textbf{case}\ zb\ \textbf{of} \\ & & \quad \begin{array}{ll} \mathrm{i}\ yb & \mapsto (xa,\, Y_c(xa, ya, yb)) \\ \mathrm{j}\ (yb, ya') & \mapsto (X_c(xa, xb), ya') \end{array} \end{array}$$

[6] I apologise for switching notation here, but for honesty's sake it seemed necessary to show explicitly that the destination of transitions in an indexed transition system depends also on the index.

By essentially the same argument as in 3.2.8, if X is wellfounded, and $Y(xa)$ is wellfounded for each $xa : X_A$, it follows that $(\sum xa : X_A)\ Y(xa)$ is wellfounded. (It does not however hold that if xa is accessible in X, and yb in $Y(xa)$, then (xa, yb) is accessible in $(\sum xa : X)\ Y(xa)$.)

In the case in which the family of transition systems does not depend on the element of X, the construction is the same as Cantor's definition of the lexicographic product $X \times Y$ (in which X takes precedence over Y).

A variety of other product operations which crop up in the theory of rewriting can be conveniently defined as operations on transition systems. For example, the following (symmetric) operation appears in the work of Stefan Kahrs. Given two transition systems

$$\begin{array}{l} A_1 = (A_1, \lambda a.\,(B_1(a), \lambda b.\ a[b]_1)) \\ A_2 = (A_2, \lambda a.\,(B_2(a), \lambda b.\ a[b]_2)) \end{array}$$

then $A = A_1 \otimes A2 = (A, \lambda a.\,(B(a), \lambda b.\ a[b]))$ is the following.

$$\begin{array}{lcl} A & = & A_1 \times A_2 \\ (a_1, a_2) & = & B_1(a_1) + B_2(a_2) + B_1(a_1) \times B_2(a_2) \\ (a_1, a_2)[b] & = & \textbf{case}\ b\ \textbf{of} \\ & & \quad \begin{array}{ll} \mathrm{i}\ b_1 & \mapsto (a_1[b_1], a_2) \\ \mathrm{j}\ b_2 & \mapsto (a_1, a_2[b_2]) \\ \mathrm{k}\ (b_1, b_2) & \mapsto (a_1[b_1], a_2[b_2]) \end{array} \end{array}$$

The transition structure $A_1 \otimes A_2$ contains transitions in which there is *simultaneously* a transition in A_1 and a transition in A_2.

3.2.10 The well ordering type

Let the set W be the so−called 'W−type' [67, pages 79–86] over a family of sets $(A : \mathrm{Set}, B : A \to \mathrm{Set})$. W may be considered to be the initial algebra for the functor $F(X) = (\sum a : A)\ B(a) \to X$ on the category of sets. Let it have the constructor $Sup : F(W) \to W$. We can put on W the following transition structure $\lambda wa.\,(W_B(wa), \lambda wb.\ W_c(wa, wc))$

$$\begin{array}{lcl} W_B(Sup(a, f)) & = & B(a) \\ Sup(a, f)[b] & = & f(b) \end{array}$$

It is wellfounded.

3.3 Interactive structures

Transition structures serve as models for situations in which an object moves from state to state atomically; that is, where the moves have no interesting or relevant substructure. It isn't really necessary to argue for the value of such models, as their applications are pervasive in computer science.

There are however situations of practical interest in which we would prefer a mathematical structure that can take account of structure within the transitions themselves. For example, there is a certain metaphor of 'clients' and 'servers' that is very prevalent in the computer industry, and the electronic marketplace which computer businesses are building. You (the customer) send a server a request, it does something (*e.g.* exchanges goods for money), and sends you back a reply. The model is familiar to anyone who knows how to buy things in shops, and so is an important paradigm for widespread electronic commerce. Here the customer or client has the initiative, and makes a request or issues an instruction 'spontaneously', in the sense that initiating a transaction can be blamed on them. The server or service has responsibility for concluding or performing the transaction. Not only is the client/server metaphor important merely to sell things to the general public, it is also very pervasive in systems programming as a design paradigm, when writing system software like operating systems, or their components. At the level of procedural code, the same bipolar structure is present even in calls to and returns from the humble subroutine.

This section contains the definition of a class of structures which I shall call 'interactive systems'. These are more intricate than transition systems in that a transition is no longer considered atomic, but to have two parts, which have a different 'polarity' reflecting the spontaneous nature of a client's request, and the reactive nature of a server's response. Because of this, we are able to model notions of relevance to the design of system software − such as freedom from deadlock.

The structures I shall describe appeared first in [84], where they arose in connection with modelling formal grammars and parse trees in type theory. There is a later description in [80], where it is pointed out that the trees identified by Petersson and Synek are in a strong sense a paradigm for a broad class of inductive definitions, in which what is defined is a family of sets over a given (fixed) index set, and there is mutual recursion across indices. The class consists of those which can be expressed as the least fixed points of predicate transformers in 'disjunctive normal form'.

What is new here is that I describe a rôle for these structures in modelling interactive systems, and spell out the significance in that application not only of wellfounded trees, but also their duals. I also show how some conventional program combinators (such as sequential composition) can be given a semantics in terms of predicate transformers induced by an interactive system.

An *interactive system* consists of the following data[7]:

$$\begin{array}{l} A : \mathrm{Set} \\ B(a : A) : \mathrm{Set} \\ C(a : A, b : B(a)) : \mathrm{Set} \\ d(a : A, b : B(a), c : C(a, b)) : A \end{array}$$

These components have the following names and interpretations.

- A set A. This is the set of *states* of the interactive system; the remaining components of the structure (which follow) comprise an *interactive structure* on that set of states.

- A predicate $B : Pow(A)$, which is to say a Set$-$valued function on A. If a is a state, then $B(a)$ can be thought of as the set of *instructions* b that it would be correct for the client to issue in that state. This set may be empty in some states, meaning that no further instructions may be issued.

- A relation $C : (\prod a : A\,)\ B(a) \to \mathrm{Set}$ between a state $a : A$ and a proof that B holds in that state. If a is a state and b is an instruction that it would be correct for a client to issue in that state, then $C(a, b)$ can be thought of as the set of *results* c that a server might correctly return when the instruction has been carried out. Carrying out the instruction is thought of as a state transition that happens at some instant between the instruction being issued and the result being returned. The set $C(a, b)$ may be empty for some states a and instructions b.

- A function $d(a, b, c) : A$ of variables $a : A, b : B(a), c : C(a, b)$ giving the *next state* to which the system moves. One might say that the function d gives the 'semantics' of the instructions and results, as it allows the next state to be computed from the current state, the instruction, and the result of its execution. I shall often use the notation

  $$a[b/c]$$

  instead of

  $$d(a, b, c)$$

  and resolve ambiguity where necessary by decorations of some kind, as in $a[b/c]^*$, $a[b/c]'$, $a[b/c]_1$, *etc.*. I shall call the pair b/c an *interaction*.

[7]I use the notation of Petersson and Synek. It may be objected that this notation is rather 'colourless', but in my opinion that is its advantage. The structure admits such a wide variety of interpretations that to base the notation on any one of them soon leads to incongruity. It is hopeless to fix the colour of something iridescent.

It is possible to give a more compact formulation of this notion, which may be useful. An interactive system is equivalent to a value of the following type:

$$(\sum S : \mathrm{Set})\; S \to \mathit{Fam}^2(S)$$

which we can describe categorically as a coalgebra for the functor $\mathit{Fam}^2(\_)$ (modulo concerns about equality). To the interactive system (A, B, C, d) there corresponds the coalgebra

$$(A, \lambda a.\, (B(a), \lambda b.\, (C(a, b), \lambda c.\, d(a, b, c))))$$

In practice, both representations are useful.

It is sometimes useful to generalise the notion of an interactive structure, so that the origin and destination states of an interaction can lie in a different state space. I shall call a structure

$$\Phi : A \to \mathit{Fam}^2(A')$$

an interactive structure *from* A to A'. In this guise an interactive structure is a morphism in the Kleisli category for the composite functor $\mathit{Fam}^2(\_)$.

It is also useful to have the notion of a *pointed* interactive system, which is an interactive system with a distinguished 'start' state.

Besides the client/server interpretation of interactive structures there are a number of other interpretations that are useful.

For example, one may obviously think of the states as positions in a certain kind of 2−player game, in which there is (corresponding to the client) a player called 'White' (who goes first), and (corresponding to the server) a player called 'Black' who responds to White's moves with countermoves. White's objective is to 'beat' Black, by bringing about a position in which there is a move for which Black has no countermove. Black's objective is to avoid being beaten, possibly (but not necessarily) by bringing about a position in which White has no moves.

Perhaps more interestingly, there is what I shall refer to as a 'rule−set' interpretation. Here the states $a : A$ correspond to unanalysed statements of some kind. Associated with statement a there is a set $B(a)$ of inference steps with conclusion a. (The inferences are not schematic, neither do they discharge hypotheses.) For each such inference step $b : B(a)$, there is a family $\{\ a[b/c] \mid c : C(a, b)\ \}$ of statements which make up the premises of the inference step. The notion of rule−set was introduced by Aczel in [4]. Aczel defines a rule set (for a given set S of statements) to be an element of $\mathit{Pow}(\mathit{Pow}(S) \times S)$.

He might equivalently have taken them to be elements of $S \to Pow(Pow(S))$, in view of the isomorphism

$$\begin{array}{llll} \text{between} & Pow(X \times Y) & \text{and} & Y \to Pow(X) \\ \text{which sends} & R \subseteq X \times Y & \text{to} & \lambda y.\, \{\ x : X \mid R(x,y)\ \} : Y \to Pow(X) \end{array}$$

As demonstrated in Dybjer's paper [28], rule sets play a fundamental rôle with respect to principles of inductive definition. He puts them to use in constructing realisability interpretations of systems of dependent types with strong universe principles. We obtain an 'isotope' of Aczel's notion by replacing $Pow(\_)$ with $Fam(\_)$.

3.3.1 Predicate transformers induced by an interactive structure

In the previous section I defined the notion of an interactive system $\Phi = (A, B, C, d)$. In essence, an interactive system associates with each state $a : A$ a doubly and dependently indexed family of states $\{\ a[b/c] \mid b : B(a), c : C(a,b)\ \}$. In this subsection, I fix an interactive system Φ from A_1 to A_2, and define two general families of monotone predicate transformers $P \mapsto \Phi^\circ(P)$ and $P \mapsto \Phi^\bullet(P)$ which arise in connection with this doubly indexed structure. Since Φ is fixed I shall write simply ${}^\circ P$ and ${}^\bullet P$. In a certain sense, Φ° gives a transformation in disjunctive normal form, and $\Phi^\bullet$ in conjunctive normal form. The definition follows.

$$\begin{array}{l} \lambda P.\, P^\circ, \lambda P.\, P^\bullet : Pow(A_2) \to Pow(A_1) \\ P^\circ(a) = (\sum x : B(a))\ (\prod y : C(a,x))\ P(a[x/y]) \\ P^\bullet(a) = (\prod x : B(a))\ (\sum y : C(a,x))\ P(a[x/y]) \end{array}$$

A predicate of the form ${}^\circ P$ holds of a state $a : A_1$ in which there is an instruction $b : B(a)$ that can be issued correctly and a function $f : (\prod y : C(a,x))\ P(a[x/y])$ applicable to values $y : C(a,x)$ that might be correctly returned when the instruction has been performed, yielding a proof that $P(a[x/y])$. In other words, if you know that ${}^\circ P$ holds in the current state, you have an instruction of which you know that it results only in states in which P holds. (There may be no such result states, in which case the instruction can be issued but not performed.)

A predicate of the form ${}^\bullet P$ holds of a state in which, for any instruction $x : B(a)$ that might be correctly issued, there is a result $y : C(a,x)$ and a next state $a[x/y]$ such that P holds. In other words, if you know that ${}^\bullet P$ holds in the current state, you have a way of correctly performing any instruction that might be issued, in such a way that P holds whatever the result state.

Both these predicate transformers are obviously monotone, since the predicate being transformed occurs only in a strictly positive position in its transform.

By application of the Knaster–Tarski theorem (for example as in [114], section 5.5.) to these monotone functions on the complete lattice $Pow(A)$, both predicate transformers have both least and greatest fixed points. Of these the greatest fixed point of $^\circ$ and the least fixed point of $^\bullet$ are degenerate. They are the predicates which are constantly true and constantly false, respectively. More interesting are the least fixed point of $^\circ$ which I shall call Bar, and the greatest fixed point of $^\bullet$ which I shall call Pos.

3.3.2 Bar, the least fixed point of $^\circ$

For Bar, we have the following formation and introduction rules:

$$\begin{array}{l} Bar : Pow(A) \\ bar : {}^\circ Bar \subseteq Bar \end{array}$$

which make the pair (Bar, bar) an algebra for the endofunctor $^\circ$.

For weak initiality, we require a recursion or fold combinator $BarRec$, satisfying the following typing and computational equation.

$$\begin{array}{l}
BarRec : (\prod G : (\prod a : A)\ Bar(a) \to \mathrm{Set}, \\
\qquad\qquad g : (\prod a : A, b : B(a), \\
\qquad\qquad\qquad f : (\prod c : C(a,b))\ Bar(a[b/c]), \\
\qquad\qquad\qquad f' : (\prod c : C(a,b))\ G(a[b/c], f(c))) \\
\qquad\qquad\quad \to\ G(a, bar(a,(b,f))), \\
\qquad\qquad a : A, \\
\qquad\qquad t : Bar(a)) \\
\qquad\quad \to\ G(a,t) \\
BarRec(G, g, a, t) = \\
\qquad \textbf{case}\ t\ \textbf{of}\ \ bar(a,(b,f)) \to g(a,b,f,\lambda c.\, BarRec(G,g,a[b/c],f(c)))
\end{array}$$

For brevity's sake, it is useful to adopt the convex 'lens'[8] bracket notation popular among functional programmers for 'catamorphisms', which are initial algebra morphisms. This notation is introduced in [9]. Using it, one writes $BarRec(G, g)$ in the form

$$(\!|G, g|\!) : (\prod a : A, t : Bar(a))\ G(a,t)$$

[8]Some time after settling on the 'lens' terminology for the concept introduced in the next chapter, which seems to me to be entirely appropriate, I discovered that it clashed with the (notationally inspired) functional programming terminology.

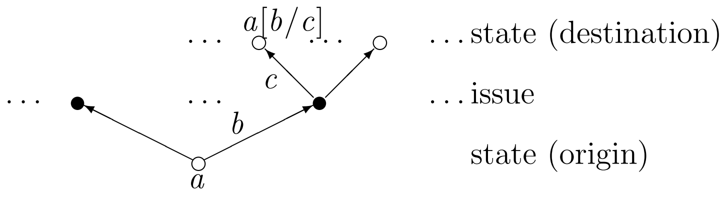

Figure 3.1: The local structure of an interactive system

It should be noted that the arguments

$$\begin{array}{l} b : B(a) \\ f : (\prod c : C(a,b)\,)\ Bar(a[b/c]) \\ f' : (\prod c : C(a,b)\,)\ G(a[b/c], f(c)) \end{array}$$

of the function g above can be expressed in the form

$$(b, \lambda c.\,(f(c), f'(c))) : {}^{\circ}(\lambda x.\,(\sum t : Bar(x)\,)\ G(x,t))(a)$$

so that if we allow ourselves the use of pattern−matching quantifiers, the type of g can be written more compactly as follows.

$$g : (\prod a : A, (b, \lambda c.\,(f(c), z(c))) : {}^{\circ}(\lambda x.\,(\sum t : Bar(x)\,)\ G(x,t))(a)\,)\ G(a, bar(a,(b,f)))$$

The interpretation of the predicate Bar is that if you have a proof of $Bar(a)$ for a given state a, then your proof can be used as a program by the client of an interactive system to drive the server inevitably into a deadlocked state. To follow this program, do as follows. When in state $a : A$, calculate the canonical form of the proof, which will be $bar(a,(b,f))$ where $b : B(a)$ is an instruction, and $f : (\prod c : C(a,b)\,)\ Bar(a[b/c])$ is a function applicable to results which the server may return. Issue the instruction b, and if a result $c : C(a,b)$ is forthcoming, then proceed according to the program $f(c)$.

A proof of $Bar(a)$ has the structure of a certain kind of wellfounded tree, with 'white' and 'black' nodes, as in the usual pictorial representation of games. A white node represents a state (or position in a game), and a black node the issue of an instruction. If you use the proof as a program, then subsequent interactions will trace a path up the tree, that inevitably terminates in a 'bad' black node.

3.3.3 Pos, the greatest fixed point of $^{\bullet}$

y For Pos, we have the following formation and co−introduction rules:

$$\begin{array}{l} Pos : Pow(A) \\ pos : Pos \subseteq {}^{\bullet}Pos \end{array}$$

which make the pair (Pos, pos) an coalgebra for the endofunctor $^\bullet$. It is convenient to express pos in the following form.

$$\begin{array}{l} pos = \lambda a, p.\, (\lambda b.\, \rho(a,p,b), \lambda b.\, \delta(a,p,b)) : Pos \subseteq {}^\bullet Pos \\ \textbf{where}\ \ \rho : (\prod a : A, p : Pos(a), b : B(a)\,)\ C(a,b) \\ \qquad\quad\ \ \delta : (\prod a : A, p : Pos(a), b : B(a)\,)\ Pos(a[b/\rho(a,p,b)]) \end{array}$$

The function ρ (which I shall call the *output* function) gives a result for any given input, like the output function of a Mealy machine [92], and the function δ (which I shall call the *dynamic* function) gives what can be thought of as the next internal configuration of the machine[9]. Perhaps one can call ρ and δ *extractors*.

For a *pos* to be a final coalgebra, we require a coiteration[10] or unfold combinator Pos_ν. For the sake of readability the equations that this combinator should satisfy are given in the following discussion.

$$\begin{array}{l} Pos_\nu : (\prod G : Pow(A), \\ \qquad\qquad g_\rho : (\prod a : A, x : G(a), b : B(a)\,)\ C(a,b), \\ \qquad\qquad g_\delta : (\prod a : A, x : G(a), b : B(a)\,)\ G(a[b/g_\rho(a,x,b)]), \\ \qquad\qquad a : A, \\ \qquad\qquad x : G(a)) \\ \qquad\quad \rightarrow\ Pos(a) \end{array}$$

I shall abbreviate g_ρ, g_δ to g. It is useful to adopt the concave 'lens' brackets notation from the 'Algebra of Programming' community (for 'anamorphisms', which are final coalgebra morphisms). Instead of $Pos_\nu(G, g)$ I shall write

$$[\![G, g]\!] : (\prod a : A, x : G(a)\,)\ Pos(a)$$

The equations that follow require that $[\![G, g]\!]$ is an coalgebra morphism from (G, g) to (Pos, pos), so that for each state a and proof x of $G(a)$, there is an element $[\![G, g]\!](a, x)$ of $Pos(a)$ which can be used as a program for a server to avoid deadlock when starting in state a.

The first equation requires that the output function of $[\![G, g]\!](a, x)$ is given by the output (g_ρ) part of the coalgebra.

$$\rho(a, [\![G, g]\!](a,x)) = g_\rho(a,x) : (\prod b : B(a)\,)\ C(a,b)$$

The second equation requires that the dynamic function of $[\![G, g]\!](a, x)$ is given by composition with the dynamic (g_δ) part of the coalgebra.

$$\begin{array}{l} \delta(a, [\![G, g]\!](a,x)) = \lambda b.\ [\![G, g]\!](a[b/g_\rho(a,x,b)], g_\delta(a,x,b)) \\ \qquad\qquad\qquad\qquad\quad : (\prod b : B(a)\,)\ Pos(a[b/g_\rho(a,x,b)]) \end{array}$$

[9] I would call it the next state function, but that is a different sense of state. Here we have states of the server, there we have states of the interactive system, in the sense of 'protocol'.

[10] Recent investigations by myself and Anton Setzer suggest that it would be preferable to have a true *corecursion* rather than coiteration combinator. The computational behaviour of coiteration is not fully satisfactory.

The interpretation of the predicate *Pos* is that if you have a proof p of $pos(a)$ for a given state a, then your proof can be used as a program for the server of an interactive system which starts in state a, as follows. Wait for an instruction b to be issued. Apply p to this instruction, and calculate the canonical form of this application, which will be a pair (c, p'). Return c as the result of the instruction, and then follow the program p'. Following this program will ensure that in all subsequent states, there is a way of correctly performing any instruction that is correctly issued, and so of avoiding deadlock.

This interpretation of *Pos*, as a family of sets of deadlock avoiding programs had occurred also to Martin−Löf, and to Thierry Coquand. I am grateful to them for pointing out to me the coinductive nature of *Pos*. I have adopted Martin−Löf's names *Bar* for the least fixed point of $^\circ$ and *Pos* for the greatest fixed point of $^\bullet$.

It should be pointed out that there are considerable challenges in providing a foundation for coinductive types and predicates in wellfounded type theory, at least in the absence of a general identity relation.

3.3.4 Two forms of sequential composition

In this section we define two kinds of sequential composition.

Suppose we are given two interactive structures as follows.

$$\begin{array}{lcl}
\Phi_1 & : & A_1 \to \mathit{Fam}^2(A_2) \\
\Phi_2 & : & A_2 \to \mathit{Fam}^2(A_3) \\
\Phi_1 & = & \lambda a.\,(B_1(a), \lambda b.\,(C_1(a,b), \lambda c.\, a[b/c])) \\
\Phi_2 & = & \lambda a.\,(B_2(a), \lambda b.\,(C_2(a,b), \lambda c.\, a[b/c]))
\end{array}$$

Then we can find $\Phi = (\Phi_1 \mathbin{\hat{\circ}} \Phi_2) : A_1 \to \mathit{Fam}^2(A_3)$ such that

$$(\Phi_1 \mathbin{\hat{\circ}} \Phi_2)^\circ = (\Phi_1)^\circ \circ (\Phi_2)^\circ : \mathit{Pow}(A_3) \to \mathit{Pow}(A_1)$$

The required components of Φ are

$$\begin{array}{lcl}
B(a) & = & (\Phi_1)^\circ(B_2, a) \\
C(a, (b_1, \lambda c_1.\, b_2(c_1))) & = & (\sum c_1 : C_1(s, b_1)\,)\; C_2(a[b_1/c_1], b_2(c_1)) \\
a[(b_1, \lambda c_1.\, b_2(c_1))/(c_1, c_2)] & = & a[b_1/c_1][b_2(c_1)/c_2]
\end{array}$$

A dual construction $\hat{\bullet}$ can be given, such that

$$(\Phi_1 \mathbin{\hat{\bullet}} \Phi_2)^\bullet = (\Phi_1)^\bullet \circ (\Phi_2)^\bullet : \mathit{Pow}(A_3) \to \mathit{Pow}(A_1)$$

In this case, the components of the rule structure $^\bullet$−representing composition are as follows.

$$\begin{array}{lcl} B(a) & = & (\Phi_1)^\bullet(B_2, a) \\ C(a, \lambda b_1. (c_1(b_1), b_2(b_1))) & = & (\sum b_1 : B_1(a)\,)\ C_2(a[b_1/b_2(b_1)], b_2(b_1)) \\ a[\lambda b_1. (c_1(b_1), b_2(b_1))/(b_1, c_2)] & = & a[b_1/c_1(b_1)][b_2(b_1)/c_2] \end{array}$$

As a representation of the identity predicate transformer on a state space A we may take the following interactive system **skip**:

$$\begin{array}{lcl} B(a) & = & N_1, \\ C(a, n_0) & = & N_1, \\ a[n_0/n_0] & = & a \end{array}$$

We have $\mathbf{skip}^\circ(X) = \mathbf{skip}^\bullet(X) = X$, so that under either interpretation (by $^\circ$ or $^\bullet$) **skip** represents the identity predicate transformer. It serves therefore as a unit for both forms of sequential composition.

3.3.5 Two forms of alternative composition

We now consider ways of combining interactive structures in which one or other of the parties gets to choose between the two structures. Here I take the point of view of the client, and so refer to the combination in which it is the client who gets to choose as 'angelic', and that in which the server gets to choose as 'demonic'.

Suppose that we have two interactive structures typed as follows.

$$\begin{array}{lcl} \Phi_1 & : & A_1 \to \mathit{Fam}^2(A_2) \\ \Phi_2 & : & A_1 \to \mathit{Fam}^2(A_2) \\ \Phi_1 & = & \lambda a. (B_1(a), \lambda b. (C_1(a, b), \lambda c. a[b/c])) \\ \Phi_2 & = & \lambda a. (B_2(a), \lambda b. (C_2(a, b), \lambda c. a[b/c])) \end{array}$$

The 'angelic' choice $\Phi = (\Phi_1 \sqcup \Phi_2)$ is defined as follows.

$$\begin{array}{lcl} B(a) & = & B_1(s) + B_2(s), \\ C(a, b) & = & \textbf{case } b \textbf{ of } \ \mathrm{i}\ b_1 \to C_1(a, b_1) \\ & & \qquad\qquad\ \ \mathrm{j}\ b_2 \to C_2(a, b_2), \\ a[b/c] & = & \textbf{case } b \textbf{ of } \ \mathrm{i}\ b_1 \to a[b_1/c] \\ & & \qquad\qquad\ \ \mathrm{j}\ b_2 \to a[b_2/c] \end{array}$$

In the client/server metaphor, the client gets two instruction sets from which to choose. The server has to be able to respond in whichever is chosen. In the game theoretical metaphor, the player 'White' (who goes first) chooses in which of the two games the next interaction will occurs.

It is easily seen that

$$\begin{array}{rcl} (\Phi_1 \sqcup \Phi_2)^\circ(X) & = & \Phi_1^\circ(X) \cup \Phi_2^\circ(X), \\ (\Phi_1 \sqcup \Phi_2)^\bullet(X) & = & \Phi_1^\bullet(X) \cap \Phi_2^\bullet(X) \ . \end{array}$$

Clearly this binary construction can be generalised to take the join of a small family of interactive structures. In the degenerate case of an empty family, we have the interactive structure called **abort** (by for example Back and von Wright in [8]) which has

$$B(s) = N_0$$

for which

$$\begin{array}{rcl} \textbf{abort}^\circ(X) & = & \mathit{False} \\ \textbf{abort}^\bullet(X) & = & \mathit{True} \end{array}$$

Dually, we may consider a game built 'demonically' from two games, in which the player who goes first offers a choice of two moves to his opponent. Let us call this $\Phi = (\Phi_1 \sqcap \Phi_2)$.

$$\begin{array}{lcl} B(a) & = & B_1(a) \times B_2(a), \\ C(a,(b_1,b_2)) & = & C_1(a,b_1) + C_2(a,b_2), \\ a[(b_1,b_2)/c] & = & \textbf{case } c \textbf{ of } \ \mathrm{i}\ c_1 \rightarrow a[b_1/c_1] \\ & & \qquad\qquad\quad \mathrm{j}\ c_2 \rightarrow a[b_2/c_2] \end{array}$$

It is easy to see that:

$$\begin{array}{rcl} (\Phi_1 \sqcap \Phi_2)^\circ(X) & = & (\Phi_1)^\circ(X) \cap (\Phi_2)^\circ(X) \\ (\Phi_1 \sqcap \Phi_2)^\bullet(X) & = & (\Phi_1)^\bullet(X) \cup (\Phi_2)^\bullet(X) \end{array}$$

Again this binary construction can be generalised to take the meet of a small family of interactive structures. In the degenerate case of an empty family we have the structure called **magic** by Back and von Wright, defined as follows.

$$\begin{array}{rcl} B(a) & = & N_1, \\ C(a,n_0) & = & N_0 \end{array}$$

There is some arbitrariness here, in that any inhabited set can be taken as the set of instructions available to the client. The server can make no response to any of instructions.

$$\begin{array}{rcl} \textbf{magic}^\circ(X) & = & \mathit{True}, \\ \textbf{magic}^\bullet(X) & = & \mathit{False} \end{array}$$

3.3.6 Some basic interactive structures and spaces

In [8] Back and von Wright introduced a notion of contract regulating the behaviour of independent agents. They gave a certain algebra of 'contracts', called refinement calculus. This calculus has its origins in the stepwise refinement method for program construction advocated by Dijkstra [19] and Wirth [115]. They interpret contracts as predicate transformers. These are awkward to handle directly in the predicative framework of dependent type theory. My idea is to interpret them instead using the interactive structures.

