CHAPTER X

Types in Logic, Mathematics and Programming

Robert L. Constable
Computer Science Department, Cornell University
Ithaca, New York 14853, USA

Contents

HANDBOOK OF PROOF THEORY
Edited by S. R. Buss

1. Introduction

Proof theory and computer science are jointly engaged in a remarkable enterprise. Together they provide the practical means to formalize vast amounts of mathematical knowledge. They have created the subject of *automated reasoning* and a digital computer based *proof technology*; these enable a diverse community of mathematicians, computer scientists, and educators to build a new artifact — a globally distributed digital library of formalized mathematics. I think that this artifact signals the emergence of a new branch of mathematics, perhaps to be called *Formal Mathematics.*

The theorems of this mathematics are completely formal and are processed digitally. They can be displayed as beautifully and legibly as journal quality mathematical text. At the heart of this library are completely formal proofs — created with computer assistance. Their correctness is based on the axioms and rules of various foundational theories; this formal accounting of correctness supports the highest known standards of rigor and truth. The need to formally relate results in different foundational theories opens a new topic in proof theory and foundations of mathematics.

Formal proofs of interesting theorems in current foundational theories are very large rigid objects. Creating them requires the speed and memory capacities of modern computer hardware and the expressiveness of modern software. Programs called *theorem provers* fill in tedious detail; they recognize many kinds of "obvious inference," and they automatically find long chains of inferences and even complete subproofs or proofs. The study of these theorem provers and the symbolic algorithms that make them work is part of the subject of automated reasoning. This science and the proof technology built on it are advancing all the time, and the new branch of mathematics that they enable will have its own standards, methods, surprises and triumphs.

This article is about the potent mixture of proof theory and computer science behind automated reasoning and proof technology. The emphasis is on proof theory topics while stressing connections to computer science.

Computer science is concerned with automating computation. Doing this well has made it possible to formalize real proofs. Computing well requires fast and robust hardware as well as expressive high level programming languages. High level languages are partially characterized by their *type systems*; i.e., the organization of *data types* expressible in the language. The evolution of these languages has led to type systems that resemble mathematical type theories or even computationally effective set theories. (This development underlines the fact that high level programming is an aspect of computational mathematics.) *This article will focus mainly on relating data types and mathematical types.*

The connection between data types and mathematical types in the case of formal mathematics and automated reasoning is even tighter than the general connection. Here is why. To preserve the highest standards of rigor in formalized mathematics built with computer assistance (the only way to produce it), it is necessary to reason

about programs and computations. This is what intuitionists and constructivists do at a very high level of abstraction. So as the programming languages for automating reasoning become more abstract and expressive, constructive mathematics becomes directly relevant to Formal Mathematics and to the "grand enterprise" of building it using theorem provers. We will see that connections are quite deep.

It turns out that proof technology is relevant to other technologies of economic and strategic importance. For instance, the *type checkers* in commercial programming languages like ML are actually small theorem provers. They check that arguments to a function match the type of the function (see section 3). Industrial *model checkers* systematically search for errors in the design of finite state systems, such as hardware circuits or software protocols. More general tools are *program verification systems.* These combine type checkers, model checkers, decision procedures, and theorem provers that use formalized mathematics. They are employed to prove that programs have certain formally specified properties. Such proofs provide the highest levels of assurance that can be given that programs operate according to specifications. There are also software systems based on proof technology which synthesize correct programs from proofs that specifications are realizable. We will examine the proof theory underlying some of these systems.

My approach to the subject comes from the experience of designing, studying, and using some of the earliest and then some of the most modern of these theorem provers. Currently my colleagues and I at Cornell are working with the system we call Nuprl ("new pearl").[1] We call it a *proof development system* in Constable et al. [1986], but some call it a *problem solving environment* (PSE) or a *logical framework* (LF). From another point of view it is a *collaborative mathematics environment*, c.f., Chew et al. [1996]. Whatever Nuprl is called, I am concerned with systems like it and their evolution. We will examine the logical features common to a variety of current systems of a similar kind, such as ACL2, Alf, Coq, HOL, IMPS, Isabelle, Kiv, LA, Lego, Mizar, NqThm and Otter. So while I will refer to Nuprl from time to time, most of the ideas are very general and will apply to the systems of the 21st century as well. Before saying more about the article, let me put the work into historical perspective. Doing this will allow me to state my goals more exactly (especially after each topic of Section 1.1).

1.1. Historical perspective on a grand enterprise 1875-1995. From *Begriffsschrift* [1879] onwards until *Grundgesetze* [1903], logic was re-surveyed by Gottlob Frege, and the ground was cleared to provide a firm foundation for mathematics.[2] In *Principia Mathematica*, Whitehead and Russell [1925-27] revised Frege's flawed architectural plans, and then using these plans, Hilbert [1926] laid out a formalist

[1] We have released Version 4.2, see http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html. Version 5 and "Nuprl Light" will be available at this World Wide Web site in 1999.

[2] *Begriffsschrift* ("concept script") analyzed the notion of a proposition into function and argument, introduced the quantifiers, binding, and a theory of identity. This created the entire predicate calculus. *Grundgesetze* presented a theory of classes based on the comprehension principle and defined the natural numbers in terms of them.

program to build the completely formal theories which would be used to explain and justify the results and methods of mathematics. His program would defend mathematical practice against critics like Brouwer who saw the need to place the foundation pilings squarely on the natural numbers and build with constructive methods.[3]

Hilbert called for workers, training some himself, and began with them the task which proved to be so compelling and attractive to many talented mathematicians like Church, von Neumann, Herbrand, Gentzen, Skolem, Turing, Tarski, Gödel, and many more. Boring deep into the bedrock to explore the foundation site, Kurt Gödel [1931] unexpected limitations to the planned activity. It could never be completed as envisioned by Hilbert.[4] His surprising discovery changed expectations, but the tools Gödel created transformed the field and stimulated enormous interest in the enterprise. More remarkable discoveries followed.

Within two decades, computer science was providing new "power tools" to realize in software the formal structures needed to support mathematics. By 1960 computer hardware could execute programming languages like Lisp, c.f. McCarthy [1963], designed for the symbolic processing needed to build formal structures. Up in the scaffolding computer scientists began to encounter their own problems with "wiring and communications," control of resource expenditure, design of better tools, etc. But already even in the 1970's poised over the ground like a giant drilling rig, the formal structures supported still deeper penetration into the bedrock designed to support mathematics (and with it the mathematical sciences and much of our technical knowledge). The theory of computational complexity, arising from Hartmanis and Stearns [1965], led to further beautiful discoveries like Cook's $P = NP$ problem, and to a theory of algorithms needed for sophisticated constructions, and to a theory of *feasible mathematics* (see Buss [1986], Leivant [1994b,1994a,1995]), and to ideas for the *foundations of computational mathematics.*

By 1970 the value of the small formal structure already assembled put to rest the nagging questions of earlier times about why mathematics should be formalized. The existing structure provided economic benefit to engineering, just as Leibniz dreamed, Frege foresaw, McCarthy planned [1962], and many are realizing.

Even without the accumulating evidence of economic value, and without counting the immediate utility of the software artifacts, scientists in all fields recognized that the discoveries attendant on this "grand enterprise" illuminate the very nature of knowledge while providing better means to create and manage it. The results of this enterprise have profound consequences because all scholars and scientists are in the business of processing information and contributing to the accumulation and dissemination of knowledge.

The construction of the foundational structure goes on; it is forming a new kind

[3] I refer to Hilbert's *formalist program* founded on a finitistic analysis of formal systems to prove their consistency and to justify non-constructive reasoning as a (possibly meaningless) *detour* justified by the *consistency* of a formal system.

[4] Gödel showed that *consistency* is not sufficient to justify the detour because there are formulas of number theory such that both P and $\neg P$ can be consistently added (P an unprovable formula).

of place, like a biosphere made out of bits. We might call it a "cybersphere" since it encloses the space we call "cyberspace." Many people now live in this space... which supports commerce and recreation as well as scholarship and science.

It is in the context of this "grand enterprise" that I have framed the article. I see it concerned with two of the major modes of work in assembling the formal structures — logical analysis and algorithmic construction. I will briefly mention the aspects of these activities that I treat here.

Logical analysis. When looking back over the period from 1879 to now, we see that the formal analysis of mathematical practice started with logical language. Frege [1879] said:

> "To prevent anything intuitive from penetrating [into an argument] unnoticed, I had to bend every effort to keep the chain of inferences free of gaps. In attempting to comply with this requirement in the strictest possible way I found the inadequacy of language to be an obstacle. This deficiency led me to the present ideography...
>
> Leibniz, too, recognized — and perhaps overrated the advantages of adequate notation. His idea of a... *calculus philosophicus*... was so gigantic that the attempt to realize it could not go beyond the bare preliminaries. The enthusiasm that seized its originator when he contemplated the immense increase in intellectual prover of mankind that [the calculus would bring] caused him to underestimate the difficulties.... But even if this worthy goal cannot be reached in one leap, we need not despair of a slow step by step approximation."

So Frege began with very limited goals and took what he characterized as "small steps" (like creating all of predicate logic!). He did not include a study of computation and its language; he limited his study of notation to logical operators, and he ruled out creating a natural expression of proofs or classifying them based on how "obvious" they are.

In addition, Frege focused on understanding the most fundamental types, natural numbers, sequences, functions, and classes. He adopted a very simple approach to the domain of functions, forcing them all to be *total.* He said (in his *Collected Papers*) "the sign $a + b$ should always have a reference, whatever signs for definite objects may be inserted in place of 'a' and 'b'." *Principia* took a different approach to functions, introducing types, but also it excluded from consideration an analysis of computation or natural proofs or the notational practices of working mathematics. It too developed only basic mathematics with no attempt to treat abstract algebra or computational parts of analysis.

Principia Mathematica, the monumental work of Whitehead and Russell [1925-27], was indeed the first comprehensive rendering of mathematics in symbolic logic. Gödel's celebrated 1931 paper "On Formally Undecidable Propositions of *Principia Mathematica* and Related Systems" begins:

> "The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one

> can prove any theorems using nothing but a few mechanical rules. The most comprehensive formal systems that have been set up hither to are the system of *Principia Mathematica* (PM) on the one hand and the Zermelo-Fraenkel axiom system of set theory... on the other."

Principia presents a logic based on types and derives in it a theory of classes, while ZF set theory provided informal *axioms* and, logic was incidental.[5] *Principia* deals with the topics that I find fundamental in my own work of "implementing mathematics" in the Nuprl system, Constable et al. [1986]. Thus much of what I say here is related to PM. Indeed, in a sense Nuprl is a modern style *Principia* suitable for computational mathematics.

Hilbert introduced a greater degree of formalization, essentially banishing semantics, and he began as a result to deal with computation in his metalanguage. But he took a step backwards from *Principia* in terms of analyzing when expressions are meaningful. He reduced this to an issue of parsing and "decidable type checking" of formulas as opposed to the semantic judgments of *Principia.*

It wasn't until Gentzen that the notion of proofs as they occur in practice was analyzed, first in *natural deduction* and then in *sequent calculi.* It wasn't until Herbrand, Gödel, Church, Markov, and Turing that computation was analyzed and not until de Bruijn that the organization of knowledge (into a "tree of knowledge" with explicit contexts) was considered. De Bruijn's Automath project also established links to computing, like those simultaneously being forged from computer science.

Recently Martin-Löf has widened the logical investigation to reintroduce a semantic approach to logic, to include computation as part of the language, and to make manifest the connections to knowledge. Martin-Löf [1983,p.30] says:

> "to have proved = to know = to have understood, comprehended, grasped, or seen. It is now manifest, from these equations, that proof and knowledge are the same. Thus, if proof theory is construed, not in Hilbert's sense, as metamathematics but simply as the study of proofs in the original sense of the word, then proof theory is the same as theory of knowledge..."

We are now in a position where mathematical logic can consider all of these elements: an analysis of the basic judgments of typing, truth, and computational equality; an analysis of natural proofs and their semantics; the integration of computational concepts into the basic language; an analysis of the structure of knowledge and its role in practical inference; and classification of inference according to its computational complexity.

We will attempt a consideration of logic that takes all this into account and is linked to computing practice, and yet is accessible. I begin the article with an account of typed logic that relates naturally to the Automath conception. The connection is discussed explicitly in Section 2.12.

[5] *Principia* is not *formal* in the modern sense. There are semantic elements in the account which Wittgenstein [1953,1922] objected to. Hilbert made a point of formalizing logic, and we follow in that purely formal tradition.

The article stresses the nature of the underlying logical language because that is so basic — everything else is built upon it. The structures built are so high that a small change in the foundation can cause a large movement at the top of the structure. So any discoveries that improve the foundation for formal mathe atics are among the most profound in their effect. As it is, we are standing on the shoulders of giants. I take the time in section 2 to review this heritage that is so crucial to everything else.

Algorithmic construction. Computer science completely transformed the "grand enterprise." First it introduced computational procedure and *procedural knowledge*, and it gradually widened the scope of its successes. It could check formulas and synthesize them; later it could check proofs and synthesize them. In all this, the precision that Gödel referred to reached new levels of *mechanical precision.* The vast change of scale from processing a few hundred lines by hand to tens of thousands by machine (now hundreds of millions) caused a qualitative change and created new fields like Automated Deduction and Formal Mathematics in which formalisms became usable.

The success of procedural knowledge created the questions of relating it to declarative knowledge, a question at the heart of computer science, studied extensively in the database area, also in "logic programming" and in AI. It is a question at the heart of AD, McAllester [1989], as is clearly seen in Bundy's work on *proof plans* [1991]. From the AI perspective, one can see this impact as reintroducing "mind" and "thought" into the enterprise McAllester [1989]. From the logical perspective one can see this as reintroducing the study of *intension* and Frege's notion of *sense* into logic. As Jean-Yves Girard put it in Girard, Taylor and Lafont [1989,p.4]:

> "In recent years, during which the algebraic tradition has flourished, the syntactic tradition was not of note and would without a doubt have disappeared in one or two more decades, for want of any issue or methodology. The disaster was averted because of computer science — that great manipulator or syntax — which posed some very important theoretical problems."

Computer science produced new high level languages for expressing algorithms. These have evolved to modern programming languages such as ML (for Meta Language) designed to help automate reasoning. ML and its proposed extensions have such a rich system of data types that the type system resembles a constructive theory of mathematical types. We discuss this observation in section 3. Our concern for the relationship between data types and mathematical types is a reason that I will talk so much about typed logic in section 2.

Computer science also created a new *medium* for doing mathematics — the digital electronic medium now most visible through the World Wide Web. This affects every aspect of the enterprise. For example, the "surface" or concrete syntax can be disconnected from the abstract syntax, and we can *display* the underlying terms in a large variety of forms. To take a trivial point, the typed universal quantifier, "for

all x of type A" can be displayed as $\forall x : A.$ or as $\bigwedge x : A.$ or as "For all x in A" or $\forall x \in A.$ or $\forall x^A : .$ The key parts of the abstraction are the operator name, *all*, a binding variable, x, and a type, A.

The new medium provides *hypertext* and *hyperproofs*, e.g. Nuprl's proof editor and Hyperproof (Barwise and Etchemendy [1991]).[6] It is difficult to render these proofs on paper as we will see in the appendix; one must "live in the medium" to experience it. This medium also creates a new approach to communication and doing mathematics. It is routine to embed computation in documents and proofs, e.g. *Mathematica* notebooks, Wolfram [1988]. Databases of definitions, conjectures, theorems, and proofs will be available on line as *digital mathematics libraries.* There will be new tools for collaborating remotely on proofs as we can now do on documents (with OLE, OpenDoc, and similar tools).

These are the overt changes brought by computer science, but as Girard says, the theoretical questions raised are very important to logic. We can see some of these by looking at the stages of work in automated reasoning.

Stage 1. From the late 50's to the late 60's was an *algorithmic phase* during which the basic symbolic procedures and low level data structures were discovered and improved. The basic matching, unification, and rewriting algorithms were coded and tested in the resolution method and various decision procedures. We learned the extent and value of these basic symbolic procedures, and they were made very efficient in systems like Otter (Wos et al. [1984]) and Prolog. Now there are links being formed with other non-numerical computing methods from symbolic algebra. A deep branch of computational mathematics has been formed, and communities of scientists are using the tools.

Stage 2. The 70's saw the creation of several systems for use in writing more reliable software. These were called *program verifiers*, e.g. the Stanford Pascal Verifier (Igarashi, London and Luckham [1975]) and Gypsy (Good [1985]) were targeted to Pascal and NqThm (Boyer and Moore [1979]) for pure Lisp, PL/CV (Constable, Johnson and Eichenlaub [1982]) for a subset of PL/I, and LCF (Gordon, Milner and Wadsworth [1979]) for a higher order functional programming language. These systems could also be seen as implementing *logics of computable functions* (hence LCF) or *programming logics.* The motivation (and funding) for this work came from computer science. The goal was to "prove that programs met their specifications." This technology drew on the algorithms from the earlier period, but also contributed new techniques such as *congruence closure* (Kozen [1977], Nelson and Oppen [1979], Constable, Johnson and Eichenlaub [1982]) and new decision procedure such as Arith (Chan [1982]) and SupInf (Bledsoe [1975], Shostak [1979]) and the notion of theorem proving *tactics* (from LCF). During this period there also appeared systems for *checking* formalizing mathematics such as Automath (de Bruijn [1970], Nederpelt, Geuvers and Vrijer [1994]) and FOL (Weyrauch [1980]).

[6]This capability can be explored at the Nuprl project home page www.cs.cornell.edu/Info/Projects/NuPrl.

Stage 3. In the 80's and 90's the so-called program verification idea was refined a great deal. The Larch (Guttag, Horning and Wing [1985]) system was multi-lingual, and it represents the change in emphasis from "verification" to *property checking.* The LCLint system is a result of this evaluation (see www.larch.lcs.mit.edu). The prover is seen as a *bug detector* or "falsification system." The LCF system spawned HOL (Gordon and Melham [1993]), Nuprl (Constable et al. [1986]), Isabelle (Paulson [1994]) and many others.[7] Nuprl, in turn, spawned others like Coq (Coquand and Huet [1988]), Lego (Pollack [1995]), PVS (Owre et al. [1996]), and Nuprl-Light (Hickey [1997]).

This period saw also the second generation efforts to build computer systems to help create formalized mathematics. The Mizar checking effort (Trybulec [1983]) included starting the *Journal of Formalized Mathematics* (see www.mcs.anl.gov/qed). The Nuprl system focused on tools for synthesizing proofs and will be discussed in this article, but numerous other systems were created, e.g. Alf, Coq, HOL, IMPS, Isabelle, PVS, STeP by Constable et al. [1986], Paulin-Mohring and Werner [1993], Farmer [1990], Farmer, Guttman and Thayer [1991], Owre, Rushby and Shankar [1992], Owre et al. [1996], Bjørner et al. [1996], and others.[8] Some of these were integrated systems with special editors, a library of theories and various "logic engines." The 90's is seeing a move toward more modular and open systems, and in the 21st century we will see cooperative systems such as Howe's HOL-Nuprl effort [1996b].

This article will introduce some of the theory behind tactic-oriented theorem proving and will show examples of modern synthesized proofs. It will relate this to developments in typed programming languages.

1.2. Outline. Section 2 covers Typed Logic. The development moves from a *pure* logic of typed propositions to a calculus with specific types—**N** the natural numbers, cartesian products of types, *lists* over any types, functions from one type to another, sets built over a type, and types defined by imposing a new equality on our existing type... so called quotient or congruence types.

The exposition is very similar to the way logic is presented in Nuprl, but instead of deriving logic from type theory, we start with logic in the spirit of *Principia.* We also present logic in the manner of Automath, so one might call this logic "*AutoMathematica.*" It contrasts with the *polymorphic predicate calculus* of LCF (Gordon, Milner and Wadsworth [1979]) and HOL (Gordon and Melham [1993]).

Section 3 covers Type Theory. This is essentially an introduction to *Martin-Löf semantics* using an axiomatic theory close to his Intuitionistic Type Theory circa 1982-88 and its Nuprl variants (Martin-Löf [1982,1984,1983]). My approach is very "expository" since there are many accessible accounts in the books of Nordstrom, Petersson and Smith [1990], Martin-Löf [1984], Thompson [1991], the articles of Backhouse [1989], Martin-Löf [1982], Allen [1987a], and the theses of Allen [1987b],

[7] A comprehensive history of theorem proving up to 1970 can be found in Siekmann and Wrightson [1983]; see MacKenzie [1995] for recent developments.

[8] The web site www-formal.stanford.edu/clt/ARS/systems.html at Stanford lists these systems.

Palmgren [1991,1995b], Poll [1994], Setzer [1993], Rezus [1985], Helmink [1992] which provide technical detail. I use a small fragment to illustrate the theory and present Stuart Allen's non-type theoretic account [1987a] as well as a *semantics of proofs as objects.* Many of the ideas of type theory are discussed for the *impredicative theories* based on Girard's system F (see Girard, Taylor and Lafont [1989]). There is a large literature here as well Coquand and Huet [1988], Reynolds [1974], Luo [1994], Poll [1994]. Dependent types and universes are in this section. These types can then be added to the Typed Logic. So in a sense, this section extends Typed Logic.

Section 4 covers Typed Programming. Here I explain the notion of "partial types" common in programming and relate them to type theory. This account is expository, designed to make connections. But I discuss the recursive types that Constable and Mendler [1985] introduced which are closely related to the subsequent accounts for Coq and Alf of Coquand and Paulin-Mohring [1990], Dybjer [1994]. I then discuss a new type constructor due to Jason Hickey [1996a] — the *very dependent type.* These recursive and very dependent types can be added to the type theory and hence, to the Typed Logic. So this section too can be viewed as extending the logic of Section 2. This section provides the theoretical basis for understanding tactic-oriented proving, but there is no space to treat the subject further.