In this subsection, we define interactive systems that correspond to the basic contract statements of Back and von Wright [8] (especially section 11.3), namely *functional update*, *assertion*, and *assumption*.

3.3.6.1 Functional updating

If $f : A \to A$ is a state transformer then we can update the state deterministically with a command **update**(f), for which Back and von Wright use the notation $\langle f \rangle$.

$$\textbf{update}(f): \qquad B(a) \;=\; N_1, \qquad C(a, n_0) \;=\; N_1, \qquad a[n_0/n_0] \;=\; f(a)$$

In particular **skip** $=$ **update**$(\lambda a.\, a)$.

3.3.6.2 Assertion

More generally, for F a state predicate, we have the assertion command **assert**(F), for which Back and von Wright use the notation $\{ F \}$. This can be issued only when F holds, and its execution is a mere acknowledgement, leaving the state undisturbed.

$$\textbf{assert}(F): \qquad B(a) \;=\; F(a), \qquad C(a, f) \;=\; N_1 \qquad a[f/n_0] \;=\; a$$

Then

$$\begin{aligned} \textbf{assert}(F)^{\circ}(X, a) &= F(a) \times X(a) \\ \textbf{assert}(F)^{\bullet}(X, a) &= F(a) \to X(a) \end{aligned}$$

In particular, **abort** $=$ **assert**(False).

3.3.6.3 Assumption

If F is a state predicate, we have the 'assume' command **assume**(F). Considered as an instruction, this can always be issued, but can be executed only when F

holds, leaving the state undisturbed. Back and von Wright use the notation $[\,F\,]$ for $\mathbf{assert}(F)$.

$$\mathbf{assume}(F): \qquad B(a) = N_1, \qquad C(a, n_0) = F(a), \qquad a[n_0/f] = a$$

Then

$$\begin{array}{rcl} \mathbf{assume}(F)^{\circ}(X,a) & = & F(a) \to X(a) \\ \mathbf{assume}(F)^{\bullet}(X,a) & = & F(a) \times X(a) \end{array}$$

In particular, **magic** = $\mathbf{assume}(\mathit{False})$.

3.3.7 Loops and recursion

From the previous subsections, we may suspect that there are operations on interactive structures that correspond to many of the commands of Back and von Wright's refinement calculus. It is natural therefore to look for a construction that corresponds to the formation of recursive commands. It turns out that there is, and that the construction is a isotope of the inductive definition of 'general trees' by Petersson and Synek, given in [84]. (We add "leaves" to the trees.) This plays a central rôle in the analysis of the "programmer's predicament" below.

Given a monotone predicate transformer $\theta : \mathit{Pow}(A) \to \mathit{Pow}(A)$, we define its *closure* $\theta^* : \mathit{Pow}(A) \to \mathit{Pow}(A)$ inductively to be the least solution ϕ of the following equation.

$$\begin{array}{rcl} \phi & : & \mathit{Pow}(A) \to \mathit{Pow}(A) \\ \phi(X) & = & X \cup \theta(\phi(X)) \end{array}$$

It is easily seen that θ^* is a closure operator on $\mathit{Pow}(A)$, in the sense that for arbitrary $X; \mathit{Pow}(A)$, we have $X \subseteq \theta^* X$ (so θ^* is inflationary) and $(\theta^*)^2 X \subseteq \theta^* X$ (so θ^* is idempotent).

Suppose now that $\Phi = \lambda a.\,(B(a), \lambda c.\,(C(a,b), \lambda p.\, a[b/c]))$ is an interactive structure on a set S. For any Φ, we can find

$$\begin{array}{l} \Phi^{\infty} = \lambda a.\,(B^{\infty}(a), \lambda bt.\,(C^{\infty}(a, bt), \lambda cs.\, a[bt/cs]^{\infty})) \\ \Phi^{\bullet\bullet} = \lambda a.\,(B^{\bullet\bullet}(a), \lambda tb.\,(C^{\bullet\bullet}(a, tb), \lambda sc.\, a[tb/sc]^{\bullet\bullet})) \end{array}$$

such that

$$\begin{array}{rcl} (\Phi^{\infty})^{\circ} & = & (\Phi^{\circ})^{*} \\ (\Phi^{\bullet\bullet})^{\circ} & = & (\Phi^{\bullet})^{*} \quad . \end{array}$$

The construction $\_^{\infty}$ is defined as follows.

$$\begin{array}{lcl} B^{\infty} & = & (\Phi^{\circ})^{*}(\mathit{True}) \\ C^{\infty}(a, \mathrm{i}\ n_0) & = & N_1 \\ C^{\infty}(a, \mathrm{j}\ (b, \lambda c.\, bt(c))) & = & (\sum c : C(a,b)\,)\ C^{\infty}(a[b/c], bt(c)) \\ a[\mathrm{i}\ n_0/n_0]^{\infty} & = & s \\ a[\mathrm{j}(b, \lambda c.\, bt(c))/(c, cs)]^{\infty} & = & a[b/c][bt(c)/cs] \end{array}$$

The equations specifying B^∞ and C^∞ should be understood as defining B^∞ and C^∞ to be their *least* solutions.

The elements of $B^\infty(a)$ can be thought of as programs in the same sense as proofs of $Bar(a)$, except that they may contain certain 'escape points', or special instructions $\mathrm{i}\ n_0$ which allow the server to evade deadlock. They *need* not contain any such special instructions, and so all proofs of $Bar(a)$ are inhabitants of $B^\infty(a)$. If $bt : B^\infty(a)$, the elements of $C^\infty(a, bt)$ can be thought of as 'escape routes', by which I mean sequences of server responses which lead to an escape point of bt. If $bt : B^\infty(a)$ and $cs : C^\infty(a, bt)$, then $a[bt/cs]^\infty$ can be thought of as the state obtaining at the end of the escape route cs.

The construction $\_^{\bullet\bullet}$ seems to be only a curiosity[11], in view of the mixture of white and black bullets which occur in the property above. It is defined as follows.

$$
\begin{array}{lcl}
B^{\bullet\bullet} & = & (\Phi^\bullet)^*(\mathit{True}) \\
C^{\bullet\bullet}(a, \mathrm{i}\ n_0) & = & N_1 \\
C^{\bullet\bullet}(a, \mathrm{j}\ (\lambda b.\,(c(b), tb(b)))) & = & (\sum b : B(a)\,)\ C^{\bullet\bullet}(a[b/c(b)], tb(b)) \\
a[\mathrm{i}\ n_0/n_0]^{\bullet\bullet} & = & a \\
a[\mathrm{j}\ (\lambda b.\,(c(b), tb(b)))/(b, bs)]^{\bullet\bullet} & = & a[b/c(b)][tb(b)/bs]
\end{array}
$$

3.3.8 The programmer's predicament

Suppose we are given two interactive structures as follows.

$$
\begin{array}{lcl}
\Phi_1 & : & A_1 \to \mathit{Fam}^2(A_1) \\
\Phi_2 & : & A_2 \to \mathit{Fam}^2(A_2) \\
\Phi_1 & = & \lambda a.\,(B_1(a), \lambda c.\,(C_1(a, b), \lambda c.\ a[b/c])) \\
\Phi_2 & = & \lambda a.\,(B_2(s), \lambda c.\,(C_2(a, b), \lambda c.\ a[b/c]))
\end{array}
$$

The two predicate transformers ${\Phi_1}^\bullet$ and ${\Phi_2}^\circ$ can be made to act jointly on the argument places of a binary relation $X : A_1 \to Pow(A_2)$ in essentially two interesting ways, giving rise to two relation transformers which I write $(\Phi_1, \Phi_2)^{\bullet\circ}$, and $(\Phi_1, \Phi_2)^{\circ\bullet}$. The definitions are as follows:

$$
\begin{array}{lcl}
\Phi^{\bullet\circ}, \Phi^{\circ\bullet} & : & (A_1 \to Pow(A_2)) \to A_1 \to Pow(A_2) \\
(\Phi_1, \Phi_2)^{\bullet\circ}(X, a_1, a_2) & = & (\prod b_1 : B_1(a_1)\,)\ (\sum b_2 : B_2(a_2)\,) \\
 & & (\prod c_2 : C_2(a_2, b_2)\,)\ (\sum c_1 : C_1(a_1, b_1)\,) \\
 & & X(a_1[b_1/c_1], a_2[b_2/c_2]) \\
(\Phi_1, \Phi_2)^{\circ\bullet}(X, a_1, a_2) & = & (\sum b_1 : B_1(a_1)\,)\ (\prod b_2 : B_2(a_2)\,) \\
 & & (\sum c_2 : C_2(a_2, b_2)\,)\ (\prod c_1 : P_1(a_1, b_1)\,) \\
 & & X(a_1[b_1/c_1], a_2[b_2/c_2])
\end{array}
$$

[11] In fact it is detritus surviving from efforts to check the delusion that $(\Phi^{\bullet\bullet})^\bullet = (\Phi^\bullet)^*$.

It is obvious that both relation transformers $\Phi^{\bullet\circ}$ and $\Phi^{\circ\bullet}$ determined by a rule structure Φ are monotone, as the input relation occurs only positively in their defining expressions.

The interpretation of a relation $(\Phi_1, \Phi_2)^{\bullet\circ}(X)$ is that for any instruction that might be issued in Φ_1, there is an instruction which can be issued in Φ_2, and a way of mapping responses to that instruction back to responses to the original instruction, so that the relation obtains between the resulting states in the two systems. By analogy with the case of transition systems, I call a relation $X : A_1 \to Pow(A_2)$ a *simulation* from Φ_1 to Φ_2 if it satisfies the inclusion $X \subseteq (\Phi_1, \Phi_2)^{\bullet\circ}(X)$.

The interpretation of a relation $(\Phi_1, \Phi_2)^{\circ\bullet}(X)$ is that there exists an instruction that may be issued in Φ_1, and a way of responding to any instruction which might be issued in Φ_2, such that for any response to the original instruction the relation obtains between the resulting states in the two systems. I call a relation $X : A_1 \to Pow(A_2)$ a *anti–simulation* from Φ_1 to Φ_2 if it satisfies the inclusion $(\Phi_1, \Phi_2)^{\circ\bullet}(X) \subseteq X$.

The predicament of a programmer is usually as follows: they have to write a program (a collection of procedures) to present at one interface a certain 'abstract' behaviour, using only lower level or 'concrete' resources made available at a second interface by the efforts of other programmers. The 'value' of such a program is a certain increase in abstraction. In this section I suggest a way of modelling such programs mathematically.

In the setting of interactive systems, we need a relation $a_{abs} \preceq a_{conc}$ (a_{abs} is within a_{conc}) between the states of two interactive systems. The work of the programmer (if it is successful) results in something that witnesses this relation. A proof that $a_{abs} \preceq a_{conc}$ can be thought of as an implementation of the abstract state a_{abs} using the concrete state a_{conc}. The analysis I suggest is that the relation $\preceq$ should be a simulation from Φ_{abs} (the abstract interactive system) to ${\Phi_{conc}}^{\infty}$ (the closure of the concrete interactive system).

$$
\begin{array}{l}
(\prod a_{abs} : A_{abs}, a_{conc} : A_{conc}\,)\ a_{abs} \preceq a_{conc} \to \\
\qquad\qquad (\prod b_{abs} : B_{abs}(a_{abs})\,)\,(\sum b_{conc} : {B_{conc}}^{\infty}(a_{conc})\,) \\
\qquad\qquad (\prod c_{conc} : {C_{conc}}^{\infty}(a_{conc}, b_{conc})\,)\,(\sum c_{abs} : C_{abs}(a_{abs}, b_{abs})\,) \\
\qquad\qquad a_{abs}[b_{abs}/c_{abs}] \preceq a_{conc}[b_{conc}/c_{conc}]
\end{array}
$$

In general, an element of a set $a_{abs} \preceq a_{conc}$ (a 'piece of work') has the form

$$\lambda b_{abs}.\,(t(b_{abs}), \lambda c_{conc}.\,(l(b_{abs}, c_{conc}), m(b_{abs}, c_{conc})))$$

where

$$
\begin{array}{lcl}
t & : & B_{abs}(a_{abs}) \to {B_{conc}}^{\infty}(a_{conc}) \\
l & : & (\prod b_{abs} : B_{abs}(a_{abs})\,) \\
 & & \qquad {C_{conc}}^{\infty}(a_{conc}, t(b_{abs})) \to C_{abs}(a_{abs}, b_{abs}) \\
m & : & (\prod b_{abs} : B_{abs}(a_{abs}), c_{conc} : {C_{conc}}^{\infty}(a_{conc}, t(b_{abs}))\,) \\
 & & \qquad a_{abs}[b_{abs}/l(b_{abs}, c_{conc})] \preceq a_{conc}[t(b_{abs})/c_{conc}]
\end{array}
$$

Here the component t is a mapping from abstract instructions $b_{abs} : B_{abs}(a_{abs})$ to concrete programs $t(b_{abs}) : {B_{conc}}^{\infty}(a_{conc})$ (*i.e.* instructions in the closure of the concrete interactive system); the component l is a mapping from a trace $p : {C_{conc}}^{\infty}(a_{conc}, t(b_{abs}))$ of the results obtained by running such a concrete program to a response to the original abstract instruction; and m maps the two two inputs (the abstract instruction b_{abs} and the concrete trace c_{conc}) to a proof that the new abstract state $a_{abs}[b_{abs}/l(b_{abs}, c_{conc})]$ is within the new concrete state $a_{conc}[t(b_{abs})/c_{conc}]$.

All models are wrong, in that a model (say, of parabolic motion for projectile) is inevitably a compromise between mathematical tractability (the theory of conic sections) and sordid reality (friction, turbulence, flocks of passing birds). My suggestion for modelling program components is wrong, for example because in reality a programmer is very often concerned with modules which respond to 'abstract' requests by issuing several requests in parallel to more than one 'concrete' resource, while contriving to maintain the appearance of sequential behaviour at its abstract interface. These and other issues of practical importance are simply not addressed. Worse, it isn't clear that the model is mathematically tractable; on the contrary this issue has to be left hanging in the air, awaiting a satisfactory account of coinduction in type theory. For one wants to define the abstraction relation $\preceq$ as the *greatest* post−fixed point of the relation transformer $\_^{\bullet\bullet}$, and then show that the relation is at least a partial order (preferably, a category of some kind) and moreover equipped with enough structure to make it a convenient framework for calculating with and reasoning about implementations which witness the abstraction relation.

As a converse issue, the programmer's predicament is in fact very often (or even mostly) that they have to demonstrate *bugs* in a module, rather than implement new modules. To find bugs in a module, one writes a test harness that provides an environment for it (both abstract and concrete) that drives it inevitably into a situation where the module either exhibits incorrect behaviour, or fails to respond when it should. To model this kind of task, it may be useful to consider the dual relation transformer $\_^{\infty}$, and its *least* pre−fixed points. An advantage here is that one is not concerned with coinduction, but induction, which is better understood in a type theoretical framework. On the other hand, in contrast with the abstraction relation, which one would hope to realise as (something like) the hom−set relation of a category, with bugs (or programs which cause them to be exhibited) it isn't so clear what structure to aim for – if there were such a structure, it would probably be called an *anti*−category.

3.3.8.1 Composition of implementations

In the previous subsection I suggested an approach to defining a relation of abstraction between interactive structures, or more precisely between *pointed* interactive structures. I pointed out that the suggestion suffers from the problem that there is (as far as I know) no adequately developed theory of coinduction in a type−theoretical context which would allow investigation of its mathematical properties to begin. If one puts this problem on one side, in the cheerful expectation that sooner or later we will understand how to deal with coinduction, there are nevertheless reasons to expect that this relation is at least a partial order. I indicate those reasons in this subsection.

As for reflexivity, surely the proof 1_a that $a \preceq a$, where a is a state in an interactive system would satisfy the equation $1_a = \lambda b.\,(b, \lambda c.\,(c, 1_{a[b/c]}))$. We cannot however take this to be a definition in wellfounded type theory. Operationally, the proof copies a request from its abstract interface straight through to its concrete interface, and copies the response from the concrete interface straight through back to the abstract interface. This is sometimes known as the 'copy−cat' or 'tit−for−tat' strategy in categories of games, as in [1][Example 2.2].

To prove transitivity of the relation $\preceq$ would amount to showing that if we plug two modules together, so that the lower−level interface of the first is identical with the higher−level interface of the second as indicated in the picture in figure 3.2 on the next page then the assembly of the two can be defined as a single module. It is difficult to check this in detail; nevertheless it is plausible, as I hope the following intuitive reasoning makes plain.

If we have a module that implements one interface α using another μ, then we have a function which maps each instruction i that might be issued over the abstract interface α to a wellfounded tree t_i, or 'subroutine' to be run over the concrete interface μ. Each node in such a tree t_i carries an instruction for the concrete interface μ, and for each result which might be returned for that instruction, there is a branch leaving the node. The tree t_i may also contain exit points, which play the rôle of instructions which cause control to return from the subroutine represented by t_i. Now if the concrete interface μ is in its turn

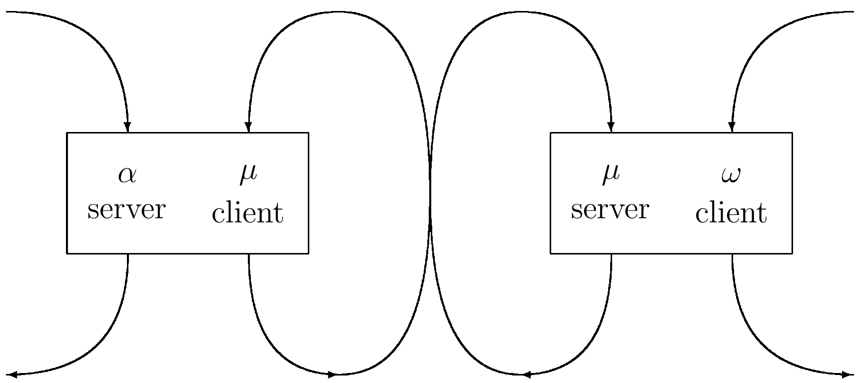

Figure 3.2: Plugging two modules together.

implemented with a yet lower−level interface ω, we have for each *node* in the first tree t_i carrying an instruction j, another *tree* t_j whose exit points are mapped to indices for the immediate subtrees of the node. So part of what we have to show is that given a tree t_i, and a function from the instructions which occur in it to lower−level trees we can by recursion on trees define a 'great big tree', in which the instructions occurring in the first tree have been 'macro'−expanded (or 'inlined', to use programming jargon) into entire trees, grafted together in the appropriate way. We also have to show that the paths which lead to the exit points in such a great big tree are concatenations of paths in the constituent trees, and so can be mapped back, or to responses for the original abstract instructions.

I ruefully acknowledge that the last paragraph is rank hand−waving. A great deal of work is required to substantiate the intuition of 'inlining' with which it conjures. A severe problem is to strive for a more abstract and more compact description which would allow the construction to be written down rigorously in a manageable form. For all that, I suggest it is plausible that it can be carried through – *modulo* the overarching problem of accounting for the coinductive definition required to 'tie the final knot'.

At the heart of the construction is a certain operation of grafting a forest onto the exit points of a given tree 'in the appropriate way'. This operation is relatively straightforward to define. Suppose that $\Phi = \lambda a.\,(B(s), \lambda b.\,(C(a,b), \lambda c.\, a[b/c]))$ is an interactive structure on a set A. Then we define the grafting operation, together with an associated concatenation operation on paths as follows.

$$
\begin{array}{l}
\mathit{graft} : (\prod\ a : A, bt : B^{\infty}(a) \\
\qquad\qquad ((\prod cs : C^{\infty}(a, bt))\ B^{\infty}(a[bt/cs]^{\infty}))) \to B^{\infty}(a) \\
\mathit{concat} : (\prod\ a : A, bt : B^{\infty}(a), \\
\qquad\qquad f : (\prod cs : C^{\infty}(a, bt))\ B^{\infty}(a[bt/cs]^{\infty}), \\
\qquad\qquad cs : C^{\infty}(a, bt)) \\
\qquad\qquad\qquad C^{\infty}(a[bt/cs]^{\infty}, f(cs)) \\
\qquad\qquad\qquad \to C^{\infty}(\mathit{graft}(a, bt, cs, f)) \\
\mathit{graft}(a, bt, f) \\
\quad = \textbf{case}\ bt\ \textbf{of} \\
\qquad \mathrm{i}\ n_0 \qquad\qquad\quad \mapsto f(n_0) \\
\qquad \mathrm{j}\ (b, \lambda c.\ bt(c)) \mapsto \mathrm{j}\ (b, \lambda c.\ \mathit{graft}(a[b/c], bt(c), \lambda cs. f((c, cs)))) \\
\mathit{concat}(a, bt, f, cs, \mathit{cstail}) \\
\quad = \textbf{case}\ bt\ \textbf{of} \\
\qquad \mathrm{i}\ n_0 \qquad\qquad\quad \mapsto \mathit{cstail} \\
\qquad \mathrm{j}\ (b, \lambda c.\ bt(c)) \mapsto \textbf{case}\ cs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad (c, cs') \mapsto (c, \mathit{concat}(a[b/c], bt(c), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda bt'. f(b, bt'), cs', \mathit{cstail}))
\end{array}
$$

3.4 Concluding remarks

In this chapter I have shown one approach to formalising the notions of transition system and interactive system in type theory, together with a number of illustrative constructions. Among these were counterparts of some of Cantor's arithmetic operations on linear orders (in the case of transition systems), and counterparts of some of Back and von Wright's operations on contracts expressed in their refinement calculus. Besides the intrinsic interest of these constructions, I wanted to show how much could be done without a notion of equality on the states of these systems, if only one is prepared to consider unconventional representations of seemingly 'obvious' notions such as that of a binary relation on a set, or of a subset of a set. In the course of this, I have run into (and to some extent, exposed) what appear to be quite challenging problems, centering around the foundations of coinductive definitions and corecursion in dependent type theory.

It is remarkable how fully the basic notions exploit the circumstance of working in a *dependent* type theory. The basic structure of an interactive system can be written as follows.

$$
\begin{array}{lcl}
A & : & \mathrm{Set}, \\
B & : & A \to \mathrm{Set}, \\
C & : & (\prod a : A)\ B(a) \to \mathrm{Set}, \\
d & : & (\prod a : A, b : B(a))\ C(a, b) \to A
\end{array}
$$

$a : A$	$b : B(a)$	$c : C(a, b)$	$a[b/c]$
sort	constructor	selector	component sort
statement	inference	premise	premise statement
neighbourhood	partition	part	new neighbourhood
game	attack	defence	new state
interrogation	question	answer	new state
interface	call	return	new state
universe	observation	reading	new state
knowledge	experience	result	new state
dialogue	thesis	antithesis	synthesis

Figure 3.3: Applications of interactive systems

One might well arrive at such a structure merely by reflecting on the general syntactical form of a dependent context, in advance of considering what its applications might be. It is to my mind astonishing how rich the notion turns out to be. The table in figure 3.3 on this page illustrates some possible interpretations. We might well then ask what emerges if we are content with a simpler example of a dependent context, such as the following.

$$\begin{array}{lcl} A & : & \mathrm{Set}, \\ B & : & A \to \mathrm{Set}, \\ c & : & (\prod a : A\,)\; B(a) \to A \end{array}$$

This is nothing but a transition system. Pressing still further in the direction of simplicity, one arrives at the structure

$$\begin{array}{lcl} A & : & \mathrm{Set}, \\ b & : & A \to A \end{array}$$

This structure is called a *dynamical system*, in Lawvere and Schanuel's book [58]. This trinity of 'systems' can be formulated in another way as coalgebras for the first three iterates of the functor $Fam(\_)$, as follows.

a dynamical system

$$(\textstyle\sum A : \mathrm{Set}\,)\; A \to A$$

A typical element has the form $(A, \lambda a.\, b(a))$.

a transition system

$$(\textstyle\sum A : \mathrm{Set}\,)\; A \to \underbrace{(\textstyle\sum B : \mathrm{Set}\,)\; B \to A}_{Fam(B)}$$

A typical element has the form $(A, \lambda a.\, (B(a), \lambda b.\, c(a, b)))$.

an interactive system

$$(\sum A : \mathrm{Set}\,)\ A \to \underbrace{(\sum B : \mathrm{Set}\,)\ B \to \underbrace{(\sum C : \mathrm{Set}\,)\ C \to A}_{Fam(A)}}_{Fam(Fam(A)))}$$

A typical element has the form $(A, \lambda a.\,(B(a), \lambda b.\,(C(a,b), \lambda c.\ d(a,b,c))))$.

It is natural to ask what emerges if one pushes further in the direction of more complex structures. Although it is of course possible to follow the general pattern further, it seems that the structures that lie beyond have no real meaning.

A striking similar trinity of structures arises from an entirely different perspective in a paper by Claire Martin, Oege de Moor, and Paul Gardiner [34]. They aimed to understand the connection between two frameworks for program semantics, namely the categorical algebra of relations advocated for example in Bird and de Moor's book [9], and the use of predicate transformers introduced by Dijkstra [23]. They arrived at the following triple of *preorder−enriched* categories. (A preorder enriched category $(\mathcal{C}, \sqsubseteq)$ is a category with a preorder $\sqsubseteq$ defined on homsets, with respect to which the categorical composition is monotonic in both arguments.)

1. The category $(Set, =)$ of sets and total functions. The objects are sets, and the morphisms from A to B are total functions $f : A \to B$. In this case, the preorder is trivial.

2. The category $(Rel, \subseteq)$ of relations, whose objects are sets, and in which the morphisms from A to B are binary relations $R \subseteq A \times B$. Morphisms are partially ordered by inclusion.

3. The category $(Pow, \sqsubseteq)$ of monotonic predicate transformers. In this the objects are powersets $Pow(A)$ where A is a set, and the morphisms are monotonic predicate transformers, ordered pointwise, so that if $p, q : Pow(A) \to Pow(B)$, then $p \sqsubseteq q$ is defined to mean that $p\ X \subseteq q\ X$ for all $X \subseteq A$.

In their terms, the structures of concern in this chapter have been endomaps in these three categories, or rather data structures which represent these endomaps (in the last case *via* the construction $\_^{\circ}$). They discovered an elegant systematic connection between these three categories, which one can tabulate as follows.

$$\begin{array}{lcl} Set = Map(Rel) & ; & Rel = Span(Set) \\ Rel = Map(Pow) & ; & Pow = Span(Rel) \end{array}$$

A *map* in the category *Rel* is a relation which is total and single−valued. In general, *Map* refers to a construction which restricts the morphisms $f : A \to B$ in a homset of an preorder−enriched category to have 'weak' inverses $f^\star : B \to A$ in the sense that $1_A \sqsubseteq f^\star \circ f$ and $f \circ f^\star \sqsubseteq 1_B$. Thinking classically, a relation between A and B can be represented (in more than one way) as a pair of functions to A and B respectively with a common source (for example, the projections from the relation itself). Such a pair is called a *span*. In general, a preorder enriched category satisfying a certain condition admits of a general construction *Span* which defines a notion of morphism between such spans, and a preorder between them. The construction can be seen as a generalisation of the construction of the integers from the natural numbers. The condition is that the category has *pullovers*, this being a weakening of the standard notion of pullback that refers to the order−enrichment of homsets. (It is needed to define composition between spans.) It happens that the category $(Pow, \sqsubseteq)$ does not satisfy this condition. This may help to explain (up to a point) the apparent 'holiness' of the trinity of dynamic, transition, and interactive structures.

To shed some rain on the notion of an interactive system, it should be pointed out that Coquand has found[12] one way in which to define the coinductive *Pos* predicate (discussed in 3.3.3) for an interactive system in wellfounded type theory, but only by assuming that a simulation relation is given between the states. This enrichment leads to a notion close to the categorical notion of a *covering system*, explained for example in exercise 5 of [62][page 524]. This may indicate that they are not yet rich enough, and need to be equipped with simulation relations. There is also the problem that there is no obvious way to define a notion of morphism between bare interactive systems.

[12]In an unpublished note.

Chapter 4

Lenses

The subject of this chapter is the justification of principles of transfinite recursion in some weak predicative type theories. We construct a *recursor* for each term t in a notation system for the Schütte−Feferman ordinal Γ_0. A recursor is an expression in an extension of type theory by a single fresh variable X typed as a predicate of ordinal notations. The type of a recursor for t is the statement that if X is progressive with respect to the transition structure on notations induced by the ordinals they denote, then X holds of t.

One might call this a *justification* of a schema of recursion on the structure of proofs that a notation is accessible, based on these weak principles. The usual justification of structural recursion uses impredicative notions. The idea here is to replace the impredicative notion of arbitrary predicate by the notion of a free, fresh predicate variable. We can then represent a proof of an impredicative statement (universally quantified over arbitrary predicates) by a predicative object in type theory with a generic predicate. So in a certain sense we are reversing the usual order of justification between recursion principles and impredicative quantification. In our case, the only form of recursion we need is induction on the structure of natural numbers. What we do need is closure of the predicate−transformers expressible in the theory under a few operations involving substitution, implication, intersection of decreasing countable sequences, and crucially a certain next−universe construction.

I shall speak a little loosely of recursors as proofs of wellfoundedness; more precisely a recursor for an ordinal notation t a proof that the segment of the notation system below it is wellfounded, *i.e.* all notations for smaller ordinals are accessible with respect to the ordinal predecessor structure between notations.

The main contribution here is the notion of 'lens', which may be some value in systematising proofs of wellfoundedness in systems which are themselves explicitly type theories, or can be regarded as such by means of the Curry−Howard correspondence. So we are concerned here with establishing *lower* bounds for the least countable ordinal inexpressible in a type theory. The question of establishing *upper* bounds, interesting as it is, is regrettably not addressed here.

Very broadly speaking, the underlying idea is of a certain connection between logic and arithmetic; a little more precisely, the logical constants (or type constructions) under which a type system is closed correspond closely to the arithmetical functions which can be expressed in the system, where by arithmetic I mean (not cardinal but) ordinal arithmetic. It may not be entirely absurd to suggest that this idea can be traced back as far as Archimedes, with his problem of devising a notation for the number of grains of sand required to fill the physical universe[1]. At any rate the idea is expressed in several passages in Gentzen's publications (for example, at the end of [36], where what I have referred to as a 'correspondence' is called a 'general affinity'). Another expression of the idea occurs in Girard's book on proof theory [38], in the slogan "One quantifier equals one exponential". In my opinion Girard's slogan sacrifices precision for pizzazz. What equals one exponentiation is one occurrence of a quantified conditional statement, *i.e.* the statement form which we use to express closure of a predicate under a function. As an addendum to Girard's slogan, we can add less pithily that "One universe layer equals one nesting of the Veblen hierarchy.".

4.1 Ordinals

This section contains some general, elementary discussion of the notions of 'ordinal', and ordinal notation systems. As there are many text-books which contain a polished mathematical treatment of the notion of ordinal in the context of classical set-theory (for example [44], [86], [94], [42]), I will focus on the underlying informal notions, and some aspects which are sensitive to issues of constructivity. It is probably fair to say that the constructive theory of ordinals is quite undeveloped in comparision with the classical set theoretical treatment. The same can be said of almost any branch of mathematics: constructive mathematics is hard and few mathematicians see any point in doing it. In the case of the notion of ordinal, there are special circumstances. It was deeply involved in the very conception of

[1]According to the heliocentric system of Aristarchus of Samos, it would take about 10^{63} grains. (Even by modern estimates, 2^{2^8} would be more than enough.) In the Ionian notation known to Archimedes, it would have taken an even more astronomical quantity of papyrus simply to write the number down. In his book 'The sand reckoner', Archimedes invented a notational system to write down the number feasibly, apparently a place−value system with a base of a *myriad*, *i.e.* 10^8. If Archimedes did not invent iterated exponential notation, or realise its connection with iteration of functions of higher type, he was arguably on the brink of it.

classical set theory, and has played a central[2] rôle in the development of classical foundations.

Origins The idea of transfinite ordinals seems to have arisen first in connection with iteration of operations on subsets of the reals. In Cantor's case[3] the operation was formation of so-called 'derived' sets[4], that play a rôle in connection with the representation of functions by trigonometric series. We can form intersections and unions of infinite families of such subsets, such as the family $P, P', P'', \ldots$ of sets obtained by finite iteration of this operation starting with a given set P of reals. Beside intersection and union, there are many other naturally arising notions of infinitary limit for families of subsets of a set, and so it is natural to consider iterative exponents beyond the usual finite ordinals. If the original need for ordinals was to index iterates of an operation, then the essential property of an ordinal is that it should be 'built up from below'. This means that it should be a fit argument for a function introduced by a scheme of structural recursion, allowing the value of a function at an argument to depend on the family of values it attains at the structural components of that argument.

Relational formulation One way to analyse the notion of structural component is in terms of a binary relation between values of some kind, which obtains when the first value contains the second as a structural component (perhaps transitively). From this perspective it is natural to consider objects which consist of a base set S : Set, and a transitive relation $R : S \to Pow(S)$ on the base. (That $R\, s_1\, s_2$ should be read as saying that s_2 is a structural component of s_1, so that the partial application $R\, s$ denotes the predicate over S which holds of the structural components of s.) A structure preserving mapping (a morphism) between (S, R) and (S', R') is a function from the first base set to the second which carries the first relation into the second. The relations should be *wellfounded*, in the intuitive sense of being 'built up from below'. (We shall consider how to formulate a mathematical definition in a moment.) The first definition of the idea of an ordinal, by Cantor, was in essence that an ordinal is an isomorphism class of wellfounded objects of the above kind (though in fact Cantor required also that the relation R should be linear). (The same general strategy was used to analyse the idea of a cardinal, in this case as an isomorphism class in the category in

[2]Almost literally! The set-theoretical universe (the cumulative hierarchy of sets) is often drawn as a cone, with a line up the middle that represents the ordinals.

[3]Two excellent references for Cantor's work are [21], and [43].

[4]If P is a set of points the *derived* set P' consists of the accumulation points of P.

which the relations are equivalence relations.) One can then think of a particular object in an isomorphism class α as a notation system for the predecessors of α. An element of the base set is a notation denoting the isomorphism class of the segment of the base below that element.

Formulations of wellfoundedness How should the notion of 'built up from below' be formulated mathematically? The following are (classically) equivalent.

- All elements of S are accessible with respect to R (*cf* 3.2.1), that is, belong to every progressive subset of S, where $P \subseteq S$ is progressive if $\{ s \mid R\, s \subseteq P \} \subseteq P$. In essence, this expresses the validity of proof by induction over the relation R.

- Every non-empty subset of S contains a R-minimal element, where s is minimal in $P \subseteq S$ if $s \in P$ and $P \cap R\, s$ is empty. The equivalence of this formulation with the one above can be seen as follows. I write Q for the complement $-P$ of $P \subseteq S$, and $s_1 < s_2$ for $R\, s_2\, s_1$.

$$\begin{array}{ll}
 & P \neq \{\} \Rightarrow \exists s \,.\, P\, s \wedge \{\ y : S \mid y < s \Rightarrow P\, y\ \} = \{\} \\
= & \quad \{\ \textit{re}-\textit{express in terms of } Q\ \} \\
 & Q \neq S \Rightarrow \exists s \,.\, (\forall y \,.\, y < s \Rightarrow Q\, y) \wedge -Q\, s \\
= & \quad \{\ -(A \Rightarrow B) = A \wedge -B\ \} \\
 & Q \neq S \Rightarrow \exists s \,.\, -[(\forall y \,.\, y < s \Rightarrow Q\, y) \Rightarrow Q\, s] \\
= & \quad \{\ \textit{de Morgan}\ \} \\
 & Q \neq S \Rightarrow -(\forall s \,.\, (\forall y \,.\, y < s \Rightarrow Q\, y) \Rightarrow Q\, s) \\
= & \quad \{\ \textit{contraposition}\ \} \\
 & (\forall s \,.\, (\forall y \,.\, y < s \Rightarrow Q\, y) \Rightarrow Q\, s) \Rightarrow Q = S
\end{array}$$

  Note that if the relation $<$ is linear, we can simplify the condition somewhat to require that any non−empty subset of S should contain a *least* element; this is commonly known as the *least element principle*. Conversely, the least element principle implies that $<$ is linear. A transitive relation which is wellfounded and linear is commonly said to be a *wellordering*.

- For any $f : \mathbb{N} \to S$, there exists $n : \mathbb{N}$ such that the pair $(f(n+1), f\, n)$ does not lie in the relation R. Intuitively, this is one way of saying that all chains (finite or infinite sequences) that descend with respect to R are in fact finite. The equivalence of this formulation with the one above can be seen by considering what happens if S is *not* wellfounded with respect to R. Then there is some non−empty subset of S without minimal elements. This allows us[5] to define a sequence $f : \mathbb{N} \to S$ where $\forall n : \mathbb{N} \,.\, f(n{+}1) \in f\, n$.

[5] In the presence of a consequence of the axiom of choice known as the axiom of countable dependent choice: see for example [86, page 142].

Conversely, given any such f, its range constitutes a non−empty set with no minimal element.

Constructively, matters are more subtle. In the first place, the requirement that a relation $<$ is linear is most usually expressed using some form of 3−way disjunction, or trichotomy: $\forall x, y \,.\, x < y \vee x = y \vee y < x$. Under a constructive interpretation, this requires that we have an algorithm which applied to a given pair of elements of S decides which of these three possibilities holds. This turns out to be far too strong in general[6]. Moreover, the various formulations of wellfoundedness given above are not equivalent constructively. For example, the third condition (that descending chains are finite) quantifies over the set of functions $f : \mathbb{N} \to S$, while the first condition (that induction over R is valid) quantifies over the type of predicates over S. The assertion that the third condition implies the first is close to the content of Brouwer's so−called 'Bar theorem' (see [26]), which concerns a notion of infinitely proceeding sequence more extensive than constructive functions with domain $\mathbb{N}$.

Despite all these reservations, it is probably necessary at this point to give a definition of 'ordinal'.

Definition An *ordinal* is an isomorphism class of structures $(S, <)$ where S is a set, and $<$ is a binary relation on S which is

- transitive: $\forall x, y, z : S.\ x < y \to y < z \to x < z$
- linear: $\forall x, y : S.\ x < y \vee x = y \vee y < x$
- wellfounded: $\forall f : \mathbb{N} \to S.\, \exists n : \mathbb{N}.\ f(n+1) \not< f(n)$

It must be reiterated that this definition is quite unsuitable as a basis for a constructive theory of ordinals.

von Neumann ordinals Some decades after Cantor's pioneering investigations, the notion of set came to be understood in terms of the cumulative hierarchy of iterations of the powerset operation. From this perspective, everything is a set, and the binary membership relation $\in$ is defined between any pair of sets. According to the axiom of regularity (formulated by Zermelo, and independently Von Neumann), the membership relation is wellfounded. It is awkward to deal with ordinals as equivalence classes, which are proper classes and not sets. It would be much more convenient to fix on particular sets that serve as

[6]It is a reasonable requirement for 'concretely presented' ordinal notation systems.

canonical representatives of these equivalence classes. It is natural to look for a representation in which the transitive, linear and wellfounded relation is simply the membership relation, so that each element of representative is actually equal to the set of its predecessors. This was accomplished by von Neumann (roughly 4 decades after Cantor introduced the notion of ordinal), and it is nowadays common to use the term 'ordinal' (or sometime 'von Neumann ordinal') for the particular representatives identified by him. In this scheme, the finite ordinals are identified with the following sequence of sets.

$$
\begin{array}{lcl}
0 & = & \{\,\} \\
1 & = & \{\{\,\}\} \\
2 & = & \{\{\,\},\{\{\,\}\}\} \\
3 & = & \{\{\,\},\{\{\,\}\},\{\{\,\},\{\{\,\}\}\}\} \\
\ldots & & \\
n+1 & = & n \cup \{n\}
\end{array}
$$

The set of all such finite ordinals is called ω. Each isomorphism class of well-orderings contains exactly one von Neumann ordinal. A careful development of von Neumann's representation of the ordinals may be found for example in [94, Ch4, sec 3].

Some varieties of ordinal For examples of ordinals one can take ω, $\omega + 1$, $\omega \times 2$ (two copies of ω, one after the other), ω^2 (ω copies of ω, one after another), ω^ω (which may be identified with the 'eventually majorises' order between polynomials with natural numbers as coefficients), ϵ_0 (which may be described as the limit of the sequence $0, \omega^0 = 1, \omega = \omega^1, \omega^\omega, \omega^{\omega^\omega}, \ldots$), and other ordinals known from proof theory such as Γ_0 (which we will encounter later in this chapter). Of the examples just given, all but the second are limit ordinals, meaning ordinals which contain zero and are closed under the successor operation. They are all *countably* infinite, meaning that they all have the same cardinal as ω. A *regular* ordinal is a limit ordinal κ which is closed not only under zero and successor but also under limits of 'autonomous' sequences meaning sequences indexed by a predecessor of κ. Examples of regular ordinals are ω (the set of all finite ordinals), Ω (the first uncountable ordinal, which is equal to the set of all ordinals that are finite or countable infinite), Ω_2 (the first ordinal after Ω with a higher cardinal), but *not* the ordinal Ω_ω which is the union of the evident sequence Ω_n. Although this ordinal has a greater cardinal than its predecessors, it is the limit of a (merely) countable sequence. However the ordinal $\Omega_{\omega+1}$ is regular. A (weakly) inaccessible ordinal is a regular ordinal which is closed not only under zero, successor and autonomous limits, but also under the step from an ordinal to the next greater regular. The notion of inaccessibility is somewhat more technical. In the presence of the generalised continuum hypothesis, every weakly inaccessibile is inaccessible. A full account of the different notions of inaccessibility that have been considered by various authors may be found in [6].

The statement that inaccessible ordinals exist is independent of the axioms of ZF-set theory. Yet it is difficult to imagine serious grounds for objecting to it as an axiom, or indeed stronger axioms asserting the existence of yet larger ordinals. It is not a question of whether one 'believes in' them, as one may or not believe in Santa Claus, or the perfectibility of human beings. The reference [54] contains an extensive survey of such axioms. These axioms may seem rather abstruse, but their presence or absence in a formal system of set theory makes a difference in quite concrete terms, for example to the set of Turing machines which can be proved to terminate (a Π^0_2 statement), to the provability of Π^0_1 statements arising even in mainstream mathematics[7], and the provability of wellfoundedness for notation systems for countable ordinals (a Π^1_1 statement).

Countable ordinals In proof theory, we are directly interested only in countable ordinals. These are either 0, of successor form $S\ a$ (with a single immediate component a), or a limit $lim_n\, a_n$ of a countable sequence a_n of ordinals which are its immediate subcomponents. This is because a finitary formal system has only countably many proofs, and so the longest initial segment of the ordinals which it can represent (*i.e.* without gaps) with a formal proof of accessibility must be countable. There is an excellent elementary introduction to ordinal theoretic proof theory of arithmetic in [105][chapter 10].

From a computer science perspective, one reason to be interested specifically in countable ordinals is that we can think of a countable ordinal as a program that at each stage of its execution, does one of three things: either terminates, executes an action of some kind (say, 'beeps'), or reads the next (concrete data structure encodable as a) natural number available on an input stream.

0	– *Cease execution*
$S\ a$	– *Perform a machine cycle*
$lim_n\, a_n$	– *Read input, and take respective branch*

Arguably any finite communication can be encoded as a natural number; even if infinite objects are communicated (such as a program), this is accomplished by communicating some finite description of the infinite object, to be interpreted

[7]A number of such statements have been found recently by H. Friedman.

by the recipient. As a paradigm for a broad class of terminating programs the interest of countable ordinals should be clear.

Ordinals in Type Theory There are various approaches to the analysis of ordinals in type theory. One approach is to model sets in the cumulative hierarchy as wellfounded trees (elements of a W-type $W(U, T)$, where (U, T) is a suitably capacious universe) under a recursively defined equivalence relation, and then transfer (what one can of) the classical theory of ordinals to a constructive setting.

Another approach is to model the notion of a structural component by means of a transition system, rather than a full-blooded relation. The immediate subcomponents of a data value are typically given as a family (exhaustively[8]), rather than a predicate (separatively). A problem with this approach is that to define an ordinal as an isomorphism class one needs a notion of morphism between transition systems, and of equality between morphisms. Without some further structure (perhaps a simulation relation between states), transition systems do not seem to support a notion of morphism. Nevertheless, one can directly represent arithmetical operations on wellfounded relations (such as the ordered sums defined in sections 3.2.2 and 3.2.9) by corresponding operations on transition systems (as demonstrated in the previous chapter).

A third approach which may hold some promise is to represent ordinals as universes of a certain kind, paralleling the construction of the von Neumann ordinals. Such an approach is currently being explored by Peter Dybjer.

In conventional set theory the ordinals form a proper class. One may wonder whether, or to what extent it is possible to characterise the ordinals 'once and for all' in type theory, rather than 'number class by number class'. To deal with ordinals (in general) in type theory a proper type of ordinals would seem to be necessary, comparable to the type Set. However, although sets are decoded to types by the (implicit) function El, ordinals should (one presumes) be decoded to sets.

4.2 ϵ_0

This section contains a description of a notation system for ordinals up to Cantor's ϵ_0.

This ordinal played a key rôle in Cantor's original study of ordinal arithmetic [11]. He investigated addition, subtraction, multiplication, division, exponentia-

[8] See 3.1

tion and logarithms, and was able to show that any ordinal α has a unique base ω expression in *Cantor normal form*, or CNF_ω:

$$\begin{aligned}&\alpha = \omega^{\alpha_1} + \ldots + \omega^{\alpha_k}\\ &\alpha \geq \alpha_1 \geq \ldots \geq \alpha_k, k \geq 0.\end{aligned}$$

This gives rise to a radix-based ordinal notation system (generalising so-called 'scientific notation', *e.g.* 295×10^{73}) for an initial segment of the countable ordinals, with a decision procedure for linear comparison. (More generally, one can take an arbitrary basis ordinal $\gamma \geq 2$, with appropriate coefficients in the notation.) What makes the ordinal ϵ_0 special is that it is the first nonzero ordinal for which we do not in fact have $\alpha > \alpha_1$. (The ϵ-numbers are precisely the ordinals inexpressible 'from below' in CNF_ω.)

4.2.1 A datatype of ordinal notations

Expressions in Cantor normal form are awkward to deal with as elements of an inductively defined data structure. The definition of their order must be given simultaneously with that of the datatype itself. This affects the definition of functions by recursion on the structure of *CNF*-notations, as the structure involves proofs (albeit of decidable propositions). For some purposes it is better to drop the normality condition, and tolerate more intensional notations in which the chain of exponents of a sum of ω-powers can 'ascend'. As constructors, we can use a single constant 0 and a single binary infix operator operator $(\_\!\uparrow\!\_)$ representing $\lambda\alpha, \beta.\, \alpha + \omega^\beta$. In this section we use Æ for the set of arithmetic expressions. (Later on we extend it, and call this $Æ_g$, and the extension $Æ_v$.) So we have the following formation and introduction rules:

$$\begin{aligned}&Æ : \mathrm{Set}\\ &0 : Æ\\ &\alpha \uparrow \beta : Æ \quad \textbf{where} \quad \alpha, \beta : Æ\end{aligned}$$

The notations are then just binary trees, such as the following table showing their intuitive meanings on the left, in which I have taken the binary operator to be left associative.

$$\begin{array}{rcl}
1 & = & 0\uparrow 0 \\
2 & = & 0\uparrow 0\uparrow 0 \\
n & = & 0(\uparrow 0)^n \\
\omega & = & 0\uparrow 1 \\
\omega + n & = & \omega(\uparrow 0)^n \\
\omega \times n & = & 0(\uparrow 1)^n \\
\omega \times n + m & = & 0(\uparrow 1)^n(\uparrow 0)^m \\
\omega^n \times m & = & 0(\uparrow n)^m \\
\omega^k \times n_k + \ldots + \omega^0 \times n_0 & = & 0(\uparrow k)^{n_k}\ldots(\uparrow 0)^{n_0} \\
\omega^\omega & = & 0\uparrow\omega
\end{array}$$

4.2.2 Transition structure

There are really only four forms of the notations.

1. 0. The notation for zero.

2. $\alpha\uparrow 0$. The notation for the successor $\alpha+\omega^0$ of α.

3. $\alpha\uparrow(\beta\uparrow 0)$. The notation for a limit ordinal of the basic form $\alpha+\omega^{\beta+1}$. Such a limit is approximated by (among others) the increasing sequence of ordinals

$$\begin{array}{rl}
& \alpha, \alpha+\omega^\beta, \alpha+\omega^\beta+\omega^\beta, \ldots, \alpha(+\omega^\beta)^n, \ldots \\
= & \alpha, \alpha\uparrow\beta, \alpha\uparrow\beta\uparrow\beta, \ldots, \alpha(\uparrow\beta)^n, \ldots
\end{array}$$

   (Recall that $\uparrow$ is taken to be left associative.)

4. Anything else. This will be a notation for a limit of the form $\alpha+\omega^\lambda$, where λ itself (*i.e.* of this form or the previous one). If we chase up the rightmost branch of the tree, we will (after at least one hop) find a limit of the first kind. If the sequence $\lambda_n, n = 0, 1, \ldots$ approximates λ, then the sequence $\alpha+\omega^{\lambda_n}, \ldots$ approximates $\alpha+\omega^\lambda$. (This is because exponentiation is continuous in the exponent.)

Guided by these considerations, we set up a transition structure on Æ

$$\begin{array}{l}
\lambda\alpha : \text{Æ}.\,(C(\alpha), \lambda i : C(\alpha).\,\alpha[i]) \\
: \text{Æ} \to \mathit{Fam}(\text{Æ})
\end{array}$$

Here C is a set valued function that gives the 'cofinality' of its argument, in the form of an index set for its family of immediate predecessors. This is either $\{\,\}$ (no predecessors) for zero, a singleton set $\{0\}$ for a successor, or the set $\mathbb{N}$ of natural numbers for a limit. The function C is defined by the computation rules:

$$\begin{array}{lcl} C(0) & = & \{\,\} \\ C(\alpha \uparrow 0) & = & \{0\} \\ C(\alpha \uparrow \beta) & = & \mathbb{N} \end{array}$$

(In this and in all such subsequent tables, each clause has an implicit 'and none of the above', so that the order of the rows is significant.) These should be compared with the computation rules for a universe which decodes an element of its domain as a set.

The indexing function is defined by the following computational rules:

$$\begin{array}{lcl} \_[\_] & : & (\prod \alpha : \text{Æ})\ C(\alpha) \to \text{Æ} \\ (\alpha \uparrow 0)[0] & = & \alpha \\ (\alpha \uparrow (\beta \uparrow 0))[0] & = & \alpha \\ (\alpha \uparrow (\beta \uparrow 0))[n+1] & = & x \uparrow \beta,\ \textbf{where}\ x = (\alpha \uparrow (\beta \uparrow 0))[n] \\ (\alpha \uparrow \beta)[n] & = & \alpha \uparrow (\beta[n]) \end{array}$$

These can be thought of as elimination and computation rules. (Normalisation means that $\ldots[n]$'s can always be eliminated.)

It may be interesting that this notation system can be quite easily represented in a functional programming language. Here is all one needs to do to code it in Haskell. (It is an advantage here that Haskell accommodates without fuss data structures that are lazily infinite. Note however that the notations themselves are finite wellfounded structures.)

```
> infixl `W`              -- a `W` b = W a b

> data AE  = N            -- Nought, represents 0
>          | W AE AE      -- W a b represents a + w^b
```

```
For each expression e, I define a (possibly infinite) list (pd e)
of its immediate predecessors.
```

```
> pd :: AE -> [AE]
> pd N                  = []
> pd (a `W` N)          = [a]
> pd (a `W` (b `W` N)) = iter (`W` b) a
> pd (a `W` b)          = map (a `W`) (pd b)   -- continuity
```

The standard functions `iter` and `map` may be defined as follows.

```
> iter f x = x : iter f (f x)          -- corecursive
> map f xs = case xs of [] -> []
>                       (x:xs') -> f x : map f xs'
```

One may even represent the entire infinite (but in fact wellfounded) tree of ordinal predecessors of a notation by means of a construction such as the following. The definition is corecursive in form, and so at first sight the tree associated to an expression may contain infinite paths. The point is that although it is not immediately obvious, on analysis, so long as one restricts attention to notations which are wellfounded as binary trees[9], the tree of predecessors will turn out to be wellfounded.

```
> data Tree x = Tree x [Tree x]  -- so-called 'rose' trees

> ordTree :: AE x -> Tree (AE x)
> ordTree e = Tree e (map ordTree (pd e))  -- corecursive
```

The distinction between the *syntactical* structure of an ordinal notation, and its *ordinal* structure may need some further clarification. The syntactical structure of a notation makes no reference to the transition structure on the notations. As a syntactical entity, an expression in `AE` has the following binary tree structure.

```
> synTree :: AE -> Tree AE          -- structurally recursive
> synTree N         = Tree N []
> synTree (a `W` b) = Tree (a `W` b) [syn a, syn b]
```

It is an ordinal notation, that is as an element of the transition structure expressed in the Haskell code by the function `pd` that an element of `AE` can represent something infinite but wellfounded. The laziness of Haskell is exploited in the code above partly to conveniently represent the immediate ordinal predecessors of a notation by a potentially infinite list[10], and partly to represent the tree of ordinal ordinal predecessors.

4.2.3 Notions pertaining to the transition structure

As is rather customary in mathematics, I shall not distinguish notationally between the transition system Æ, and the set Æ which it is base.

[9] That is, excludes infinite expressions such as `let t = t `W` t in t`

[10] One could use instead functions defined on the natural numbers. This would however only be more cumbersome.

When talking about *predicates* I usually mean predicates over Æ, that is to say, set valued functions with domain Æ, or inhabitants of the type $Pow(Æ)$. An inclusion of P in Q is an element of the set $(\prod \alpha : Æ)\ P(\alpha) \to Q(\alpha)$. By taking morphisms to be inclusions, and equality between morphisms to be constantly true, we can make $Pow(Æ)$ into a category, in which isomorphism is exactly extensional equality between predicates. There is further structure: a lattice structure over Æ (intersection and union of set-indexed families of predicates), an operation $X \mapsto X(.)f$ of composition with a function on states, and exponentials in the sense of a Heyting algebra. These last two closure properties allow us to express that a predicate is closed under a section of a binary function, with a predicate.