1.3. Highlights. Section 2 stresses a typed presentation of the predicate calculus because we deal with a general mechanism for making the judgment that a formula is *meaningful.* This is done first in the style of Martin-Löf as expressed in Nuprl, but I also mention the essential novelty in Nuprl's use of *direct computation* rules which go beyond Martin-Löf's inference rules by allowing reasoning about general recursive functions (say defined by the Y combinator). This is a powerful mechanism that is not widely known.

Section 2 also stresses the notion that *proof expressions* are sensible for classical logic (see Constable [1989]). This separates an important technique from its origins in constructive logic. So the account of Typed Logic covers both constructive as well as classical logic. It also lays the ground work for tactics.

Section 3 features a very simple fragment of type theory which illustrates the major design issues. The fragment has a simple (predicative) inductive semantics. We keep extending it until the semantics requires the insights from Allen [1987a].

The treatment of recursive types in Section 4 is quite simple because it deals only with partial types. It suggests that just as the notion of program "correctness" can be nicely factored into partial and total correctness, the rules for data types can also be factored this way.

Table 1 shows how similar concepts change their appearance as they are situated in the three different contexts.

2. Typed logic

Ordinary mathematical statements are usually expressed in a *typed language.* Consider the trivially true proposition: "if there is a rational number q whose

Typed Logic	Type Theory	Typed Programming
terms with binding	terms with binding	terms with binding
definitional equality	computation rules	observational equivalence & bisimulation
proofs as objects	content of proofs	programs
$Prop_i$	$Type_i$ or U_i	*Type* (partial types)
equality propositions	equality judgments	equality functions
consistency	nontriviality	representation integrity (internal & external)
standard models	inductive definitions	termination conditions
proof synthesis	element synthesis	program synthesis
tactic-tree proof	derivation	proof data type & tactics

Table 1: Type concepts

absolute value is less than the reciprocal of any natural number n, then for every natural number n, there is a real number r whose absolute value is also less than $1/n$." Symbolically we express this as follows for $\mathbf{N}$ the natural numbers, $\mathbb{Q}$ the rationals, and $\mathbb{R}$ the reals:

$$* \qquad \exists q{:}\mathbb{Q}.\ \forall n{:}\mathbf{N}.\ (|q| < 1/n) \Rightarrow \forall n{:}\mathbf{N}.\ \exists r{:}\mathbb{R}.\ (|r| < 1/n)\,.$$

We can abstract on the relation $|r| < 1/n$, and speak of any relation L on $\mathbf{N} \times \mathbb{R}$ and recognize that $\mathbb{Q}$ is a *subtype* of $\mathbb{R}$ to obtain

$$\exists q{:}\mathbb{Q}.\ \forall n{:}\mathbf{N}.\ L\,(n,q) \Rightarrow \forall n{:}\mathbf{N}.\ \exists r{:}\mathbb{R}.\ L\,(n,r)\,.$$

Abstracting further, we know that for any types A, A' and B where A' is a *subtype* of A, say $A' \sqsubseteq A$ the following is true

$$\exists a{:}A'.\ \forall x{:}B.\ L(x,a) \Rightarrow \forall x{:}B.\ \exists a{:}A.\ L\,(a,x)\,.$$

This last statement is an *abstraction* of $*$ with respect to $\mathbb{Q}, \mathbf{N}, \mathbb{R}$ and the relation $|r| < 1/n$. It is these purely *abstract typed propositions* that we want to study in the beginning. We want to know what statements are true regardless of the types and the exact propositions. We are looking for those properties of mathematical propositions that are invariant under arbitrary replacement of types and propositions.

Here logic is presented in a way that relates it closely to the type theory of section 3 and the programming language of section 4. Essentially, we will be able to lay one presentation on top of the others and see a striking correspondence as Table 1 suggests. This goal leads to a novel presentation of logic because of the role of explicit typing judgments. We begin now to gradually make these ideas more precise.

2.1. Propositions

Relation to sentences. Use of the word *proposition* in logic refers to an idea that is new in this century. The English usage is from Russell's reading of Frege. To explain propositions it is customary to talk first about *declarative sentences* in some natural language such as English. A sentence is an aggregation of words which expresses a complete thought, and when that thought is an assertion, e. g. $0 < 1$, then the sentence is declarative. The thought expressed is the *sense* of the sentence. We are interested in the conditions under which we can *assert a sentence* or *judge it to be true.*

Logic is not concerned directly with the nature of natural language and sentences. It is a more abstract subject. The abstract object corresponding to a sentence is a *proposition.* As Church [1960] says "...a proposition as we use the term, is an abstract object of the same general category as class, number or function." He says that any concept of *truth-value* is a proposition whether or not it is expressed in a natural language.[9]

This definition from Frege [1903] as explained by Church [1956] will suffice even for the varieties of constructive logic we will consider. We can regard truth-values themselves as abstractions from a more concrete relationship, namely that we know evidence for the truth of a formula; by forgetting the details of the evidence, we come to the notion of a *truth-value.* We say that the asserted sentence is *true.* Thus, when we judge that a sentence or expression is a proposition, we are saying that we know what counts as evidence for its truth, that is, we understand what counts as a proof of it.

It is useful to single out two special propositions, say $\top$ for a proposition agreed to be true, accepted as true without further analysis. We can say it is a *canonical true proposition*, a generalization of the concrete proposition $0 = 0$ in $\mathbf{N}$. We say that $\top$ is atomically true. Likewise, let $\bot$ be a canonically false proposition, a generalization of the idea $0 = 1$ in $\mathbf{N}$; it has no proof.

The category of propositions, *Prop*. In order to relate this account of typed logic to the type theory of section 3 and to programming section 4, I would like to consider the collection of all propositions and refer to it as *Prop*.[10] But we know already from *Principia* that the concept of proposition is *indefinitely extensible.* Here is how Whitehead and Russell put the matter.

> "...vicious circles...[arise] from supposing that a collection of objects may contain members which can only be defined by means of the collection as a whole. Thus, for example, the collection of *propositions* will be supposed to contain a proposition stating that 'all propositions are either true or false.' It would seem, however, that such a statement

[9]There are, of course, opposing views; Wittgenstein [1953,1922] was concerned with sentences or *formulas* as is Quine [1960], and we access propositions through specific formulas.

[10]In topos theory *Prop* and the true propositions form the *subobject classifier*, Ω, $\top$ see Bell [1988], MacLane and Moerdijk [1992], Lambek and Scott [1986].

could not be legitimate unless 'all propositions' referred to some already definite collection, which it cannot do if new propositions are created by statements about 'all propositions.' We shall, therefore, have to say that statements about 'all propositions' are meaningless. By saying that a set has 'no total,' we mean, primarily, that no significant statement can be made about 'all its members.' In such cases, it is necessary to break up our set into smaller sets, each of which is capable of a total. This is what the theory of types aims at effecting."

So we must be very careful about introducing this notion. There are *predicative* approaches to this which lead to *level restrictions* as in *Principia* and allow writing $Prop_i$ as the "smaller collections" into which *Prop* is broken (Martin-Löf [1982], Constable et al. [1986]).[11] There are *impredicative* approaches that allow *Prop* but restrict the logic in other ways (Girard, Taylor and Lafont [1989]). Ultimately we will impose level restrictions on *Prop* and relate it to another indefinitely extensible concept which we denote as *Type*. These restrictions and relationships will occupy us in section 3.

From the beginning I want to recognize that we intend to treat collections of propositions as mathematical objects — some collections will be types, some will be too "comprehensive" to be types. We will call these large collections *categories* or *classes* or *kinds* or *large types* to use names suggestive of "large collections."

The use of *Prop* as a concept in this account of logic, while not common in logic texts, provides an orientation to the subject which will gradually reveal problems that I think are both philosophically and practically important. So I adopt it here. We recognize at the onset that *Prop* will not denote *all* propositions. We will be developing an understanding of how to talk about such open-ended notions in a meaningful way. This development will confirm that at the very least we can *name them.* The issue is what else can we say?

The category of propositional functions. Frege's analysis of propositions into functions and arguments is central to modern logic. It requires us to consider the notion of a *propositional function.* Frege starts with concrete propositions such as $0 < 1$, then abstracts with respect to a term to obtain a form like $0 < x$ which denotes a *function in x.* In *Principia* notation, given a proposition ϕa, ϕx is the ambiguous value, and the propositional function is $\phi\hat{x}$; so $0 < 1$ can be factored as $\phi 1$ and abstracted as $0 < \hat{x}$. Also, in *Principia* a type is defined to be the range of significance of a propositional function; we might write the type as $type(x.\phi x)$. So the function maps this type to *Prop*. For example, $\frac{1}{3} < \frac{1}{2}$ might be abstracted to $\frac{1}{x} < \frac{1}{2}$; it is not meaningful for $x = 0$.

In the logic presented here, propositional functions also map types T to *Prop*. Given a type T we denote the category of propositional functions over T as $T \to Prop$. Instead of using $0 < x$, we denote the abstractions in the manner of Church

[11] In topos theory this leads to the Grothendieck topos which is definable in our predicative type theory of section 3.

with lambda notation, $\lambda(x.\ 0 < x)$. The details of the function notation will not concern us until section 2.9. It suffices now to say that given a specific proposition such as $0 < 1$, we require that the individuals such as $0, 1$ be elements of types, here $\mathbf{N}$. The function maps $\mathbf{N}$ to *Prop*, thus an element of the category $(\mathbf{N} \to Prop)$. Given a function P in $(T \to Prop)$ and t in the type T, then $P(t)$ denotes the application of P to t just as in ordinary mathematics.

Types. Types are structured collections of objects such as natural numbers, $\mathbf{N}$, or pairs of numbers $\mathbf{N} \times \mathbf{N}$ or lists of numbers, etc. In the section on type theory we will present specific concrete types, here we treat the notion abstractly. We think of the elements of the types as possibly given first without specifying the type; they might be built from sets for example or be the raw data in a computer (the bytes) or be physical objects or be given by constructions. Even when we are thinking of constructing objects according to a method specified by the type (as for the natural numbers based on zero and successor), still we imagine the object as existing without type information attached to it, and thus objects can be in more than one type.

The critical point about judging that T is a type is knowing what it means for an expression to be an element of T. This is what we know when we show that T is a type.

Assertions about objects require their classification into types. Moreover, we need relative or hypothetical classification because an object t occurring in a proposition may be built contingently from other objects x_i of type T_i, and will be in some type T. We understand these relative type membership judgments as follows. To judge that $t[x]$ is a member of T_2 provided x is a member of T_1 means that for any object t_1 of type T_1, $t[t_1]$ is an object of T_2. We write this as $x : T_1 \vdash t[x] \in T_2$. We extend the notation to n assumptions by writing $x_1 : T_1, \ldots, x_n : T_n \vdash t \in T$ for x_i distinct identifiers. (We write $t[x_1, \ldots, x_n]$ only when we need to be explicit about the variables; this notation is a *second order variable.*) We give concrete examples of this judgment below and discuss them at length in section 3.

We don't treat this judgment as an assertion. It summarizes what must hold for a statement about t to make sense. It is not the kind of expression that is true or false. For example, when we say $0 \in \mathbf{N}$, we are thinking that 0 is a natural number. This is a *fact.* We are thinking "$0\ is\_a\ \mathbf{N}$," and it does not make sense to deny this, thinking that 0 is something else.

In order to consider propositional functions of more than one argument, say $P(x, y)$, we explicitly construct pairs of elements, $\langle x, y\rangle$ and express $P(x, y)$ as $P(\langle x, y\rangle)$. The pair $\langle x, y\rangle$ belongs to the Cartesian product type. So our types have at least this structure.

Definition. If A, B are types, then so is their *Cartesian product,* $A \times B$. If $a \in A$ and $b \in B$ then the *ordered pair* $\langle a, b\rangle$ belongs to $A \times B$. We write $\langle a, b\rangle \in A \times B$. We write $1of\ (\langle a, b\rangle) = a$ and $2of\ (\langle a, b\rangle) = b$. Let A^n denote $A \times \cdots \times A$ taken n times.

Propositions are elements of the large type (or category) *Prop*. We use membership judgements

$$x_1 : T_1, ..., x_n : T_n \vdash P \in Prop$$

or an abbreviation $\mathcal{T} \vdash P \in Prop$ to define them. Atomic propositions are constants: $\top$ for a canonically true one and $\bot$ for a canonically false one, or propositional variables or applications of propositional function variables in the large type $(T \to Prop)$ for some type T. Compound propositions are built from the logical connectives $\&, \vee, \Rightarrow$ or the logical operators $\exists$ and $\forall$.

The membership rules are that if $\mathcal{T} \vdash F \in Prop$ and $\mathcal{T} \vdash G \in Prop$, then $\mathcal{T} \vdash (F\ op\ G) \in Prop$ for *op* a connective; and if $x_1 : T_1, ..., x_n : T_n \vdash F \in Prop$, then $\mathcal{T}' \vdash (Qx_i : T_i\,.\,F) \in Prop$ where Q is a quantification operator and where $\mathcal{T}'$ is obtained by removing the typing assumption $x_i : T_i$.

The usual names for the compound propositions are:

proposition		English equivalent	operator name
$(F \,\&\, G)$	is	F and G	conjunction
$(F \vee G)$	is	F or G	disjunction
$(F \Rightarrow G)$	is	F implies G	implication
$F(t)$	is	F at t	instance
$\forall x{:}A.F(x)$	is	for all x of type A, $F(x)$	universal quantification
$\exists x{:}A.F(x)$	is	for some x of type A, $F(x)$	existential quantification

With these definitions in place we can write examples of general typed propositions. We use *definiendum* == *definiens* to write definitions.

Definition. For any propositions F and G define

$$\begin{aligned} \neg F &== (F \Rightarrow \bot) \\ (F \Leftarrow G) &== (G \Rightarrow F) \\ (F \Leftrightarrow G) &== ((F \Leftarrow G) \,\&\, (F \Rightarrow G)). \end{aligned}$$

For $P : ((A \times B) \to Prop)$ let $P(x, y) == P(\langle x, y\rangle)$.

Examples. Let $P \in (A \to Prop), Q_i \in (B \to Prop), R \in ((A \times B) \to Prop)$ and $C \in Prop$.

1. $\forall x{:}A.\ \forall y{:}B.\ (P(x) \,\&\, Q(y)) \Leftrightarrow \forall x{:}A.\ P(x) \,\&\, \forall y{:}B.\ Q(y)$
2. $\exists x{:}A.P(x) \,\&\, \exists y{:}B.\ Q(y) \Rightarrow \exists z{:}A \times B.\ R(z)$
3. $\neg\forall x{:}A.\ P(x) \Leftrightarrow \exists x{:}A.\ \neg P(x)$
4. $\forall x{:}A.\ (C \Rightarrow P(x)) \Leftrightarrow (C \Rightarrow \forall x{:}A.\ P(x))$
5. $\exists y{:}B.\ \forall x{:}A.\ R(x, y) \Rightarrow \forall x{:}A.\ \exists y{:}B.\ R(x, y)$

2.2. Judgments and proofs

Knowledge arises when we judge a proposition to be true. A proposition P becomes an *assertion* when we judge it to be true or assert it. In *Principia* this judgment is called an *assertion* and is written $\vdash P$.

Normally we assert a proposition when we *know it to be true*, but people also make so-called "blind assertions" which are made without this knowledge but happen to be true because someone else knows this or the person "speaking blindly" discovers it later. (These "blind assertions" normally make a person, especially students in exams, anxious.)

Assertions are not the only form of knowledge. We follow Per Martin-Löf [1982] and speak also of judgments of the form $P \in Prop$ that we discussed above. These are *typing judgments*. They also convey knowledge and need to be made evident, but we consider them as a different category of knowledge from assertions. Indeed, we see that knowing these judgments is part of knowing truth judgments because we must know that the object P we are asserting is a proposition before (or as) we judge it to be true.[12]

In section 2.1 we treated this notion in the background with the notation $P \in Prop$, $P \in A \rightarrow Prop$, without explaining judgments in general. In most logic textbooks this judgment is reduced to a question of correct parsing of formulas, i.e., is made *syntactic* and is thus prior to truth. But we follow Russell and Martin-Löf in believing it to be a semantic notion that can be treated in other ways when we formalize a theory.

For the predicate calculus, we leave most of these typing judgments implicit and adopt the usual convention that all the formulas we examine represent well-formed propositions. In the full typed logic this condition cannot be checked syntactically, and explicit typing judgments must be made.

In general, to make a judgment is to know it or for it to be *evident*. It does not make sense for a judgment to be evident without a person knowing it. However, we recognize that there are propositions which are true but were not known at a previous time. So at any time there are propositions which are not known, but will be asserted in the future.

True propositions and proofs. One of the most interesting properties of a proposition is whether the thought it expresses is *true*. Here is a more concrete way to say this. To grasp the sense of a proposition is to understand what counts as evidence for its truth. To know whether a proposition is true, we must find evidence for it. Trying to find evidence is a *mathematical task*, and it is an abstract activity. [13]

The evidence for an assertion is called a *proof*. Proving a proposition is the way

[12]In Martin-Löf's theorem these judgments are *prior* to assertion; in Nuprl they can be simultaneous with assertion.

[13]But the consequences of finding a proof or a disproof can be very concrete and very significant. For instance, a purported proof might cause someone to test a device in the belief that it is safe to do so. If the proof is flawed, the device might destroy something of great value.

of knowing it.[14] When we judge that an expression is a proposition, we specify what counts as a proof of it. For the propositions $\top, \bot$ $(P\&Q), (P \vee Q), \exists x : A.\ P(x)$ it is easy to specify this. To prove $\top$ we could cite an axiom, that is, we say it is *self-evident.* As we said before, $\bot$ has no proofs. To prove $(P\&Q)$ we prove P and prove Q. To prove $(P \vee Q)$ we either prove P or prove Q. To prove $\exists x{:}A.\ P(x)$ we exhibit an element a of A and prove $P(a)$.

To say what it means to prove $(P \Rightarrow Q)$ we need to understand logical consequence or what is the same in this article, *hypothetical proof.* We write this as *hypothetical judgment* $P \vdash Q$ which reads, assuming P is true, we can prove that Q is. What this means is that if we have a proof p of P, then we can build a proof q of Q using p.

To discuss nested implications such as $P \Rightarrow (Q \Rightarrow P)$ we need to understand hypothetical judgments of the form $P_1, \ldots, P_n \vdash Q$ which means that assuming we have proofs p_i of P_i, we can find a proof q of Q.

To prove $\forall x{:}A.P(x)$ we need the hypothetical judgment $x{:}A \vdash P(x)$ which says that given any a of type A, we can find a proof of $P(a)$. Combining these two forms of hypothetical judgments we will need to consider $H_1, \ldots, H_n \vdash P$ where H_i can be either a proposition or a type declaration such as $x{:}A$.

This account of provability gives what we call a *semantics of evidence.* Depending on the interpretation of the functionality, judgments like $P \vdash Q$ or $x{:}A \vdash P(x)$ we can explain both classical and constructive logic in this way. These ideas will become clearer as we proceed.

In general, there is no systematic way to search for a proof. Indeed, the notion of proof we have in mind is not for any *fixed* formal system of mathematics. We are interested mainly in open-ended notions. Like the concept of a proposition, the concept of a proof is inexhaustible or *open* or *creative.* By Gödel [1933] and Tarski [1956] we know that for any consistent *closed non-trivial formal system,* we can systematically enlarge it, namely by adding a rule asserting its consistency or giving a definition of truth for that system.

For the collection of *pure* abstract propositions, there is a systematic way to search for a proof. If we want to describe this procedure, then it is necessary to have a *representation* of propositions as data that lets us access their structure. We can do this with the inductive definition of propositions given next, but it is more natural to build a representation that directly expresses the *linguistic structures* that we use when writing and manipulating formulas. This is the traditional approach to the study of logic and provability. We take it up in the subsections on Formulas and Formal Proofs.

[14]When we need to present evidence for a typing judgment, we will incorporate that into our proofs as well and speak of proving a typing. One might want to give this a special name, such as a *derivation,* but in Nuprl we use the term proof. These typing "proofs" never have interesting computational content.

2.3. Pure propositions

The traditional approach to studying logic is to first isolate the propositional calculus and then the first-order predicate calculus, and to study completeness results for these calculi, since completeness results tie together the semantic and proof-theoretic concepts. These topics are covered by defining restricted formal languages. We will take this approach to typed logic starting in section 2.4 on formulas. But many of the concepts can also be presented by a mathematical analysis of propositional functions without resort to linguistic mechanisms and formulas.

There is one outstanding technical problem about the propositional calculus which benefits from this direct analysis, namely a functional approach to completeness of the intuitionistic propositional calculus. The most widely known completeness theorem for this calculus is based on Kripke semantics (Kripke [1965]); this account of propositional calculus semantics, while illuminating and technically elegant, and even constructively meaningful, is not *faithful* to constructive semantics. In particular, it is not based on the type-theoretic semantics we offer in part 3. Providing a completeness result for a *constructively faithful* semantics is an open area, and it seems to me that Martin-Löf's inductive semantics has created new opportunities here to produce definitive results.

The key to a constructive semantics might be a careful study of a functional approach to propositions that allows us to express the functional *uniformity of proofs* that is central to completeness.[15] As a start, we can try to understand the basic concept of a pure propositional function without resort to formal languages.

Consider a propositional function of one argument, $P \Rightarrow P$. This can be understood as a function from $Prop$ to $Prop$.[16] The function $P \Rightarrow (Q \Rightarrow P)$ in variables P, Q is a two-argument function, most naturally from $Prop$ to $(Prop \rightarrow Prop)$ as would be clear from writing the function as

$$\lambda(P.\ \lambda(Q.\ P \Rightarrow (Q \Rightarrow P))).$$

We could also think of this as a mapping from $Prop \times Prop$ to $Prop$ if we took pairs as arguments (writing $\lambda(z.\ 1of(z) \Rightarrow (2of(z) \Rightarrow 1of(z)))$ in $Prop^2 \rightarrow Prop$). For ease of analysis, we will consider propositional functions from the Cartesian power, $Prop^n$, into $Prop$. The constants $\top$ and $\bot$ are regarded as zero-ary functions, and for convenience define $Prop^0 = \mathbf{1}$ for $\mathbf{1}$ the *unit type*. Then $f(x) = \top$ and $f(x) = \bot$ are in $Prop^0 \rightarrow Prop$.