By a predicate transformer, I mean an operation from predicates to predicates. To put it a little fancifully, if a predicate is a set with an Æ-shaped hole in it, then a predicate transformer is a predicate with a predicate shaped hole in it. Concretely, a predicate transformer in a type theory is represented by a predicate term which may contain a free predicate variable. A predicate transformer need not be monotone. In fact the predicate transformers with which we shall be concerned are not monotone.

4.2.3.1 Induced predicate transformer

Associated with any transition system is the predicate transformer

$$\begin{array}{lcl} B & : & Pow(Æ) \to Pow(Æ) \\ B(X,\alpha) & = & (\prod i : C(\alpha))\ X(\alpha[i]) \end{array}$$

which I shall call the predicate transformer induced by the transition structure. (It is the predicate transformer written $\_^{\bullet}$ in 3.2.1)

A predicate $P : Pow(Æ)$ is *progressive* if $B(P) \subseteq P$. This means that we have the following constants and operators.

- A constant $0_p : P(0)$.
- An Æ−indexed family of unary operators $S_p(\alpha) : P(\alpha) \to P(\alpha \uparrow 0)$
- An Æ^2−indexed family of infinitary operators

$$\begin{array}{l} L_p(\alpha,\beta) : (\prod n : \mathbb{N})\ P(\alpha(\uparrow\beta)^n) \\ \qquad \to P(\alpha\uparrow(\beta\uparrow 0)) \end{array}$$

- An Æ^4−indexed family of similarly infinitary operators

$$\begin{array}{l} M_p(\alpha,\beta,\gamma,\delta) : (\prod n : \mathbb{N})\ P(\alpha\uparrow(\beta\uparrow(\gamma\uparrow\delta))[n]) \\ \qquad \to P(\alpha\uparrow(\beta\uparrow(\gamma\uparrow\delta))) \end{array}$$

A predicate transformer which transforms progressive predicates into progressive predicates is said to *maintain progress.* Not only does it transform predicates, it transforms algebras (proofs of progressivity), by some construction from the operations of the input algebra to operations on the output algebra. We shall focus on predicate transformers which maintain progress. Observe that B itself maintains progress, just because B is monotone.

An element of Æ is *accessible* if it is in the intersection of all progressive predicates, and the transition system Æ is *wellfounded* if all its elements are accessible. (This definition appeals to the impredicative notion of an arbitrary predicate over Æ, or to put it another way, the definition of a set by a Π^1_1 condition.)

4.2.3.2 The internal order

Let $\alpha < \beta$ be the relation $(\sum i : C^+(\beta)\,)\ \alpha = \beta[i]^+$, where $=$ is syntactical equality between expressions, and $(C^+, \_[\_]^+)$ is the transitive closure of the transition structure $(C, \_[\_])$. I call this relation the *internal* order, and say that α is an internal predecessor of β, as it captures the idea that α actually occurs somewhere inside (*i.e.* as an ordinal predecessor of) β.

I shall not need to refer to it, but for $n : \mathbb{N}$, let $\alpha <_n \beta$ be the relation which holds between α and β if $\alpha < \beta$, and all transitions referred to in the proof are less than n. Clearly, $\alpha < \beta$ if and only if for some n $\alpha <_n \beta$. The notation system (Æ, $<$) is in fact the colimit in the category of linear orders of the chain (Æ, $<_n$) with the obvious embeddings from (Æ, $<_n$) to (Æ, $<_{n+1}$).

The notation system has some good properties with respect to the internal order relation.

lemma 1 *If α is any notation of limit form, then for all $i : \mathbb{N}$ $\alpha[i] <_1 \alpha[i+1]$.*

4.2.4 Two proofs that Æ is wellfounded

There are (as far as I know) essentially two ways to prove Æ is wellfounded. The first proof uses recursion over the structure of proofs of accessibility, and so appeals to a principle of generalised (that is, non−finitary) inductive definition. This principle is unavailable to us in first order arithmetic, or a type theory of the kind we are considering. The second proof uses instead a predicate transformer under which the predicates expressible in first order arithmetic are closed.

4.2.4.1 By transfinite recursion

The first proof uses transfinite recursion on proofs of wellfoundedness. We simply prove $Wf(\alpha) \to Wf(\beta) \to Wf(\beta \uparrow \alpha)$ by recursion on the structure of the proof that $Wf(\alpha)$, with a nested recursion on the structure of the proof that $Wf(\beta)$.

4.2.4.2 By closure of predicates under logical operations

Define the exponential depth of a notation as follows:

$$\begin{array}{lcl} depth(0) & = & 0 \\ depth(\alpha\uparrow\beta) & = & \max(depth(\alpha)+1, depth(\beta)) \end{array}$$

A notation with depth $< n+1$ has the form $0\uparrow\beta_1\uparrow\ldots\uparrow\beta_k$, where $\beta_1,\ldots\beta_k$ have depth $< n$. Intuitively, the depth of a notation is the number of levels of superscripting if $\alpha\uparrow\beta$ is typeset as $\alpha+\omega^{\beta}$. The only expression of depth 0 is 0, the only expressions of depth at most 1 are the predecessors of ω, the only notations of depth at most 2 are the predecessors of ω^{ω}, and so on.

Similarly, we define the implicational depth of a type built up by $\rightarrow$ from atomic types with zero depth as follows.

$$\begin{array}{lcl} depth(a) & = & 0 \\ depth(A\rightarrow B) & = & \max(depth(A)+1, depth(B)) \end{array}$$

For the second proof, consider the predicate transformer

$$G(X,\alpha) = X\subseteq X\circ(\uparrow\alpha)$$

This transforms a predicate X into the predicate (one deeper) which holds of α if X is closed under the right section $(\uparrow\alpha)$.

lemma 2 *G maintains progress.*

proof: Suppose that $X : Pow(Æ)$ is progressive. The proof that $G(X)$ is progressive is by cases.

Since X is progressive, it is closed under $(\uparrow 0)$. From this it follows that $G(X,0)$.

Suppose that α has limit form, and that for all n $G(X,\alpha[n])$, *i.e.* that X is closed under $(\uparrow\alpha[n])$. Since X is progressive, it holds of a limit if it holds of all its immediate predecessors. It follows that X is closed under $(\uparrow\alpha)$. It is crucial here that $(\uparrow)$ is continuous in its second argument.

As for successor form, suppose that $G(X,\alpha)$, *i.e.* X is closed under $(\uparrow\alpha)$. Then X is closed under all finite iterates $(\uparrow\alpha)$, which have (pointwise) limit $(\uparrow(\alpha\uparrow 0))$. Since X contains a limit when it includes the sequence of its predecessors, we have $G(X,\alpha\uparrow 0)$.

From this, we can construct in a very concrete way a proof of the accessibility of an arbitrary notation in Æ, without any use of transfinite recursion, but instead only that the universe of predicates is closed under the predicate transformer G.

Zero

$$\frac{\textit{Immediate}}{B(X) \subseteq X \vdash X(0)}$$

Non−zero

$$\frac{\dfrac{\textit{i.h. for } \alpha,\ X}{B(X) \subseteq X \vdash X(\alpha)} \quad \dfrac{\textit{lemma 2} \text{ for } X}{B(X) \subseteq X \vdash B(G(X)) \subseteq G(X)} \quad \dfrac{\textit{i.h. for } \beta,\ G(X)}{B(G(X)) \subseteq G(X) \vdash G(X, \beta)}}{B(X) \subseteq X \vdash X(\alpha \uparrow \beta)}$$

Figure 4.1: Wrapping a Æ notation in the proof of its accessibility

theorem 1 *All notations of exponential depth* $< (n+2)$ *can be proved accessible in type theory without universes, even if recursion over* $\mathbb{N}$ *is restricted to sets with implicational depth below* n*. It is sufficient that the type structure can express that a predicate of notations is closed under an unary function* $Æ \to Æ$*.*

The intuition for the proof is that we can 'clothe' or 'wrap' an ordinal notation with a proof that it satisfies an arbitrary progressive predicate. The idea is illustrated in the picture in figure 4.1 on the current page. (There is a formal proof in appendix B.)

We proceed from the root of the syntax tree of the notation. The root expression is wrapped with the 'problem' $B(X) \subseteq X \vdash X(\_)$. If the notation has the form $\alpha \uparrow \beta$, then α is wrapped with $B(X) \subseteq X \vdash X(\_)$, and β with $B(G(X)) \subseteq G(X) \vdash G(X, \_)$. The latter problem can be reduced, by cutting with the proof of lemma 2, to the problem $B(X) \subseteq X \vdash G(X, \_)$. Eventually we arrive, at the leaves of the syntax tree with notations has the form 0, with problems of the form $B(G^n(X)) \subseteq G^n(X) \vdash G^n(0)$, which are trivial to solve. When the process is complete, we have constructed a proof that our notation is accessible.

The predicate transformer G is an example of a *lens* for the function $0 \uparrow \alpha$. A formal definition is given later, but the idea is that a lens for a function f is a predicate transformer F which maintains progress, and satisfies $F(X) \subseteq X \circ f$ for progressive predicates X. The term 'lens' was chosen to suggest (optical) magnification; a lens is a device which helps us to see further into 'the transfinite'.

There is a certain inevitability about Gentzen's lens, which can perhaps be put in the following way. Suppose we would like to prove that a binary function on notation systems, written $\_ \dagger \_$, preserves accessibility. Then it would suffice to have *two* predicate transformers Φ and Ψ (one for each argument of the function), which preserve progressivity, and are such that if X is a progressive predicate, then $\Phi(X, \alpha)$ and $\Psi(X, \beta)$ together imply $X(\alpha \dagger \beta)$. How might we attack this problem? Let us make it easier for ourselves by *defining*

$$\Phi(X, \alpha) = \Psi(X) \subseteq X \circ (\dagger\alpha)$$

This takes care of the second property, by sheer cookery. As for preservation of progressivity, let us again take the path of least resistance and define Ψ to be the identity. So the definition of Φ becomes even simpler:

$$\Phi_{simple}(X) = \{\ \alpha \mid X \subseteq X \circ (\dagger\alpha)\ \}$$

What remains is to prove that Φ_{simple} preserves progressivity. This is the only part of the problem which requires any insight into the nature of the $\dagger$ operator. Of course, in some specific case, there may be a non−trivial Ψ which preserves progressivity, and for which it is possible to prove that the more general Φ (defined as in the first equation above) preserves progressivity[11].

4.3 Γ_0

In the previous section, we made use of closure of the type structure of a type theory under the the predicate transformer G to find a lens for the function ω^α. Using that lens, we showed how to construct in that theory proofs of accessibility for successive terms of a sequence of ordinals approximating ϵ_0.

In this section, we describe how to make use of closure of the type structure of a system such as Martin-Löf's under a step 'to the next universe' in order to construct proofs of accessibility for terms approximating Γ_0. By 'the step to the next universe', I mean the rules for forming from a given family of sets a family which contains it as a proper subfamily and is closed under particular forms of $\prod$ and $\sum$. By Γ_0 I mean the first (non-zero) ordinal closed under not only $\beta + \omega^\alpha$ but also Veblen's 2-place function $\phi_\alpha(\beta)$, where ϕ_0 is the function $\lambda\beta.\,\epsilon_\beta$, and for $\alpha > 0$, ϕ_α enumerates the common fixed points of all functions ϕ_β with $\beta < \alpha$. Each new universe gives us another level of nesting of the ϕ function; thus with no universes, we get to ϵ_0, with one to $\phi_{\epsilon_0}(0)$, with two to $\phi_{\phi_{\epsilon_0}(0)}(0)$, and so on. Although the construction makes intensive use of lenses, I have not been able to see how to express it as a lens for (say) the function $\phi_\alpha(0)$. Instead, the key step

[11]It should be clear how to generalise these considerations to operators of arity higher than two. I am not certain that the definition of lens used in this thesis succeeds in 'carving out' exactly the right notion.

in using a new universe is to encode the notion of a lens for a function ϕ_α as a predicate, and to prove it progressive.

The results have been known since the 70's[12]. However, to establish a lower bound on the strength of the system, it was necessary (or at least convenient) to use an indirect argument, proceeding by way of so-called 'hatted' theories $\widehat{ID_0}$, $\widehat{ID_1}, \ldots$ axiomatising (not necessarily least) fixed points for certain monotone operators on predicates, over classical logic. It is perhaps fair to say that despite these technical results, no-one had much idea what a proof of wellfoundedness carried out in this type theory would actually look like. In contrast, the constructions I am about to describe have been formalised as concrete programs or 'λ-terms' in a proof checker[13].

4.3.1 Arithmetic expressions

We first need to extend the datatype of ordinal expressions with a further binary operator. To 0 and $\alpha \uparrow \beta$ representing $\alpha + \omega^\beta$ we add $\alpha \Uparrow \beta$ representing $\phi_\alpha\beta$, where $\phi_\alpha : \Omega \to \Omega$ is the $\alpha-$th Veblen derivative of $\phi_0(\beta) = \epsilon_\beta$, meaning that for $\alpha > 0$, ϕ_α enumerates the common fixed points of the set of normal (continuous and strictly increasing) functions $\{\ \phi_\beta : \Omega \to \Omega \mid \beta < \alpha\ \}$. (It is convenient to start with ϵ_β rather than the more usual ω^β in order that $\alpha \Uparrow \beta$ is always a limit, and to avoid overlap with $(\uparrow)$.) I call the resulting datatype $Æ_v$, or in this section simply Æ.

Now the computation rules for the function which assigns to an arithmetic expression the family of its immediate predecessors become:

expression γ	$C(\alpha)$	$\alpha[i]$ *for* $i : C(\gamma)$
0	$\{\}$	(undefined)
$\alpha \uparrow 0$	$\{0\}$	α
$\alpha \uparrow (\beta \uparrow 0)$	$\mathbb{N}$	$\alpha (\uparrow \beta)^i$
$\alpha \uparrow \beta$	$\mathbb{N}$	$\alpha \uparrow \beta[i]$
$0 \Uparrow 0$	$\mathbb{N}$	$(0 \uparrow)^i 0$
$0 \Uparrow (\beta \uparrow 0)$	$\mathbb{N}$	$(0 \uparrow)^i ((0 \Uparrow \beta) \uparrow 0)$
$(\alpha \uparrow 0) \Uparrow 0$	$\mathbb{N}$	$(\alpha \Uparrow)^i 0$
$(\alpha \uparrow 0) \Uparrow (\beta \uparrow 0)$	$\mathbb{N}$	$(\alpha \Uparrow)^i (((\alpha \uparrow 0) \Uparrow \beta) \uparrow 0)$
$\alpha \Uparrow 0$	$\mathbb{N}$	$\alpha[i] \Uparrow 0$
$\alpha \Uparrow (\beta \uparrow 0)$	$\mathbb{N}$	$\alpha[i] \Uparrow ((\alpha \Uparrow \beta) \uparrow 0)$
$\alpha \Uparrow \beta$	$\mathbb{N}$	$\alpha \Uparrow \beta[i]$

[12] In the case of one universe, the first results were obtained by Aczel in 1974.

[13] The system first used was the 'Chalf' system written (in `C`) by Dan Synek, based on ideas and a prototype written (in Haskell) by Thierry Coquand. I am particularly indebted to Anton Setzer help with this code during a week's visit to Edinburgh in 1997. I have subsequently 'ported' the proof to the 'Agda' system; the code itself is contained in Appendix C.

The definition of the (family−assigning) function is by recursion on the syntactic structure of the arithmetic expression, and needs only a weak form of first order primitive recursion.

The ordinal notation system can be coded directly as a data type in Haskell, as mentioned in section 4.2.2. The code now becomes the following.

```
> infixl `W`               -- a `W` b = W a b
> infixr `V`               -- a `V` b = V a b

> data AE  = N             -- Nought, represents 0
>          | W AE AE      -- W a b represents a + w^b
>          | V AE AE      -- V a b represents phi(a,b)

> suc :: AE -> AE
> suc a = a `W` N

> pd :: AE -> [AE]
> pd N                   = []
> pd (a `W` N)           = [a]
> pd (a `W` (b `W` N)) = iter (`W` b) a
> pd (a `W` b)           = map (a `W`) (pd b)
> pd (N `V` N)           = iterate (N `W`) N
> pd (N `V` (a `W` N)) = iterate (N `W`) t  where t = suc (N `V` a)
> pd ((a `W` N) `V` N) = iterate (a `V`) N
> pd ((a `W` N) `V` ( b `W` N))
>                        = iterate (a `V`) t  where t = suc (a `V` b)
> pd (a `V` N)           = map (`V` N) (pd a)
> pd (a `V` (b `W` N)) = map (`V` t) (pd a) where t = suc (a `V` b)
> pd (a `V` b)           = map (a `V`) (pd b)
```

Recall that a predicate P : Æ → Set is *progressive* if satisfies the following condition, at each α : Æ.

$$((\prod i : C(\alpha)\,)\; P(\alpha[i])) \rightarrow P(\alpha)$$

It is easily checked that progressive predicates are closed under intersections of set−indexed families. In categorical terms, a progressive predicate is (equipped with a function which makes it) an algebra in the category of predicates and inclusion maps over Æ, for the following endofunctor (*i.e.* monotone predicate transformer).

$$B(P) = \{\ \alpha : Æ \mid (\prod i : C(\alpha)\,)\ P(\alpha[i])\ \}$$
$$B(f : P \subseteq Q) = \lambda\alpha : Æ, i : C(\alpha).\, f(\alpha[i]) : B(P) \subseteq B(Q)$$

The initial algebra of this functor is the *accessibility* predicate for the assignment of predecessors *pd*. An element of a set $Acc(\alpha)$ for some $\alpha : Æ$ is a wellfounded tree which represents the predecessor structure of α.

The existence of this least fixed point requires minimally the principle of generalised inductive definition, iterated once, a principle which is unavailable to us in the type theory we are considering. If we make use of this principle, it is straightforward to establish the following lemma. The purpose of this section is to show that we can establish each *instance* of the lemma in a type theory with only ordinary (not generalised) induction, but instead an external sequence of universes closed under $\prod$ and $\sum$. We cannot expect to prove the lemma itself in that theory; according to [33], its proof theoretic strength is exactly $Gamma_0$. (As far as I am aware, no direct proof of this fact is known.)

lemma 3 *All expressions in Æ are accessible.*

proof: Since Æ is inductively defined, we need to show that its constructors preserve accessibility. It is immediate that 0 is accessible. We consider the constructors $\alpha \uparrow \beta$ and $\alpha \Uparrow \beta$ in turn.

The structure of the proof that $Acc(\alpha) \to Acc(\beta) \to Acc(\alpha \uparrow \beta)$ is recursion on the proof that $Acc(\beta)$. The case $\beta = 0$ is trivial. In the successor case $\beta = \gamma \uparrow 0$, by ordinary numerical recursion we can strengthen the induction hypothesis to $Acc(\alpha) \to Acc(\alpha(\uparrow \gamma)^n)$ where $n : \mathbb{N}$. It is crucial here that the induction hypothesis 'has an arrow'; the proof of $Acc(\alpha)$ is an argument, not a parameter. From this $Acc(\alpha) \to Acc(\alpha \uparrow (\gamma \uparrow 0))$ follows by closure of progressive predicates under limits. The argument in the limit case is again by closure of progressive predicates under limits. (This repeats the argument of section 4.2.4.1.)

The structure of the proof that $Acc(\alpha) \to Acc(\beta) \to Acc(\alpha \Uparrow \beta)$ is recursion on the proof that $Acc(\alpha)$, with nested recursion on the proof that $Acc(\beta)$. There are three forms of expression to consider. In each case, the nested recursion (on β) makes use of a slightly different observation for the zero and successor cases. The limit case is always by continuity.

$0 \Uparrow \beta$. The observation is that since $(\uparrow)$ preserves accessibility, in particular $(0 \uparrow)$ and all its finite iterates preserve accessibility.

$(\alpha \uparrow 0) \Uparrow \beta$. The observation is that by hypothesis the function $(\alpha \Uparrow)$ and all its finite iterates preserve accessibility.

$\alpha \Uparrow \beta$ where $C(\alpha) = \mathbb{N}$. The observation is that by hypothesis the functions $(\alpha[n] \Uparrow)$ preserve accessibility for all $n : \mathbb{N}$.

4.3.2 The theory of lenses

In this section, we set out the basic theory of lenses. A machine−checked development of the theory of lenses is given in appendix C. This is a fairly simple theory of predicate transformers of a particular kind arising from the transition structure on a notation system. We will be particularly interested in predicate transformers defined by iteration, and so we need a set for this iteration to 'happen' in, which is to say, a universe of small sets, closed under the necessary constructions.

The mathematical content of this theory is modest. The interest is that we can define a really quite simple construction on lenses corresponding to Veblen's derivative operation without any use of transfinite recursion. Instead we assume only that the universe satisfies some weak closure properties, principally closure of predicates under intersections of certain countable sequences. The simplicity of the construction is a virtue if one thinks of the task as a programming problem.

Definition A lens for a function $\phi : \text{Æ} \to \text{Æ}$ is a predicate transformer which

1. preserves progressivity with respect to the transition structure on Æ. That is, $\mathit{Prog}(\Phi(X))$ for all progressive $X : \mathit{Pow}(\text{Æ})$.
2. ensures $\Phi(X) \subseteq X \circ \phi$ for all progressive $X : \mathit{Pow}(\text{Æ})$.

In the following lemmas, It is (an ordinary form of) the iteration functional, mapping a natural number and a set S to the set $(S \to S) \to S \to S$. It_T is its tail recursive variant.

$$
\begin{array}{l}
\mathit{It}, \mathit{It}_T :: \mathbb{N} \to (\prod S : \mathit{Set}\,)\ (S \to S) \to S \to S \\
\mathit{It}(0, S, f, s) = s \\
\mathit{It}(n+1, S, f, s) = f(\mathit{It}(n, S, f, s)) \\
\mathit{It}_T(n, S, f) = \mathit{It}(n, S \to S, \lambda g.\, g \circ f, \mathit{id})
\end{array}
$$

I shall omit the set parameter (*i.e.* the second parameter), as it can usually be inferred.

lemma 4 *1. If Φ and Ψ preserve progressivity, then so does $\Phi \circ \Psi$.*

2. If Φ and Ψ are lenses for f and g respectively, then $\Phi \circ \Psi$ is a lens for $g \circ f$.

3. *If Φ is a lens for f then for any $n : \mathbb{N}$, $It_T(n, \Phi)$ is a lens for $It(n, f)$, and $It(n, \Phi)$ for $It_T(n, f)$.*

The proofs are straightforward.

lemma 5 *The intersection of any (countable) sequence of progressive predicates is progressive.*

The proof depends only on the fact that the predicate transformer $(\_)^\bullet$ (in the notation of section 3.2.1) induced by the transition system is monotone.

lemma 6 *If a function f on ordinal notations is the pointwise limit of the sequence f_n, and Φ_n is a lens for f_n for each n, then $\Phi = \lambda X, a. (\prod n : \mathbb{N})\ \Phi_n(X, a)$ is a lens for f.*

That Φ preserves progressivity follows from the last lemma. That $\Phi(X) \subseteq X \circ f$ for progressive X follows from $\Phi(X) = \bigcap_n \Phi_n(X) \subseteq X \circ f_n$ for all n, and the fact that X is closed under limits. (The counterpart of this lemma in the formal proof in appendix C is called `SupLens`.)

lemma 7 *Given a lens Φ for a function f, we can find a lens Φ' for the identity (i.e. such that any progressive predicate X is a prefixed point of Φ'), such that for progressive X, $\Phi'(X)$ is preserved by*

1. *f.*

2. *All finite iterates f^n.*

3. *f^ω.*

Proof: $\Phi'(X, a) \triangleq (\prod n : \mathbb{N})\ It(n, \Phi, X, a)$. Trivially $\Phi'(X) \subseteq X$. If X is progressive, then $\Phi^n(X)$ is progressive for all n, so $\Phi'(X)$ is progressive. If $\alpha \in \Phi'(X)$, then $\alpha \in \Phi^{n+1}(X)$ and so $f(\alpha) \in \Phi^n(X)$ for any n, so that $\Phi'(X)$ is preserved by f for progressive X. This establishes the first part of the lemma. (The counterpart of this lemma in the formal proof in appendix C is called `MkClens`.) The second part of the lemma is a trivial consequence of the first, and the third follows from the second since progressive predicates are closed under limits. (The counterpart of the last part of the lemma in the formal proof in appendix C is called `ItClens`.)

lemma 8 *Given Φ', and f as in the last lemma, we can find*

1. *a lens Φ'' for the function $\nabla f = \lambda\alpha.\,(f \circ (+1))^\alpha(f(0))$.*

2. *a lens for the derivative $f'(\alpha) = \nabla f^\omega$.*

It should be stressed that the proof of this lemma does not use transfinite recursion, but depends only on the definitional equalities satisfied by f'.

Proof: For the first part, $\Phi''(X) = \Phi'(X) \circ (\nabla f)$. It is simple to check that Φ'' preserves progressivity, and that $\Phi''(X) \subseteq X \circ (\nabla f)$ for progressive X. (The counterpart of this lemma in the formal proof in appendix C is called `NablaLens`.) The second part of the lemma is proved by applying the first part to f^ω.

lemma 9 *(Gentzen) We can find a lens for the function $(\uparrow \alpha) = (+\omega^\alpha)$.*

Proof: $\lambda X, \alpha.\, X \subseteq X \circ (+\omega^\alpha)$. The proof of this lemma has already been given in 4.2.4.2. (Its counterpart in the formal proof in appendix C is called `GentzenLens`.)

lemma 10 *1. We can find a lens for the function $(0 \Uparrow)$.*

2. If we have a lens for the function $(\alpha \Uparrow)$, then we can find one for $((\alpha \uparrow 0) \Uparrow)$.

3. If we have lenses for each of $(\alpha[i] \Uparrow)$, (where α is not of the form $\alpha \uparrow 0$ or 0, i.e. is a limit), then we can find a lens for $(\alpha \Uparrow)$.

Proof: For the first part, since $(0 \Uparrow)$ is the derivative of $\lambda\alpha.\,\omega^\alpha$, we can just apply lemma 8 to the Gentzen lens. The second part is a direct applications of the second part of lemma 8. The third follows from lemma 8 and lemma 6. (The formal counterparts of these arguments in appendix C are called `ZeroLens`, `SuccLens` and `LimLens` respectively.)

So we have the induction hypotheses which would be sufficient to derive by transfinite recursion a lens for any function of the form $(\alpha \Uparrow)$ – were we not abstaining from transfinite recursion. Teetering on this brink, we notice:

lemma 11 *If we have a lens for a function f, then the accessible terms (i.e. those in the intersection of all progressive predicates) are closed under f.*

Proof: Suppose α satisfies any progressive predicate; we want to see that $f(\alpha)$ does too. Suppose Φ is a lens for f. Given a progressive predicate X, we also have that $\Phi(X)$ is progressive, so holds of α. Therefore X holds of $f(\alpha)$. (The formal counterpart of this argument is called `LensLemmaV` in appendix C.)

This is all the theory of lenses we need for present purposes. The rest has to do with universes.