The idea is to define the pure propositional functions inductively as a subtype of $Prop^n \rightarrow Prop$ constructed using only constant functions, simple projections like $proj_i^n(P_1, \ldots, P_n) = P_i$ and the operations $\&, \vee, \Rightarrow$ lifted up to the level of functions.

Each connective $\&, \vee, \Rightarrow$ can be lifted to the functions $Prop^n \rightarrow Prop$, namely given f and g, define $(f\ op\ g)(\bar{P}) = f(\bar{P})\ op\ g(\bar{P})$ where $\bar{P} \in Prop^n$. For example,

[15] Läuchli [1970] tries to express this uniformity using permutations.

[16] We will deal later with the issue of equality on $Prop$, which seems necessary to talk about *functions*.

if $f(P,Q) = P$ and $g(P,Q) = (Q \Rightarrow P)$ then $f \Rightarrow g$ is a function h such that $h(P,Q) = (P \Rightarrow (Q \Rightarrow P))$.

We can now define the general abstract propositional functions of n variables call the class $\mathcal{P}_n$ as the inductive subset of $Prop^n \to Prop$ whose base elements are the constant and projection functions,

$$C_\top(\bar{P}) = \top \qquad C_\perp(\bar{P}) = \perp$$
$$proj_i^n(\bar{P}) = P_i \quad \text{where } \bar{P} = \langle P_1, \ldots, P_n \rangle \text{ and } 1 \leq i \leq n.$$ [17]

Then given $f, g \in \mathcal{P}_n$ and given any lifted connective op, we have $(f\ op\ g) \in \mathcal{P}_n$. Nothing else belongs to $\mathcal{P}_n$. When we want to mention the underlying type, we write $\mathcal{P}_n$ as $\mathcal{P}(Prop^n \to Prop)$. Let $\mathcal{P} = \bigcup_{n=0}^{\infty} \mathcal{P}_n$; these are the *pure* propositions. Note that $\mathcal{P} = \bigcup_{n=0}^{\infty} \mathcal{P}(Prop^n \to Prop)$ is inductively defined. The *valid* elements of $\mathcal{P}$ are those functions $f \in \mathcal{P}$ such that for $f \in \mathcal{P}_n$ and $\bar{\mathcal{P}}$ *any* element of $Prop^n$, $f(\bar{\mathcal{P}})$ is true. Call these $True(\mathcal{P})$.

Using these concepts we can express the idea of a uniform functional proof. The simplest approach is probably to use a Hilbert style axiomatic base. If we take Heyting's or Kleene's axioms for the intuitionistic propositional calculus, then we can define $Provable_H(\mathcal{P})$ inductively. The completeness theorem we want is then $True(\mathcal{P}) = Provable_H(\mathcal{P})$.

We can use the same technique to define the pure typed propositional functions. First we need to define *pure type functions* $\mathcal{T}$ as a subset of $\text{Type}^n \to \text{Type}$ for $n = 1, 2, \ldots$. We take $n \geq 1$ since there are as yet no constant types to include. An example is $t(A,B) = A \times B$. Next we define the typed propositional functions $p : t(\bar{T}) \to Prop$.

In general we need to consider functions whose inputs are n-tuples of the type

$$(t_1(\bar{T}) \to Prop) \times \ldots \times (t_n(\bar{T}) \to Prop)$$

and whose output is a $Prop$. We do not pursue this topic further here, but when we examine the proof system for typed propositions we will see that it offers a simple way to provide abstract proofs for pure typed propositions that use only rules for the connectives and quantifiers — say a *pure proof*. There are various results suggesting that if there is any proof of these pure propositions, then there is a *pure proof*. These are *completeness results* for this typed version of the predicate calculus. We will not prove them here.

2.4. Formulas

Propositional calculus. Consider first the case of formulas to represent the pure propositions. The standard way to do this is to inductively define a class of *propositional formulas*, *PropFormula*. The base case includes

[17] Since we do not study any mapping of formulas to pure propositions, I have not worried about relating elements of P_n and P_m, $n < m$, by coherence conditions.

the propositional constants, $Constants = \{\top, \bot\}$, and propositional variables, $Variables = \{P, Q, R, P_1, Q_1, R_1, \ldots\}$. These are propositional formulas. The inductive case is

If F, G are *PropFormulas*, then so are $(F \,\&\, G)$, $(F \vee G)$, and $(F \Rightarrow G)$. Nothing else is a *PropFormula*.

We can assign to every formula a mathematical meaning as a pure proposition. Given a formula F, let $P_1, \ldots, P_n$ be the propositional variables occurring in it (say ordered from left to right); let $\bar{P}$ be the vector of them. Define a map from n variable propositional formulas, $PropFormulas_n$, into $(Prop^n \rightarrow Prop)$ inductively by

$$\begin{array}{lcl} [\![P_i]\!] & = & proj_i^n \\ [\![(F \,\&\, G)]\!] & = & [\![F]\!] \,\&\, [\![G]\!] \\ [\![(F \vee G)]\!] & = & [\![F]\!] \vee [\![G]\!] \\ [\![(F \Rightarrow G)]\!] & = & [\![F]\!] \Rightarrow [\![G]\!]. \end{array}$$

For each variable P_i, corresponds to the projection function $proj_i^n(\bar{P}) = P_i$. Say that F is *valid* iff $[\![F]\!]$ is a valid pure proposition.

Boolean valued formulas. If we consider a single-valued relation from propositions to their truth values, taken as Booleans, then we get an especially simple semantics. Let $\mathbb{B} = \{tt, ff\}$ and let $\boldsymbol{B} : Prop \times \mathbb{B} \rightarrow Prop$ such that $P \Leftrightarrow \boldsymbol{B}(P, tt)$.

In classical mathematics one usually assumes the existence of a *function* like $\boldsymbol{b}$, say $\boldsymbol{b} : Prop \rightarrow \mathbb{B}$ where $P \Leftrightarrow \boldsymbol{b}(P) = tt$ in $\mathbb{B}$. But since $\boldsymbol{b}$ is not a computable function, this way of describing the situation would not be used in constructive mathematics. Instead we could talk about "decidable propositions" or "boolean propositions."

$$BoolProp == \{\langle P, v\rangle : Prop \times \mathbb{B} | P \Leftrightarrow (v = tt \; in \; \mathbb{B})\}$$

Then there is a function $\boldsymbol{b} \in BoolProp \rightarrow \mathbb{B}$ such that $P \Leftrightarrow (\boldsymbol{b}(P) = tt \text{ in } \mathbb{B})$.

If we interpret formulas as representing elements of pure *boolean propositions*, then each variable P_i denotes an element of $\mathbb{B}$. An *assignment* α is a mapping of variables into $\mathbb{B}$, that is, an element of $Variables \rightarrow \mathbb{B}$. Given an assignment α we can compute a boolean value for any formula F. Namely

$$\begin{array}{ll} Value(F, \alpha) = & \text{if } F \text{ is a variable, then } \alpha(F) \\ & \text{if } F \text{ is } (F_1 \; op \; F_2) \text{ then } Value(F_1, \alpha) \; bop \; Value(F_2, \alpha) \end{array}$$

where *bop* is the boolean operation corresponding to the propositional operator *op* in the usual way, e. g.

P	Q	P $\&_b$ Q	P $\vee_b$ Q	P $\Rightarrow_b$ Q
tt	tt	tt	tt	tt
ff	tt	ff	tt	tt
tt	ff	ff	tt	ff
ff	ff	ff	ff	tt

Typed propositional formulas. To define typed propositional formulas, we need the notion of a type expression, a term, and a *type context* because formulas are built in a type context. Then we define propositional variables and propositional-function variables which are used along with terms to make atomic propositions in a context. From these we build compound formulas using the binary connectives $\&, \vee, \Rightarrow$, and the *typed quantifiers* $\forall x : A, \exists x : A$. We let *op* denote any binary connective and $Qx{:}A$ denote either of the quantifiers.

Type expressions. Let $A_1, A_2, \ldots$ be type variables, then A_i are type expressions.

If T_1, T_2 are type expressions, then so is $(T_1 \times T_2)$.
Nothing else is a type expression for now.

Terms. Let $x_1, x_2, \ldots$ be individual variables (or element variables); they are *terms.* If s, t are terms, then so is the ordered pair $\langle s, t\rangle$. Nothing else is a term for now.

Typing contexts. If $T_1, \ldots, T_n$ are type expressions and x_i, $i = 1, \ldots, n$ are *distinct* individual variables, then $x_i : T_i$ is a *type assumption* and the list $x_1{:}T_1, \ldots, x_n{:}T_n$ is a *typing context.* We let $\mathcal{T}, \mathcal{T}', \mathcal{T}_j$ denote typing contexts.

Typing judgments. Given a typing context, $\mathcal{T}$, we can assign types to terms built from the variables in the context. The judgment that term t has type T in context $\mathcal{T}$ is expressed by writing

$$\mathcal{T} \vdash t \in T.$$

If we need to be explicit about the variables of $\mathcal{T}$ and t, we use a *second-order variable* $t[x_1, \ldots, x_n]$ and write

$$x_1{:}T_1, \ldots, x_n{:}T_n \vdash t[x_1, \ldots, x_n] \in T$$

When using a second-order variable we know that the only variables occurring in t are x_i. We call these variables of t *free variables.*

Later, we give rules for knowing these judgments. Just as we said in section 2.2, it should be noted that $t \in T$ is *not a proposition*; it is *not* an expression that has a truth value. We are saying what an ordered pair *is* rather than giving a property of it. So the judgment $t \in T$ is giving the meaning of t and telling us that the expression t is *well-formed* or meaningful. In other presentations of predicate logic these judgments are incorporated into the syntax of terms, and there is an algorithm to check that terms are meaningful before one considers their truth. We want a more flexible approach so that typing judgments need not be decidable.

We let $P_1, P_2, \ldots$ denote *propositional variables*, writing $P_i \in Prop$, for propositional function variables, writing $P_i \in (T \to Prop)$ for T a type expression.

If $\mathcal{T} \vdash t \in T$ and $P \in (T \to Prop)$, then $P(t)$ is an *atomic formula in the context* $\mathcal{T}$ with the variables occurring in t *free*; it is an *instance* of P. Note, we abbreviate $P(\langle t_1, \ldots, t_n\rangle)$ by $P(t_1, \ldots, t_n)$. If t is a variable, say x, then $P(x)$ is

an *arbitrary instance* or *arbitrary value* of P with free variable x. A propositional variable, $P_i \in Prop$, is also an atomic formula.

If F and G are formulas with free variables $\bar{x}, \bar{y}$ respectively in contexts $\mathcal{T}$, and if op is a connective and $Qx{:}A$ a quantifier, then

$$(F\ op\ G)$$

is a formula with free variables $\{\bar{x}\} \cup \{\bar{y}\}$ in context $\mathcal{T}$ and with *immediate subformulas* F and G;

$$Qv{:}T.F$$

is a formula in context $\mathcal{T}'$ where $\mathcal{T}'$ is $\mathcal{T}$ with $v{:}A$ removed; this formula has leading *binding operator* $Qv{:}A$ with *binding occurrence* of v whose *scope* is F, and its free variables are $\{\bar{x}\}$ with v removed, and all free occurrences of v in F become *bound* by $Qv{:}A$; its immediate subformula is F.

A formula is *closed* iff it has no free variables; such a formula is well-formed in an empty context, but its subformulas might only be well-formed in a context. A *subformula* G of a formula F is either an immediate subformula or a subformula of a subformula.

Examples. Here are examples, for $P_1 : A \to Prop$, $P_2 : B \to Prop$, $P_3 : A \times B \to Prop$.

$$(\forall x{:}A.\ \exists y{:}B.\ P_3(x,y) \Rightarrow (\exists x{:}A.\ P_1(x)\ \&\ \exists y : B.\ P_2(x)))$$

is a closed formula. $\forall x{:}A.\ \exists y{:}B.\ P_3(x,y)$ is an immediate subformula which is also closed, but $\exists y{:}B.\ P_3(x,y)$ is not closed since it has the free variable $x{:}A$; this latter formula is well-formed in the context $x{:}A$.

The atomic subformulas are $P_1(x), P_2(y)$, and $P_3(\langle x,y\rangle)$ which are formulas in the context $x{:}A,\ y{:}B$, and the typing judgment $x{:}A,\ y{:}B \vdash \langle x,y\rangle\ \in A \times B$ is used to understand the formation of $P_3(x,y)$ (which is an abbreviation of $P_3(\langle x,y\rangle)$).

2.5. Formal proofs

There are many ways to organize formal proofs of typed formulas, e. g. natural deduction, the sequent calculus, or its dual, tableaux, or Hilbert style systems to name a few. We choose a sequent calculus presented in a top-down fashion (as with tableaux). We call this a *refinement logic* (RL). The choice is motivated by the advantages sequents provide for automation and display.[18] Here is what a simple proof looks like for $A \in Type$, $P \in A \to Prop$; only the relevant hypotheses are mentioned and only the first time they are generated.

[18]This is the mechanism used in Nuprl and HOL; PVS uses multiple conclusion sequents.

	$\vdash \forall x : A.\ (\forall y : A.\ P(y) \Rightarrow \exists x : A.\ P(x))$	by $\forall R$
1.	$x : A \vdash \forall y : A.\ P(y) \Rightarrow \exists x : A.\ P(x)$	by $\Rightarrow R$
1.1	$f : (\forall y : A.\ P(y)) \vdash \exists x : A.\ P(x)$	by $\forall L$ on f with x
1.1.1	$l : P(x) \vdash \exists x : A.\ P(x)$	by $\exists R$ with x
1.1.1.1	$\vdash P(x)$	by *hyp* l
1.1.1.2	$\vdash x \in A$	by *hyp* x
1.1.2	$\vdash x \in A$	by *hyp* x

The schematic tree structure with path names is

$$
\begin{array}{c}
\vdash G \\
| \\
1.\quad H_1 \vdash G_1 \\
| \\
1.1\quad H_2 \vdash G_2 \\
/\ \backslash \\
1.1.1\quad H_3 \vdash G_2 \qquad 1.1.2\quad H_2 \vdash G_3 \\
/\ \backslash \\
1.1.1.1\quad H_3 \vdash G_4 \qquad 1.1.1.2\quad H_3 \vdash G_3
\end{array}
$$

Sequents. The nodes of a proof tree are called *sequents.* They are a list of *hypotheses* separated by the assertion sign, $\vdash$ (called turnstile or proof sign) followed by the *conclusion.* A hypothesis can be a typing assumption such as $x : A$ for A a type or a *labeled assertion*, such as $l : P(x)$. The label l is used to refer to the hypothesis in the rules. The occurrence of x in $x : A$ is an *individual variable*, and we are assuming that it is an object of type A. So it is an assumption that A is inhabited. Here is a sequent,

$$\boxed{x_1 : H_1, \ldots, x_n : H_n \vdash G}$$

where H_i is an assertion or a type and x_i is either a label or a variable respectively. The x_i are all distinct. G is always an unlabeled formula. We can also refer to the hypothesis by number, $1 \ldots n$, and we refer to G as the 0-th component of the sequent. We abbreviate a sequent by $\bar{H} \vdash G$ for $\bar{H} = (x_1 : H_1, \ldots, x_n : H_n)$; sometimes we write $\bar{x} : \bar{H} \vdash G$.

Rules. Proof rules are organized in the usual format of the single-conclusion sequent calculus. They appear in a table shortly. We explain now some entries of this table. There are two rules for each logical operator (connective or quantifier). The *right*

rule for an operator tells how to decompose a conclusion formula built with that operator, and the *left rule* for an operator tells how to decompose such a formula when it is on the left, that is when it is a hypothesis. There are also trivial rules for the constants $\top$ and $\bot$ and a rule for hypotheses. So the rules fit this pattern and are named as shown.

	Left	Right
&	&L	&R
$\vee$	$\vee$L	$\vee$Rl $\vee$Rr
$\Rightarrow$	$\Rightarrow$L	$\Rightarrow$R
$\forall$	$\forall$L	$\forall$R
$\exists$	$\exists$L	$\exists$R
$\top$	–	$\top$R
$\bot$	$\bot$L	–

$$x_1 : H_1, \ldots, x_n : H_n \vdash H_i \text{ by } hyp\ x_i$$
$$1 \leq i \leq n$$

The $\vee$R rule splits into two, $\vee$Rl and $\vee$Rr. To prove $\bar{H} \vdash P \vee Q$ we can prove $\bar{H} \vdash P$ or we can prove $\bar{H} \vdash Q$. The first rule is $\vee$Rl because we try to prove the *left* disjunct. We pronounce these somewhat awkwardly as "or right left" and "or right right."

The other rules all have obvious pronunciations, e. g. "and left" for &L, "and right" for &R, etc. Some of them have ancient names, such as *ex falso libet* for "false left" (meaning "anything follows from false"). In Nuprl we use *any* for $\bot$L. Sometimes $\vee$L is called "cases", and instead of "by $\vee$L l" we might say "by cases on l" for l a label. For $\forall$L on l with a term t we might say "instantiating l with t."

The rule $\Rightarrow L$ is similar to the famous *modus ponens* rule usually written as

$$\frac{A \quad A \Rightarrow B}{B}.$$

In top down form it would be

$$\begin{array}{l} A, A \Rightarrow B \vdash B \quad \text{by } \Rightarrow L \\ \quad A, B \vdash B \\ \qquad A \vdash A \end{array}$$

Some of the rules such as $\forall$L and $\exists$R require *parameters*. For example, to decompose $\forall x : T.P(x)$ as a hypothesis, we need a term $t \in T$. So the rule is $\forall$L on t. For $\exists x{:}T.\, P(x)$ as a goal, to decompose it, we also need a term $t \in T$; the decomposition generates the subgoal $P(t)$.

Magic rule. These rules do not allow us to prove the formula $P \vee \neg P$ nor $\neg\neg P \Rightarrow P$ nor any equivalent formula. If we add one of these formulas as an axiom scheme then we can prove the others. We can also prove them by adopting the *proof by contradiction* rule

$$\begin{array}{l} H \vdash P \text{ by } \textit{contradiction} \\ H, \neg P \vdash \bot \end{array}$$

My preference is to base arguments for these formulas on the axiom scheme $P \vee \neg P$ called the *law of excluded middle* because these arguments have a special status in relating logic to computation and because this law is so important in philosophical and foundational discussions. In the organization I adopt, this is the only rule which does not fit the sequent pattern and it is the only rule not constructively justifiable as we will see later. I sometimes call the rule "magic" based on the discussion of justification to follow.

Justifications. The rule names and parameters to them make up a very important part of the proof called the *justification* of the inference step. We can think of the justification as an *operator on sequents* which decomposes the goal sequent into a subgoal sequents. This format for the justification reveals that role graphically.

$$\begin{array}{ll} \bar{x} : \bar{H} \vdash G & \text{by } r(\bar{x}; \bar{t}) \\ \quad 1.\bar{H}_1 \vdash G_1 & \\ \quad \vdots & \\ \quad k.\bar{H}_k \vdash G_k & \end{array}$$

For example

$$\begin{array}{l} \bar{H} \vdash (P \vee Q) \text{ by } \vee Rl \\ \quad 1.\bar{H} \vdash P \end{array}$$

The justification takes the variables and labels of $\bar{x}$ plus some parameters $\bar{t}$ and *generates* the k subgoals $\bar{H}_i \vdash G_i$. The hypothesis rule generates no subgoals and so terminates a branch of the proof tree. Such rules are thus found at the leaves.

By putting into the justifications still more information, we can reveal all the links between a goal and its subgoals. To illustrate this information, consider the $\Rightarrow$L rule and the &L rule.

$$\boxed{\begin{array}{l} \bar{H}, f : (P \Rightarrow Q), \bar{J} \vdash G \text{ by } \Rightarrow L \text{ on } f \\ 1.\ \bar{H}, f : (P \Rightarrow Q), \bar{J} \quad \vdash P \\ 2.\ \bar{H}, f : (P \Rightarrow Q), \bar{J}, y{:}Q \vdash G \end{array}}$$

$$\boxed{\begin{array}{c} \bar{H}, pq{:}P \,\&\, Q \vdash G \text{ by } \&L \\ \bar{H}, pq{:}P \,\&\, Q, p{:}P, q{:}Q, \bar{J} \vdash G \end{array}}$$

If the $\Rightarrow$R justification provided the label y, then all the information for generating the subgoal would be present. If the &L rule provided the labels p, q then the data is present for generating its subgoals as well. So we will add this information to form a *complete justification.*

Notice that these labels behave like the variable names x_i in the sense that we can systematically *rename* them without changing the meaning of a sequent or a justification. They act like *bound variables* in the sequent. The phrase new u, v in a justification allows us to explicitly name these bound variables.

Structural rules. Sequents as defined here are based on lists of formulas, so the rules for decomposing on the left must refer to the position of the formula. This is indicated by supplying a context around the formula, typically of the form $\bar{H}, x : F, \bar{J} \vdash G$. The *cut* rule specifies the exact location at which the formula is to be introduced into a hypothesis list, and *thin* does the same.

By combining applications of *cut* and *thin*, hypotheses can be moved (exchanged) or contracted. The so-called *structural rules* are included among these rules.

2.6. Proof expressions and tactics

Complete justifications. If there is enough information in a justification to generate the subgoals, then the tree of justifications and the top goal can *generate* the whole proof. Moreover, the tree of justifications can be combined into a single "algebraic expression" describing the whole proof. Indeed, the proof tree stripped of the sequents is just a parse tree for this expression.

If we present the justifications in the right way we can read the rules annotated by them as an *attribute grammar* (c.f. Reps and Teitelbaum [1988], Reps [1982], Griffin [1988a]) for generating an expression describing the proof called a *proof expression.* Consider the case of the $\Rightarrow L$ and $\&L$ rules again. Suppose we let p and g denote proof expressions for the subgoals, then

$$\begin{array}{ll} \bar{x} : \bar{H}, f : (P \Rightarrow Q), \bar{J} \vdash G \text{ by } \Rightarrow L \text{ on } f & \\ \quad — \quad \vdash P & \text{by } p(\bar{x}) \\ \quad —, y : Q \vdash G & \text{by } g(\bar{x}, y) \end{array}$$

If we think of the proof expressions $p(\bar{x})$ and $g(\bar{x}, y)$ as being *synthesized* up from the subtrees, then the complete proof information for the goal sequent is

$$\boxed{\Rightarrow L \text{ on } f \text{ from } p(\bar{x}) \text{ and from } g(\bar{x}, y) \text{ with new } y}$$

Organizing this into a more compact expression and recognizing that y is a new bound variable, a suggestive expression is

$$\boxed{\Rightarrow L(f; p(\bar{x}); y\,.g(\bar{x}, y))}$$

Here we use the "dot notation" $y.g(\bar{x}, y)$ to indicate that y is a new *bound label* in the proof expression $g(\bar{x}, y)$. The dot notation is used with quantifiers as in $\forall x{:}A.\, F$ to separate the binding operator $\forall x{:}A$ from the formula F. Likewise, in the lambda notation, $\lambda(x.b)$, the dot is used to indicate the beginning of the scope of the binding of x.

In the case of $\&L$, the rule with proof expressions looks like

$$\bar{x}{:}\bar{H}, z{:}P\&Q \vdash G \text{ by } \&L \text{ in } z \text{ with new } u, v$$
$$\bar{x}{:}\bar{H}, u{:}P, v{:}Q \vdash G \text{ by } g(\bar{x}, u, v)$$

A compact notation is

$$\&L(z; u, v.\, g(\bar{x}, u, v))$$

Here u, v are new labels which again behave like bound variables in the proof expression.

The justification for $P \vee \neg P$ will be the term $magic(P)$. This is the only justification term that requires the formula as a subterm.

With this basic typed predicate logic as a basis, we will now proceed to add a number of specific types, namely natural numbers, lists, functions, sets over a type, and so-called quotient types. Each of these shows an aspect of typed logic. Note, in these rules we are presupposing that P, Q, and the formulas in $\bar{H}$ are well-formed according to the definition of a formula and that the type expressions are also well-formed in accordance with the typing rules. As we introduce more types, it will be necessary to incorporate typing judgments as subgoals. The Nuprl logic of Constable et al. [1986] relies on such subgoals from the beginning so that the caveat just stated for this table of rules is unnecessary there.

Tactics. Complete justifications will generate the entire proof given the goal formula because the rule name, and labeling formation and parameters are enough data to generate subgoals from the goals. So the subgoals are computable from the part of the justification that does not include the proof expression for the subproofs (the synthesized expressions). This fact suggests a way to automate interactive proof generation. Namely, a program called a *refiner*, takes a goal and a complete justification and produces the subgoals. Nuprl works this way.

Nuprl also adapts tactics from LCF (Gordon, Milner and Wadsworth [1979]) into the proof tree setting to get a notion of *tactic-tree proof* (Allen et al. [1990], Basin and Constable [1993], Griffin [1988b]). In this setting the justifications are called primitive refinement tactics. These can be combined using procedures called *tacticals*. For example, if a refinement r_o generates subgoals $G_1, \ldots, G_n$ when applied to sequent G_o, then the compound refinement tactic written r_o THENL$[r_1; \ldots; r_n]$ executes r_o, then applies r_i to subgoal G_i generated by r_o.

There are many tacticals (c.f. Jackson [1994a], Constable et al. [1986]); two basic ones are ORELSE and REPEAT. The ORELSE tactical relies on the idea that a refinement might fail to apply, as in trying to use $\&R$ on an implication. In r_o ORELSE r_1, if r_o fails to decompose the goal, then r_1 is applied.

Table of justification operators

	Left(L)	Right(R)
&	$\bar{H},\ x{:}P\&Q,\ \bar{J} \vdash G$ by $\&L(x;u,\ v.\ g(u,v))$ 1. $\bar{H},\ x{:}P\&Q,\ u{:}P,\ v{:}Q,\ \bar{J} \vdash G$ by $g(u,v)$	$\bar{H} \vdash P\&Q$ by $\&R(p;q)$ 1. $\bar{H} \vdash P$ by p 2. $\bar{H} \vdash Q$ by q
$\vee$	$\bar{H},\ x{:}P \vee Q,\ \bar{J} \vdash G$ by $\vee L(x;u.\ g_l(u);v.\ g_r(v))$ 1. $\bar{H},\ x{:}P \vee Q, u{:}P,\ \bar{J} \vdash G$ by $g_l(u)$ 2. $\bar{H},\ x{:}P \vee Q, v{:}Q,\ \bar{J} \vdash G$ by $g_r(v)$	$\bar{H} \vdash P \vee Q$ by $\vee Rl(p)$ 1. $\bar{H} \vdash P$ by p $\bar{H} \vdash P \vee Q$ by $\vee Rr(q)$ 1. $\bar{H} \vdash Q$ by q
$\Rightarrow$	$\bar{H},\ x{:}P \Rightarrow Q,\ \bar{J} \vdash G$ by $\Rightarrow L(x;p;y.\ g(y))$ 1. $\bar{H},\ x{:}P \Rightarrow Q,\ \bar{J} \vdash P$ by p 2. $\bar{H},\ x{:}P \Rightarrow Q,\ \bar{J},\ y{:}Q \vdash G$ by $g(y)$	$\bar{H} \vdash P \Rightarrow Q$ by $\Rightarrow R(x.\ q(x))$ $\bar{H},\ x{:}P \vdash Q$ by $q(x)$
$\forall$	$\bar{H},\ x{:}\forall z{:}A.\ P(z),\ \bar{J} \vdash G$ by $\forall L(x;a;y.\ g(y))$ 1. $\bar{H},\ x{:}\forall z{:}A.\ P(z),\ \bar{J} \vdash a \in A$ 2. $\bar{H},\ x{:}\forall z{:}A.\ P(z),\ \bar{J},\ y{:}P(a) \vdash G$ by $g(y)$	$\bar{H} \vdash \forall z{:}A.P(z)$ by $\forall R(z.\ p(z))$ new w $\bar{H},\ w{:}A \vdash P(w)$ by $p(w)$
$\exists$	$\bar{H},\ x{:}\exists z{:}A.\ P(z),\ \bar{J} \vdash G$ by $\exists L(x;u,v.\ g(u,v))$ new u,v $\bar{H},\ x{:}\exists z{:}A.\ P(z),\ u{:}A, v{:}P(u),\ \bar{J} \vdash G$ by $g(u,v)$	$\bar{H} \vdash \exists z{:}A.\ P(z)$ by $\exists R(a;p)$ 1. $\bar{H} \vdash a \in A$ 2. $\bar{H} \vdash P(a)$ by p
$\bot$ $\top$	$\bar{H},\ x{:}\bot,\ \bar{J} \vdash G$ by $any(x)$	$\bar{H} \vdash \top$ by true
H_i	$x_1{:}H_1, \ldots, x_n{:}H_n \vdash H_i$	by $hyp\, x_i$ $i = 1, \ldots, n$ (recall x_i are *distinct*)
cut	$\bar{H},\ \bar{J} \vdash G$ by $\mathrm{cut}(x.g(x);c)$ $\bar{H},\ x{:}C,\ \bar{J} \vdash G$ by $g(x)$ $\bar{H},\ \bar{J} \vdash C$ by c	Assert C @ i where i locates C in $\bar{H}, \bar{J}$.
thin	$\bar{H},\ x{:}P,\ \bar{J} \vdash G$ by g $\bar{H},\ \bar{J} \vdash G$ by g	Thin @ i where i locates $x{:}P$ in $\bar{H},\ \bar{J}$.

Magic: $H \vdash P \vee \neg P$ by $magic(P)$

We will use tacticals to put together compound justifications when the notation seems clear enough.

2.7. Natural numbers

One of the most basic mathematical types is **N**, the natural numbers. This type is formed by the rule $H \vdash \mathbf{N} \in \text{Type}$. The type is inductively defined by the rules which say that $0 \in \mathbf{N}$, and if $n \in \mathbf{N}$ then $suc(n) \in \mathbf{N}$. The typing judgments we need are

$H \vdash 0 \in \mathbf{N}$	type_of_zero
$H \vdash suc(n) \in \mathbf{N}$ $H \vdash n \in \mathbf{N}$	type_of_successor

To express the fact that **N** is inductively defined we use the rule of mathematical induction. In its unrestricted form, this essentially says that nothing else is a member of **N** except what can be generated from 0 using *suc*. But the form of the rule given here does not quantify over all propositional functions on **N**, so it is not a full statement of the principle.

Suppose $P:(\mathbf{N} \times A) \rightarrow Prop$, then

$$\begin{array}{l} \bar{x}:\bar{H},\ n:\mathbf{N} \vdash P(n,\bar{x}) \text{ by } ind(n; p_o; u, i,\ p_s(u,i,\bar{x})) \\ \quad \bar{x}:\bar{H}, n:\mathbf{N} \vdash P(0) \text{ by } p_o \\ \bar{x}:\bar{H}, n:\mathbf{N}, u:\mathbf{N}, i:P(u,\bar{x}) \vdash P(suc(u),\bar{x}) \text{ by } p_s(u,i,\bar{x}) \end{array}$$

Arithmetic. When we display proofs of arithmetical propositions, we will assume that there is an automatic proof procedure which will prove any true quantifier free conclusion in a sequent involving 0, $suc(n)$, $+$, $-$, $*$, $=$ and $<$. So for example, here are some arithmetic facts in this category

$$0 < x, y < suc(z) \vdash y * x < suc(z * x).$$

Although there is no proof procedure with this power (the problem is undecidable), there are good arithmetical proof procedures for *restricted arithmetic* (Arith, see Boyer and Moore [1988], Church [1960]) and *linear arithmetic* (SupInf, see Chan [1982], Shostak [1979], Bledsoe [1975]). We refer the interested reader to the citations for details. The use of Arith allows us to present proofs in a form close to that of Nuprl (Constable et al. [1986], Jackson [1994c]). Here are two proofs of the same trivial theorem, one inductive, one not. We write Arith$(l_1, \ldots, l_n)$ to show which labeled hypotheses are used by the proof procedure. For readability we intentionally elided parts of the justification, using —.

$$\begin{array}{rl} \vdash \forall x{:}\mathbf{N}.\ \exists y{:}\mathbf{N}.\ (x < y) & \text{by } \forall R(x, \_\_) \\ x{:}\mathbf{N} \vdash \exists y{:}\mathbf{N}.\ (x < y) & \text{by } \exists R(s(x); \_\_) \\ \vdash x < suc(x) & \text{by Arith} \end{array}$$

The complete proof expression is $\forall R(x.\ \exists R(suc(x); \text{Arith}))$.

$$\begin{array}{ll}
\vdash \forall x{:}\mathbf{N}.\ \exists y{:}\mathbf{N}.\ (x < y) & \text{by } \forall R(x.\ ---) \\
x{:}\mathbf{N} \vdash \exists y{:}\mathbf{N}.\ (x < y) & \text{by } ind(x; ---;\ ) \\
\vdash \exists y{:}\mathbf{N}(0 < y) & \text{by } \exists R(suc(0); \text{Arith}) \\
x{:}\mathbf{N}, u{:}\mathbf{N}, i{:}\ \exists y{:}\mathbf{N}.\ (u < y) \vdash \exists y{:}\mathbf{N}.\ (suc(u) < y) & \text{by } \exists L(i; y_o, ---) \\
x{:}\mathbf{N}, u{:}\mathbf{N}, y_o{:}\mathbf{N}, l{:}(u < y_o) \vdash \exists y{:}\mathbf{N}(suc(u) < y) & \text{by } \exists R(suc(y_o); ---) \\
\vdash (suc(u) < suc(y_o)) & \text{by Arith}(l)
\end{array}$$

The complete proof expression is

$$\forall R(x.\ ind(x; \exists R(suc(0);\ \text{Arith}); u, i.\ \exists L(i; y_0, l.\ \exists R(suc(y_0);\ \text{Arith}(l))))).$$

The following example will provide another compact proof expression. It shows that integer square roots exist without using Magic (O'Leary et al. [1995]). First we specify these roots. Let $Root(r, n) == r^2 \leq n < (r+1)^2$.

Theorem. $\vdash \forall n{:}\mathbf{N}.\ \exists r{:}\mathbf{N}.\ Root(r, n)$.

by $\forall R$ THEN *ind new u*
base case
1. $n{:}\mathbf{N}$
   $\vdash \exists r{:}\mathbf{N}.\ Root(r, 0)$
   by $\exists R\ 0$ THEN Arith

induction case
1. $n{:}\mathbf{N}$
2. $u{:}\exists r{:}\mathbf{N}.\ Root(r, n)$
   $\vdash \exists r{:}\mathbf{N}.\ Root(r, suc(n))$
   by $\exists L$ *on u new* r_o, v
3. $r_o{:}\mathbf{N}$
4. $v{:}Root(r_o, n)$
   $\vdash \exists r{:}\mathbf{N}.\ Root(r, suc(n))$
   by *cut* $(r_o+1)^2 \leq suc(n) \ \vee\ suc(n) < (r_o+1)^2$ with label d THENA Arith.
   (This rule generates two subgoals. The "auxiliary one" is to prove the cut formula. That subgoal can be proved by Arith, so we say THENA Arith to indicate this.)

(The "main" subgoal is this one.)
5. $d{:}(r_o+1)^2 \leq suc(n) \ \vee\ suc(n) < (r_o+1)^2$
   $\vdash \exists r{:}\mathbf{N}.\ Root(r, n)$
   by VL *on d*
   (This is case analysis on the cases in hypothesis 5.)
6. $(r_o+1)^2 \leq suc(n)$
   $\vdash \exists r{:}\mathbf{N}.\ Root(r, n)$
   by $\exists R(r_o+1)$ THEN *SupInf*
   (Since $r_o^2 \leq n < (r_o+1)^2$, from $(r_o+1)^2 \leq suc(n)$ we know $(r_o+1)^2 \leq suc(n) < ((r_o+1)+1)^2$ since $n+1 < (r_o+1)^2+1$. The SupInf procedure can find this proof.)

6. $suc(n) < (r_o + 1)^2$
   $\vdash \exists r{:}\mathbf{N}.\ Root(r, n)$
   by $\exists R r_o$ THEN Arith
   (Since $r_o^2 \leq n$ we know immediately that $r_o^2 \leq suc(n)$.)

The proof expression corresponding to this is

$$\forall R(n.\ ind(n; \exists R(0; Arith);$$
$$n, u.\ \exists L(u; r_o, v.\ cut(d.\ VL(d; \exists R(r_o + 1; SupInf);$$
$$\exists R(r_o; Arith)); Arith)))).$$

In the appendix, section 6.2, we consider another simple arithmetic example and show a complete Nuprl proof.

2.8. Lists

Sequences are basic forms of construction in mathematics, often written $(a_1, \ldots, a_n)$. With the widespread use of programming languages we have come to distinguish several data types associated with sequences as distinct types. There are *lists*, *arrays*, and *sequences*, and they have different mathematical properties. We first look at lists.

If A is a type, then so is A list. We could write an informal rule like this

$$\frac{A \in Type}{A\ list \in Type}.$$

The elements of an A list are nil and $(a.l)$ where $a \in A$ and $l \in A$ list. If A is $\mathbf{N}$, these are lists:

$$nil \qquad (2.\ nil) \qquad (1.\ (2.\ nil))$$

All lists are built up from nil and the operation of pairing an element of A with a list. The typing rules are

$$\bar{H} \vdash nil \in A\ list \quad type\_of\_nil \qquad \begin{array}{l} \bar{H} \vdash (a.\ l) \in A\ list \quad type\_of\_cons \\ \quad \bar{H} \vdash a \in A \\ \quad \bar{H} \vdash l \in A\ list \end{array}$$

Equality on lists is given by these rules.

$$\begin{array}{l} \bar{H} \vdash (h.\ t) = (h'.\ t') \text{ in } A\ list \text{ by } list - eq \\ \quad \bar{H} \vdash h = h' \text{ in } A \\ \quad \bar{H} \vdash t = t' \text{ in } A\ list \end{array}$$

$$\bar{H} \vdash nil = nil \text{ in } A\ list.\ \text{ by } nil - eq$$

For every type A, A list is an inductive type with base case of nil. The inductive character of the type is given by its induction rule stated below for $P \in (A\ \text{list} \times T \Rightarrow Prop)$

$$\bar{x}:\bar{H}, l:A\ list \vdash P(l,\bar{x}) \text{ by } list\_ind\ (l; p_o; h,t,i.\ p(h,t,i,\bar{x}))$$
$$\bar{x}:\bar{H} \qquad \vdash P(nil,\bar{x}) \text{ by } p_o$$
$$\bar{x}:\bar{H}, l:A\ list, h:A, t:A\ list, i:P(t,\bar{x}) \vdash P((h.\ t),\bar{x}) \text{ by } p(h,t,i,\bar{x})$$

We can define the usual head-and-tail functions by induction. Define $Compound(x) == \exists h:A.\ \exists t:A\ list.\ x = (h.\ t)$ in $A\ list$.

$$\vdash \forall x:A\ list.\ (x = nil \text{ in } A\ list\ \vee\ Compound(x))$$
by $\forall R$
$$x:A\ list \vdash (x = nil \text{ in } A\ list\ \vee\ Compound(x))$$
by $list\_ind$
$$\vdash nil = nil \text{ in } A\ list\ \vee\ Compound(x)$$
by VRl
$$x:A\ list, h:A, t:A\ list \vdash (h.\ t) = nil \text{ in } A\ list \vee Compound(h.\ t))$$
by VRr
$$\vdash \exists h':A.\ \exists t':A\ list.\ (h.\ t) = (h'.\ t') \text{ in } A\ list.$$
by $\exists R\ h$ THEN $\exists R\ t$

We can also prove

$$\vdash \forall x:A\ list.\ \exists! h:A.\ \exists! t:A\ list.\ (\neg(x = nil) \Rightarrow x = (h.\ t) \text{ in } A\ list)$$

where $\exists!$ expresses unique existence (see the end of section 2.9).

2.9. Functions

The function type is one of the most important in modern mathematics. As we have noted, Frege patterned his treatment of logic which we are following on the concept of a function. In some ways this type represents the divide between abstract and concrete mathematics. By quantifying over functions we enter the realm of abstract mathematics. Indeed, the very notion of obtaining a function from an expression is called *abstraction.*

The day-to-day notation for functions at the beginning of the century was that one wrote phrases like "the function $sin(x)$ in x or e^x in x". Russell's notation, $\phi\hat{x}$, and Church's lambda notation, $\lambda x.e^x$, brought flexibility to the notation, creating a single name for the function with a binding operator (λ) to indicate the arguments. The modern *working notation* in mathematical articles and books (used in Bourbaki for example) is $x \mapsto b$ for a function with argument x and value given by the expression b in x; for example $x \mapsto x$ for the identity, $x \mapsto e^x$ for the exponential.

As we did for propositional functions, we will adopt the lambda notation in the form $\lambda(x.\ b)$ for $x \mapsto b$. In Nuprl one can *display* this in a variety of ways, including $x \mapsto b$ or $b\hat{x}$ or *fun* $x \Rightarrow b$. The important points are:

- There is an operator name, *lambda* that distinguishes functions. Their canonical value is $\lambda(x.\ b)$.
- A *binding phrase*, $x.\ b$ is used to identify the name of the *argument* (or formula parameter), x, and the body of the function.

- The usual rules about binding phrases apply concerning bound variables, scope, and α-equality.

Essentially the only way to use a function is to apply it to an argument.[19] Informal *notation* for applying a function f to an argument a is to write $f(a)$ or fa or even to show the substitution of "actual" argument for the "formal" one as in $sin(a)$ or e^a. We adopt an operator name to remind ourselves that application is a distinct *operation.* So we write $ap(f;a)$. But again, Nuprl can *display* this anyway the user pleases, e.g. as $f(a)$ or fa or even $f.\ a$ or $f@a$.

One of the major discoveries from a systematic study of function notations, especially the lambda calculus and combinatory calculus and later programming languages, is that rules for *formally calculating* with functions can be given independently of their meaning, especially independently of types.

The rules for calculation or for "definitional equality" can be expressed nicely as evaluation rules. Here is the so called "call_by_name" evaluation rule.

$$\frac{f \downarrow \lambda(x.\ b) \quad b[z/x] \downarrow c}{ap(f;a) \downarrow c}$$

The "call_by_value" rule is this

$$\frac{f \downarrow \lambda(x.\ b) \quad a \downarrow a' \quad b[a'/x] \downarrow c}{ap(f;a) \downarrow c}$$

Closed expression functions like $I == \lambda(x.\ x)$ or $K == \lambda(x.\ \lambda(y.\ x))$ are called *combinators*; these two are "polymorphic" in that we can compute their values regardless of the form of the input. Thus $ap(\lambda(x.\ x);K) \downarrow K$ and $ap(\lambda(x.\ x);0) \downarrow 0$, and $ap(K;I) \downarrow \lambda(x.\ I)$.

Other functions like $\lambda(z.1of(z))$ or $\lambda(z.add(1of(z);2of(z)))$ can only be reduced to values on inputs of a specific form, and others like $\lambda(x.suc(x))$ or $\lambda(x.\ 4/x)$ reduce to meaningful values (typed values) only on specific inputs. For example, $ap(\lambda(z.1of(z);0) \downarrow 1of(0)$ but $1of(0)$ is not a canonical value let alone a sensible value. In the case $ap(\lambda(x.\ suc(x));pair(0;0))$ the result of evaluation is the value $suc(pair(0;0))$, but this value has no type.