4.3.3 Miniaturisation of the theory of lenses

The theory of lenses involves two universes: the 'small' universe of the predicates, and the 'big' universe above it in which the theory itself is expressed, in which we quantify universally over small predicates, and existentially over predicate transformers. Our aim in this section is to demonstrate the following

theorem 2 1. *If α and β are in the intersection of all small progressive predicates, then so is $\alpha \uparrow \beta$.*

2. *If α is in the intersection of all big progressive predicates, and β in the intersection of all small progressive predicates, then $\alpha \Uparrow \beta$ is in the intersection of all small progressive predicates.*

The first part of the theorem has already been proved in 4.2.4.2. The proof of the second part of the theorem is by cooking the definition of 'big', so that the predicate $\lambda\alpha.\, Lens(\alpha \Uparrow 0)$ is big. By lemma 10 in the previous section, it is progressive. So if α is in the intersection of all large progressive predicates, there is a lens for $(\alpha \Uparrow)$. By lemma 11 in the previous section, the intersection of all small progressive predicates is closed under $(\alpha \Uparrow)$.

The definitions and propositions of the theory of lenses have a particularly simple form, well illustrated by the notion of lens itself:

$$\begin{array}{rcl} Lens(f : S \to S) & \triangleq & (\sum \Phi : Pow(S) \to Pow(S)) \\ & & (\prod X : Pow(S)) \\ & & B(X) \subseteq X \to (B \circ \Phi)(X) \subseteq \Phi(X) \subseteq (\circ f)(X) \end{array}$$

Inspection of the definitions and proofs of the theory of lenses shows that they can be carried out if the universe of 'big' sets contains the universe of small sets and is closed under

1. universal quantification over small predicates. (The $\prod$ quantifier.)
2. function spaces. (A special case of the $\prod$ quantifier.)
3. universal quantification over Æ.
4. the predicate transformer B.
5. existential quantification over small predicates. (The *sum* quantifier.)
6. conjunction. (A special case of the $\sum$ quantifier.)

Such a 'miniaturisation' is actually carried out in the package called `small` in appendix C.

It is regrettable that I have had to require that the universes are closed under $\sum$, as this weakens the result. My intuition has been that requiring universes to be closed under the $\sum$ quantifier adds nothing to the proof$-$theoretic strength of type theory, at least in the context of the weak systems with which I have been concerned. The problem has been that the use of $\sum-$types in some form (for example, to express the notion of lens) has been in practice necessary to keep the more bureaucratic details of the construction under control. Perhaps they can be dispensed with 'in principle'; however I wished actually to construct proofs of accessibility in *practice*.

It is also regrettable that I have not been able to bring the formal proof into a clean enough form to actually present it in this chapter (rather than merely to allude to it). As well as a certain amount of tool support, and perhaps a formal proof language designed for the purpose, the 'literate' presentation of formal proofs seems to require a very high level of taste, or art.

Chapter 5

Conclusions

This thesis has been concerned with dependent type theories. It has focussed on three areas, all of them more or less loosely connected with the representation, use and calibration of proofs of wellfoundedness in such theories. This concluding chapter tries to draw these strands together, and gather up the loose ends.

The first main chapter was concerned with the representation of type theories in a logical framework based on a name−free substitution calculus. The topic here is the fundamental mechanisms on which the representation of inductive definitions and recursion principles is based. One can perhaps compare this with mechanics at the level of particle physics; the issues that arise here are rather delicate. What exactly is the logical framework itself? What is it for? How should it be structured? What kind of equational reasoning is involved in the general representational mechanisms of the logical framework, and where is the boundary between this general equational reasoning, and the computational rules of a specific type theory represented in the framework? To what extent can we expect to capture the notion of inductive definition formally within such a setting?

The second main chapter was concerned with the use of inductively defined datatypes in programming, to model dynamic behaviour. The central models were transition systems and interactive systems. Some twenty years after Martin−Löf first suggested that type theory could be considered as a programming language, we are perhaps beginning to suspect that this was not just an abstract possibility, or a flight of philosophical fancy, but a fully serious suggestion. How then do we write real programs in type theory, that 'do things'? We learn from the last chapter of [80] (entitled 'Programming in Martin−Löf's Type Theory') how to divide by 2, and how write a program to solve Dijkstra's problem of the Dutch national flag. I do not wish to criticise the focus on small problems, but how are we to write a program to run a web server, or fly an aeroplane? How do we write an editor, or a transaction monitor, or an operating system? To my mind the fundamental issue is one of understanding how to write interactive systems in type theory. This is not just a technical problem, or a mathematical one, but to some degree a conceptual problem; it may even be to some extent a problem of having the courage to begin to think about the issues. It cannot be long before good implementations of functional programming languages with dependent type systems are within reach. What facilities for interactive programming , and 'I/O' (input−output) will be needed? My hope is that further investigation of models of the kind described in this chapter will contribute to a the development of a monadic I/O interface for such a programming language; perhaps one in which the semantics of the I/O primitives can be precisely specified through their types.

The third main chapter was concerned with 'quantitative' investigation into the proof theoretic strength of the principles of induction and recursion available in a specific type theory. The locus of this question is the old−established subject of ordinal theoretic proof theory. The content of the chapter was a construction of the provable ordinals in a weak form of Martin−Löf's type theory, having a sequence of cumulative universes closed under finitary inductive definitions, and the $\prod$ and $\sum$ quantifiers. The issues have a much more technical character than in the previous two chapters; the contribution here is the notion of 'lens' which I hope may serve somehow to organise the investigation of the proof theoretic strength of stronger type theories. It is remarkable, and to my mind rather unsatisfactory that proof theoretical investigations of constructive type theories proceed by reducing them as quickly as possible to subsystems of classical arithmetic, or forms of classical Kripke−Platek set theory. It is one thing to conclude at the end of a sequence of ingenious proof−theoretical reductions that the proof theoretic strength of a certain type theory can be identified by such−and−such an ordinal; it is another to actually see its limits emerge by a sequence of exhaustive constructions within the type theory itself.

5.1 Loose ends

I am aware of several shortcomings in the work presented in the previous chapters. Some of these may have been avoidable; I may simply have been unaware of work that is directly relevant, or failed to appreciate work of which I was aware, or failed to pursue thoroughly my own ideas. On the other hand some shortcomings may just be in the nature of things; I may have followed paths that lead more or less obviously to dead ends, or require new ideas beyond my reach. On the grounds that failure can sometimes be as valuable as success[1], for which it is necessary that they are in plain view, in this section I shall draw together some of these shortcomings.

5.1.1 Logical framework

It is clear enough that there are a number of mechanisms which occur again and again in the presentation of deductive systems, which centre round the notion of a *schema*, form or pattern capable of being instantiated in one way or other to obtain an axiom, construction, or inference step that may occur in one or more contexts in a deduction or program. The suspicion or hope is surely justified that it is possible to abstract from the panorama of logical and deductive systems that have arisen in logic and computer science a certain core of general principles that pertain to this notion, and the notions of instantiation and equality between instantiations of different schemata that are connected with it. The idea of such a general logic or meta−logic may have emerged first in the 1930's in publications of Post, Carnap and Tarski. With the rise of computer science, and the renaissance in logic provoked by it, the idea has re−emerged in a new form, called 'Logical Frameworks', as for example in [45] and the collections [51] and [52].

It is difficult to understand *precisely* what problem a logical framework is supposed to solve. There seem to be at least two points of view about it.

The first is that one wants to represent as wide a variety of logics as possible, by representing their *judgements* as types in the logical framework. So for example, if one wants to represent a type theory there will (it seems) be types for each of the judgement forms

$$\begin{array}{ll} A : \mathsf{Type}\ \Gamma & \qquad A = B : \mathsf{Type}\ \Gamma \\ a : \Gamma \rightarrow A & \qquad a = b : \Gamma \rightarrow A \end{array}$$

The framework itself will be as *weak* as possible, and in particular 'agnostic' about computational equations, so that (ideally) one never encounters anything called a logic (a modal, linear, or programming logic) that requires any extension of or disturbance in the framework). A problem with such ambition is that a theory of everything is in danger of being a theory of nothing. A framework that is capable of representing (in a technical sense) logics in general is in danger of being incapable of representing (in a practical sense) any particular one of them. To the extent that I have been able to understand the 'judgements as

[1]In this connection one can think of Cantor, who failed to settle the continuum hypothesis, Frege, who failed to reduce mathematics to logic, Hilbert, who failed to reduce it to combinatorial reasoning.

types' point of view it seems utterly impractical to actually attempt to use such a framework as a setting in which to carry out constructions in a Martin−Löf style type theory. The difficulties here are (apparently) of such a degree that they cannot be dismissed as mere matters of presentation, to be overcome with 'computer assistance'. This may well be a travesty of the 'judgements as types' point of view. There may be a wide spectrum of deductive calculi arising in computer science for which the approach is very apt.

The second point of view is that one is interested not so much in formalising and implementing a framework for logics in general, but rather a framework for logics belonging to a particular 'open−ended' family – such as for example the family to which I have been referring by phrases such as 'type theories in Martin−Löf style'. It is often said that one may add to Martin−Löf's system (whatever that is) new data types according to general patterns of which particular data types such as the natural numbers, or the so−called 'wellordering' datatypes are instances. There is a lot of truth in this. Programmers have hardly learnt yet how to exploit the positive data−types described for example in [61][section 9.2.2]; indeed we have scarcely learnt how to make full use of the data−types available in functional languages such as Haskell or ML. Yet it is *intrinsically* impossible to describe formally a general framework (even allowing undecidable type−checking) which will serve for all the inductive definitions (and more generally reflection principles) that 'we might ever need'. As soon as we have such a purported description, we can diagonalise, to obtain a false negative. There is more to this than a recursion theoretic trick. For example, Anton Setzer's Mahlo universe, and his Π_3−reflecting universe have revealed delicate issues in type−checking fixed−point definitions in T. Coquand's implementations of frameworks for 'type theories in Martin−Löf style', even though these are not formal systems.

The principal shortcoming that permeates chapter 2 is therefore is that (not to put to fine a point on it) I have failed to understand the very concept of a logical framework. This may well be of course purely (and uninterestingly) a problem peculiar to me. It may however indicate a need for a clearer exposition of the basis of the concept by those who do have a firm grasp of it.

Other shortcomings include the following.

- I have chosen to present the logical framework as a dependent type theory over a family of ground types representing a specific type theory. The type theory over the framework contains not only $\prod$ types, but $\sum$ types and even $+$ and finite types $\{\,\}$ and $\{*\}$. These types were characterised by universal properties, in a category theoretic sense. This implies that the rules for equational reasoning at the framework level include not only analogues of the $\eta-$rules for $\prod-$types, but also the so called 'fusion' rules described for example in [9][pages 48,141,42,72,39,37,40]. These seem to be related to the 'permutative contractions' [104][page 139] discussed in proof theoretical literature. The rules for equational reasoning are thus highly extensional[2]. There are almost certainly many of difficult meta−theoretic questions begged by including such type constructions. For example, is it guaranteed that type checking is decidable? This problem has not yet received a satisfactory solution (about which there is general consensus) even in the simpler context of cartesian closed categories, let alone a dependently typed framework.

- To start from a name free substitution calculus, and introduce notational conventions and abbreviations in a wellfounded incremental fashion till one can discuss the kind of constructions that arise in connection with interactive systems is perhaps comparable to starting from the physics of sub−atomic particles and trying to derive the kind of physics necessary to design a shock−absorber for a bus. At any rate, I cannot claim to have accomplished a smooth development.
- I have not been able to describe a uniform schema for inductive definitions adequate for those needed in chapter 3.
- Still less have I been able to describe a framework for coinductive definitions such as those needed in chapter 3.

5.1.2 Transition systems, Interactive structures

The treatment of transition system contains constructions on transition systems analogous to Cantor's operations of addition and multiplication of linear wellorderings. (Multiplication is a special case of the sum of an ordered series, in the same way that $\times$ is a special case of $\sum$.) Conspicuously, I have not offered a counterpart of Cantor's exponential construction, nor of any related construction such as of the multiset ordering. I do not know whether this is because there is some intrinsic difficulty; I simply became overwhelmed by the details, and gave up.

It would have been appropriate to present analogues of the lens constructions in chapter 4 for addition, multiplication and exponentiation of transition systems.

[2] In contrast to 'extensional type theory', this has nothing to do with rules for identity types.

(These would establish that the operations preserve wellfoundedness, but without the use of recursion.) I was unable to bring my efforts in this direction into a presentable form.

It seems to be extremely difficult to show, without the use of generalised inductive definitions that if one is given a wellfounded transition system, then the natural transition system on multisets in the original order is wellfounded. Yet (at least in the case of linear orders) if one starts with an order of type α, then the order type of the multiset order is ω^α. So one would expect it to be possible to show that the multiset order construction preserves wellfoundedness in first order arithmetic, without the use of recursion on proofs of accessibility, or any other form of transfinite recursion. I believe that this question has been considered by Buchholtz, among other proof theorists, but so far with no success.

As regards interactive systems, it may be that one can give a satisfactory model for terminating programs in a type−theoretical context. In terms of the game metaphor one can perhaps model adequately strategies for the player who starts first in a game to drive the player who responds into a deadlock. It is much less clear how to model strategies for the responding player to evade deadlock. This is a specific case of the general problem of accounting for coinductive datatypes and corecursive definitions in type theory. It is clear enough how to deal with infinite streams $[A]$. They can be modelled adequately by functions $\mathbb{N} \to A$ in ordinary (wellfounded) type theory. The question is rather one of *justifying* definitions such as that of the greatest fixed point on a predicate transformer

$$P(a) = (\prod b : B(a))\,(\sum c : C(a,b))\,P(a[b/c])$$

for an interactive system

$$\begin{array}{lcl} A & : & \text{Set}, \\ B & : & A \to \text{Set}, \\ C & : & (\prod a : A)\,B(a) \to \text{Set}, \\ d & : & (\prod a : A, b : B(a))\,C(a,b) \to A \end{array}$$

In section 3.3.3 I have formulated rules which one expect to be justified for reasoning about such a predicate.

Corecursion is an enormously important facility in functional programming. It seems to arise whenever the natural description of a solution to a programming problem is in terms of a notion of state. It is inconceivable that one will be able to regard dependent type theory as a framework for practical programming until the foundations for corecursion are clear. The problem presents itself in a particularly acute form in connection with modelling 'the programmers predicament' in section 3.3.8 on page 86.

It should be added that there has been a good deal of work connected with coinduction in type theory, particularly in the case of simple (non−dependent) type theory and in impredicative type theories.

1. One problem is simply to formulate general schemes for introducing coinductively defined datatypes together with their associated schemes of corecursion.

   The problem is not so great, as one can be guided by 'dualising' standard schemes for inductive definition. For examples of presentations of schemes of inductive definition, one may cite Martin−Löf's early paper [63] (given in the context of a natural deduction presentation of the intuitionistic predicate calculus), the scheme of inductive definition for strictly positive functors given in Luo's book [61], and perhaps most generally the presentation of simultaneous inductive−recursive definition given by Peter Dybjer in [28]. Of course ideas from category theory are particularly relevant here. Just as inductively defined types are analysed categorically as initial algebras for endofunctors on suitable categories of types, so coinductive types can be analysed with the dual notion of terminal coalgebras for such functors. Even inductive−recursive definitions can be treated in this way, as endofunctors on certain slice categories Type/D, where Type is a suitable category of types.

   In the context of dependent type theories, Eduardo Giménez [37] has introduced an extension of the Calculus of Constructions with inductive and co−inductive types. Mendler in [69] has also formulated such schemes in the context of Martin−Löf's type theory with predicative universes. Mendler's formulations are inspired by categorical considerations.

   In the context of simple type theory, there has been a great deal of work, and some quite interesting formulations have been proposed. In particular, one may cite recent work by Uustalu and Vene, for example the papers [107] and [108] . One problem has been to deal properly with the analogues of the 'side arguments' in recursion schemes, that distinguish recursion from mere iteration. Uustalu and Vene have managed (in [106]) to systematise a variety of recursive schemes into a 'cube', analogous to Barendregt's cube of pure type systems. I know of no reason why it should not be possible to extend their work to the context of dependent type theory.

2. When one has formulated general schemes of coinduction in dependent type theory, a further problem is to work out the metatheoretical properties of

such schemes. Of course, one would like to preserve normalisation properties, at least in the sense of that expressions constructed in adherence to one of these schemes should evaluate to weak head−normal form. (This is primarily in order to preserve the decidability of type checking for expressions constructed in adherence to one of these schemes.)

The extension of Coq proposed by Giménez preserves strong normalisation for a lazy computation relation. This extension considerably enlarges the expressiveness of the system, enabling a direct translation of recursive programs, while keeping a relatively simple collection of typing rules. The proof of strong normalisation is by a model construction using saturated sets.

3. The greatest problem of all is not to formulate schemes, or to analyse them theoretically, but to understand them. In this connection, there is not a lot of work which one can cite. There is a paper by Martin−Löf [68] which does not deal directly with coinduction, but rather with non−standard type theory, which is type theory together with a single constant ∞, for an infinite natural number. Although it does not deal directly with coinduction, it refers frequently to the 'primordial' example of a coinductive type, that of the type $S(A)$ of infinite streams of objects of a given type A. Another very penetrating paper, which is frequently referred to in the literature[3] but as far as I am aware has not been published, is T. Coquand's "Infinite Objects in Type Theory". Coquand's idea is to use constructors (rather than destructors) to define coinductive types. In the case of streams this means we have the constructor

$$\mathit{push} : (\textstyle\prod a : A)\ S(A) \rightarrow S(A)$$

We may use recursive definitions to define streams but they have to be *guarded* by constructors. In a sense this is dual to structural recursion for wellfounded types. For example, the definition

$$\begin{array}{l} \mathit{nats} : \mathbb{N} \rightarrow S(\mathbb{N}) \\ \mathit{nats}(n) = \mathit{push}(n, \mathit{nats}(s(n))) \end{array}$$

is guarded because the recursion is 'inside' a constructor. If recursions are guarded, normalisation is preserved provided that reduction does not take place under a constructor (such as *push*). The restriction is quite natural, and analogous to a restriction on reduction that is necessary in the case of

[3]For example it is referred to by Giménez as an inspiration for his work on Coq.

wellfounded types. For example, we can define the recursor on the type of natural numbers by recursion.

$$
\begin{array}{l}
R : (\prod P : \mathit{Nat} \to \mathit{Set}\,)\; P(0) \to \\
\qquad\qquad\qquad\qquad ((\prod n : \mathit{Nat}\,)\; P(n) \to P(s(n)) \to \\
\qquad\qquad\qquad\qquad (\prod n : \mathit{Nat}\,)\; P(n) \\
R = \lambda P, a, b, n.\, \mathbf{case}\ n\ \mathbf{of}\ \ \begin{array}[t]{ll} 0 & \to a \\ s(m) & \to b(m, R(P, a, b, m)) \end{array}
\end{array}
$$

For normalisation, it is clear that reduction must not take place inside the case expression.

These ideas are explored further in C. Coquand's thesis [17].

One way to understand coinduction is to model it in terms of induction and exponential types. Thus, we have the sequence of (rather well−known) natural isomorphisms

$$
\begin{array}{lcl}
\nu X.\, A \times X & = & (\mu X.1 + X) \to A \\
\nu X.\, A \times X^2 & = & (\mu X.1 + 2 \times X) \to A \\
\nu X.\, A \times X^B & = & (\mu X.1 + B \times X) \to A
\end{array}
$$

The intuition here is that where we have a type of infinite objects (for example the type $\nu X.\, A \times X$ of infinite streams of objects of type A), we may analyse it as a function which when applied to a (finite, wellfounded) *position* in such an object, returns the information to be found at that position. By extension, if we have an infinite binary tree (an object of type $\nu X.\, A \times X^2$) it may be represented as an A−valued function from strings of bits (an object of type $(\mu X.1 + 2 \times X) \to A$). This representation, or analysis, can be expressed by giving isomorphisms.

$$
\begin{array}{lcl}
\mathit{decode} & : & (\nu X.\, A \times X^B) \to ((\mu X.1 + B \times X) \to A) \\
\mathit{encode} & : & ((\mu X.1 + B \times X) \to A) \to (\nu X.\, A \times X^B)
\end{array}
$$

The isomorphism from the coinductive type to the exponential can be defined as follows.

$$
\mathit{decode}\,(a, f) = \lambda bs.\, \mathbf{case}\ bs\ \mathbf{of}\ \begin{array}[t]{ll} \\ \mathrm{i}\,0 & \mapsto a \\ \mathrm{j}(b, bs') & \mapsto \mathit{decode}(f(b), bs') \end{array}
$$

The inverse of *decode* can be defined as follows.

$$
\mathit{encode}\, f = (f(\mathrm{i}\,0), \mathit{encode}(\lambda bs.\, f(\mathrm{j}(b, bs))))
$$

It is tempting to think that a 'position' is always a finite list of some kind.

Something that may be a little surprising is the following isomorphism.

$$
\begin{array}{lcl}
\mathit{decode} & : & (\nu F.\, \lambda X.\, X \times F^2(X))(A) \to ((\mu X.1 + X^2) \to A) \\
\mathit{encode} & : & ((\mu X.1 + X^2) \to A) \to (\nu F.X \times F^2(X))(A)
\end{array}
$$

Here the 'positions' are not lists, but finite binary trees. A suitable isomorphism may be defined as follows.

$$\begin{array}{lcl} decode((a,t),x) & = & \textbf{case}\ t\ \textbf{of}\ \ \mathrm{i}\,0 \qquad\ \to a \\ & & \qquad\qquad\quad \mathrm{j}(t_1,t_2) \to decode(decode(x,t_1),t_2) \\ encode(f) & = & (f(\mathrm{i}\,0), encode(\lambda t_1.\, encode(\lambda t_2.\, f(t_1,t_2)))) \end{array}$$

In fact, the isomorphism may be generalised considerably.

$$\begin{array}{c} (\nu\, F.\, \lambda X.\, X^{C_0} \times (F(X))^{C_1} \times \ldots (F^k(X))^{C_k})(A) \\ (\mu.\, C_0 + C_1 \times X + \ldots + C_k \times X^k) \to A \end{array}$$

It is as if the 'logarithm' of $(\nu\, F.\, \lambda X.\, X^{C_0} \times (F(X))^{C_1} \times \ldots (F^k(X))^{C_k})(A)$ (to the 'base' A) is $(\mu\, X.\, C_0 + C_1 \times X + \ldots + C_k \times X^k)$. The 'positions' in the infinite object are in this case objects of a polynomial data type.

In certain cases, as illustrated above in the context of simple type theory, we can analyse or represent non−wellfounded objects in terms of functions defined on wellfounded objects. It is however extremely challenging to extend this analysis to dependent types. For example, how should be analyse the following coinductive analogue of the W−type? The objects of this type are *non*−wellfounded trees in which the nodes are labelled by objects from type A, and the branching index of a node with label $a : A$ is the type $B(a)$.

$$\nu X.(\textstyle\sum a : A)\ X^{B(a)}$$

The prospects of extending the analysis even further to such a coinductive type as the predicate *Pos* discussed in section 3.3.3 seem at present quite remote.

Another problem with an analysis of infinite objects of the type illustrated above, besides its somewhat limited scope, is that although there are (in general several) isomorphisms between ν types and exponential types of the kind illustrated above, the computational behaviour of objects which correspond under the isomorphisms is quite different.

I am very grateful to Thorsten Altenkirch and Martin Wehr for discussion on these matters. The insight that 'positions' in infinite structures are not always lists is due to Altenkirch. He has extended the isomorphism beyond polynomial data types to 'regular' data types that are additionally closed under a 'μ' operator.

It should be mentioned that a quite different approach to the analysis of infinite objects in type theory may be attempted by treating their types as inverse limits. Such an approach is taken by Lindström in [59]. It depends heavily on a generic notion of equality, not yet available in a framework such as `half` or `Agda`.

5.1.3 Lenses

As regards shortcomings in chapter 4 about the notion of lens, those of which I am aware include the following.

- It would have been far preferable to have given a 'literate' presentation of the formal constructions on which chapter 4 is based. The raw code is presented in appendix C.

- I should like to have made a more extensive study of the kind of predicate transformers which emerge when establishing lower bounds on the proof theoretic strength of type theories. In particular, I should like to have begun to test the idea of 'lens' in elucidating lower bounds on the proof−theoretic strength of type theories incorporating new kinds of universe rules that have arisen in the past decade in the work of Setzer, Rathjen and Palmgren. (I have however been able to conjecture lower bounds on the strength of the system with an a single superuniverse, an external sequence of superuniverses, and other simple extended universes that are correct according to results of Rathjen.) A problem here seems to be that the notion of lens is not yet in a suitably general form; this is shown to some extent already in the proof in appendix C; there is no 'Veblen lens'.

- I should have liked to be able to demonstrate a very simple technique for directly establishing *upper* bounds on the proof−theoretic strength of type theories, at least in the case of Gödel's T. There are perhaps grounds for hope that such a technique may not be far away. (These are indicated in appendix A.)

- As suggested in the final section of the introductory chapter, there is in my opinion a genuine need to explain the interest of ordinal theoretic proof theory in terms which are relevant to the beginning of this century, rather than the the last one. I had hoped to have outlined an example of the kind of explanation I believe to be necessary. I can only suggest that this kind of proof theory, concerned as it is with the extent to which a given formal system can accommodate proofs of wellfoundedness may shed some light on the implications of choosing a particular type theory in which to write interactive programs.

  The naïve idea which has from the beginning been at the back of my mind was that the capacity of a theory to write (provably) terminating programs must also manifest itself as a limitation on the complexity of (provably) non−terminating (or deadlock−avoiding) programs. In some sense, these are graphs that are most naturally expressed using labels and (terminating) programs associated with and yielding such labels. Perhaps (I had hoped) it is possible to formulate precisely a notion of complexity for potentially non−terminating programs, and establish a link with the more familiar (if still rather mysterious) notion of proof−theoretic strength. Nailing this down was beyond me.

Appendix A

Arithmetical Combinators

A.1 Introduction

There are many combinatorially complete sets of combinators. These are 'instruction sets' to which the λ-calculus can be 'compiled'. The most famous are perhaps $\{S, K\}$ and $\{B, C, K, W\}$. The latter set of combinators is sometimes preferred because the first two of its combinators are affine (*i.e.* use their arguments exactly once), the first three are 'linear' (*i.e.* use their arguments at most once), and W alone uses an argument more than once. For these reasons, they prove convenient for the analysis of systems of linear logic. Several authors have observed that there is also a set of 'arithmetical' combinators $\{A, M, E, N\}$[1] arising from the Church numerals, which is combinatorially complete. For example, this has been observed by Stenlund [101] page 21, Schwichtenberg [104] page 19, Burge [10] page 35, Rosenbloom [93] page 122, Church [14] page 10, and Fitch (according to Stenlund). These combinators share the advantage of $\{B, C, K, W\}$ that they mesh well with concerns about linearity. However their arithmetical combinators are improvable. In this appendix I define, and show the combinatorial completeness of, a set of combinators $\{(+), (\times), (\hat{}), 0\}$ which perhaps better deserve to be called the 'arithmetical combinators', because they arise naturally from a certain calculus of iterative exponents.

Over the last 50 years several systems of ordinal notations have been devised, based upon functionals of higher type over the ordinals, in which this calculus of iterative exponents plays a central rôle. The simplest of these is a notation system for ϵ_0 that uses 'ω-iteration' functionals of finite type, that are easily defined in Gödel's system T. It is natural to wonder whether the combinatorial completeness of the arithmetic combinators provides a clue for a new way of establishing an upper bound for the proof theoretic ordinal of this system. The final section of

[1] Surely, the last word in combinators!

this appendix contains some remarks on this possibility.