Typing functions. The space of functions from type A to type B is denoted $A \rightarrow B$. The *domain* type is A, the *range* (or *co-domain*) is B. The typing rule for functions is intuitively simple. We say that $\lambda(x.\ b) \in A \rightarrow B$ provided that on each input $a \in A$, $ap(\lambda(x.\ b);a) \in B$. This judgment is usually made symbolically by assuming $x \in A$ and judging by typing rules that $b \in B$. This is the form of typing judgment we adopt. So the typing rule has the form

$$\begin{array}{l} \bar{H} \vdash \lambda(x.\ b) \in A \rightarrow B \text{ by fun\_type} \\ \bar{H}, x{:}A \vdash b \in B \end{array}$$

[19] Although if functional equality is defined intensionally, then it is also possible to analyze their structure. Of course, function can also be passed as data.

More generally, given an expression f we allow

$$\bar{H} \vdash f \in A \to B \text{ by fun\_type}$$
$$\bar{H}, x{:}A \vdash ap(f;x) \in B$$

In the course of judging that an expression t has a type T, we allow replacing t by any term t' that is definitionally equal or by a term t' that t evaluates to. So if t in T and $t \downarrow t'$, then $t \in T$. In the logic over $(A \to B)$ we add the rule for function equality

$$\bar{H} \vdash f = g \text{ in } A \to B \text{ by extensional\_equalityR}$$
$$\bar{H}, x{:}A \vdash ap(f;x) = ap(g;x) \text{ in } B$$

$$\bar{H}, f = g \text{ in } A \to B \vdash ap(f;a) = ap(g;b) \text{ in } B \text{ by extensional\_equalityL}$$
$$\vdash a \in A$$

Here is Cantor's interesting argument about functions based on the method of *diagonalization.* It illustrates the rules for functions. (See the appendix for a Nuprl proof.)

Definition. Call f in $(A \to B)$ *onto* iff $\exists g{:}(B \to A)$ such that $\forall y{:}B.\ f(g(y)) = y$ in B.

Cantor shows that for inhabited types A with two distinct elements there is no function from A onto $(A \to A)$—essentially because $(A \to A)$ is "too big" to be enumerated by A. We state the condition on A using functions. We require that there is a function $diff \in A \to A$ such that $diff(x) \neq x$ for all x in A. The theorem is

Cantor's Theorem.
$(\exists\, diff : (A \to A).\ \forall x{:}A.\ diff(x) \neq x \text{ in } A) \Rightarrow (\neg\exists e{:}A \to (A \to A).\ e \text{ is onto})$

Proof. by $\Rightarrow R$ THEN $\Rightarrow R$

1. $\exists\, diff : (A \to A).\ \forall x{:}A.\ diff(x) \neq x$ in A
2. $\exists e{:}A \to (A \to A).\ e$ is onto

$\vdash \bot$

Next use $\exists L$ on 2 THEN unfold "onto" THEN $\exists L$

2. $e{:}A \to (A \to A)$
3. $g{:}(A \to A) \to A$
4. $\forall h{:}(A \to A).\ e(g(h)) = h$ in $(A \to A)$

Next $\exists L$ on 1 to replace 1 by 1.1 $diff : A \to A$, 1.2 $\forall x{:}A.\ diff(x) \neq x$ in A

Let $h_0 == \lambda(x.\ diff(e(x)(x)))$

Now $\forall L$ on 4 with h_0

5. $e(g(h_0)) = h_0$ in $A \to A$

Let $d == g(h_0)$, by *extensional_equalityL*

6. $e(d)(d) = h_0(d)$ in A

Now evaluate $h_0(d)$ to rewrite 6 as

6. $e(d)(d) = diff(e(d)(d))$

Now by $\forall L$ on 1.2 with $e(d)(d)$

7. $diff(e(d)(d)) \neq e(d)(d)$ (which is $(diff(e(d)(d)) = e(d)(d)) \rightarrow \bot$)

$\vdash \bot$

Finish by $\Rightarrow L$ on 7. and 6. □

Implicit functions from relations. A common way to define functions is implicitly in terms of relations. Suppose R is a relation on $A \times B$ and we know that for every $x \in A$ there is a unique y in B such that $R(x, y)$. Then we expect to have a function $f \in A \rightarrow B$ such that $R(x, f(x))$. How do we specify this function?

To facilitate consideration of this matter, let us define $\exists! y{:}A.\ P(y)$ to mean there is a y satisfying P, and any z that satisfies it is y. Thus

Definition. $\exists! y{:}A.\ P(y) == \exists y{:}A.\ P(y)\ \&\ \forall z{:}A.\ (P(z) \Rightarrow y = z \text{ in } A)$.

We expect the following formula to be true.

Function Comprehension. $\forall x : A.\ \exists! y : B.\ R(x, y) \Rightarrow \exists f : A \rightarrow B.\ \forall x : A.\ R(x, f(x))$.

For many instances of types A, B and relation R we can prove this formula by exhibiting a specific function. For example, if we define $Root(n, r)$ for n, r in N as $r^2 \leq n\ \&\ n < (r+1)^2$ then not only can we prove $\forall x{:}N.\ \exists! r{:}N.\ Root(n, r)$ but we can also define a function root by primitive recursion, namely

$$root(0) = 0$$
$$root(suc(n)) = \text{if } (root(n) + 1)^2 \leq n \text{ then } root(n) + 1 \textit{ else } root(n).$$

We know that $\lambda(x.\ root(x)) \in N \rightarrow N$ and $Root(n, root(n))$ is true. So perhaps if there are enough expressions for defining functions, we can prove the conjecture.

In set theory, functions are usually *defined* as single-valued total relations, i.e., a relation R on $A \times B$ is a function iff for all x in A there is a unique y in B such that $R(x, y)$. The relation R is a subset of $A \times B$, and this R is taken to *be* the function.

If the underlying logic has a choice function (or Hilbert $\in$-operator) as in Bourbaki [1968b] or HOL (Gordon and Melham [1993]), then the value of the function defined by R on input x is $choice(y.\ R(x, y))$ and a λ form for the function is $\lambda(x.\ choice(y.\ R(x, y))$.

The choice operator would not only prove the implicit function conjecture, but it would prove the closely related axiom of choice as well. That axiom is

Axiom of Choice. $\forall x{:}A.\ \exists y{:}B.\ R(x, y) \Rightarrow \exists f{:}(A \rightarrow B).\ \forall x{:}A.\ R(x, f(x))$.

We will see in section 3 that in constructive type theory this axiom is provable because the theory has enough expressions for functions.

2.10. Set types and local set theories

Another of the most fundamental concepts of modern mathematics is the notion of *set* or class. Class theory arose out of Frege's foundation for mathematics in *Grundgesetze* and in *Principia* along similar lines. Even before 1900 Cantor was creating a rich *naive set theory* which was axiomatized in 1908 by Zermelo and improved by Skolem and Fraenkel into modern day axiomatic set theories such as ZF (Bernays [1958]) and BG (Gödel [1931]) and Bourbaki's set theory ([1968b]).

We could formulate a full blown axiomatic set theory based on the type Set. But the type theory of section 3 is an alternative into which ZF can be encoded (Aczel [1986]). So instead we pursue a much more modest treatment of sets along the lines of *Principia's* classes. In *Principia*, given a propositional function $\phi\hat{x}$ whose range of significance is the type A, we can form the class $\hat{x}(\phi x)$ of those elements of A satisfying ϕ. We write this as $\{x : A \mid \phi(x)\}$. We call this a *set type* or a *class*. Given two classes α, β we can form the usual combinations of union, $\alpha \cup \beta$, intersection, $\alpha \cap \beta$, complement, $\bar{\alpha}$, universal class, A, and empty class, ϕ.

The typing judgment associated with a set type is what one would expect. Suppose A is a type and $P \in A \to Prop$, then

$$\begin{array}{l} \bar{H} \vdash a \in \{x{:}A \mid P(x)\} \text{ by } setR \\ \bar{H} \vdash a \in A \\ \bar{H} \vdash P(a) \end{array}$$

The rule for using an assumption about set membership is

$$\begin{array}{l} \bar{H}, y : \{x{:}A \mid P(x)\} \vdash G \quad by\ setL \\ \bar{H}, y{:}A,\ P(y) \vdash G \end{array}$$

As with the other rules, we can choose to name the assumption $P(y)$ by using the justification by *setL new u*. In Nuprl there is the option to "hide" the proof of $P(y)$. This hidden version is the default in Nuprl. A hypothesis is hidden to prevent the proof object from being used in computations. This is necessary because the set membership rule, *setR*, does not keep track of the proof $P(a)$; so the constructive elimination rule is

$$\begin{array}{l} \bar{H}, y{:}\{x{:}A \mid P(x)\}, \bar{J} \vdash G \text{ by } IsetL, new\ u \\ \bar{H}, y{:}A, [u{:}P(y)], \bar{J} \vdash G. \end{array}$$

In local set theories, the concept of the power set, $\mathcal{P}(A)$ is introduced (c.f. Bell [1988], MacLane and Moerdijk [1992]). This type collects all sets built over A and $Prop$. If A is a type, then $\mathcal{P}(A)$ is a type.

In order to express rules about this type, we need to treat the judgments $A \in Type$ and $P \in A \to Prop$ in the rules. Thus far we have expressed these judgments only implicitly, not as explicit goals, in part because $Type$ and $A \to Prop$ are not types themselves, but "large types." However, it makes sense to write a rule such as

$$\begin{array}{c} \bar{H} \vdash \{x{:}A \mid P(x)\} \in P(A) \\ \bar{H} \vdash A \in Type \\ \bar{H} \vdash P \in A \to Prop \end{array}$$

We can also imagine the rule

$$\bar{H}, X{:}\mathcal{P}(A) \vdash \exists P{:}A \to Prop.\,(X = \{x{:}A \mid P(x)\} \text{ in } \mathcal{P}(A))\,.$$

This introduces the large type, $(A \to Prop)$ into the type position. Treating this concept precisely requires that we consider explicit rules for $Type$ and $Prop$, especially their stratification as $Type_i$ and $Prop_i$. We defer these ideas until section 3.7.

Let us note at this point that the notion of $Prop$ and set types be at the heart of *topos theory* as explained in Bell [1988]. Essentially, the *subobject classifier*, Ω and $T : 1 \to \Omega$, of topos theory is an (impredicative) notion of $Prop$ and the subtype of true propositions. The notion of a *pullback* is used to define subtypes of a type A by "pulling back" a characteristic function $P : A \to Prop$ and the truth arrow $T : 1 \to Prop$ to get the domain of P, $\{x : A \mid P(x)\}$. A topos is essentially a category with Cartesian products (n-ary) a subobject classifier and power objects. In other words, it is an abstraction of a type theory which has $Prop$, a collection of true propositions, subtypes and a power type, $\mathcal{P}(A)$ for each type. The notion of a *Grothendieck topos* (c.f. Bell [1988], MacLane and Moerdijk [1992]) is essentially a *predicative* version of this concept. It can be defined in Martin-Löf type theory and in Nuprl, but that is beyond the scope of these notes. (However, see section 5.)

2.11. Quotient types

The equality relation on a type, written $s = t$ in T or $s =_T t$, defines the element's referential nature. The semantic models we use in section 3.9 take a type to be a partial equivalence relation (per) on a collection of terms.

Given a type T, other types can be defined from it by specifying new equality relations on the elements of T. For example, given the integers $\mathbb{Z}$, we can define the *congruence integers* $\mathbb{Z}//mod\ n$ to be the type whose elements are those of $\mathbb{Z}$ related by

$$x = y\ mod\ n \text{ iff } n\ divides\ (x - y).$$

More symbolically, let $n \mid m$ mean that n divides m, i.e., $\exists k{:}\mathbf{N}.\ m = k \star n$. Then $x = y\ mod\ n$ iff $n \mid (x - y)$. If $rm(x, n)$ is the remainder when x is divided by n, then clearly $x = y\ mod\ n$ iff $rm(x, n) = rm(y, n)$ in $\mathbb{Z}$. It is easy to see that $x = y\ mod\ n$ is an equivalence relation on $\mathbb{Z}$. In general, this is all we require to form a quotient type. If A is a type and E is an equivalence relation on A,

then $A//E$ is a new type, the quotient of A by E. The equality rule is $x = y$ in $A//E$ iff $E(x, y)$ for x, y in A. Here are the new rules.

$A//E$ is a type iff A is a type and E is an equivalence relation on A

$\bar{H} \vdash a$ in $A//E$ by *quotient_member*
$\bar{H} \vdash a$ in A

$\bar{H}, x{:}A//E, \bar{J} \vdash b[x]$ in B by *quotientL*
$\bar{H}, x{:}A, \bar{J} \vdash b[x]$ in B
$\bar{H}, x{:}A, x'{:}A, E(x, x'), \bar{J} \vdash b[x] = b[x']$ in B

For P to be a propositional function on a type A, we require that when $a = a'$ in A then $P(a)$ and $P(a')$ are the same proposition. If we consider atomic propositions $P(x)$ iff $x = t$ in $A//E$, then $a = t$ in $A//E$. The rules for equality of expressions built from elements of $A//E$ will guarantee the functional nature of propositions over $A//E$. We discuss the topic in detail in section 3.9 and in the literature on Nuprl Constable et al. [1986], Allen [1987b].

The quotient type is very important in many subjects. We have found it especially natural in automata theory (Constable et al. [1998]), rational arithmetic and of course, for congruences. For congruence integers we have proved Fermat's little theorem in this form:

Theorem. $\forall p{:}\{x{:}\mathbf{N} \mid prime(p)\}\,.\,\forall x{:}\mathbb{Z}//mod\ p.\ (x^p = x)$

Here the display mechanism suppresses the type on equality when it can be immediately inferred from the type of the equands.

Equivalence classes. It is noteworthy that quotient types offer a *computationally tractable* way of treating topics normally expressed in terms of equivalence classes. For example, if we want to study the algebraic properties of $\mathbb{Z}//mod\ n$ it is customary to form the set of equivalence classes of $\mathbb{Z}$ where the equivalence class of an element z is $[z] = \{i{:}\mathbb{Z} \mid i = z\ mod\ n\}$. The set of these classes is denoted $\mathbb{Z}/mod\ n$. The algebraic operations are extended to classes by

$$[z_1] + [z_2] = [z_1 + z_2],$$
$$[z_1] \star [z_2] = [z_1 \star z_2], \quad \text{etc.}$$

All of this development can be rephrased in terms quotient types. We show that $+$ and $\star$ are well-defined on $\mathbb{Z}//mod\ n$, and the elements are ordinary integers instead of equivalence classes. What changes is the equality on elements.

2.12. Theory structure

So far we have introduced a typed mathematical language and a few examples of specific types and then rules—for $\mathbf{N}$, lists, Cartesian products, functions, subsets, and quotients. The possibilities for new types are endless, and we shall see more of

them in sections 3 and 4. For example, we could introduce the type *Set* and explore classical and computational set theories. We can introduce partial objects via the *bar types* that Constable and Smith [1993] developed. As we have seen, we can use the Magic rule or not or various weaker forms of it.

Some choices of rules are inconsistent, e.g. bar types and Magic or the impredicative $\triangle$ type of Mendler [1988] and dependent products on the fixed point rule with all types. How are we to keep track of the consistent possibilities?

One method is to postulate *fixed theories* in the typed logic such as Heyting Arithmetic (HA) (c.f. Troelstra [1973]) or Peano Arithmetic (HA + Magic) or IZF (c.f. Beeson [1985], Friedman and Scedrov [1983], Joyal and Moerdijk [1995], Moerdijk and Reyes [1991]) or Intuitionistic Type Theory (ITT) or Higher Order Logic (HOL). We rely on a community of scholars to establish the consistency of various collections of axioms. Books like Troelstra [1973] study relationships between dozens of these theories. The space of them is very large.

Another possibility is to explore the "tree of knowledge" formed by doing mathematics in various *contexts* determined by the definitions and axioms used for any result. We can think of definitions and axioms as establishing contexts. N.G. de Bruijn [1980] has proposed a way to organize this knowledge, including derivation of inconsistency on certain paths.

Essentially de Bruijn defined typed mathematical languages, PAL, Aut-68, Aut-QE, Aut-Π, which were used for writing definitions and axioms.[20] He proposed a *logical framework* for organizing definitions, axioms and theorems into *books*. We will explore these typed languages in the next section. They are more primitive than our typed logic.

The apparatus of Automath is completely *formal*; it is a mechanism whose meaning is to be found completely in its ability to organize information and classify it without regard for content. Extending this attitude to the mathematics being expressed leads to the formalist philosophy of mathematics espoused by Hilbert [1926]. This is de Bruijn's view in fact, and it surely contrasts with *Principia* which found its meaning in the logical truths written into a fixed foundational theory. It will contrast as well to the Martin-Löf view, Girard's [1987] view, the views of Coquand and Huet in Coq and my own view (as expressed to a large extent in Nuprl) in which the logical framework is organized to express computational meaning. It is noteworthy that the three influential philosophical schools—Formalism, Logicism, and Intuitionism, can be characterized rather sharply in this setting (and coexist!).

An Automath book is a sequence of *lines*. A line has four parts as indicated in Table 2. Each line introduces a unique identifier which is either a *primitive notion*, PN, or a *block opener* or is defined. The category part provides the grammatical category; type is a built-in category, defined types like *nat* are another.

The lines form two structures, one the *linear order* and the other a *rooted tree*.

[20] "Automath is a language which we claim to be suitable for expressing very large parts of mathematics, in such a way that the correctness of the mathematical contents is guaranteed as long as the rules of the grammar are obeyed." de Bruijn [1980].

indicator	identifier	definition	category
0	nat	PN	type
0	n	—	nat
0	real	PN	type
n	x	—	real

Table 2: Sequence of lines

The nodes of the tree are identifiers, x, and the edges are from x to the indicator of the line having x as its identifier part. The *complete context* of x is the list of indicators from x back to the root. So each line uses as its indicator the last block opener in its context. When the definition and category components are included with x, the result is what de Bruijn calls the *tree of knowledge.*

Nuprl has a similar structure to its knowledge base, called a *library.* A library consists of lines. Each one is uniquely named by an identifier. These can include the equivalent of block openers, called *theory delimiters* (begin_thyname, end_thyname). The library is organized by a *dependency graph* which indicates the logical order among theories (the lines between delimiters). Unlike in Automath, the theory structure is a *directed acyclic graph* (dag). Theories can also be linked to a file system or a database which provides additional "nonlogical" structuring.

The Nuprl 5 system also provides a structured library with mechanisms to control access to theories. There are two modes of accessing information. One is by collecting axioms, definitions, and theorems into *controlled access theories.* These theories can only use the specific rules and axioms assembled at its root. Each type such as N or $S \times T$ is organized into a small theory consisting of its rules.[21] More complex theories are built by collecting axioms.[22] We will be specifying certain important theories later. One of them is Nuprl 4, the fixed logic in the Nuprl 4.2 release. Another theory could be Nuprl 4_bar, the theory with partial objects (Constable and Smith [1993]) or NuIZF, the formulation of IZF in type theory.

Another way to use the library we might call *free access.* A user can prove theorems using any rules whatsoever, even inconsistent collections. Once a theorem is proved, the system can define its *root_system*, the collection of all rules and definitions used to state and prove it. The root_system determines the class of theories into which the result can be "planted."

2.13. Proofs as objects

The notion of proof plays a fundamental role in logic as we have seen here. Hilbert's *proof theory* is a study of proofs, and for philosophical reasons he conceived

[21]The associated tactics are attached as well, see Hickey [1996b,1997].

[22]The associated tactics can also enforce global constraints on the theory such as "decidable type checking."

of it as a constructive theory, and a *metatheory*.[23] Given the central role of proofs in all of mathematics, it is not a great leap to begin thinking about proofs as mathematical objects with the same "reality" as numbers. This viewpoint is central to intuitionistic and constructive mathematics, and it seems to be coherent classically as well. De Bruijn designed the Automath formalisms around notion of *formal* proofs as objects, and ordinary objects such as functions could depend on proofs. In order to treat what was called classical mathematics he had to add a *principle of irrelevance of proofs*.[24] However, to bring proof expressions fully into the mathematics as objects means more than allowing them into the language. As the proof irrelevance principle shows, they can be regarded as part of the underlying linguistic apparatus.[25] To make proofs *explicit objects* with a referential character, we must define equality on them (the kind of equality called *book equality* in Automath as opposed to definitional equality which holds for all terms whether referential or not).

There are two sources to guide the discovery of equality rules for proof objects. We can turn to intuitionistic mathematics and its semantics for the logical operators or we can look to proof theory and the reduction (or normalization rules). Neither account is definitive for classically conceived mathematics. In the case of using intuitionistic reasoning as a guide, we must handle classical rules, such as contradiction, or classical axioms like the law of excluded middle "magic". There are various ways to approach this with promising results (Allen et al. [1990], Murthy [1991], Girard [1991]). The subject is still very active.

Another approach is suggested by the normalization theorems for classical and constructive logics natural deduction systems, or N-systems (due to Prawitz [1965]), and the body of results on cut elimination in the sequent calculi, or L-systems (arising from Gentzen [1935]). Unfortunately, the results give somewhat conflicting notions of proof equality (c.f. Zucker [1974,1977], Ungar [1992]). It is perhaps premature to suggest the appropriate classical theory, so instead we will sketch the constructive ideas and leave the technical details to section 3 where we will explore carefully Martin-Löf's interpretation in which the *computational content* of a proof is taken as the object.

Another prerequisite to treating proofs as objects is that we understand the domain of significance, the type of assertions about proofs. This is another point that is not entirely clear. For instance, the views of Kreisel [1981], Scott [1976], and Tait [1967,1983] differ sharply from those of Martin-Löf [1982,1983] and Girard [1987].

One of the key points is whether we understand a proof p as a proof of a proposition P, p *proves* P, or whether provability is a relation on proofs so that $Proves(p, P))$ is the appropriate relationship. In the latter case there arises the

[23]That it had to be so was part of *Hilbert's Program* for a formal foundation of mathematics. Classical parts of mathematics were to be considered as ideal elements ultimately justified by constructive means.