A.2 The calculus of iterative exponents

Suppose we write application backwards, with the function after the argument, as mathematicians sometimes do. The notation is usually called 'algebraic notation', but I will refer to it as 'exponential notation', for reasons that will soon emerge. Perhaps it is congenial if you visualise data as starting on the left and flowing through successive functions towards the right.

Let us use $\_{1}\, \hat{}\, \_{2}$ as an infix notation for application written backwards. (Juxtaposition $(\_{2}\ \_{1})$ is more often used in real life, or at least in functional languages, so we need some visual clue to or sign of the reverse operator.) The symbol is chosen to evoke 'exponential' associations, as we shall shortly introduce the familiar arithmetical operators $\times$, 1, $+$, 0. We take all binary operators to be right associative, so that for example $a\,\hat{}\,b\,\hat{}\,c$ is bracketed $a\,\hat{}\,(b\,\hat{}\,c)$, corresponding to the left associativity of the juxtaposition operator.

In connection with the normal application notation the need soon emerges to introduce the normal composition notation, with its unit 1. In connection with exponential notation, for mirror-image reasons it is just as urgent to introduce a multiplication operator $\_\times\_$, which is the transpose of the usual composition operator.

$$\begin{array}{lcl|lcl} (f \circ g)\ a & = & f\ (g\ a) & a\,\hat{}\,(g \times f) & = & (a\,\hat{}\,g)\,\hat{}\,f \\ 1\ a & = & a & a\,\hat{}\,1 & = & a \end{array}$$

What is more, on the arithmetical side there is good use for the addition operator $\_+\_$ and its neutral element 0. There does not seem to be any common notation for their counterparts in the normal column, so I shall use the operator $\_\hat{\circ}\_$, and constant 0:

$$\begin{array}{lcl|lcl} (f \mathbin{\hat{\circ}} g)\ a & = & f\ a \circ g\ a & a\,\hat{}\,(g + f) & = & a\,\hat{}\,g \times a\,\hat{}\,f \\ 0\ a & = & 1 & a\,\hat{}\,0 & = & 1 \end{array}$$

Let us take the 'ζ' rule for granted:

$$\frac{x\,\hat{}\,a = x\,\hat{}\,b}{a = b} \text{ where } x \text{ is fresh to } a \text{ and } b$$

This is a mild form of extensionality, or perhaps exponentiality. Using it, we can prove from the definitions the following algebraic facts:

- $(\_\times\_, 1)$ forms a monoid: i.e. $\_\times\_$ is associative, and has 1 as a left and right neutral element.

```
-- 'take over' operators
import Prelude hiding ((*),(^),(+))

-- usual precedence
infixr 8 ^
infixr 7 *
infixr 6 +

m + n       = \f -> f ^ m * f ^ n
m * n       = \f -> (f ^ m) ^ n
m ^ n       = n m
nop f z     = z
```

Figure A.1: Haskell code for the arithmetical combinators.

- $(\_+\_, 0)$ forms a monoid.
- $\_\times\_$ distributes over $\_+\_$ from the left:

$$\begin{array}{lcl} a \times (b+c) & = & a \times b + a \times c \\ a \times 0 & = & 0 \end{array}$$

Curiously, these laws[2] are (roughly[3]) the main ones that continue to hold when arithmetic is extended to arbitrary linear orderings (wellfounded or not). Other laws that hold only in finite arithmetic fail. For example, the commutativity of multiplication fails because it says that order of composition does not matter.

It is simple to translate arithmetical notation into Haskell. The code is in figure A.1 on page 134. This gives us the possibility of using (+), (*), and (^) as if they were 'numbers', that *represent* the arithmetical operations in the sense that the following laws hold:

$$\begin{array}{lcl} b\,\hat{}\,a\,\hat{}\,(\hat{}) & = & a\,\hat{}\,b \\ b\,\hat{}\,a\,\hat{}\,(\times) & = & a \times b \\ b\,\hat{}\,a\,\hat{}\,(+) & = & a + b \end{array}$$

These are the 'arithmetical combinators' of Stenlund [101] page 21, Schwichtenberg [104] page 19, Burge [10] page 35, Rosenbloom [93] page 122, Church [14] page 10, and Fitch, except for a small point. All these authors define the multiplication combinator to be the transpose of $(\times)$, namely $(\circ)$, which equals the classic 'B' combinator. Their addition combinator is correspondingly different. (It is $\_\hat{\circ}\_$.) My definitions have the authority of Cantor, who appreciated exponential notation, and said (after working with them for a while) that laws such

[2] I have seen such a structure called a 'half-ring'. See [22].

[3] We do not have $1\,\hat{}\,a = 1$, nor $0 \times a = 0$.

as:

$$a\hat{}(f \times g) = (a\hat{}g)\hat{}f$$

were 'repulsive' (*abstoßende* in German) [4] [5].

A.3 Completeness

That the arithmetical combinators are functionally complete, means that we can introduce λ-abstraction as a façon-de-parler, and compile untyped lambda terms into equivalent applicative terms ('code') with only the arithmetical combinators. Because of the arithmetical setting, it is natural to use logarithmic notation '$\log_x b$' instead of '$\lambda x.b$'.

To compile into arithmetical combinators, first consider *affine* λ-abstraction $\log_x b$, where the bound variable x may have at most one occurrence in the body b. The following cases are typical. I have used 'section notation', replacing $a\hat{}(\hat{})$ by $(a\hat{})$, and so on.

- $\log_x x = 1$. (1 can be defined as $\mathsf{Junk}\hat{}0$, where Junk is anything, *e.g.* 0.)
- $\log_x (x\hat{}f) = f$, if x does not occur in f. (The 'η' rule.)
- $\log_x (a\hat{}x) = \log_x (x\hat{}(a\hat{})) = (a\hat{})$, if x does not occur in a.
- $\log_x (a_k\hat{}\cdots\hat{}a_1\hat{}M\hat{}a) = (\log_x M) \times a \times (a_1\hat{}) \times \cdots \times (a_k\hat{})$, if x does not occur in $a_1, \ldots, a_k, a$.
- $\log_x a = 0 \times (a\hat{})$, if x does not occur in a.

Here is arithmetical code for the classical (Shönfinkel[6]) combinators C, B and K.

$$\begin{array}{rcl} C & = & (\times) \times ((\hat{})\times) \\ B & = & (\times)\hat{}C \\ K & = & 0\hat{}C \end{array}$$

The compilation of non-affine terms is handled by the first of the following two 'laws of logarithms'.

- $\log_x (M \times N) = (\log_x M) + (\log_x N)$.

[4] I learnt this from page 120 of Potter's [86].

[5] I suggest that a correct definition of the multiplication combinator can be read into Wittgenstein's Tractatus [116], at or around remarks 6.241, 6.02, and 6.021. Wittgenstein used exponential notation in connection with iteration of operations, and in the 1910's thought of numbers *à la* Church (to put it anachronistically).

[6] He called them T, Z and C respectively.

- $\log_x 1 = 0$.

As for the W combinator[7], consider first it's transpose:

$$W' = \log_y(\log_x(y\,\hat{}\,y\,\hat{}\,x))$$

Using the laws above for linear logarithms, one has $W' = \log_y((y\,\hat{}\,) \times (y\,\hat{}\,))$. Then, by the first of the 'laws of logarithms' mentioned above,

$$W' = (\,\hat{}\,) + (\,\hat{}\,) \quad ,$$

so

$$W = ((\,\hat{}\,) + (\,\hat{}\,))\,\hat{}\,C \quad .$$

Since C plays such a key rôle in the translations of B, K and W, it may be worth noting that

$$\begin{array}{lcl} a\,\hat{}\,C & = & (\,\hat{}\,) \times (a\times) \\ b\,\hat{}\,a\,\hat{}\,C & = & a \times (b\,\hat{}\,) \quad . \end{array}$$

To compile a logarithmic expression into arithmetical code you can always do it by 'brute force', that is by translating it first to affine form by introducing W combinators and then by compilation of the resulting affine term. (In particular cases you can sometimes do better, as is shown by the second of the two 'laws of logarithms' shown above.)

The translation works just as well for the simply typed λ-calculus. The types of the arithmetical combinators are as follows.

$$\begin{array}{rcl} (\,\hat{}\,) & : & A \to (A \to B) \to B \\ (\times) & : & (C \to A) \to (A \to B) \to C \to B \\ (+) & : & (D \to C \to A) \to (D \to A \to B) \to D \to C \to B \\ 0 & : & A \to B \to B \end{array}$$

It is noteworthy that all affine terms (that is, terms which do not contain $(+)$) are well-typed. There is no way in which the Hindley-Milner type inference algorithm can fail for such terms, since it is impossible that non-unifiable type schemas can be inferred for two occurrences of the same variable.

The exponential function entails a certain shift in type. You can see this if you think of numbers as iterating operations – for example, handing another coin to a shopkeeper. Doing such a thing $n + m$ times is just doing it n times, and then m more. Doing a thing $n \times m$ times is doing m blocks in each of which it it is done n times, as when handing over m piles which are each of n coins. Doing a thing n^m times also involves doing something m times, but now the operation is

[7] Judging by Stenlund [101] p 22, the following definition is (essentially) due to F. B. Fitch.

(even) higher order. It is the operation on a arbitrary concrete operation which iterates it n times. (There is also a certain 'gearing-up' involved in the step from addition to multiplication, since we have to think of the operation as something that can be replaced by blocks or multiples of itself.)

The laws given above show how 'logarithms' interact with exponentiation, and general multiplication (where the 'base' occurs in both factors). It may be of some interest to see how it interacts with multiplication by 'constants'. The following are representative.

- $\log_x (a \times M) = (\log_x M) \times (a\times)$ if x does not occur in a.
- $\log_x (M \times a) = (\log_x M) \times (\times) \times (a\hat{})$ if x does not occur in a.
- $\log_x (b + a \times x) = (a\times) \times (b+)$ if x does not occur in a or b.
- $\log_x (b + x \times a) = (\times) \times (a\hat{}) \times (b+)$ if x does not occur in a or b.
- $\log_x (a \times x + b) = (a\times) \times (+) \times (b\hat{})$ if x does not occur in a or b.
- $\log_x (x \times a + b) = (\times) \times (a\hat{}) \times (+) \times (b\hat{})$ if x does not occur in a or b.

Another observation that may have some interest is that the usual pairing $(a, b) = \log_c (c\ a\ b)$ comes out as $(a\hat{}) \times (b\hat{})$, with projections $(K\hat{})$ and $(0\hat{})$.

A.4 A clue for ordinal analysis of Gödel's T?

Systems of ordinal notations based on functionals of higher type over the ordinals seem to have been studied first by W. Neumer, in a sequence of articles [73] [74] [75] [76] [77], [78] [79] published in the 1950's. The functionals that Neumer was concerned with involved a kind of diagonalisation akin to Veblen's derivative operator. He called these functionals 'facients'. Similar systems were developed by Peter Aczel, who was influenced by Neumer's papers, and certain ideas of S. Feferman, reported in a paper [30] called 'Hereditarily replete functionals over the ordinals'). Aczel's systems are described in [3]. Feferman's ideas were developed further by Weyrauch [113]. At this time, it was hoped that functionals of higher type would prove a viable alternative to Bachmann's use of higher number classes in the development of notation systems for large ordinals. Bachmann's approach (involving explicit assignment of fundamental sequences) had been extended by Pfeiffer and Isles to the very boundaries of human endurance. Unfortunately, the ordinals obtained by the use of functionals of higher type were not large; different ideas due to Aczel, Bridge, Bucholtz and Feferman soon emerged for the description of vast initial segments of the countable ordinals (still making use of higher number classes). These set the basis for the ordinal notation systems which are in use today. Interest in ordinal functionals of higher type apparently subsided in the early 70's.

Aczel's paper [3] describes two hierarchies of functionals (with the same type hierarchy): a 'simple' one in which the functionals express a form of ω-iteration, and a more powerful one in which the functionals express a form of diagonalisation, akin to Veblen's derivative functional. The simpler hierarchy was the inspiration for my conjecture at that time about the proof theoretic strength of Martin-Löf's system, as it seemed that it should in principle be possible to formalise the hierarchy in that system. (In fact, his second hierarchy also suggests what the ordinal should be if one were to extend the type theory by a non-iterated generalised inductive definition: namely the ordinal $\phi_{F_{\Omega_2}(0)}(0)$ in the notation used by Aczel, an ordinal somewhat above the Bachmann-Howard ordinal.) I also began to look for an 'interpretation' of Gödel's T using iteration functionals of finite type. Some initial investigations showed that the idea of such an 'interpretation' could be carried through for primitive recursive arithmetic with induction restricted to ground types, the corresponding system over the constructive second number class, and indeed some artificial fragments of Gödel's T. What I wanted was to find for the subject of proof-theoretic ordinal analysis something comparable to the method of computability arguments at the time developed by Tait, Howard, Girard, Martin-Löf, Prawitz, *et al.*. However I failed to find a suitable notion of 'interpretation' applicable to Gödel's T itself.

Recently, systems of functionals of finite type over the ordinals have again been reconsidered by Danner [20] and Leivant, and in unpublished work of Simmons. Their results have the same flavour as those of Schwichtenberg and Statman on numeric functions representable in the simply typed lambda calculus. Interest in ordinal functionals seems to be awakening again.

In view of the combinatorial completeness of the arithmetical combinators, it is difficult to resist the notion that there may after all be a suitable sense of 'interpretation' in which Gödel's T can be interpreted in a system of ω-iteration functionals over the ordinals, in such a way as to give a simple analysis of its proof theoretic ordinal. After all, we have an interpretation, even a compilation of the simply typed λ-calculus into a system of combinators based on four of the five ingredients of Cantor Normal Form, namely 0, $(+)$, $(\times)$, and $(\hat{\ })$. That leaves only ω, and surely there is some way of introducing a combinator with that name so as to interpret the recursor R? To my intense frustration, a solution still evades me. The rest of this section indicates how I have approached the problem.

The approach I have taken is to consider extending the simply typed λ-calculus with *streams*; that is, I have added a unary type constructor written $[A]$ for the type of infinite streams of objects of type A, together with combinators for certain pointwise operations on streams, and (crucially) a form of 'cumulative product' over streams of functions.

A.4.1 Construction and deconstruction

We add constants for pushing something on the front of a stream, and operators $\_{}_0$ and $\_'$ for taking the head and tail.

$$\frac{a : A \qquad as : [A]}{(a, as) : [A]}$$

$$\frac{a : [A]}{a_0 : A} \qquad\qquad \frac{as : [A]}{as' : [A]}$$

$$\begin{array}{rcl} (a, as)_0 & = & a \\ (a, as)' & = & as \end{array}$$

A.4.2 Pointwise operations

We add a constant ! ('bang') for a form of infinite repetition, and operators $[\hat{\ }]$, $[\times]$ and $[+]$. The operator $[\hat{\ }]$ represents pointwise application of a stream of functions to a stream of arguments, while $[\times]$ and $[+]$ are similarly pointwise liftings of $\times$ and $+$.

$$\frac{a : A}{a! : [A]}$$

$$\frac{as : [A] \qquad fs : [A \to B]}{as[\hat{\ }]fs : [B]}$$

$$\frac{as : [A \to B] \qquad bs : [B \to C]}{as[\times]bs : [A \to C]}$$

$$\frac{as : [A \to B \to C] \qquad bs : [A \to C \to D]}{as[+]bs : [A \to B \to D]}$$

These constants are postulated to satisfy the following equations.

$$\begin{array}{lcl@{\qquad\qquad}lcl}
(a!)_0 & = & a & (a!)' & = & a! \\
(as[\hat{}]fs)_0 & = & as_0\hat{}fs_0 & (as[\hat{}]fs)' & = & as'[\hat{}]fs' \\
(as[\times]bs)_0 & = & as_0 \times bs_0 & (as[\times]bs)' & = & as'[\times]bs' \\
(as[+]bs)_0 & = & as_0 + bs_0 & (as[+]bs)' & = & as'[+]bs'
\end{array}$$

A.4.3 Cumulative product

$$\frac{fs : [A \to A]}{\Pi fs : [A \to A]}$$

This combinator is stipulated to satisfy the following equations.

$$\begin{array}{lcl}
(\Pi fs)_0 & = & 1 \\
(\Pi fs)' & = & (fs_0!)[\times](\Pi fs')
\end{array}$$

The motivation behind the equations for Πf may become apparent if one contemplates the equation

$$\Pi(f_0, f_1, f_2, \ldots) = (1, f_0, f_0 \times f_1, f_0 \times f_1 \times f_2, \ldots)$$

and bears in mind that $f \times g$ equals $g \circ f$.

Note that if we order the arguments of the recursion combinator (in Gödel's T) as follows:

$$R : (N \to A \to A) \to N \to A \to A$$

with equations

$$\begin{array}{lcl}
R(b, 0, a) & = & a \\
R(b, S(n), a) & = & b(n, R(b, n, a))
\end{array}$$

then we can rewrite the equations in the form

$$\begin{array}{lcl}
R(b, 0) & = & 1 \\
R(b, S(n)) & = & R(b \circ S, n) \circ b(0) \\
 & = & b(0) \times R(S \times b, n)
\end{array}$$

In view of these calculations, it is natural to suppose that the Π operation will be what one needs to interpret the recursion combinator.

A.4.4 Cumulative sum

It is natural to propose also (or at least hold in reserve) a cumulative sum operator.

$$\frac{fs : [A \to B \to B]}{\Sigma fs : [A \to B \to B]}$$

This combinator would satisfy the following equations.

$$\begin{array}{rcl}(\Sigma fs)_0 & = & 0 \\ (\Sigma fs)' & = & (fs_0!)[+](\Sigma fs')\end{array}$$

Surely if anything deserves to be called ω it is $\Sigma(1!)$.

Appendix B

Formal proofs of accessibility of ϵ_0

In 4.2.4.2 a proof of accessibility of notations for ordinals below ϵ_0 was described in intuitive terms. In this appendix there is a formal proof, expressed in the language accepted by the type checker 'Agda', designed and implemented by the Programming Logic group at the Department of Computing Science, Göteborg University and Chalmers University of Technology, Göteborg, Sweden. A web page for the system can currently (April 2000) be found using the following URL.

`http://www.cs.chalmers.se/~catarina/agda`.

Agda is quite similar to the programming language 'Cayenne', devised by Lennart Augustsson, and currently accessible via the following URL.

`http://www.md.chalmers.se/~augustss/cayenne/index.html`

It is possible to create Agda scripts using a structured editor called 'Alfa', that can display the material in a variety of styles and notations.

The proof was originally written using an earlier experimental type checker from Chalmers called 'half'. Agda has a number of sophisticated features which are not used here.

Concerning the proof, which is quite small, there are two points to note.

- There is no use of transfinite recursion. The only uses of recursion are over the (finitary) datatype of notations, and the iterator and recursor over the natural numbers. (It should be possible to code the notations as natural numbers; I have not tried it.)

- The proof of the final theorem requires closure of the universe of sets under the predicate transformer `G`. On the other hand, each instance of the theorem (for a specific ordinal notation) can be proved without the use of a universe.

```
{- [2000-04-07 21:18:43]
   Proof of accessibility of a system of notations
   for epsilon_0 .
-}

{- Datatype of notations for epsilon_0 -}
AE :: Set  = data O | W (a :: AE) (b :: AE)

{- some less barbarous notation, so we can use infix -}
w (a :: AE) (b :: AE) :: AE = W@_ a b

{- zero  -}
zeroAE :: AE = O@_
{- succ -}
succAE (a :: AE) :: AE = a `w` zeroAE

{- Some auxiliary data structures -}

{- empty set -}
N0 :: Set = data
{- ex falso quodlibet -}
efq (X :: Set) (x :: N0) :: X = case x of {}

{- singleton set -}
N1 :: Set = data n0
{- its element -}
star :: N1 = n0@_

{- natural numbers -}
Nat :: Set = data Z | S (p :: Nat)

{- some less barbarous notation -}
zero :: Nat = Z@_
succ (p :: Nat) :: Nat = S@_ p

{- recursion on Nat -}
Rec (X :: Nat -> Set)      -- predicate
```

```
    (x :: X zero)          -- basis
    (f :: (n :: Nat)-> X n -> X (succ n))  -- step
    (n :: Nat) :: X n
  = case n of (Z)   -> x
              (S p) -> f p (Rec X x f p)

{- iteration of operations on AE -}
It (f :: AE -> AE) (n :: Nat) (a :: AE) :: AE
  = Rec (\(_::Nat)->AE) a (\(_::Nat)->f) n

{- Cofinality of a notation -}
C (x :: AE) :: Set
  = case x of (O)     -> N0
              (W _ b) -> case b of
                           (O)     -> N1
                           (W _ _) -> Nat

{- Immediate predecessors of a notation -}
pd (a :: AE) (t :: C a) :: AE
  = case a of
      (O)       -> case t of {}
      (W a1 a2) -> case a2 of
           (O) -> a1
           (W b1 b2) -> case b2 of
                 (O) -> It (\(x::AE) -> x `w` b1) t a1
                 (W _ _) -> a1 `w` (pd a2 t)

Pred :: Type = AE -> Set
PT :: Type   = Pred -> Pred

{- PT induced by transition structure -}
B :: PT = \(X :: Pred) -> \(a :: AE)->
          (t :: C a) -> X (pd a t)

{- Progressivity of a predicate -}
Prog (X :: Pred) :: Set  = (a :: AE) -> B X a -> X a
```

```
{- Accessibility of a notation with respect to
   a given predicate -}
Acc :: PT = \(X :: Pred) -> \(a :: AE) -> Prog X -> X a

{- Gentzen's PT -}
G :: PT = \(X :: Pred) -> \(b :: AE) ->
          (a :: AE) -> X a -> X (a `w` b)

{- G preserves progressivity -}
lemma (X :: Pred) :: Prog X -> Prog (G X)
  = \(p :: Prog X) ->
    \(b :: AE) ->
    \(h :: B (G X) b) ->
    \(a :: AE) ->
    \(xa :: X a) ->
    let  arg :: B X (a `w` b)
          = case b of
            (O) -> (\(t::N1) -> xa)
            (W b1 b2) ->
             case b2 of
               (O)  -> let f :: AE -> AE
                             = \(x :: AE) -> x `w` b1
                           itf (n :: Nat) :: AE
                             = It f n a
                       in Rec (\(n::Nat) -> X (itf n))
                              xa  (\(n::Nat) -> h star (itf n))
               (W _ _) -> (\(t::Nat) -> h t a xa)
    in p (a `w` b) arg

{- All notations are accessible -}
{- RECURSIVE over AE, with a large predicate -}
theorem (a :: AE) (X :: Pred) :: Acc X a
  = \(pX :: Prog X) ->
    case a of
      (O)       -> pX zeroAE (\(t::N0)->efq (X (pd zeroAE t)) t)
      (W a' b') -> theorem b' (G X)
(lemma X pX) a' (theorem a' X pX)
```

Appendix C

Formal development of theory of lenses

In 4.3.2 the basic theory of lenses was sketched. In this appendix there is a formal development of this theory, expressed in the language accepted by the type checker 'Agda'. (See appendix B for further information on this system.)

The material is split into several files, listed in separate sections. There is at least one non-presentational problem with it of which I am aware.

- I have used a notation system different from the one described in chapter 4. The system is perhaps 'morally' the same as the one in the body of the thesis, but the problem of justifying its use remains. I have used a different version because of its simplicity. It seems that use of the 'official' system of notations (set out in section C.4) would precipitate an explosion in the size of the code.

  The crucial thing is that the use of transfinite recursion has been avoided, except for three instances used in setting up the notation system. In the proofs of accessibility, there is no use of transfinite recursion. This has to be verified by inspection. (It might have been possible to arrange the code so as to obtain greater assurance of this.)

Of course, the presentation is not particularly lucid, and the code style not at all what I would wish. (It was ported rather rapidly from another proof checker which had 'stopped working'.)

C.1 Basic amenities

The following file defines some basic notions, many (but not necessarily all) of which are used throughout the code.

```
{- logical constants, and common amenities -}

package Amenities where

  Fam (A :: Type) :: Type
    = sig { I :: Set ; i :: I -> A }

  Fam_map (A, B :: Set) (f :: A -> B)
    :: Fam A -> Fam B
    = \(h::Fam A) ->
      struct  I = h.I
              i = \(i::I) -> f (h.i i)

  Pow (A :: Type) :: Type
    = A -> Set

  Pow_map (A, B :: Set) (f :: A -> B)
    :: Pow B -> Pow A
    = \(X :: Pow B) ->
      \(a :: A) ->
      X (f a)

  Rel (A, B :: Type) :: Type
    = A -> Pow B

  {--------- functions ---------------------}

  Pi  (A :: Set) (B :: Pow A) :: Set = (a::A) -> B a
  arr (A :: Set) (B :: Set) :: Set
      = Pi A (\(_::A) -> B)
  op  (A :: Set) :: Set = A `arr` A
  op2 (A :: Set) :: Set = A `arr` op A

  implies :: Set -> Set -> Set = arr

  {---------- multiplicative things ----------}
```

```
Si  (A :: Set) (B :: Pow A) :: Set
  = sig fst :: A ; snd :: B fst

and (A, B :: Set) :: Set
    = Si A (\(_::A) -> B)
and2 (A :: Set) (B :: Set) (C :: Set) :: Set
    = (A `and` B) `and` C

andIn (A, B :: Set) :: A -> B -> A `and` B
    = \(a::A) -> \(b::B) ->
      struct  fst = a
              snd = b
andElimL (A, B :: Set) :: A `and` B -> A
    = \(h::and A B) -> h.fst
andElimR (A, B :: Set) :: A `and` B -> B
    = \(h::and A B) -> h.snd

sigIn (A :: Set) (B :: Pow A)
    :: (a :: A) -> B a -> Si A B
    = \(a::A) -> \(b::B a) ->
      struct  fst = a
              snd = b

sigOutL (A :: Set)
        (B :: Pow A)
        (h :: Si A B)
        :: A
    = h.fst
sigOutR (A :: Set)
        (B :: Pow A)
        (h :: Si A B)
        :: B (sigOutL A B h)
    = h.snd

split (A :: Set) (B :: Pow A) (C :: Pow (Si A B))
      (c :: (a :: A) -> (b :: B a) -> C (sigIn A B a b))
      (h :: Si A B)
```

```
      :: C h
    = c (sigOutL A B h) (sigOutR A B h)

times :: Set -> Set -> Set = and
pair  :: (A, B :: Set) -> A -> B -> A ‘and‘ B
      = andIn
p0    :: (A, B :: Set) -> A ‘and‘ B -> A = andElimL
p1    :: (A, B :: Set) -> A ‘and‘ B -> B = andElimR

{---------- additive things ---------------}

or (A, B :: Set) :: Set
    = data inl (a :: A) | inr (b :: B)
orInL (A, B :: Set) :: A -> A ‘or‘ B
    = \(h::A) -> inl@_ h
orInR (A :: Set) (B :: Set) :: B -> A ‘or‘ B
    = \(h::B) -> inr@_ h
orElim (A, B, C :: Set)
       :: (A -> C) -> (B -> C) -> A ‘or‘ B -> C
    = \(ha::A -> C) -> \(hb::B -> C) -> \(h0::or A B) ->
      case h0 of
        (inl a) -> ha a
        (inr b) -> hb b

plus  :: Set -> Set -> Set = or
copair:: (A, B, C :: Set) ->
         (A -> C) -> (B -> C) -> A ‘plus‘ B -> C
      = orElim
i0 :: (A, B :: Set) -> A ‘and‘ B -> A = andElimL
i1 :: (A, B :: Set) -> A ‘and‘ B -> B = andElimR

{---------- extremal things ----------------}

{- empty set, absurdity -}
N0 :: Set = data
falsum :: Set = N0
{- ex falso quodlibet -}
```

```
efq (X :: Set) (x :: N0) :: X = case x of {}

{- singleton set, vacuity  -}
N1   :: Set = data n0
verum  :: Set = N1
star :: N1 = n0@_
```

The following file defines the set of natural numbers, their constructors, and the constant for recursion on the natural numbers.