[24]"...we extend the language by proclaiming that proofs of one and the same proposition are always definitionally equal. This extra rule was called 'proof irrelevance'...."

[25]This is quite different from taking them to be *metamathematical* objects as is done in proof theory...a theory that could be formalized in Automath.

danger of an infinite regress since we will require a proof p' of (p *Proves* P). At some level it seems that provability must be a basic judgment, like the typing judgment $t \in T$.

If we start with the view of the relationship p *proves* P as a typing judgment, then we are led to the view that the type of a proof is the proposition that it proves. Thus propositions play the role of types according to the *propositions_as_types principle.*

This principle is designed into Automath (but can be regarded as "linguistic"), and it is the core of both Martin-Löf type theory (Martin-Löf [1982,1984,1983], Nordstrom, Petersson and Smith [1990]) and Girard type theory (Stenlund [1972], Constable et al. [1986], Girard, Taylor and Lafont [1989]). According to this principle, a proposition P is provable (constructivists would say *true*) iff there is a proof p whose type is P, that is

$$\vdash P \quad \text{iff} \quad \text{for some } p,\ \vdash p \in P$$

Indeed, on this interpretation and recognizing that proof expressions p denote proofs, we can see the sequent notation $\bar{H} \vdash P$ by p as just another way of writing $\bar{H} \vdash p \in P$.

The $\bar{H} \vdash P$ by p judgment form can be considered *implicit.* Attention is focused on P, and the main concern is that there is some inhabitant. The $\bar{H} \vdash p \in P$ form is *explicit*, and attention is focused on the actual proof. The rules could all be presented in either implicit (logical) form or explicit (type theoretic) form. Consider the $\forall L$ and $\forall R$ rules, for example. Here is an implicit form.

$$\begin{array}{l} H, f{:}\forall x{:}A.\ P(x,)\bar{J} \vdash G \quad \text{by} \quad \forall L(f; a; y.\ g[y]) \\ H, f{:}\forall x{:}A.\ P(x), \bar{J}, y{:}P(a) \vdash G \quad \text{by} \quad g[y] \\ \qquad\qquad\qquad\qquad \bar{H} \vdash A \quad \text{by} \quad a \end{array}$$

$$\begin{array}{l} H \vdash\ \forall x{:}A.\ P(x) \quad \text{by} \quad \forall R(x.\ p[x]) \\ \quad H, x{:}A \vdash P(x) \quad \text{by} \quad p[x] \end{array}$$

Here is the explicit form of the $\forall L$ rule.

$$\begin{array}{l} \bar{H}, f{:}\forall x{:}A.\ P(x), \bar{J} \vdash \forall L(f; a; y.\ g[y]) \in G \\ \bar{H}, f{:}\forall x{:}A.\ P(x), \bar{J}, y{:}P(a) \vdash g[y] \in G \\ \qquad\qquad\qquad\qquad \bar{H}\ \vdash\ a \in A \end{array}$$

We will discover in section 3.11 that there is a reasonable notion of reduction on proof expressions (which can either be considered as computation or definitional equality) and that this gives rise to a minimal concept of equality on proofs that is sufficient to give them the status of mathematical objects.

2.14. Heyting's semantics

Here is Heyting's interpretation of the judgment p *proves* P.

1. For atomic P we cannot base the explanation on propositional components of P because there aren't any. But it might depend on an analysis of terms and

their type which could be compound.

We recognize certain atomic propositions, such as $0 = 0$ *in* N as "atomically true." That is, the proofs are themselves atomic, so the proposition is an axiom. In the case when the terms are atomic and the type is as well, there is little left to analyze. But other atomic propositions can be reduced to these axioms by computation on terms, say $5 * 0 = 1 * 0$ *in* N.
Some atomic propositions are proved by *computation on terms and proofs.* For example, $suc(suc(suc(0))) = suc(suc(suc(0)))$ *in* N is proved by thrice iterating the inference rule *suc_eq*

$$\frac{n = m}{suc(n) = suc(m)}$$

We might take the object $suc\_eq(suc\_eq(suc\_eq(zero\_eq)))$ as a proof expression for this equality. On the other hand, in such a case we can just as well consider the proof to be a computation procedure on the terms whose result is some token indicating success of the procedure.
In general, the proofs of atomic propositions depends on an analysis of the terms involved and the underlying type and its components. For example, $a = b$ *in* $A//E$ might involve a proof the proposition $E(a, b)$.

So we cannot say in advance what all the forms of proof are in these cases. As a general guide, in the case of completely atomic propositions such as $0 = 0$ *in* N in which the terms and type are atomic, we speculate that the proof is atomic as well. For these atomic proofs we might have a special symbol such as *axiom.*[26]

2. A proof of $P \,\&\, Q$ is a pair $\langle p, q\rangle$ where p proves P and q proves Q.
3. A proof of $P \vee Q$ is either p or q where p proves P and q proves Q. To be more explicit we say it is a pair $\langle tag, e\rangle$ where if the tag designates P then e is p and if it designates Q, then e is q.
4. A proof of $P \Rightarrow Q$ is a procedure f which maps any proof p of P to $f(p)$, a proof of Q.
5. A proof of $\exists x{:}A.\ P[x]$ is a pair $\langle a, p\rangle$ where $a \in A$ and p proves $P[a]$.
6. A proof of $\forall x{:}A.\ P[x]$ is a procedure f taking any element a of A to a proof $f(a)$ of $P[a]$.

Note, we treat $\neg P$ as $P \Rightarrow \bot$, so these definitions give an account of negation, but there are other approaches, such as Bishop [1967].

We will see a finer analysis of this definition in the section on type theory; there following Martin-Löf [1982] and Tait [1967,1983], we will distinguish between *canonical proof expressions* and non-canonical ones such as $add(suc(0); suc(suc(0)))$ (which reduces to a canonical one $suc(suc(suc(0))))$. In this more refined analysis

[26]In Martin-Löf type theory and in Nuprl all proofs of atomic formulas are reduced to a token (*axiom* in Nuprl). Information that might be needed from the proof is kept only at the metalevel.

we say that the above clauses define the *canonical proofs*, e.g. a canonical proof of $P \,\&\, Q$ is a pair $\langle p, q\rangle$, but $\Rightarrow L(\Rightarrow R(x.\langle x, q\rangle); p)$ is a noncanonical proof of $P \,\&\, Q$ which reduces to $\langle p, q\rangle$ when we "normalize" the proof.

Although this is a suggestive semantics of both proofs and propositions, several questions remain. Given a proposition P, can we be sure that all proofs have the structure suggested by this semantics? Suppose $P \,\&\, Q$ is not proved by proving P and proving Q but instead by a case analysis or by decomposing an implication and then decomposing an existential statement, etc.; so if t proves $P \,\&\, Q$, do we know t is a pair?

If proofs are going to be objects, then what is the right equality relation on them? If t proves $P\&Q$ then is t at least *equal* to a pair $\langle p, q\rangle$? What is the right equality on propositions? If $P = Q$ and p proves P does p prove Q? How can we make sense of Magic as a proof object? It is a proof of $P \vee \neg P$ yet it has no structure of the kind Heyting suggests. We will see that the type theories of the next section provide just the right tools for answering these questions.

3. Type theory

3.1. Introduction

Essential features. In this section I want to give a nontechnical overview of the subject I am calling *type theory.* I will discuss these points:

- It is a *foundational theory* in the sense of providing definitions of the basic notions in logic, mathematics, and computer science in terms of a few primitive concepts.
- It is a *computational theory* in the sense that among the primitive built-in concepts are notions of algorithm, data type, and computation. Moreover these notions are so interwoven into the fabric of the theory that we can discuss the computational aspects of every other idea in the theory. (The theory also provides a foundation for *noncomputational* mathematics, as we explain later.)
- It is *referential* in the sense that the terms denote *mathematical objects.* The referential nature of a term in a type T is determined by the *equality relation* associated with T, written $s = t$ in T. The equality relation is basic to the meaning of the type. All terms of the theory are *functional* over these equalities.
- When properly formalized and *implemented*, the theory provides practical tools for expressing, performing, and reasoning about computation in all areas of mathematics.

A detailed account of these three features will serve to explain the theory. Understanding them is essential to seeing its dynamics. In a sense, the axioms of the theory serve to provide a very abstract account of mathematical data, its transformation by effective procedures, and its assembly into useful knowledge. I summarized my ideas on this topic in Constable [1991].

Language and logic. In a sense, the theory is *logic free.* Unlike our account of typed logic, we do not start with propositions and truth. Instead we begin with more elementary parts of language, in particular, with a theory of computational equality of terms (or expressions). In *Principia* these elementary ideas are considered as part of the meaning of propositions. We separate them more clearly. We examine the mechanism of naming and definition as the most fundamental and later build upon this an account of propositions and truth.

This analysis of language draws on the insights of Frege, Russell, Brouwer, Wittgenstein, Church, Curry, Markov, de Bruijn, Kolmogorov, and Martin-Löf, and it draws on technical advances made by numerous computer scientists and logicians. We can summarize the insights in this way. The notion of *computability* is grounded in rules for processing language (Church [1940], Curry and Feys [1958], Markov [1949]). In particular, they can be organized as rules for a basic (type free) *equality on expressions* closely related to Frege's theory of identity in [1903]. The rules explain when two expressions will have the same reference if they have any reference. (We call these *computation rules*, but they could also be considered simply as general rules of *definitional equality* as in Automath.) De Bruijn showed that to fully understand the definitional rules, we need to understand how expressions are organized into *contexts* in a *tree of knowledge* as we discussed in section 2.12.

Frege not only realized the nature of identity rules, but he explained that the very notion of an *object* (or mathematical object) depends on rules for equality of expressions which are intended to denote objects. The equality rules of a theory serve to define the objects and prepare the ground for a *referential language*, one in which the expressions can be said to denote objects.

Frege also believed that the equality rules were not arbitrary but expressed the primitive truths about *abstract objects* such as numbers and classes. We build on Brouwer's theme that an understanding of the *natural numbers* $\mathbf{N}$ is an especially clear place to begin, and we try to build as much as possible with them. Here the insights of Brouwer [1975] (see van Stigt [1990]) show how to connect intuitions about number to the rules for equality of expressions. Brouwer shows that the idea of natural number and of pairing numbers are meaningful because they arise from mental operations. Moreover, these are the same abilities needed to manipulate the language of expressions (see Chomsky [1988]).[27]

So like Frege and Brouwer (and unlike formalists), we understand type theory to be *referential*, that is, the theory is about mathematical objects, and the meaningful expressions denote them.

Following Russell, we believe that a referential theory is created by classifying expressions into types. Not every expression is meaningful, for example, school children know that 0/0 is not. We sometimes say that the meaningful expressions are those that refer to mathematical objects, but this seems to presuppose that we

[27] For Brouwer this language is required by an individual only because of the limits and flaws in his or her mental powers. But for our theory, language is essential to the communication among agents (human and artificial or otherwise) needed to establish public knowledge.

know what such objects are. So we prefer to say that the task of type theory is *to provide the means to say when an expression is meaningful.* This is done by classifying expressions into types. Indeed to define a type is to say what *expressions* are of that type. This process also serves to define mathematical objects.[28]

Martin-Löf suggested a particular way of specifying types based on ideas developed by W. W. Tait [1967,1983]. First designate the standard irreducible names for elements of a type, say $t_1, t_2, \ldots$ belong to T. Call these *canonical values.* Then based on the definition of evaluation, extend the membership relation to all t' such that t' evaluates to a canonical value of T; we say that membership is extended by *pre-evaluation.*

Level restrictions. Russell [1908] observed that it is not possible to regard the collection of all types as a type itself. Let Type be this collection of all types. So Type is not an element of Type. Russell suggested schemes for layering or stratifying these "inexhaustible concepts" like Type or Proposition or Set. The idea is to introduce notions of types of various *levels.* In our theory these levels are indicated by *level indexes* such as Type_i. They will be defined later.

Architecture of type theory. What we have said so far lays out a basic structure for the theory. We start with a class of *terms.* This is the linguistic material needed for communication. We use variables and substitution of terms for variables to express relations between terms. Let x, y, z be variables and s, t be terms. We denote the substitution of term s for all free occurrences of variable x in t by $t[s/x]$. The details of specifying this mechanism vary from theory to theory. Our account is conventional and general.

Substitution introduces a primitive *linguistic relationship* among terms which is used to define certain basic *computational equalities* such as $ap(\lambda(x.b); a) = b[a/x]$.

There are other relations expressed on terms which serve to define computation. We write these as evaluation relations

$$t \; evals\_to \; t' \text{ also written } t \downarrow t'.$$

Some terms denote types, e. g. $\mathbf{N}$ denotes the type of natural numbers. There are type forming operations that build new types from others, e. g. the Cartesian product $T_1 \times T_2$ of T_1 and T_2. Corresponding to a type constructor like $\times$ there is usually a constructor on elements, e. g. if $t_1 \in T_1$, $t_2 \in T_2$ then $pair(t_1; t_2) \in T_1 \times T_2$. By the Tait pre-evaluation condition above

$$\frac{t' \; evals\_to \; pair(t_1; t_2)}{t' \in T_1 \times T_2}$$

[28]The interplay between expressions and objects has seemed confusing to readers of constructive type theory. In my opinion this arises mainly from the fact that computability considerations cause us to say more about the underlying language than is typical, but the same relationship exists in any formal account of mathematics.

Part of defining a type is defining equality among its numbers. This is written as $s = t$ in T. The idea of defining an equality with a type produces a concept like Bishop's sets (see Bishop [1967], Bishop and Bridges [1985]), that is Bishop [1967,p.63] said "... a *set* is defined by describing what must be done to construct an element of the set, and what must be done to show that two elements are *equal*."

The basic forms of judgment in this type theory are

- t is a term
  This is a simple context-free condition on strings of symbols that can be checked by a parser. We stress this by calling these *readable expressions*.
- T *is_a type*
  We also write $T \in \mathsf{Type}$ and prefer to write capital letters, S, T, A, B for types. This relationship is not decidable in general and cannot be checked by a parser. There are rules for inferring typehood.
- $t \in T$ (type membership or elementhood)
  This judgement is undecidable in general.
- $s = t$ in T (equality on T)
  This judgement is also undecidable generally.

Inference mechanism. Since Post it has been the accepted practice to define the class of formulas and the notion of proof inductively. Notice our definition of formula in section 2.4, also, for example, a *Hilbert style proof* is a sequence of closed formulas $F_1, \ldots, F_n$ such that F_i is an axiom or follows by a rule of inference from F_j, F_k for $j < i, k < i$. A typical inference rule is expressed in the form of hypotheses above a horizontal line with the conclusion below as in *modus poneus*.

$$\frac{A,\ A \Rightarrow B}{B}$$

This definition of a proof includes a specific presentation of evidence that an element is in the class of all proofs.

The above form of a rule can be used to present any inductive definition. For example, the natural numbers are often defined inductively by one rule with no premise and another rule with one.

$$0 \in \mathbf{N} \qquad \frac{n \in \mathbf{N}}{suc(n) \in \mathbf{N}}$$

This definition of $\mathbf{N}$ is one of the most basic inductive definitions. It is a pattern for all others, and indeed, it is the clarity of this style of definition that recommends it for foundational work.

Inductive definitions are also prominent in set theory. The article of Aczel [1986] "An Introduction to Inductive Definitions" surveys the methods and results. He bases his account on sets Φ of rule instances of the form $\frac{X}{x}$ where X are the *premises* and x the *conclusions*. A set Y is called Φ-closed iff $X \subseteq Y$ implies $x \in Y$. The set inductively defined by Φ is the intersection of all subsets Y of A which are Φ-closed.

3.2. Small fragment — arithmetic

We build a small fragment of a type theory to illustrate the points we have just made. The explanations are all inductive. We let S and T be metavariables for types and let, s, t, s_i, t_i, also s', t', s'_i, t'_i denote terms.

We arrange the theory around a *single judgment*, the equality $s = t$ in T. We avoid membership and typehood judgments by "folding them into equality" just to make the fragment more compact. First we look at an informal account of this theory.

The intended meaning of $s = t$ in T is that T is a type and s and t are equal elements of it. Thus a premise such as $s = t$ in T implies that T is a type and that s and t are elements of T (thus subsuming membership judgment).[29]

The only atomic type is $\mathbf{N}$. If S and T are types, then so is $(S \times T)$; these are the only compound types.

The canonical elements of $\mathbf{N}$ are 0 and $suc(n)$ where n is an element of $\mathbf{N}$, canonical or not. The canonical elements of $(S \times T)$ are $pair(s;t)$ where s is of type S and t of type T. The expressions $1of(p)$ and $2of(p)$ are noncanonical. The evaluation of $1of(pair(s;t))$ is s and of $2of(pair(s;t))$ is t.

The inference mechanism must generate the evident judgments of the form $s = t$ in T according to the above semantics. This is easily done as an inductive definition. The rules are all given as clauses in this definition of the usual style (recall Aczel [1977] for example).

We start with *terms* and their evaluation. The only atomic terms are 0 and $\mathbf{N}$. If s and t are terms, then so are $suc(t), (s \times t), pair(s;t), 1of(t), 2of(t)$. Of course, not all terms will be given meaning, e.g. $(0 \times \mathbf{N})$, $suc(\mathbf{N})$, $1of(\mathbf{N})$ will *not* be.

Evaluation. Let s and t be terms.

$$0 \; evals\_to \; 0 \quad \mathbf{N} \; evals\_to \; \mathbf{N} \quad suc(t) \; evals\_to \; suc(t) \quad pair(s;t) \; evals\_to \; pair(s;t)$$

$$1of(pair(s;t)) \; evals\_to \; s \qquad 2of(pair(s;t)) \; evals\_to \; t$$

Remark: $s(\mathbf{N}) \; evals\_to \; s(\mathbf{N})$, $1of(pair(\mathbf{N};0)) \; evals\_to \; \mathbf{N}$. So evaluation applies to meaningless terms. It is a purely formal relation, an *effective calculation.* Thus the base of this theory includes a formal notion of *effective computability* (c.f. Rogers [1967]) compatible with various formalizations of that notion, but not restricted necessarily to them (e.g. Church's thesis is not assumed). Also note that *evals_to* is idempotent; if $t \; evals\_to \; t'$ then $t' \; evals\_to \; t'$ and t' is a value.

general equality

$$\frac{t_1 = t_2 \text{ in } T}{t_2 = t_1 \text{ in } T} \qquad \frac{t_1 = t_2 \text{ in } T \quad t_2 = t_3 \text{ in } T}{t_1 = t_3 \text{ in } T} \qquad \frac{t_1 = t_2 \text{ in } T \quad t_1 \; evals\_to \; t'_1}{t'_1 = t_2 \text{ in } T}$$

[29] In the type theory of Martin-Löf [1982], a premise such as $s = t$ in T presupposes that T is a type and that $s \in T, t \in T$. This must be known *before* the judgment makes sense.

typehood and equality

$$0 = 0 \text{ in } \mathbf{N} \qquad \frac{t = t' \text{ in } \mathbf{N}}{suc(t) = suc(t') \text{ in } \mathbf{N}} \qquad \frac{s = s' \text{ in } S \quad t = t' \text{ in } T}{pair(s;t) = pair(s';t') \text{ in } (S \times T)}$$

The inductive nature of the type **N** and of the theory in general is apparent from its presentation. That is, from *outside* the theory we can see this structure. We can use induction principles from the informal mathematics (the *metamathematics*) to say, for example, every canonical expression for a number is either 0 or $suc(n)$. But so far there is no construct *inside* the theory which expresses this fact. We will eventually add one in section 3.3.

Examples. Here are examples of true judgments that we can make: $suc(0) = suc(0)$ in **N**. This tells us that **N** is a type and $suc(0)$ an element of it. Also $pair(0; suc(0)) = pair(0; suc(0))$ in $(\mathbf{N} \times \mathbf{N})$ which tells us that $(\mathbf{N} \times \mathbf{N})$ is a type with $pair(0; suc(0))$ a member. Also $1of(pair(0; a))$ belongs to **N** and $suc(1of(pair(0; a)))$ does as well for arbitrary a.

Here is a derivation that $suc(1of(pair(0; suc(0)))) = 2of(pair(0; suc(0)))$ in **N**.[30]

$$\begin{array}{rcl}
 & & 0 = 0 \text{ in } \mathbf{N} \\
0 = 0 \text{ in } \mathbf{N} suc(0) & = & suc(0) = suc(o) \text{ in } \mathbf{N} \\
pair(0; suc(0)) & = & pair(0; suc(0)) \text{ in } \mathbf{N} \times \mathbf{N} \\
1of(pair(0; suc(0))) & = & 1of(pair(0; suc(0))) \text{ in } \mathbf{N} \quad 1of(pair(0; suc(0))) \; evals_to \; 0
\end{array}$$

$$2of(pair(0; suc(0))) = 2of(pair(0; suc(0))) \text{ in } \mathbf{N} \quad 2of(pair(0; suc(0))) \; evals\_to \; suc(0)$$

$$1of(pair(0; suc(0))) = 0 \text{ in } \mathbf{N} \qquad 2of(pair(0; suc(0))) = suc(0) \text{ in } \mathbf{N}$$

$$suc(1of(pair(0; suc(0)))) = suc(0) \text{ in } \mathbf{N} \qquad suc(0) = 2of(pair(0; suc(0))) \text{ in } \mathbf{N}$$

$$\overline{suc(1of(pair(0; suc(0)))) = 2of(pair(0; suc(0))) \text{ in } \mathbf{N}}$$

Analyzing the fragment. This little fragment illustrates several features of the theory.

First, *evaluation is defined prior to typing.* The *evals_to* relation is purely formal and is grounded in language which is a prerequisite for communicating mathematics. Computation does not take into account the meaning of terms. This definition of computability might be limiting since we can imagine a notion that relies on the information in typehood, and it is possible that a "semantic notion" of computation must be explored in addition, once the types are laid down.[31] Our approach to

[30] In type theory, we will write the derivations in the usual bottom-up style with the conclusion at the bottom, leaves at the top.

[31] In IZF this is precisely the way computation is done, based on the information provided by a membership proof.

computation is compatible with the view taken in computation theory (c.f. Rogers [1967]).

Second, the semantics of even this simple theory fragment shows that the concept of a proposition involves the notion of its meaningfulness (or well-formedness). For example, what appears to be a simple proposition, $t = t$ in T, expresses the judgments that T is a type and that t belongs to this type. These judgments are part of understanding the judgment of truth.

To stress this point, notice that by postulating $0 = 0$ in $\mathbf{N}$ we are saying that $\mathbf{N}$ is a type, that 0 belongs to $\mathbf{N}$ and that it equals itself. The truth judgment is entirely trivial; so the significance of $t = t$ in T lies in the well-formedness judgments implicit in it. These judgments are normally left *implicit* in accounts of logic.

Notice that the well-formedness judgments cannot be *false*. They are a different category of judgment from those about truth. To say that $0 \in \mathbf{N}$ is to *define* zero, and to say $\mathbf{N}$ is a type is to define $\mathbf{N}$. We see this from the rules since there are no separate rules of the form "$\mathbf{N}$ is_a type" or 0 *is_a* $\mathbf{N}$." Note, because $t = t$ whenever t is in a type, *the judgment $t = t$ in T happens to be true exactly when it is well-formed.*

Finally the points about $t = t$ in T might be clarified by contrasting it with $suc = suc$ in 0. This judgment is meaningless in our semantics because 0 is not a type. Likewise $suc = suc$ in $\mathbf{N}$ is meaningless because although $\mathbf{N}$ is a type, suc is not a member of it. Similarly, $0 = suc$ in $\mathbf{N}$ is meaningless since suc is not a member of $\mathbf{N}$ according to our semantics. None of these expressions, which read like propositions, is false; they are just *senseless*. So we cannot understand, with respect to our semantics, what it would mean for them to be false.

Third, notice that the *semantics of the theory were given inductively* (although informally), and the proof rules were designed to directly express this inductive definition. This feature will be true for the full theory as well, although the basic judgments will involve variables and will be more complex both semantically and proof theoretically.

Fourth, the semantic explanations are *rooted in the use of informal language.* We speak of terms, substitution and evaluation. The use of language is critical to expressing computation. We do not treat terms as mathematical objects nor evaluation as a mathematical relation. To do this would be to conduct metamathematics *about* the system, and that metamathematics would then be based on some prior informal language. When we consider implementing the theory, it is the informal language which we implement, translating it to a programming notation lying necessarily outside of the theory.

Fifth, although the theory is grounded in language, it *refers to abstract objects.* This abstraction is provided by the equality rules. So while $1of(pair(0; suc(0)))$ is not a canonical integer in the term language, we cannot observe this linguistic fact in the theory. This term denotes the number 0. The theory is referential in this sense.

Sixth, the theory is *defined by rules.* Although these rules reflect concepts that we have mastered in language, so are meaningful, and although all of the judgments we assert are evident, it is the rules that define the theory. Since the rules reflect a semantic philosophy, we can see in them answers to basic questions about the objects

of the theory. We can say what a number is, what 0 is, what successor is. Since the fragment is so small, the answers are a bit weak, but we will strengthen it later.

Seventh, the theory is *open-ended.* We expect to extend this theory to formalize ever larger fragments of our intuitions about numbers, types, and propositions. As Gödel showed, this process is never complete. So at any point the theory can be extended. By later specifying how evaluation and typing work, we provide a framework for future extensions and provide the guarantees that extensions will preserve the truths already expressed.

3.3. First extensions

We could extend the theory by adding further forms of computation such as a term, prd, for predecessor along with the evaluation

$$prd(suc(n))\ evals\_to\ n.$$

We can also include a term for addition, $add(s;t)$ along with the evaluation rules

$$add(0;t)\ evals\_to\ t \qquad \frac{add(n;t)\ evals\_to\ s'}{add(suc(n);t)\ evals\_to\ suc(s')}$$

We include, as well, a term for multiplication, $mult(s;t)$ along with the evaluation rule

$$mult(0;t)\ evals\_to\ 0 \qquad \frac{mult(n;t)\ evals\_to\ m \quad add(m;t)\ evals\_to\ a}{mult(suc(n);t)\ evals\_to\ a}$$

These rules enable us to type more terms and assert more equalities. We can easily prove, for instance, that

$$add(suc(0);suc(0)) = mult(suc(0);add(suc(0);suc(0))) \text{ in } \mathbf{N}.$$

But this "theory" is woefully weak. It cannot

- internally express general statements such as $prd(suc(x)) = x$ in $\mathbf{N}$ or $add(suc(x);y) = suc(add(x;y))$ for any x because there is no notion of variable, but these are true in the metalanguage.
- express function definition patterns such as the *primitive recursions* which were used to define add, multiply and for which we know general truths.
- express the inductive nature of $\mathbf{N}$ and its consequences for the uniqueness of functions defined by primitive recursion.

Adding capability to define new functions and state their "functionality" takes us from a concrete theory to an abstract one; from specific equality judgments to *functional judgments.* These functional judgments are the essence of the theory, and they provide the basis for connecting to the propositional functions of typed logic. So we add them next.

The simplest new construct to incorporate is one for constructing any object by following the pattern for the construction of a number. We call it a (*primitive*) *recursion combinator*, R. It captures the pattern of definition of *prd*, *add*, *mult* given above. It will later be used to explain induction as well.

The defining property of R is its rule of computation and its respect for equality. We present the computation rule using substitution.[32] The simplest way to to this as to use the standard mechanism of *bound variables* (as in the lambda calculus or in quantifier notation). To this end we let u, v, w, x, y, z be variables, and given an expression *exp* of the theory, we let $u.exp$ or $u, v.exp$ or $u, v, x.exp$ or generally $u_1, \ldots, u_n.exp$ (also written $\bar{u}.exp$) be a *binding phrase*. We say that the u_i are *binding occurrences* of variables whose *scope* is *exp*. The occurrences of u_i in *exp* are *bound* (by the smallest binding phrase containing them). The unbound variables of *exp* are called *free*, and if x is a free variable of $\bar{u}.exp$, then $\bar{u}.exp[t/x]$ denotes the substitution of t for every free occurrence of x in *exp*. If any of the u_i occur free in t, then as usual $\bar{u}.exp[t/x]$ produces a new binding phrase $\bar{u}'.exp'$ where the binding variables are renamed to prevent *capture* of free variables of t.[33]

$$\frac{b[t/v]\ evals\_to\ c}{R(0; t; v.b; u, v, i.h)\ evals\_to\ c}$$

$$\frac{R(n; t; v.b; u, v, i.h)\ evals\_to\ a \quad h[n/u, t/v, a/i]\ evals\_to\ c}{R(suc(n); t; v.b; u, v, i.h)\ evals\_to\ c}$$

Here is a typical example of R used to define addition in the usual primitive recursive way.

$$R(n; m; v.v; u, v, a.suc(a))$$

We see that

$$\begin{gathered} R(0; m; --)\ evals\_to\ m\ ,\ i.e.\ \ 0 + m = m \\ R(suc(n); m; --)\ evals\_to\ suc(R(n; m; --)), \\ \text{i.e. } \ suc(n) + m\ evals\_to\ suc(n + m) \end{gathered}$$

Once we have introduced binding phrases into terms, the format for equality and consequent typing rules must change. Consider typing R. We want to say that if $v.b$ and $u, v, i.h$ have certain types, then R has a certain type. But the type of b and h will depend on the types of u, v and i. For example, the type of $v.v$ will be T in a context in which the variable v is assumed to have type T. Let us agree to use the judgment $t \in T$ to discuss typing issues, but for this theory fragment (as for Nuprl) this notation is just an abbreviation for $t = t$ in T. We will use it when we intend to focus on typing issues. We might write a rule like

[32] R can also be defined as a combinator without variables. In this case the primitive notion is application rather than substitution.

[33] If u_i is a free variable of t then it is *captured* in $\bar{u}.exp[t/x]$ by the binding occurrence u_i.

$$\frac{n \in \mathbf{N} \quad t \in A_1 \quad \begin{array}{c} v \in A_1 \\ \hline b \in B_2 \end{array} \quad \begin{array}{c} u \in \mathbf{N} \quad v \in A_1 \quad i \in B_2 \\ \hline h \in B_2 \end{array}}{R(n; t; v.b; u, v, i.h) \in B_2}$$

The premises

$$\frac{u \in N \; v \in A_1 \; i \in B_2}{h \in B_2}$$

reads "h has type B_2 under the assumption that u has type $\mathbf{N}$, u has type A_1 and i has type B_2."

For ease of writing we render this *hypothetical typing judgment* as $u : N,\ v : A_1,\ i : B_2 \vdash h \in B_2$. The syntax $u : N$ is a variant of $u \in N$ which stresses that u is a variable. Now the typing of R can be written

$$\frac{n \in N \quad t \in N \quad v{:}A_1 \vdash b \in B_2 \quad u{:}N,\ v{:}A_1,\ i{:}B_2 \vdash h \in B_2}{R(n; t; v.b; u, v, i.\ h) \in B_2}$$

This format tells us that n, t, b and h are possibly compound expressions of the indicated types with v, u, i as variables assumed to be of the indicated types.

Following our practice of subsuming the typing judgment in the equality one, we introduce the following rule.

First let

$$\begin{aligned}
\mathit{Principle_argument} &== n = n' \text{ in } \mathbf{N} \\
\mathit{Aux_argument} &== t = t' \text{ in } \mathbf{N} \\
\mathit{Base_equality} &== v = v' \text{ in } A_1 \vdash b = b' \text{ in } B_2 \\
\mathit{Induction_equality} &== u = u' \text{ in } \mathbf{N}, v = v' \text{ in } A_1, i = i' \text{ in } B_2 \vdash h = h' \text{ in } B_2
\end{aligned}$$

Then the rule is

$$\frac{\mathit{Principle\_argument} \quad \mathit{Aux\_argument} \quad \mathit{Base\_equality} \quad \mathit{Induction\_equality}}{R(n; t; v.\, b; u, v, e.\, h) = R(n'; t'; v'.\, b'; u', v', e'.\, h') \text{ in } B_2}$$

Unit and empty types. We have already seen a need for a type with exactly one element, called a unit type. We take **1** as the type name and $\bullet$ as the element, and adopt the rules:

$$\bullet = \bullet \text{ in } \mathbf{1}$$

We adopt the convention that such a rule automatically adds the new terms $\bullet$ and **1** to the collection of terms. We also automatically add

$$\bullet \; \mathit{evals\_to} \; \bullet \qquad \mathbf{1} \; \mathit{evals\_to} \; \mathbf{1}$$

to indicate that the new terms are canonical unless we stipulate otherwise with a different evaluation rule.

We will have reasons later for wanting the "dual" of the unit type. This is the empty type, **0**, with no elements. There is no rule for elements, but we postulate **0** *is_a type* from which we have that we **0** as a term and **0** *evals_to* **0**

An interesting point about handling **0** is to decide what we mean by *assuming* $x \in \mathbf{0}$. Does

$$x : \mathbf{0} \vdash x \in \mathbf{0}$$

make sense? Is this a sensible judgment? We seem to be saying that if we assume x belongs to **0** and that **0** is type, then x indeed belongs to **0**. We clearly know functionality vacuously since there are no closed terms t, t' with $t = t'$ in **0**. It is more interesting to ask about such anomalies as

$$x : \mathbf{0} \vdash x \in \mathbf{N} \quad \text{or} \quad x : \mathbf{0} \vdash x \in \mathbf{1}$$

or even the possible nonsense

$$x : \mathbf{0} \vdash \mathbf{N} \in \mathbf{N}.$$

What are we to make of these "boundary conditions" in the design of the theory?

According to our semantics and Martin-Löf's typing judgments, even $x : \mathbf{0} \vdash (suc = \mathbf{N} \text{ in } \mathbf{N})$ is a true judgment because we require that **0** is a type and for t, t' in **0**, if $t = t'$ in **0**, then $suc \in \mathbf{N}, \mathbf{N} \in \mathbf{N}$ and $suc = \mathbf{N}$ in **N**. Since anything is true for all t, t' in **0**, the judgment is true.

This conclusion is somewhat bizarre, but we will see later that there will be other types, of the form $\{x : A \mid P(x)\}$ whose emptiness is unknown. So our recourse is to treat types uniformly and not attempt to make a special judgment in the case of assumptions of the form $x : T$ for which T might be empty.

List types. The list data type is almost as central to computing as the natural numbers. We presented this type in the logic as well, and we follow that example even though we can see lists as a special case of the recursive types to be discussed later (section 4). The rules are more compact and pleasing to examine if we omit the typing context $\mathcal{T}$ and use the typing abbreviation of $t \in T$ for $t = t$ in T. So although we will write a rule like[34]

$$\boxed{\frac{a \in A,\ l \in list(A)}{cons(a; l) \in list(A)}}$$

Without its typing context, we intend the full rule

$$\frac{\mathcal{T} \vdash a = a' \text{ in } A \quad \mathcal{T} \vdash l = l' \text{ in } list(A)}{\mathcal{T} \vdash cons(a; l) = cons(a'; l') \text{ in } list(A).}$$

[34]In this section we use $list(A)$ instead of $A\,list$ to stress that we are developing a different theory than in section 2.

We also introduce a form of primitive recursion on lists, the combinator L whose evaluation rule and typing rules are:

$$\frac{b[t/v]\ evals\_to\ c}{L(nil; p; v.b; h, t, v, i.g)\ evals\_to\ c}$$

$$\frac{L(l, s, v.b, h, t, v, i.g)\ evals\_to\ c_1 \quad g[a/h, l/t, s/v, c_1/i]\ evals\_to\ c_2}{L(cons(a; l); s; v.b; h, t, v, i.g)\ evals\_to\ c_2}$$

Let $L[x; b, g] = L(x; v.\ b; h, t, v, e.\ g)$, and

$H_B == v = v'$ in $S \vdash b = b' \in B$,

$H_S == h = h'$ in A, $t = t'$ in $list(A)$, $v = v'$ in S, $i = i'$ in $B \vdash g = g'$ in B,

$C_A == \vdash a = a'$ in A,

$C_S == \vdash s = s'$ in S, and

$C_{Alist} == \vdash l = l'$ in $list(A)$, then

$$\frac{H_B \qquad H_S \qquad C_A \qquad C_S \qquad C_{Alist}}{L[cons(a; l), b; g] = L[cons(a'; l'), b', g'] \text{ in } list(A)}$$

$$L(nil; v.b; h, t, v, i.g) = L(nil; v.b'; h', t, v, i.g') \text{ in } list(A)$$

Here are typical generalizations of the functions add, mult, exp to N list to illustrate the use of L. For the list (3, 8, 5, 7, 2) the operations behave as follows. Add $addL$ is $(3 + (8 + (5 + (7 + (2 + 0)))))$, $multL$ is $3 * 8 * 5 * 7 * 2 * 1$, $expL_2$ is $(((((2)^2)^7)^5)^8)^3$.

$$\begin{aligned} addL(l) &== L(l; 0; h, t, a.add(h, a)) \\ multL(l) &== L(l; 1; h, t, m.mult(h, m)) \\ expL(l)_k &== L(l; k; h, t, e.exp(h, e)). \end{aligned}$$

The induction rule for lists is expressed using L as follows. Let $H_S ==$

$$x \in list(A), y \in S,\ v \in S \vdash f[nil/x, v/y] = b \text{ in } B$$

and let $H_{list} ==$

$$x \in list(A), y \in S, h \in A, t \in list(A), v \in S, i \in B \vdash f[cons(h; t)/x, v/y] = g \text{ in } B,$$

then

$$\frac{H_S \qquad H_{list}}{x \in list(A), y \in S \vdash f = L(x; y; v.b; h, t, v, i.g) \text{ in } B}$$

This says that L defines a unique functional expression over $list(A)$ and S because the values as inductively determined by the evaluation rule completely determine functions over $list(A)$.

3.4. Functions

The judgment $x = x$ in $A \vdash b = b$ in B defines a function from A to B whose rule is given by the expression b. We know this from the functionality constraint implicit in the judgment, i.e. if $a = a'$ in the type A, then $b[a/x] = b[a'/x]$ in the type B. Likewise if b_1 is an expression in x and b' is an expression in x' then $x = x'$ in $A \vdash b = b'$ in B defines such a function. The two rules b, b' are considered equal on equal a, a' in A. Also it is part of the judgment that $b[a/x] = b'[a'/x']$. To this extent at least the notion of equality on these functions is *extensional.*

Let us look at patterns of functionality that involve functions as arguments. The addition function on N is represented by

$$z \in N \times N \vdash add(1of(z); 2of(z)) \in N$$

We also know that

$$l \in list(N) \vdash addL(l) \in N.$$

We know that the pattern of definition used to form $addL, multL, expL$ can be extended to any binary function f from $N \times N$ to N using $fL_k(l) = L(l; k; h, t, a.f(h, a))$. For any specific f we can write this function $fL_k(l)$, but we would like to express the general fact as a function of f, saying: for any function from $N \times N$ to N and any k in N, $L(l; k; h, t, a.f(h, a))$ is a functional expression in l, k and f.

In order to say this, we need a type for f. The notation $(N \times N) \to N$ is the type used in section 2. We can add $(A \to B)$ as a type expression for A and B types. But we also need canonical values for the type, what should they be? Can we use $(x \in A \vdash b \in B)$ as a notation for a function in $(A \to B)$?

It would be acceptable to use just that notation; it is even similar to the Bourbaki notation $x \mapsto b(x \in A, b \in A)$ (see Bourbaki [1968a]). But in fact we do not need the type information to define the evaluation relation nor to describe the typing rule. So we could simply use $(x \mapsto b)$. Instead we adopt the lambda notation $\lambda(x.b)$ more familiar in computer science as we did in sections 1 and 2.

We also need notation for function application. We write $ap(f; a)$ for the application of function f to argument a, but often display this as $f(a)$. The new evaluation rules are:

$$\lambda(x.b) \; evals\_to \; \lambda(x.b)$$

$$\frac{b[a/x] \; evals\_to \; c}{ap(\lambda(x.b); a) \; evals\_to \; c}$$

The typing rule is

$$\frac{x = x' \text{ in } A \;\vdash\; b = b' \text{ in } B}{\lambda(x.\, b) = \lambda(x'.\, b') \text{ in } (A \to B)}$$

This rule generates the type $(A \to B)$ as a term.[35]

3.5. Duality and disjoint unions

The types **0** and **1** are called *duals* of each other in a category theory. Here is what this means. The object **1** is called *terminal* (or final) because for every type A, there is a unique map in $A \to \mathbf{1}$, i.e. a map terminating in **1**, namely $\lambda(x.\ \bullet)$. The object **0** is *initial* since for every type A, there is a unique map initiating in **0**, i.e. $\mathbf{0} \to A$, namely $\lambda(x.x)$.[36]

The duality concept is that the arrows of the types are reversed in the definition.

1 is final iff for all A there is a unique element in $A \to \mathbf{1}$.

0 is initial iff for all A there is a unique element in $\mathbf{0} \to A$.

We will examine another useful duality next.

The type $A \times B$ can be characterized in terms of functions. In category theory this is done with a diagram

$$\begin{array}{ccccc} & & \mathrm{C} & & \\ & & \vdots & & \\ & \mathrm{f}\swarrow & \vee p & \searrow \mathrm{g} & \\ \mathrm{A} & \xleftarrow{a} & A \times B & \xrightarrow{b} & B \end{array}$$

which says that given the projection functions $a == \lambda(x.1of(x))$, $b == \lambda(x.2of(x))$ and any functions $f : C \to A$, $g : C \to B$, there is exactly one map p denoted $\langle f, g\rangle \in C \to A \times B$ such that $f = a \circ p$ and $g = b \circ p$; that is, for $z \in C$

$$\begin{aligned} f(z) &= a(\langle f, g\rangle(z)) \\ g(z) &= b(\langle f, g\rangle(z)). \end{aligned}$$

We can show that $\lambda(z.pair(f(z); g(z)))$ is the unique map $\langle f, g\rangle$.

In category theory there is a construction that is dual to the product, called *co-product.* Duals are created by reversing the arrows in the diagram, so for a dual we claim this.

$$\begin{array}{ccccc} & & \mathrm{C} & & \\ & & \wedge & & \\ & \mathrm{f}\nearrow & \vdots\, p & \nwarrow \mathrm{g} & \\ \mathrm{A} & \xrightarrow{inl} & A + B & \xleftarrow{inr} & B \end{array}$$

Given A, B with maps $inl \in A \to A + B$, $inr \in B \to A + B$ and maps $f \in A \to C$, $g \in (B \to C)$ there is a unique map $[f, g] \in A + B \to C$ such that

$$[f, g] \circ inl = f \quad \text{and} \quad [f, b] \circ inr = g.$$

In type theory we take $inl(a)$, $inr(b)$ to be canonical values with evaluation

[35] Martin-Löf would only need the premise $x{:}A \vdash b \in B$ since this means that A is a type. But in his system to prove $x{:}A \vdash b \in B$ requires proving A *is_a type*.

[36] We could also use $\lambda(x.\ a)$ for any $a \in A$ if there is one since under the assumption that $x \in 0$, $x = a$ for any a, thus $\lambda(x.\ x) = \lambda(x.\ a)$ in $\mathbf{0} \to A$.

$$inl(a)\ evals\_to\ inl(a) \qquad inr(b)\ evals\_to\ inr(b).$$

For A and B types, $A + B$ is a new type called the *disjoint union* of A and B. But the typing rules present a difficulty. If we simply write

$$\frac{a = a' \text{ in } A}{inl(a) = inl(a') \text{ in } A + B} \qquad \frac{b = b' \text{ in } B}{inr(b) = inr(b') \text{ in } A + B}$$

then we can deduce a judgment like $inl(0) = inl(0)$ in $\mathbf{N} + suc(0)$ which does not make sense because $\mathbf{N} + suc(0)$ is not a type. That is, the rules would no longer propagate the invariant that if $t = t$ in T then T is a type.