C.2 Natural Numbers

```
package Naturals where

  {- the datatype -}

  Nat :: Set = data Z | S (p :: Nat)
  zero  :: Nat = Z@_
  succ (p :: Nat) :: Nat = S@_ p

  {- A convenient abbreviation. Note, we
     cannot use it to speak of sequences of Types. -}
  Seq (A :: Set) :: Set = Nat -> A

  {- recursion on Nat -}
  Rec (X :: Nat -> Set)
         (x :: X zero)
         (f :: (n :: Nat) -> X n -> X (succ n))
         (n :: Nat)
         :: X n
    = case n of
         (Z)   -> x
         (S p) -> f p (Rec X x f p)

  {- iteration on Nat: f^n -}
  It (X :: Set) (f :: X -> X) (n :: Nat)
      :: X -> X
    = \(x::X)-> Rec (\(_::Nat)->X) x (\(_::Nat)->f) n
```

```
{- tail recursive version -}
ItT (X :: Set) (f :: X -> X) (n :: Nat) :: X -> X
  = It (X -> X)
       (\(g :: X -> X) -> \(x::X) -> g (f x))
       n
       (\(x :: X) -> x)
```

C.3 Next Universe construction

The following file defines the operation which erects a universe over a given family of sets.

```
--#include "amenities.agda"
--#include "naturals.agda"

{- package defining the ordinary kind of ‘universe over’
   operator. -}

{- There is a parameter for the type of ordinal terms -}

package UniverseOver (OT :: Set)
                     ( U :: Set)
                     ( T :: U -> Set) where
  open Amenities use N0, N1, Pi, Si
  open Naturals use Nat, NatIt = It, NatItT = ItT

  mutual

    set :: Set
      = data u | t (x :: U)
             | pi (d :: set) (p :: el d -> set)
             | si (d :: set) (p :: el d -> set)
             | n0code | n1code | natcode
             | otcode

    el :: set -> Set
      =  \ (x :: set) ->
```

```
       case x of
         (u) -> U
         (t x)    -> T x
         (pi d p) -> Pi (el d) (\(x :: el d) -> el (p x))
         (si d p) -> Si (el d) (\(x :: el d) -> el (p x))
         (n0code)  -> N0
         (n1code)  -> N1
         (natcode) -> Nat
         (otcode)  -> OT

otcode  :: set = otcode@_
natcode ::  set = natcode@_
n0code  :: set = n0code@_
n1code  :: set = n1code@_
groundU :: set = u@_
groundT :: el groundU -> set
  = \(x :: el groundU) -> t@_ x

pi  (a :: set) (p :: el a -> set) :: set = pi@_ a p
si  (a :: set) (p :: el a -> set) :: set = si@_ a p

arr (a :: set) (b :: set) :: set = pi a (\(_ :: el a) -> b)
and (a :: set) (b :: set) :: set = si a (\(_ :: el a) -> b)

op (a :: set ) :: set = a `arr` a

pin (p :: Nat -> set) :: set = pi natcode (\(n :: Nat) -> p n)
piot(p :: OT  -> set) :: set = pi otcode  (\(a :: OT ) -> p a)
```

C.4 Official notation system

The following file contains the 'official' definition of the notation system for Γ_0; it is *not* used. It is included here so that the reader will understand why I have used instead (in C.5) another notation system. The sheer size of the definition of the transition structure is horrible, as is the idea that its structure might be repeated again and again in the code. It *may* be that the complexity is only apparent, not real; there might be a simple idea or coding trick which would render use of the ‘official’ notations feasible.

There is another problem beside that of sheer size, that may be more serious. An advantage of the approach I have taken[1] is that I have been able to deal directly with sequences of functions on the ordinal notations, rather than with syntactical structures representing those sequences. It is not entirely clear to me how to re-code the proof to use a more conventional system of ordinal notations.

```
--#include "amenities.agda"
--#include "naturals.agda"

package AEdef  where
  open Amenities use N0, efq, N1, op
  open Naturals  use Nat, NatIt = It

  {- Datatype of notations for gamma_0 -}
  AE :: Set
    = data O
         | W (a :: AE) (b :: AE)
         | V (a :: AE) (b :: AE)

  {- an abbreviation -}
  ItAE :: (f::op AE) -> (n::Nat) -> op AE = NatIt AE

  {- some less barbarous notation, so we can use infix -}
  w (a :: AE) (b :: AE) :: AE = W@_ a b
  v (a :: AE) (b :: AE) :: AE = V@_ a b

  {- zero  -}
  zero :: AE = O@_

  {- succ -}
  succ :: op AE = \ (a :: AE) -> a `w` zero

  {- (naught plus) omega-to-the ... -}
  wexp :: op AE = w zero
```

[1]The idea for this approach comes from a seminar on ordinal notations given by Martin-Löf in the early 1970’s.

```
{- w with its arguments transposeded. (Convenient.) -}
wt (a, b :: AE) :: AE = w b a

{- Cofinality of a notation -}
C (a :: AE) :: Set
  = case a of
        (O)     -> N0
        (W _ b) -> case b of
                     (O)     -> N1
                     (W _ _) -> Nat
                     (V _ _) -> Nat
        (V _ _) -> Nat

{- Immediate predecessors of a notation.
   This enormous definition seems a repellent
   waste of time, energy and paper.  Instead, in the lens
   constructions I have used one or two instances
   of transfinite recursion to define the veblen hierarchy
   on a set Ord of ‘abstract ordinal notations’.
   where O :: Ord, S :: Ord -> Ord, L :: (N -> Ord) -> Ord.
   The question may be asked: isn’t this cheating?
   I’m (almost) sure it is essentially sound.
-}
pd (x :: AE) (t :: C x) :: AE
  = case x of
    (O) -> case t of {}
    (W a b)
        -> case b of
           (O) -> a
           (W b1 b2)
               -> case b2 of
                  (O) -> {- a + w^(b1 + 1) -}
                         ItAE (wt b) t a
                  (W _ _)
                      -> a ‘w‘ pd b t
                  (V _ _)
                      -> a ‘w‘ pd b t
```

```
       (V _ _) -> a `w` pd b t
(V a b)
    -> case a of
       (O) -> {- phi 0 b -}
              case b of
              (O) -> {- phi 0 0 -}
                     ItAE wexp t zero
              (W b1 b2)
                  -> case b2 of
                     (O) -> {- phi 0 (b1 + 1) -}
                            ItAE wexp t (succ (a `v` b1))
                     (W _ _)
                         -> a `v` pd b t
                     (V _ _)
                         -> a `v` pd b t
              (V _ _) -> a `v` (pd b t)
       (W a1 a2)
           -> case a2 of
              (O) -> {- phi (a1 + 1) b -}
                     case b of
                     (O) -> {- phi (a1 + 1) 0 -}
                            ItAE (v a) t zero
                     (W b1 b2)
                         -> case b2 of
                            (O) -> {- phi (a1 + 1) (b1 + 1) -}
                                   ItAE  (v a) t
                                         (succ (a `v` b1))
                            (W _ _)
                                -> a `v` pd b t
                            (V _ _)
                                -> a `v` pd b t
                     (V _ _)
                         -> a `v` pd b t
              (W _ _)
                  -> case b of
                     (O) -> {- phi 1 0 -}
                            pd a t `v` b
```

```
                         (W b1 b2)
                             -> case b2 of
                                (O) -> {- phi l (b1 + 1) -}
                                       pd a t `v` succ (a `v` b1)
                                (W _ _)
                                    -> a `v` pd b t
                                (V _ _)
                                    -> a `v` pd b t
                         (V _ _)
                             -> a `v` pd b t
                  (V _ _)
                      -> case b of
                         (O) -> pd a t `v` b
                         (W b1 b2)
                             -> case b2 of
                                (O) -> pd a t `v` succ (a `v` b1)
                                (W _ _)
                                    -> a `v` pd b t
                                (V _ _)
                                    -> a `v` pd b t
                         (V _ _)
                             -> a `v` pd b t
           (V _ _)
               -> case b of
                  (O) -> pd a t `v` b
                  (W b1 b2)
                      -> case b2 of
                         (O) -> {- phi a (b1 + 1) -}
                                pd a t `v` succ (a `v` b1)
                         (W _ _)
                             -> a `v` pd b t
                         (V _ _)
                             -> a `v` pd b t
                  (V _ _)
                      -> a `v` pd b t
Pred :: Type
    = AE -> Set
```

```
{- predicate transformer -}
PT :: Type
    = Pred -> Pred

{- PT induced by transition structure -}
Below :: PT
    = \(X :: Pred) -> \(a :: AE) ->
      (t :: C a) -> X (pd a t)

{- Progressivity -}
Prog (X :: Pred) :: Set
    = (a :: AE) -> Below X a -> X a
```

C.5 Unofficial notation system

This section contains the system of ordinal notations used in the formal development of the theory of lenses. I shall call these 'formal ordinals'. In essence, this consists in adding the following constants.

- Ord : Set, for a set of ordinals,
- constructors $\mathit{zero} : \mathit{Ord}$, $\mathit{succ} : \mathit{Ord} \to \mathit{Ord}$, $\mathit{limit} : (\mathit{Nat} \to \mathit{Ord}) \to \mathit{Ord}$. (There is no recursor.)
- function constants v, w, and $\nabla : (\mathit{Ord} \to \mathit{Ord}) \to \mathit{Ord} \to \mathit{Ord}$ (written `nabla` in the code), and $\phi : \mathit{Ord} \to \mathit{Ord}$, together with certain computation rules. These rules are instances of general (transfinite) recursion over ordinals.

To take $\nabla : (\mathit{Ord} \to \mathit{Ord}) \to \mathit{Ord} \to \mathit{Ord}$ first, its computation rules are as follows.

$$\nabla(f, \alpha) = \textbf{case } \alpha \textbf{ of } \begin{array}{ll} \mathit{zero} & \to f(\mathit{zero}) \\ \mathit{succ}(\alpha') & \to f(\mathit{succ}(\nabla(f, \alpha'))) \\ \mathit{limit}(\xi) & \to \mathit{limit}(\lambda n.\, \nabla(f, \xi(n))) \end{array}$$

This is a key part of the definition of the Veblen hierarchy. (We have for example that $\phi_{\alpha+1} = \nabla({\phi_\alpha}^\omega)$. Note that ∇ is a 'type 2' functional. Formally, its definition uses 'type 0' transfinite recursion at Ord.[2]

[2] In fact, $\nabla(f, \alpha) = (f \circ \mathit{succ})^\alpha(f(\mathit{zero}))$.

As for w, which denotes $\lambda\alpha,\beta.\,\alpha+\omega^\beta$, the most convenient way to give its computation rules is to give instead those for its (less familiar) transpose $wt = \lambda\alpha,\beta.\,w(\beta,\alpha)$. I write the first argument to wt as a subscript, and use currying.

$$wt_\alpha = \textbf{case}\ \alpha\ \textbf{of}\ \begin{array}{ll} \mathit{zero} & \to \mathit{succ} \\ \mathit{succ}(\alpha') & \to (wt_{\alpha'})^\omega \\ \mathit{limit}(\xi) & \to \lambda\beta.\,\mathit{limit}(\lambda n.\,wt_{\xi(n)}(\beta)) \end{array}$$

The notation $(\_)^\omega$ denotes $\omega-$iteration of functions: $f^\omega(\alpha) = \mathit{limit}(\lambda n.\,f^n(\alpha)$. This kind of iteration can be defined using only recursion on the natural numbers, without transfinite recursion. Note that the definition of wt uses 'type 0' transfinite recursion at Ord.[3]

As for v, which denotes $\lambda\alpha,\beta.\,\phi_\alpha(\beta)$ (based on $\lambda\alpha.\,\epsilon_\alpha$) its computation rules are as follows. I write the first argument as a subscript and use currying.

$$v_\alpha = \textbf{case}\ \alpha\ \textbf{of}\ \begin{array}{ll} \mathit{zero} & \to \nabla(\lambda\alpha.\,wt_\alpha \mathit{zero})^\omega \\ \mathit{succ}(\alpha') & \to \nabla({v_{\alpha'}}^\omega) \\ \mathit{limit}(\xi) & \to \nabla(\lambda\beta.\,\mathit{limit}(\lambda n.\,v_{\xi(n)}\beta)) \end{array}$$

```
--#include "amenities.agda"
--#include "naturals.agda"

{- This package is concerned with the Cantor and
   Veblen hierarchies, in the context of formal ordinals -}

package Ordinals where

  open Amenities use
     Pi, implies , and,
     N1, N0

  open Naturals use
     Nat, It, Rec, Seq

  Ord :: Set = data O
                  | S (a :: Ord)
                  | L (ls :: (n :: Nat)->Ord)

  {- Some less barbarous notation -}
```

[3] In fact, $wt_\beta = (\_^{\,\omega})^\beta\ \mathit{succ}$.

```
zero  :: Ord        = O@Ord
succ  :: Ord -> Ord = S@Ord
limit :: (Nat -> Ord) -> Ord = L@Ord

{------ The transition structure on Ord ---}

{- Cofinality -}
C (a :: Ord) :: Set
  = case a of
      (O)    -> N0
      (S a') -> N1
      (L ls) -> Nat

{- Predecessor -}
pd (a :: Ord) (t :: C a) :: Ord
  = case a of
      (O)    -> case t of { }
      (S a') -> a'
      (L ls) -> ls t

{--------- operations from Ords to Ords ------------}

Op :: Set = Ord -> Ord

{- Composing operations -}
Comp :: Op -> Op -> Op
   = \(f :: Op) -> \(g :: Op) -> \(a :: Ord) ->
     f (g a)

{- Iterating operations, to get a sequence -}
OpIt :: Op -> Seq Op
   = It Ord

{- pointwise limit of a seuence of operations -}
OpLim :: Seq Op -> Op
   = \(fs :: Nat -> Op) -> \(a :: Ord) ->
     limit (\(n :: Nat) -> fs n a)
```

```
{- omega iterating an operation -}
OpItw :: Op -> Op
   = \(f :: Op) -> OpLim (OpIt f)

{------------ Cantor hierarchy -------------------}

{- uses transfinite recursion (in second argument). -}
w :: Ord -> Ord -> Ord
   = \(a :: Ord) ->
     \(b :: Ord) ->
      case b of
        (O)    -> succ a
        (S b') -> OpItw (\(x :: Ord) -> x `w` b') a
        (L ls) -> limit (\(n :: Nat) -> a `w` (ls n))

{- The function a ↦ w^a -}
wexp :: Op = w zero

{------------ Veblen hierarchy -------------------}

{- \a -> (f.(+1))^a (f 0) : an essential part of derivative -}
{- uses transfinite recursion. -}
Nabla :: Op -> Op
   = \(f :: Op) -> \(a :: Ord) ->
     case a of
       (O)    -> f zero
       (S a') -> f (succ (Nabla f a'))
       (L ls) -> limit (\(n :: Nat) -> Nabla f (ls n))

{- derivative of a normal function -}
deriv :: Op -> Op
   = \(f :: Op) -> Nabla (OpItw f)

{- derivative of a sequence of normal function -}
derivl :: (Nat -> Op) -> Op
   = \(sf :: Nat -> Op) -> Nabla (OpLim sf)
```

```
{- Veblen hierarchy -}
{- uses transfinite recursion -}
v :: Ord -> Ord -> Ord
   = \(a :: Ord) ->
      let { pds :: Seq Op =
               case a of
                 (O)    -> OpIt (w a)
                 (S a') -> OpIt (v a')
                 (L ls) -> \(n :: Nat) -> v (ls n)
      } in Nabla (OpLim pds)

veb :: Op
  = \(a :: Ord) -> v a zero

{-------- The landmark ordinals ------------}

epsilon0 :: Ord = veb zero

Gamma0 :: Ord = limit (\ (n :: Nat) ->
    It Ord veb n zero)
```

C.6 The theory of lenses

```
--#include "amenities.agda"
--#include "naturals.agda"
--#include "ordinals.agda"
--#include "nextU.agda"

{- This file, together with the files mentioned
   above, contains a proof of accessibility for Gamma0,
   hence a sequence of proofs of accessibility, one for
   each term in the fundamental sequence for Gamma0.

   Each proof can be expressed
   in a type theory in which there is a next universe
   operator (ie. an external sequence of universes).
```

```
   mapping a family of sets to the least family containing
   it as an internal subfamily which is closed under certain
   operations (pi, sigma, Nat).
   (Admittedly it needs close inspection to see this.)

   Closer inspection should reveal that we can construct
   a proof of accessibility for all notations, and
   with a number of universes depending on the
   levels to which the veblen hierarchy is nested.
   (But this is a little complex to set up.)

   The main result is called Corollary and is the
   last definition of the file.
-}

open Ordinals use
    Ord, zero, limit, succ,
    C, pd,
    OpIt, OpItw, OpLim,
    Nabla,
    w, wexp, epsilon0,
    v, veb, Gamma0,
    deriv, derivl

open Amenities
   use N1, star,
       N0, efq,
       Si, and,
       op, op2,
       Pow, Fam

open Naturals
   use Nat, Seq,
       It, Rec

package small( FS :: Fam Set ) where
```

```
  Ot :: Set = Ord

  Us :: Set = FS.I
  Ts :: Us -> Set = FS.i

  {- small sets -}
  open UniverseOver Ot Us Ts use
      set, el,
      otcode,
      pi, arr,
      pin, piot

  Next :: Fam Set = struct
                      I = set
                      i = el

  {- large sets -}
  open UniverseOver Ot set el use
      SET = set,
      EL = el,
      OT  = otcode,
      ARR = arr,
      PI  = pi,
      PIN = pin,
      PIOT= piot,
      SI  = si,
      AND = and

{------------ predicates ------------------}

  {- The type of real predicates -}
  Pred :: Type  = Ot -> Set

  {- The Set of small predicates -}
  pred :: Set   = Ot -> set

  {- The Set of big predicates -}
```

```
PRED :: Set   = Ot -> SET

{- the representation of pred
   in SET.  We
   have EL PRED = pred; so PRED
   represents or reflects pred. -}

predcode :: SET = otcode@SET `ARR` u@SET

{- that X is a subset of Y -}
subset (X, Y :: pred) :: set
    = piot (\(a :: Ot)-> X a `arr` Y a)

{- That "is a subset of" is transitive -}
subsetTrans (p, q, r :: pred)
  :: el ((p `subset` q) `arr`
        ((q `subset` r) `arr` (p `subset` r)))
  = \(pq :: el (p `subset` q)) ->
    \(qr :: el (q `subset` r)) ->
    \(a :: Ot) ->
    \(pa :: el (p a)) ->
    qr a (pq a pa)

{- That "is a subset of" is reflexive -}
subsetRefl (p :: pred)
  :: el (p `subset` p)
  = \(a :: Ot) -> \(pa :: el (p a)) -> pa

{- Intersection of a sequence of predicates. -}
capn (Xs :: Seq pred) :: pred
    = \(a :: Ot) -> pin (\(n :: Nat) -> Xs n a)

{- That capn gives a lower bound for
   sequences of predicates
-}
capnLemOut (ps :: Seq pred)
  :: el (pin (\(n :: Nat) -> capn ps `subset` (ps n)))
```

```
  = \(n :: Nat) ->
    \(a :: Ot) ->
    \(psa :: el (capn ps a)) -> psa n

{- That capn is a greatest lower bound
   for sequences of predicates
-}
capnLemIn (ps :: Seq pred) (q :: pred)
  :: el (pin (\(n :: Nat) -> q `subset` (ps n))
         `arr` (q `subset` capn ps))
  = \(qps :: el (pin (\(n :: Nat) -> q `subset` ps n))) ->
    \(a :: Ot) ->
    \(qa :: el (q a)) ->
    \(n :: Nat) ->
      qps n a qa

{- Operation on a predicate which precomposes it with
   an operation. -}
sub (X :: pred) (f :: op Ot) :: pred
    = \(a :: Ot) -> X (f a)

{- Property of a predicate that it is closed
   under an operation -}
clunder (X :: pred) (f :: op Ot) :: set
    = X `subset` (X `sub` f)

{- predicate transformer induced by
   a family of operations
   = a binary operation -}
clunder2 (X :: pred) (f :: op2 Ot) :: pred
    = \(a :: Ot) -> X `clunder` f a

{- PT induced by transition structure -}
B :: pred -> pred
  = \(X :: pred) ->
    \(a :: Ot) ->
    case a of
```

```
      (O) -> n1code@set
      (S a') -> X a'
      (L ls) -> pin (\(n :: Nat) -> X (ls n))

{- That B is monotone. -}
BM (p :: pred) (q :: pred)
  :: el ((p `subset` q) `arr` (B p `subset` B q))
  = \(pq :: el (p `subset` q)) ->
    \(a :: Ot) ->
     case a of
      (O)    -> \(x :: N1) -> x
      (S a') -> pq a'
      (L ls) -> \(a' :: el (B p (limit ls))) ->
                \(n :: Nat) -> pq (ls n) (a' n)

{- Progressivity of a predicate -}
prog (X :: pred) :: set
  = piot (\(a :: Ot) -> B X a `arr` X a)

Prog (X :: pred) :: Set
  = el (prog X)

B' :: PRED -> PRED
  = \(X :: PRED) ->
    \(a :: Ot) ->
    case a of
      (O) -> n1code@SET     -- t@SET (n1code@set)
      (S a') -> X a'
      (L ls) -> PIN (\(n :: Nat) -> X (ls n))

PROG (X :: PRED) :: SET
  = PIOT (\( a :: Ot ) -> B' X a `ARR` X a)

{- Accessibility of a notation.
   Quantifies over pred. -}

Acc (a :: Ot) :: Set
```

```
  = (X :: pred) ->
        el (prog X `arr` X a)

AccO :: Acc zero
  = \(X::pred) -> \(a::Prog X) -> a zero star

AccS (a :: Ot) :: Acc a -> Acc (succ a)
  = \(h :: Acc a) ->
    \(X :: pred) ->
    \(pX :: Prog X) ->
    pX (succ a) (h X pX)

AccL (fs :: Seq Ot) :: ((n :: Nat) -> Acc (fs n)) -> Acc (limit fs)
  = \(h :: (n::Nat) -> Acc (fs n)) ->
    \(X :: pred) ->
    \(pX :: Prog X) ->
    pX (limit fs) (\(n::Nat) -> h n X pX)

acc_code (a :: Ot) :: SET
  = PI predcode (\(X :: pred) ->
        t@SET (prog X `arr` X a))

ACC (a :: Ot) :: Set
  = (X :: PRED)->
        EL (PROG X `ARR` X a)

{- That if a predicate is closed under an operation, then
   it is also closed under all iterates of that operation. -}
closureLemma (X :: pred)
             (f :: op Ot)
             (clf :: el (X `clunder` f))
  :: el (pin (\(n :: Nat) -> X `clunder` OpIt f n))
  = \(n :: Nat) ->
    \(a :: Ot) ->
    \(xa :: el (X a)) ->
    Rec ( \(k :: Nat) ->
          el (X (OpIt f k a)) )
```

```
          xa ( \(n' :: Nat) ->
               \(h :: el (X (OpIt f n' a))) ->
                  clf (OpIt f n' a) h) n


  {- That the intersection of a sequence of progressive
     predicates  is itself progressive.  -}
  capnProg (ps :: Seq pred)
    :: el ((pin (\(n :: Nat) -> prog (ps n)))
           `arr` prog (capn ps))
    =  \ (psp :: el (pin (\(n :: Nat) -> prog (ps n)) )) ->
       let L :: pred = B (capn ps)
           body (n :: Nat)
              :: el (B (capn ps) `subset` ps n) =
                let M :: pred = B (ps n)
                    step1 :: el (capn ps `subset` ps n)
                            = capnLemOut ps n
                    step2 :: el (B (capn ps) `subset` B (ps n))
                            = BM (capn ps) (ps n) step1
                in subsetTrans L M (ps n) step2 (psp n)
       in capnLemIn ps L body

{------------- predicate transformers ----------}

  PT      :: Set          = op pred
  PT'     :: SET          = predcode `ARR` predcode
  PTid    :: PT           = \(x :: pred) -> x

  {- composition of predicate transformers -}
  PTcomp  :: op2 PT       = \(f :: PT) -> \(g :: PT) ->
                            \(x :: pred) -> f (g x)

  {- iteration of predicate transformers. -}
  ItPT    :: PT -> Seq PT = It pred

  {- pointwise intersection of sequences of PTs. -}
  PTlim   :: Seq PT -> PT
```

```
  = \(pts :: Seq PT) -> \(a :: pred) ->
    capn (\(n :: Nat) -> pts n a)

{- preservation of progressivity by a PT -}

PProg (F :: pred -> pred) :: Set
  = (X :: pred) ->
      Prog X -> Prog (F X)

{- same thing reflected as a SET -}
PPROG(F :: pred -> pred) :: SET
  = PI predcode (\(X :: pred) ->
       t@SET (prog X `arr` prog (F X)))

{- Composition preserves preservation of progressivity -}

PProgComp
    (F :: PT ) (pF :: PProg F)
    (G :: PT ) (pG :: PProg G)
    :: PProg (PTcomp F G)
  = \(X :: pred) ->
    \(p :: Prog X) ->
    pF (G X) (pG X p)

{- Identity PT preserves progressivity. -}

PProgId ::  PProg PTid
  = \(X :: pred) -> \(h :: Prog X) -> h

{- Iteration preserves preservation of progressivity. -}

ItPProg (F :: PT) (pF :: PProg F)
        (n :: Nat)
    :: PProg (ItPT F n)
  = \(X :: pred) -> \(pX :: Prog X) ->
    Rec (\(k :: Nat) -> Prog (ItPT F k X))
        pX (\(k :: Nat) -> \(h' :: Prog (ItPT F k X)) ->
```

```
          pF (ItPT F k X) h’) n

{- Given a sequence of predicate transformers,
   if they each  preserve progressivity,
   then so does their pointwise limit -}
PProgLim ( pts :: Seq PT)
         ( pp :: (n :: Nat)-> PProg (pts n) )
  :: PProg (PTlim pts)
  = \(X :: pred) ->
    \(p :: Prog X) ->
    capnProg (\(n :: Nat) -> pts n X)
             (\(n :: Nat) -> pp n X p)

{- Gentzen’s PT -}
G :: PT
  = \(X :: pred) ->
       X ‘clunder2‘ (\(a, b :: Ot) -> b ‘w‘ a)

GentzensLemma :: PProg G
  = \(X :: pred) ->
    \(p :: Prog X) ->
    \(x :: Ot) ->
    \(h :: el (B (G X) x)) ->
    \(a :: Ot) ->
    \(xa :: el (X a)) ->
    let arg ::  el (B X (a ‘w‘ x))
          = case x of
            (O)    -> xa
            (S a’) ->
             let  f   :: Ot -> Ot
                      = \(x :: Ot) -> x ‘w‘ a’
                  itf :: Nat -> Ot
                      = \(k :: Nat)-> It Ot f k a
             in Rec (\(n :: Nat) -> el (X (itf n)))
                    xa (\(n :: Nat) -> h (itf n))
            (L ls) -> (\(n :: Nat) -> h n a xa)
    in p (a ‘w‘ x) arg
```

```
LensLemmaG (a :: Ot) (acca :: Acc a)
           (b :: Ot) (accb :: Acc b)
  :: Acc (w a b)
  = \(X::pred) ->
    \(pX::el (prog X)) ->
    accb (G X) (GentzensLemma X pX) a (acca X pX)

{- A protolens is a predicate transformer that
   preserves progressivity -}

ProtoLens :: Set
  = Si PT PProg

{- Coded as a large set -}
PROTOLENS :: SET    = si@SET PT' PPROG

ProtoLensId :: ProtoLens
  = struct
      fst = PTid
      snd = PProgId

ProtoLensComp :: op2 ProtoLens
  = \(f :: ProtoLens) ->
    \(g :: ProtoLens) ->
      struct
        fst = PTcomp f.fst g.fst
        snd = PProgComp f.fst f.snd g.fst g.snd

{- Finite iteration of proto-lenses -}
ProtoLensIt :: ProtoLens -> Seq ProtoLens
  = \(pl :: ProtoLens) ->
    \(n :: Nat) ->
    It ProtoLens (ProtoLensComp pl) n pl

{- The notion of Lens -}
Lens  (f :: op Ot) :: Set
```

```
  = Si ProtoLens (\(pl :: ProtoLens) ->
       (X :: pred) -> Prog X ->
       el ((pl.fst X) ‘subset‘ (X ‘sub‘ f) ))

{- Coded as a large set -}
LENS (f :: op Ot) :: SET
  = si@SET PROTOLENS (\(pl :: ProtoLens) ->
        PI predcode (\(X :: pred) ->
           t@SET (prog X ‘arr‘ (pl.fst X ‘subset‘ (X ‘sub‘ f))) ))

{- The accessible notations are closed under any operation that
   possesses a lens. -}

LensLemmaV (f :: op Ot)
          (lf :: Lens f)
          (a :: Ot)
  :: Acc a -> Acc (f a)
  = \(h :: Acc a) ->
    \(X :: pred) ->
    \(pX :: Prog X) ->
    lf.snd X pX a (h (lf.fst.fst X) (lf.fst.snd X pX))

{- The identity function possesses a lens -}

LensId :: Lens (\(x :: Ot) -> x)
  = struct
      fst = ProtoLensId
      snd = \(X :: pred) ->
            \(pX :: Prog X) ->
            \(a :: Ot) ->
            \(xa :: el (X a)) -> xa

{- Closure of lenses under composition. -}

LensComp (f :: op Ot) (lf :: Lens f)
         (g :: op Ot) (lg :: Lens g)
         :: Lens (\(a :: Ot) -> f (g a))
```

```
  = struct
      fst = ProtoLensComp lg.fst lf.fst
      snd = \(X :: pred) -> \(pX :: Prog X) ->
            \(a :: Ot) ->
            \(a' :: el (fst.fst X a)) ->
            lf.snd X pX
                   (g a) (lg.snd (lf.fst.fst X)
                                 (lf.fst.snd X pX) a a')

GentzensProtoLens :: ProtoLens
  = struct
      fst = G
      snd = GentzensLemma

GentzenLens :: Lens wexp
  = struct
      fst = GentzensProtoLens
      snd = \(X :: pred) ->
            \(p :: Prog X) ->
            \(a :: Ot) ->
            \(h :: el (G X a)) ->
            h zero (p zero star)

{- A closure lens for an operator f is a protolens (F,..),
   such that F X subset X on progressive
   predicates X (i.e. a lens for the identity
   function), and such that if X is a progressive
   predicate, then F X is closed under f.  -}

CLens (f :: op Ot) :: Set
  = Si  ProtoLens (\(pl :: ProtoLens) ->
      and ( (X :: pred) ->
            el (prog X `arr` (pl.fst X `subset` X )))
          ( (X :: pred) ->
            el (prog X `arr` (pl.fst X `clunder` f))))
```

```
{- We can make a closure lens from a lens.
   The idea is quite simple, though in a way
   the key to the whole construction.
   Just take the limit-by-intersection of the finite
   iterates of the predicate transformer.
-}
MkClens ( f :: op Ot ) :: Lens f -> CLens f
  = \(l :: Lens f) ->
    let
      pt  :: PT = l.fst.fst
      pp  :: PProg pt = l.fst.snd
      dr  :: (X :: pred) -> Prog X ->
                  el (pt X ‘subset‘ sub X f)
          = l.snd
      pt’ :: PT
          = PTlim (ItPT pt)
      pp’ :: PProg pt’
          = PProgLim (ItPT pt) (ItPProg pt pp)
      cl1 :: (X :: pred) ->
             (p :: Prog X) ->
             el (pt’ X ‘subset‘ X)
          = \(X :: pred) ->
            \(_ :: Prog X) ->
            \(a :: Ot) ->
            \(x :: el (pt’ X a)) ->
            x Naturals.zero
      cl2 :: (X :: pred) ->
             (_ :: Prog X) ->
             el (pt’ X ‘clunder‘ f)
          = \(X :: pred) ->
            \(pX :: Prog X) ->
            \(a :: Ot) ->
            \(x :: el (pt’ X a)) ->
            \(n :: Nat) ->
            dr (ItPT pt n X)
               (ItPProg pt pp n X pX)
               a
```

```
                  (x (Naturals.succ n))
     in struct
          fst = struct
                 fst = pt'
                 snd = pp'
          snd = struct
                 fst = cl1
                 snd = cl2

{- Given a closure lens for an operator,
   we can get another one for the omega-iteration
   of that operator. -}

ItClens ( f :: op Ot )
  :: CLens f -> CLens (OpItw f)
  = \(cl  :: CLens f) ->
    let   pl :: ProtoLens
             = cl.fst
          pt :: PT
             = pl.fst
          pp :: PProg pt
             = pl.snd
          pf :: (X :: pred) -> Prog X ->
                el (pt X `subset` X)
               = cl.snd.fst
          cls :: (X :: pred) -> (p :: Prog X) ->
                  el (clunder (pt X) f)
               = cl.snd.snd
          cls' :: (X :: pred) -> (p :: Prog X) ->
                  el (pt X `clunder` OpItw f)
           = \(X :: pred) ->
             \(pX :: Prog X) ->
             \(a :: Ot) ->
             \(x :: el (pt X a)) ->
             pp X pX (OpItw f a) (\(n :: Nat) ->
             closureLemma (pt X) f (cls X pX) n a x)
    in struct
```

```
            fst = pl
            snd = struct
                    fst = pf
                    snd = cls'

{- Given a sequence of lenses for a sequence of functions,
   we can form a lens for their pointwise sup. -}

{- Really uses only that progressive implies closed -}
SupLens (fs :: Seq (op Ot))
        (ls :: (n :: Nat) -> Lens (fs n))
        :: Lens (OpLim fs)
  = let
      pts (n :: Nat) :: PT
              = (ls n).fst.fst
      pps (n :: Nat) :: PProg (pts n)
              = (ls n).fst.snd
      drs (n :: Nat)
             :: (X :: pred) ->
                Prog X ->
                el ( (pts n X) `subset` (X `sub` fs n))
             = (ls n).snd
      pt :: PT = PTlim pts
      pp :: PProg pt = PProgLim pts pps
      dr (X :: pred) (pX :: Prog X)
             ::  el (pt X `subset` (X `sub` (OpLim fs)))
             = \(a :: Ot) ->
               \(xa :: el (pt X a)) ->
               pX (OpLim fs a)
                  (\(n :: Nat) ->
                        drs n X pX a (xa n))
    in struct
            fst = struct
                    fst = pt
                    snd = pp
            snd = dr
```

```
{- If we have a closure lens for f, then
   we can get a lens for Nabla f.
   It may be unhygienic that there is a case
   distinction here.
-}

NablaLens (f :: op Ot)
          (cl :: CLens f )
  :: Lens (Nabla f)
  = let
      pt  :: PT            = cl.fst.fst
      pp  :: PProg pt      = cl.fst.snd
      pf  :: (X :: pred) -> Prog X ->
                el (pt X `subset` X)
          = cl.snd.fst
      cls :: (X :: pred) -> (p :: Prog X) ->
                  el (pt X `clunder` f)
          = cl.snd.snd
      pt' :: PT
          = \(X :: pred) -> pt X `sub` Nabla f
      pp' :: PProg pt'
          = \(X :: pred) ->
            \(pX :: Prog X) ->
            \(a :: Ot) ->
            \(h ::  el (B (pt' X) a)) ->
            case a of
              (O)    -> cls X pX a (pp X pX a star)
              (S a') -> cls X pX (succ (Nabla f a'))
                             (pp X pX (succ (Nabla f a')) h)
              (L ls) -> pp X pX (Nabla f a) h
      dr' (X :: pred) (pX :: Prog X)
          :: el ((pt' X) `subset` (X `sub` (Nabla f)))
          = \(a :: Ot) -> pf X pX (Nabla f a)
    in struct
         fst = struct
                fst = pt'
                snd = pp'
```

```
          snd = dr'

  {- The following predicate is pivotal. -}
  LensPredicate (a :: Ord) :: SET
    = LENS (v a)

  SuccLens (f :: op Ot) (lf :: Lens f)
    :: Lens (deriv f)
    = NablaLens (OpItw f)
                (ItClens f (MkClens f lf))

  LimLens (fs :: Seq (op Ot))
          (lfs :: (n :: Nat) -> Lens (fs n))
    :: Lens (derivl fs)
    = let  f :: op Ot = OpLim fs
      in   NablaLens f (MkClens f (SupLens fs lfs))

  ZeroLens :: Lens (deriv wexp)
    = SuccLens wexp GentzenLens

--------------------------------------------------------
{- We are now back outside the "small" package. The file
  concludes with a proof of the accessibility of
  Gamma_0 -}

{- Next universe operator. -}

NextU (FS :: Fam Set) :: Fam Set
   = (small FS).Next

{- Accessibility relative to a family of sets -}
acc (FS :: Fam Set) :: Ord -> Set
   = (small FS).Acc

ACC (FS :: Fam Set) :: Ord -> Set
   = acc (NextU FS)
```

```
{- The set of lenses relative to a family of sets -}
Lens (FS :: Fam Set)(f :: Ord -> Ord) :: Set
   = (small FS).Lens f

PRED (FS :: Fam Set)
   :: Set
   = (small (NextU FS)).pred

{- The predicate wrt FS that Lens (v a) -}
lenspred (FS :: Fam Set)
   :: PRED FS
   = (small FS).LensPredicate

PROG (FS :: Fam Set)
   :: Pow (PRED FS)
   = (small (NextU FS)).Prog

{- That the above predicate is progressive -}
lenspredprog (FS :: Fam Set)
   :: PROG FS (lenspred FS)
   = \(a::Ord) ->
     case a of
        (O) -> \(_::N1) ->
                  (small FS).ZeroLens
        (S a') -> (small FS).SuccLens
                       (v a')
        (L ls) -> (small FS).LimLens
                       (\(n::Nat) -> v (ls n))

{- The key lemma which shows how for each
   layer of the veblen hierarchy, you can
   use another universe -}

LemmaV (FS :: Fam Set)
      (a :: Ord)
      (acca :: ACC FS a)
   :: (b :: Ord) -> acc FS b -> acc FS (v a b)
```

```
    = let vl :: Lens FS (v a)
             = acca (lenspred FS) (lenspredprog FS)
      in (small FS).LensLemmaV (v a) vl

{- Just for completeness, we pull out
   of the package above the corresponding
   lemma for the function w. -}

LemmaG (FS :: Fam Set)
      (a :: Ord)
      (acca :: acc FS a)
   :: (b :: Ord) -> acc FS b -> acc FS (w a b)
   = (small FS).LensLemmaG a acca

{- RECURSIVE
   The main theorem.
   To express the recursion by use of NatRec, we
   would need a superuniverse closed under NextU -}

Theorem (FS :: Fam Set) (n :: Nat)
    :: acc FS (pd Gamma0 n)
    =  case n of
         (Z) ->  (small FS).Acc0
         (S p) -> LemmaV FS
                    (pd Gamma0 p) (Theorem (NextU FS) p)
                     zero         (small FS).Acc0

Corollary (FS :: Fam Set)
    :: acc FS Gamma0
    = (small FS).AccL (pd Gamma0) (Theorem FS)
```

Bibliography

[1] S. Abramsky. Semantics of interaction. In A.M. Pitts and P. Dybjer, editors, *Sematics and Logics of Computations*, pages 1–31. Cambridge University Press, Cambridge, 1997.

[2] W. Ackermann. Zur widerspruchsfreiheit det zahlentheorie. *Mathematische Annalen*, 117:162–194, 1940.

[3] P. Aczel. Describing ordinals using functionals of transfinite type. *Journal of Symbolic Logic*, 37:35–47, 1972.

[4] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.

[5] A. Alternkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic: 1999*, to appear 1999.

[6] Bachmann. *Transfinite Zahlen.* Springer, Berlin, Göttingen, Heidelberg, 1955.

[7] H. Bachmann. Die normalfunktionen und das problem der ausgezeichneten folgen von ordungszahlen. *Vierteljahrschr. Naturf. Ges. Zurich*, 95:115–147, 1950.

[8] R-J Back and J. von Wright. *Refinement Calculus: a systematic introduction.* Graduate texts in computer science. Springer-Verlag, New York, 1998.

[9] Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[10] W.H. Burge. *Recursive Programming Techniques.* The Systems Programming Series. Addison-Wesley, Reading, Massachusetts, 1975.

[11] G. Cantor. Grundlagen einer allgemeinen Mannigfaltigkeitslehre. ein mathematische-philosophischer Versuch in der Lehre des Unendlichen. Leipzig, 1883.

[12] L. Cardelli. Type systems. In *The computer science and engineering handbook*, pages 2208–2236. CRC press, 1997.

[13] J. Cartmell. *Generalised Algebraic Theories and Contextual Categories*. PhD thesis, University of Oxford, July 1990.

[14] A. Church. *The Calculi of Lambda Conversion*. Princeton Univ. Press, 1941.

[15] J.H. Conway. *On Numbers and Games*. London Mathematical Society monographs,6. Academic Press, London, 1976.

[16] J.H. Conway and R.K. Guy. *The Book of Numbers*. Springer-Verlag, New York, 1996.

[17] C. Coquand. *Computation in Type Theory*. PhD thesis, Department of Computing Science, University of Göteborg, 1996.

[18] T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279, Cambridge, 1991. Cambridge University Press.

[19] O. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, chapter Notes on structured programming, pages 1–82. Academic Press, 1971.

[20] N. Danner. Ordinals and ordinal functions representable in the simply typed λ-calculus. *Annals of Pure and Applied Logic*, 97:179–201, 1999.

[21] J.W. Dauben. *Georg Cantor, His Mathematics and Philosophy of the Infinite*. Princeton University Press, New Jersy, 1979.

[22] A. S. Davis. Half-ring morphologies. In M.H. Löb, editor, *Proceedings of the Summer School in Logic, Leeds, 1967*, volume 70 of *Lecture Notes in Mathematics*, pages 253–268, Berlin, Heidelberg, New York, 1968. Springer-Verlag.

[23] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[24] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1989.

[25] J. Diller. Zur berechenbarkeit primitiv-recursiver functionale endlicher typen. In H.A. Schmidt, K. Schütte, and H Thiele, editors, *Contributions to Mathematical Logic*, pages 109–120. North-Holland, Amsterdam, 1970.

[26] M.A.E. Dummett. *Elements of Intuitionism*. Clarendon Press, New York, 1977.

[27] P. Dybjer. Internal type theory. In *TYPES workshop, Torino, June 1995*, volume 1158 of *LNCS*. Springer-verlag, 1996.

[28] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 1997.

[29] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In *Proceedings of TLCA 1999*, volume 1581 of *lncs*, pages 129–146, 1999.

[30] S. Feferman. Hereditarily replete functionals over the ordinals. In A. Kino, J. Myhill, and R.E. Vesley, editors, *Intuitionism and Proof Theory: Bufallo N.Y. 1968*, Amsterdam, 1970. North-Holland.

[31] S. Feferman. Iterated inductive fixed-point theories: Application to hancock's conjecture. In G Metakides, editor, *Patras Logic Symposium*, pages 171–196. North-Holland, 1982.

[32] S. Feferman. *In the Light of Logic*. Oxford University Press, 1998.

[33] Solomon Feferman. Iterated inductive fixed-point theories: Application to Hancock's conjecture. In G. Metakides, editor, *Patras Logic Symposium*, pages 171 – 195, Amsterdam, 1982. North-Holland.

[34] P. H. B. Gardiner, C.E. Martin, and O. de Moor. An algebraic construction of predicate transformers. *Science of Computer Programming*, 22:21–44, 1994.

[35] G. Gentzen. New version of the consistency proof for elementary number theory. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 252–286. North-Holland, Amsterdam, 1969.

[36] G. Gentzen. Provability and nonprovability of restricted transfinite induction in elementary number theory. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 287–308. North-Holland, Amsterdam, 1969.

[37] Eduardo Giménez. Structural recursive definitions in type theory. In *Proceedings of ICALP'98, LNCS series no. 1443, July 1998.*, number 1443 in LNCS, July 1998.

[38] J.-Y. Girard. *Proof Theory and Logical Complexity.* Bibliopolis, Napoli, 1987.

[39] K. Gödel. Über eine bisher noch nicht benützte erwietung des finiten standpunkts. *Dialectica*, 12:240–251, 1958. English translation in [40].

[40] K Gödel. *Collected Works, Vol. II. Publications 1938-1974.* Oxford University Press, New York, 1990.

[41] H. Gonshor. *An Introduction to the Theory of Surreal Numbers.* London Mathematical Society lecture note series, 110. Cambridge University Press, Cambridge, 1986.

[42] A. Hahnal and P. Hamburger. *Set Theory.* Number 48 in London Mathematical Society Student Texts. Cambridge University Press, 1999.

[43] M. Hallet. *Cantorian set theory and Limitation of size.* Number 10 in Oxford Logic Guides. Clerendon Press, Oxford, 1984.

[44] A.G Hamilton. *Numbers, Sets and Axioms: the apparatus of mathematics.* Cambridge University Press, 1982.

[45] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[46] D. Hilbert. The foundations of mathematics. In J. van Heijenoort, editor, *From Frege to Gödel: A Sourcebook in Mathematical Logic, 1879-1931*, pages 464–479. Harvard University Press, Cambridge, Massachusetts, 1967.

[47] M. Hofmann. *Extensional Concepts in Extensional Type Theory.* PhD thesis, University of Edinburgh, 1995.

[48] M. Hofmann. *Syntax and Semantics of Dependent Types*, pages 79–127. Cambridge University Press, 1997.

[49] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980. Reprint of a manuscript written 1969.

[50] W.A. Howard. Ordinal analysis of terms of finite type. *Journal of Symbolic Logic*, 43(3):355–374, 1980.

[51] G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, Cambridge, New York, Melbourne, 1991.

[52] G. Huet and G. Plotkin, editors. *Logical Environments*. Cambridge University Press, Cambridge, New York, Melbourne, 1993.

[53] D. Isles. Regular ordinals and normal forms. In A. Kino, J. Myhill, and R.E. Vesley, editors, *Intuitionism and Proof Theory, Buffalo N.Y. 1968*, Amsterdam, 1970. North-Holland.

[54] A. Kanamori. The mathematical development of set theory from Cantor to Cohen. *The Bulletin of Symbolic Logic*, 1(2):1–71, March 1996.

[55] A. N. Kolmogorov. Zur deutung der intuitionistischen logik. *Matematische Zeitschrift*, 35:58–65, 1932.

[56] G. Kreisel. On the interpretation of non-finitist proofs, part i. *Journal of Symbolic Logic*, 16:241–267, 1951.

[57] G. Kreisel. On the interpretation of non-finitist proofs, part ii. *Journal of Symbolic Logic*, 17:43–58, 1952. Erratum, p. iv.

[58] F. William Lawvere and Stephen J. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, Cambridge, 1997.

[59] I. Lindström. A construction of non-well-founded sets within martin-löf's type theory. Technical Report 15, Department of Mathematics, Uppsala University, 1986.

[60] P. Lorenzen. Algebraische und logistische untersuchungen über freie verbände. *Journal of Symbolic Logic*, 16:81–106, 1951.

[61] Z. Luo. *Computation and Reasoning*. Clarendon Press, Oxford, 1994.

[62] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer-Verlag, 1992.

[63] P. Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In J.E. Fenstad, editor, *Proceedings of the Second Scandanavian Logic Symposium*, pages 179–216. North-Holland, 1971.

[64] P. Martin-Löf. Infinite terms and a system of natural deduction. *Compositio Mathematica*, 24:93–103, 1972.

[65] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H.E Rose and J.C. Shepherdson, editors, *Logic colloquium 1973*, pages 73–118, Amsterdam, 1975. North-Holland.

[66] P. Martin-Löf. Constructive mathematics and computer programming. In Cohen, Los, Pfeiffer, and Podewski, editors, *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175, Amsterdam, 1982. North-Holland.

[67] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[68] P. Martin-Löf. Mathematics of infinity. In P. Martin-Löf and G. Mints, editors, *Colog88, International Conference on Computer Logic, Tallinn, December 88*, volume 417 of *lncs*, pages 146–197, Berlin and Heidelberg, 1990. Springer.

[69] N.P. Mendler. Predicative type universes and primitive recursion. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 173–184. IEEE, Amsterdam, 1991.

[70] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[71] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[72] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[73] W. Neumer. Zur konstruktion von ordnungszahlen, i. *Mathematishe Zeitschrift*, 58:319–413, 1953.

[74] W. Neumer. Zur konstruktion von ordnungszahlen, ii. *Mathematishe Zeitschrift*, 59:434–454, 1954.

[75] W. Neumer. Zur konstruktion von ordnungszahlen, iii. *Mathematishe Zeitschrift*, 60:1–16, 1954.

[76] W. Neumer. Zur konstruktion von ordnungszahlen, iv. *Mathematishe Zeitschrift*, 61:47–69, 1954.

[77] W. Neumer. Zur konstruktion von ordnungszahlen, v. *Mathematishe Zeitschrift*, 64:435–456, 1956.

[78] W. Neumer. Algorithmen für ordnungszahlen und normalfunktionen, part i. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 3:108–150, 1957.

[79] W. Neumer. Algorithmen für ordnungszahlen und normalfunktionen, part ii. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:1–65, 1960.

[80] B. Nordström, Petersson, and J. M. K., Smith. *Programming in Martin-Löf's Type Theory*. Clarendon Press, Oxford, 1990.

[81] P. Novikov. On the consistency of a certain logical calculus. *Matématicésky sbovnik*, 12(3):353–369, 1943.

[82] E. Palmgren. On universes in type theory. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory*, Oxford Logic Guides. Oxford University Press, 1998.

[83] D. Park. Concurrency and automata on infinite sequences. In *Proc. Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.

[84] K. Petersson and D. Synek. A set constructor for inductive sets in martin-löf's type theory. In *LNCS*, volume 389. Springer-Verlag, 1989.

[85] W. Pohlers. Pure proof theory: Aims, methods and results. *Bulletin of Symbolic Logic*, 2(2):159–188, June 1996.

[86] Michael D. Potter. *Sets, An Introduction*. Clarendon Press, Oxford, New York, Tokyo, 1990.

[87] D. Prawitz. *Natural Deduction.* Number 3 in Stockholm Studies in Philosophy. Almqvist and Wiksell, Stockholm, 1965.

[88] Griffor E. Rathjen, M. The strength of some martin-löf type theories. *Archiv for Mathematical Logic*, 33:347–385, 1994.

[89] M. Rathjen. The realm of ordinal analysis. In S. Cooper and J. Truss, editors, *Sets and Proofs. Proceedings of the Logic Colloquium '97, Cambridge.* Cambridge University Press, 1997. http://www.amsta.leeds.ac.uk:80/Pure/staff/rathjen/mahlo.ps.

[90] M. Rathjen. The strength of martin-lf type theory with a superuniverse. part i. *Archive for Mathematical Logic.*, 1999. To appear.

[91] M. Rathjen, Griffor E.R., and E. Palmgren. Inaccessibility in constructive set theory and type theory. *Annals of Pure and Applied Logic*, 94:181–200, 1998.

[92] V. J. Rayward-Smith. *A first course in formal language theory.* McGraw-Hill, second edition, 1995.

[93] Paul C. Rosenbloom. *The elements of mathematical logic.* Dover, New York, 1950.

[94] B. Rotman and G.T. Kneebone. *The Theory of Sets and Transfinite Numbers.* Oldbourne Mathematical Series. Oldbourne, London, 1966.

[95] L.E. Sanchis. Functionals defined by recursion. *Notre Dame Journal of Formal Logic*, 8:161–174, 1967.

[96] K. Schütte. Beweistheoretische erfassung der unendliche induction in der zahlentheorie. *Mathematische Annalen*, 122:369–389, 1950.

[97] H. Schwichtenberg. Proofs as programs. In P. Aczel, H. Simmons, and S. Wainer, editors, *Proof Theory*, pages 79–115, Cambridge, 1992. Cambridge University Press.

[98] A. Setzer. Well-ordering proofs for martin-löf type theory with w-type and one universe. *Annals of Pure and Applied Logic*, 92:113–159, 1998.

[99] A. Setzer. Well-ordering proofs in martin-lf's type theory. In G. Sambin and J. Smith, editors, *Twenty-five years of constructive type theory*, pages 245–263. Clarendon Press, Oxford, 1998.

[100] A. Setzer. Extending martin-lf type theory by one mahlo-universe. *Arch. math. log.*, page 21pp, 1999.

[101] Sören Stenlund. *Combinators, λ-Terms and Proof Theory*, volume 42 of *Synthese Library*. D. Reidel Publ. Co., Dordrecht, 1972.

[102] T Streicher. *Semantical Investigations into Intensional Type Theory*. PhD thesis, LMU Münnchen, 1993.

[103] W.W. Tait. Infinitely long terms of transfinite type. In Dummett M.A.E. Crossley, J.N., editor, *Formal Systems and Recursive Functions*, Studies in Logic and the Foundations of Mathematics, pages 176–185, Amsterdam, 1965. North-Holland.

[104] Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996.

[105] A.S Troelstra and H. Schwichtenburg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996.

[106] T. Uustalu and V. Vene. A cube of proof systems for the intuitionistic predicate mu-,nu-logic. In *Selected Papers 8th Nordic Workshop on Programming Theory*, pages 237–246, Oslo, 1997. Research Report 248, Dept. of Informatics, Univ. of Oslo.

[107] T. Uustalu and V. Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361, 1999.

[108] T. Uustalu and V. Vene. Primitive (co)recursion and course-of-values (co)iteration, categorically. *Informatica*, 10(1):5–26, March 1999.

[109] Philip Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[110] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI series, Series F: Computer and System Sciences*. Springer Verlag, 1994. Proceedings of the International Summer School at Marktoberdorf directed by F. L. Bauer, M. Broy, E. W. Dijkstra, D. Gries, and C. A. R. Hoare.

[111] Philip Wadler. How to declare an imperative (invited talk). In *International Logic Programming Symposium*, Portland, Oregon, December 1995. MIT Press.

[112] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995. (This is a revised version of [110].).

[113] R.W. Weyrauch. *Relations between some hierarchies of Ordinal Functions and Functionals*. PhD thesis, Department of Mathematics, Stanford University, Stanford, California, September 1972.

[114] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. The MIT Press, 1993.

[115] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[116] L. Wittgenstein. *Tractatus Logico-Philosophicus*. Routledge and Kegan Paul, London, 1961. First German edition in Annalen der Naturphilosophie, 1921.