We could solve this problem by including a new judgment, $T\ is\_a\ type$, into the theory. The rules would be quite clear for the types already built, namely:

$$\mathbf{N}\ is\_a\ type \quad \mathbf{1}\ is\_a\ type \quad \mathbf{0}\ is\_a\ type$$

$$\frac{A\ is\_a\ type \quad B\ is\_a\ type}{\begin{array}{l}(A \times B)\ is\_a\ type\\ list(A)\ \ is\_a\ type\\ (A \to B)\ is\_a\ type\\ (A + B)\ is\_a\ type\end{array}}$$

We can then use the rules

$$\frac{a = a' \text{ in } A \quad B\ is\_a\ type}{inl(a) = inl(a') \text{ in } A + B} \qquad \frac{b = b' \text{ in } B \quad A\ is\_a\ type}{inr(b) = inr(b') \text{ in } A + B}$$

We will see in section 3.7 how to avoid adding this new judgment $T\ is\_type$.

The map $[f, g]$ is built from a new form called $decide(d; u.f(u); v.g(v))$ whose evaluation rules are

$$\frac{f(a)\ evals\_to\ c}{decide(inl(a); u.f(u); v.g(v))\ evals\_to\ c}$$

$$\frac{g(b)\ evals\_to\ c}{decide(inr(b); u.f(u); v.g(v))\ evals\_to\ c}$$

The function $[f, g]$ is $\lambda(x.decide(x; u.f(u); v.g(v)))$. It is easy to see that

$$\boxed{\begin{array}{l}[f, g](inl(a)) = f(a) \text{ and}\\ [f, g](inr(b)) = g(b).\end{array}}$$

3.6. Metamathematical properties of the type theory fragment

The theory with base types **0**, **1**, N and type constructors $\times, list, \rightarrow$ and $+$ is sufficiently complex that it is worthwhile analyzing its properties.

First, it is based on a simple inductive model of computability and typing that is *intuitively clear*. So we could accept it based on self-evidence. Indeed it is like PRA Church [1960] in that regard—a manifestly correct theory baring mistakes of formalization of the intuitive ideas. Discussing this type evidence for the theory leads us into philosophy and Formal Methods studies of formalization which are beyond the scope of the work.

Second, we can prove various properties of the formalism by syntactic means. For instance:

Termination of Evaluation: If $\vdash t = t$ in T then there is a term t' such that t *evals_to* t' and t' *evals_to* t'.

Subject Reduction: If $\vdash t = t$ in T and t *evals_to* t' then $\vdash t' = t'$ in T.

Typehood: If $\vdash t_1 = t_2$ in T then T *is_a type*, and $\vdash t_1 = t_1$ in T and $\vdash t_2 = t_2$ in T.

Nontriviality: There is no term t such that $\vdash t = t$ in **0**.

Consistency: It is not possible to derive $0 = suc(0)$ in **N**.

Third, we can translate this theory into various well-known mathematical theories including Heyting Arithmetic of ω order, HA^{ω}, IZF set theory and ZF set theory, and the theories of Feferman [1970,1975]. There are also categorical models of this simple fragment using topoi (Bell [1988]).

3.7. Inductive type classes and large types

The types defined so far belong to an inductively defined collection according to the scheme for T *is_a type* in the last section. Let U_1 denote this inductively defined collection of types; it has the characteristic of a type in that it has elements and is structured. Evaluation is defined on the elements, e.g. **N** *evals_to* **N**, $(\mathbf{N} \times \mathbf{N})$ *evals_to* $(\mathbf{N} \times \mathbf{N})$, etc. So all of the elements are canonical and are built up inductively themselves. In this regard U_1 resembles **N**. It has all the properties of a type.

We want to make U_1 a type. So we add rules for its elements in terms of equalities. For example, there are rules $\mathbf{0} = \mathbf{0}$ in U_1 and

$$\frac{A = A' \text{ in } U_1 \quad B = B' \text{ in } U_1}{A \times B = A' \times B' \text{ in } U_1}$$

The equality rules we have in mind are these

$$\mathbf{N} = \mathbf{N} \text{ in } U_1 \quad \mathbf{0} = \mathbf{0} \text{ in } U_1 \quad \mathbf{1} = \mathbf{1} \text{ in } U_1$$

$$\frac{A = A' \text{ in } U_1 \quad B = B' \quad \text{in } U_1}{\begin{array}{ll} (A \times B) = (A' \times B') & \text{in } U_1 \\ list(A) = list(A') & \text{in } U_1 \\ (A \to B) = (A' \to B') & \text{in } U_1 \\ (A + B) = (A' + B') & \text{in } U_1 \end{array}}$$

This is a structural or *intensional equality* (used in both Nuprl and Martin-Löf [1982]). It turns out that this equality is also extensional since $A = B$ in U_1 iff $a \in A$ implies $a \in B$ and conversely. This is the only type so far whose *elements are types*, but it does not include *all* types, in particular U_1 is not in U_1 according to our semantics.

We have no way to prove that U_1 is not in U_1. We don't even have a way to say this. But it would be possible to add a recursion combinator on U_1 that expressed the idea that U_1 is the least type closed under these operations. The combinator would have the form of a primitive recursive definition

$$\begin{array}{l} f(\mathbf{0}, x) = b_0(x) \\ f(\mathbf{1}, x) = b_1(x) \\ f(N, x) = b_2(x) \\ f((A \times B), x) = h_1(A, B, f(A, x), f(B, x)) \\ \vdots \\ f((A + B), x) = h_4(A, B, f(A, x), f(B, x)) \end{array}$$

With this form of recursion and the corresponding induction rule we could prove that every element of U_1 was either $\mathbf{0}$, $\mathbf{1}$, N, a product, a union, etc.

Once we can regard types as elements of a type like U_1, then we can extend our methods for building objects, say over $\mathbf{N}$ or by case analysis over a type of Booleans, say $\mathbb{B}$ etc. to building types. Here are two examples, taking $\mathbb{B}$ as an abbreviation of $\mathbf{1} + \mathbf{1}$ and abbreviating $inl(\bullet)$ as tt and $inr(\bullet)$ as ff.

Let $T(tt) = A, T(ff) = B$, then $\lambda(x.T(x))$ is a function $\mathbb{B} \to U_1$. If we build a generalization of $\mathbb{B}$ to n distinct values, say $\mathbb{B}_n = ((\mathbf{1} + \mathbf{1}) + \cdots + \mathbf{1})$ n times defined by $\mathbb{B} = \mathbf{1}, \mathbb{B}(suc(n)) = \mathbb{B}(n) + \mathbf{1}$ with elements $1_b, \ldots, n_b$, then we can build a function $T(x)$ selecting n types, $T(i_b)$.

It is worth thinking harder about functions like $T : \mathbb{B}_n \to U_1$. This is an indexed collection of types, $\{T(1_b), \cdots, T(n_b)\}$. We can imagine putting them together to form types in various ways, for instance by products or unions or functions

$$\begin{array}{l} T(1_b) \times \cdots \times T(n_b) \text{ or} \\ T(1_b) + \cdots + T(n_b) \text{ or} \\ T(1_b) \to \cdots \to T(n_b). \end{array}$$

We could define these types recursively, say by functions Π, Σ and Θ if we could have inputs like this: m in $\mathbf{N}$, T in $\mathbb{B}_m \to U_1$,

$$\begin{array}{l} \Pi_m(0)(T) = T(i_m(1)) \\ \Pi_m(n)(T) = \Pi_m(n-1)(T) \times T(i_m(suc(n))) \end{array}$$

where $i_m(k)$ selects the k-th constant of $\mathbb{B}_n$, k_b.[37] Likewise for Σ and Θ. However, we are unable to type these functions Π, Σ, Θ with the current type constructors. We could type them with the new ones we are trying to define!

In the case of Π and Σ the operations make sense even for infinite families of types, say indexed by $T \in A \to U_1$ for any type A. We can think of Π over $T \in A \to U_1$ as functions f such that on input $a \in A$, we have $f(a) \in T(a)$. For Σ over $T \in A \to U_1$ we can use the elements a as "tags" so that elements are pairs $\langle a, t\rangle$ where $t \in T(a)$.

These ideas give rise to two new type constructors, Π and Σ over an indexed family of types $T \in A \to U_1$. We write the new constructors as $\Pi(A;T)$ and $\Sigma(A;T)$. We could use typing rules like these

$$\frac{A \in U_1 \quad T \in A \to U_1}{\begin{array}{c}\Pi(A;T) \in U_1 \\ \Sigma(A;T) \in U_1\end{array}}$$

$$\frac{x \in A \vdash f \in T(x)}{\lambda(x.f) \in \Pi(A;T)} \qquad \frac{\vdash a \in A \quad \vdash b \in T(a)}{pair(a;b) \in \Sigma(A;T)}$$

The dotted lines forming the box indicate that this is an exploratory rule which will be supplanted later. We treat $\lambda(x.f)$ and $pair(a;b)$ just as before, so we are not adding new elements to the theory, just new ways to type existing ones.

With Π and Σ and using induction over $\mathbf{N}$ we can build types that are not in this U_1. For example, let $f(0) = A, f(suc(n)) = A \times f(n)$. Then f is a function $\mathbf{N} \to U_1$ where $f(n) = A \times \cdots \times A$ taken n times. The actual function is $\lambda(n.R(n; A; u, t.A \times t))$. Now we can build types like $\Sigma(N; \lambda(n.R(n; A; u, t.A \times t)))$ and $\Pi(A; \lambda(n.R(n; A; u, t.A \times t)))$ which are not in U_1. We could imagine trying to enlarge the inductive type class U_1 by adding these operators to the inductive definition. We will take up this topic in the next section.

Dependent types. The construction of Π and Σ types over U_1 suggests something more expressive. Instead of limiting the dependent constructions to functions from $T \in A \to U_1$, we could allow dependency whenever we can form a type expression $B[x]$ that is meaningful for all x of type A. We are led to consider a rule of the form

$$\frac{\vdash A \in U_1 \quad x : A \vdash B[x] \in U_1}{\begin{array}{c}fun(A; x.B) \in U_1 \\ prod(A; x.B) \in U_1\end{array}}$$

[37] $i_n(0) = inl^{m-1}(inl(\bullet))$ and $i_m(n) = inl^{m-n}(inr(\bullet))$.

We call *fun* a *dependent function* constructor and *prod* a *dependent product*.[38] We adopt a different notation from Π and Σ to suggest the more fundamental character of the construction. If we have $T \in A \to U_1$, then $\Pi(A;T)$ is the same as $fun(A;x.T(x))$ and $\Sigma(A;T)$ is the same as $prod(A;x.T(x))$. But now we can iterate the construction without going beyond U_1. That is, we postulate that U_1 is closed under dependent functions and products.

This conception of Π and Σ is reminiscent of the collection axiom in set theory. For example, in ZF if $R(x,y)$ is a single-valued relation on sets, then we can form $\{y \mid \exists x \in A.R(x,y)\}$. Another way to think of collection is to have a function $f : A \to Set$ where $A \in Set$ and postulate the existence of the set $\{f(x) \mid x \in A\}$.

The similarity between collection and these rules is that we can consider B in $fun(A;x.B)$ to define a function $\lambda(x.B)$ from A into U_1. With the addition of dependent types, the intuitive model becomes more complex. What assurance can we offer that the theory is still consistent, e.g. that we can't derive $0 = 1$ in N or that we derive $t \in T$ but evaluation of t fails to terminate? Can we continue to understand the model inductively? If we can build an inductive model of U_1 then we can be assured of not only consistency but of a constructive explanation. We answer these questions next.

3.8. Universes

We can consider U_1 and the rules for it in the last section as partial axiomatization of the concept of *Type*. On this view, we think of U_1 as open-ended, and we do not adapt an axiom capturing its closed inductive character, such as the recursion combinator for U_1 discussed above.

On the other hand, we can also think of U_1 as a large type belonging to *Type*. On this view the axioms for U_1 *reflect* the rules of type construction on *Type* into the collection of types. The axioms postulate a certain enrichment of the concept *Type* in the same way that the axiom of inaccessible cardinals postulates an enrichment of Set. Similarly, from the foundations of category theory (Kreisel [1959]), Grothendieck's concept of a *universe* is a way of modeling large categories (and is equivalent to inaccessible cardinals).

If we take the view that U_1 is a universe (rather than *Type*), then it makes sense to form larger universes, say U_2, then U_3, etc. To form U_2 we extend U_1 by adding the type U_1 itself, like this: $U_1 = U_1$ in U_2.

Martin-Löf and Nuprl axiomatize a universe hierarchy indexed by natural numbers, U_i. The method of doing this is to add $U_i = U_i$ to U_{i+1} and to postulate *cumulativity*, that any type A in U_i belongs to all U_j for $i < j$. So the *universe rules* are:

$$U_i = U_i \text{ in } U_{i+1} \qquad \frac{A = A \text{ in } U_i}{A = A \text{ in } U_j \quad \text{for } i < j.}$$

[38] Martin-Löf calls this a *dependent sum* and writes $\Sigma(x \in A)B$. We think of it as a generalization of $A \times B$ and display the constructor in Nuprl as $x : A \times B$.

It is possible to extend the universe hierarchy further, say indexed by ordinal numbers *ord*. It is possible to postulate closure of *Type* under various schemes for generating larger universes; Palmgren [1991] considers such matters.

Nuprl has been designed to facilitate index free, or "polymorphic", treatment of U_i. Generally, the user simply writes a universe as U_i and the system keeps track of providing relative level numbers among them in terms of expressions called *level expressions* which allow forming $i+1$ and $max(i,j)$. The theoretical basis for this is in Allen [1987b] and was implemented by Howe and Jackson (see Jackson [1994c]).

3.9. Semantics: PER models

The principal mathematical method that we have used to prove the soundness of Nuprl (and Martin-Löf type theory) has been to interpret equality relations on a type as partial equivalence relations ("pers") over terms — thereby building a variety of *term model* (see Stenlund [1972]). We use a method pioneered by Stuart Allen [1987a,1987b] to define the model inductively.[39] In his thesis Allen compares his models to those of Aczel [1986], Beeson, and Smith [1984]. The modeling techniques also borrow from Tait [1967,1983] in that the membership relation is extended from values to all terms by the pre-evaluation relation; in that regard it follows closely Martin-Löf's informal semantics.[40]

Allen's method has been remarkably potent in all of our work. Mendler [1988] used the technique to model the recursive types defined in Mendler, Constable and Panangaden [1986]), and Smith [1984] used it to model our bar types for partial objects Constable and Smith [1993]. Harper [1992] gave one of the most accessible accounts; I draw heavily on the accounts of Allen, Mendler, and Harper to explain the method.

The first step is to fix the collection of terms. The next step is to equip the terms with an evaluation relation, written now as $t \downarrow t'$. Allen [1987b] gives an abstract account of the syntax of terms and the properties of evaluation. We follow Mendler and Harper in supplying less detail.

Assume that on *closed* terms t evaluation satisfies E1 and E2:

E1. if $t \downarrow t'$ and $t \downarrow t''$ then $t' = t''$, so $\downarrow$ is *deterministic*, and

E2. if $t \downarrow t'$ then $t' \downarrow t'$, so $\downarrow$ is *idempotent*.

If $t \downarrow t'$ then we call t' a (*canonical*) *value*.

Our task now is to specify those terms which are intended to be expressions for mathematical objects and to specify those terms which are expressions for types. We carry this out for the types built from **N** using products and dependent functions. We *distinguish* these as two tasks. The first one is to consider *membership*, and the

[39] In his introduction Allen says "The principal content of this thesis is a careful development of ... a semantic reinterpretation [of type theory] with the intention of making the bulk of type-theoretic practice ... independent of its original type-theoretic and constructive basis. ... Moreover, in the unfamiliar domain of intuitionistic type theory, the reinterpretation can serve as a *staff made of familiar mathematical material*."

[40] We say that if $t \downarrow t'$ and $t' \in T$, then $t \in T$. This is the *preevaluation* relation of t' to t.

second is to determine *type expressions.* We look at membership first since it is more basic.

According to Martin-Löf, to specify a type is to say what its members are, i.e., which terms are members, and to say what equality means on those terms which are members. Equality will be an equivalence relation, E, on some collection of terms. Considered over the entire collection, the relation need only be partial. The *field* of the relation (those elements in the relation to themselves, xEx) are the members of the type. These relations are called *partial equivalence relations* or per for short.

The built-in notion of computation places an additional requirement on the pers, namely, they must respect evaluation. That is, if $t \downarrow t'$ and tEr, then $t'Er$. We can say this succinctly by defining *Kleene equality* on terms, $t \simeq t'$ means that if $t \downarrow s$ or $t' \downarrow s$, then $t \downarrow s$ and $t' \downarrow s$, i. e. if either term has a value, then both have that same value. So we require that $t \simeq t'$ and tEr implies $t'Er$. We next introduce notation for this notion of type, starting with the idea that types themselves are mathematical objects with an equality defined on them.

Let us see how this notion of type membership looks for the natural numbers, i. e. for the type $\mathbf{N}$. Define the relation *Neq* on terms inductively by

$$\begin{array}{l} 0 \ \mathit{Neq}\ 0 \\ a \ \mathit{Neq}\ b \text{ implies } \mathit{suc}(a)\ \mathit{Neq}\ \mathit{suc}(b) \\ a' \ \mathit{Neq}\ b' \text{ and } a \downarrow a' \text{ and } b \downarrow b' \text{ implies } a\ \mathit{Neq}\ b. \end{array}$$

Neq is a partial equivalence relation which determines a minimal notation for numbers on which we can compute by primitive recursion (lazily). That is, we know what elements are zero, and for nonzero numbers, we can find the predecessor.

Next, we define a membership per for the Cartesian product of two types A and B with α and β as the membership relations. Let α^* denote the pre-evaluation relation of a relation α, that is $a\alpha b$ iff there are a', b' such that $a'\alpha b'$ and $a \downarrow a', b \downarrow b'$. Define $\alpha \otimes \beta$ as $\{\langle pair(a;b), pair(a';b')\rangle \mid a\alpha a' \ \&\ b\beta b'\}^*$.

We can see that if α and β are value respecting pers, then so is $\alpha \otimes \beta$. It clearly defines membership in a Cartesian product according to our account of products.

Finally, we need a membership condition for the dependent function space constructor, $fun(A; x.\ B)$. This is a bit more complex because for each element a of A, $B[a/x]$ is a type. So we need to consider a family of membership relations indexed by a type. The members of the function type will be lambda terms, $\lambda(x.\ b)$. Let α be a value respecting per and for each a such that $a\alpha a$, let $\Phi(a)$ be a value respecting per. Define $\Pi\alpha\Phi$ as the following partial equivalence relation:

$$\{\langle \lambda(x.\ b), \lambda(x'.\ b')\rangle \mid \forall a, a'.\ a\alpha a' \Rightarrow b[a/x]\ \Phi(a)\ b'[a'/x']\}^*.$$

In order for this per to define type membership for the function space, we require that whenever $a\alpha a'$, then $\Phi(a) = \Phi(a')$. We have in mind that these membership conditions are put together inductively. This is made explicit by the following inductive definition of a relation $\boldsymbol{K}$ on pers.

$\boldsymbol{K}(Neq)$
$\boldsymbol{K}(\alpha\otimes\beta)$ if $\boldsymbol{K}(\alpha)$ and $\boldsymbol{K}(\beta)$
$\boldsymbol{K}(\Pi\alpha\Phi)$ if $\boldsymbol{K}(\alpha)$ and $\forall a, a'.\ a\alpha a' \Rightarrow \Phi(a) = \Phi(a')$ & $\boldsymbol{K}(\Phi(a))$.

We can prove inductively that all the pers in $\boldsymbol{K}$ are value respecting and all define type membership. $\boldsymbol{K}$ provides a per semantics for the small type theory based on N, products and dependent functions. Notice equality on pers is *extensional.*

Type expressions. The inductively defined set $\boldsymbol{K}$ determines a collection of membership pers which represent types, but it does not relate these to the terms used to name types, e. g. terms such as $N, N \times N$, $fun\,(N; x.\ decide(s; u.\ N; v.\ N \times N))$ and so forth. We establish this relationship next by modifying the definition of $\boldsymbol{K}$ to include names for types. Let $\boldsymbol{M}$ be the following inductively defined binary relation.

$$\begin{array}{rcl} N & \boldsymbol{M} & Neq \\ A \times B & \boldsymbol{M} & \alpha\otimes\beta \text{ if } A\boldsymbol{M}\alpha \text{ and } B\boldsymbol{M}\beta \\ fun(A; x.\ B) & \boldsymbol{M} & \Pi\alpha\Phi \text{ if } A\boldsymbol{M}\alpha \text{ and } \forall a,a'. a\alpha a' \Rightarrow \Phi(a)=\Phi(a') \text{ and } B[a/x]\boldsymbol{M}\Phi(a) \end{array}$$

This is an ordinary inductive definition of a binary relation. Also, it is easy to see that $A\boldsymbol{M}\alpha$ implies $\boldsymbol{K}\alpha$. The only membership pers described by $\boldsymbol{M}$ are those whose constituents are also described by $\boldsymbol{M}$. Moreover, all the membership pers are represented by terms, i. e. are related to terms by $\boldsymbol{M}$. This is critical for the $\Pi\alpha\Phi$ pers because it guarantees that Φ is represented by a term. Here are three critical facts about $\boldsymbol{M}$.

Fact 1 $A\boldsymbol{M}\alpha \Rightarrow \boldsymbol{K}(\alpha)$
Fact 2 $A\boldsymbol{M}\alpha$ and $A\boldsymbol{M}\alpha' \Rightarrow \alpha = \alpha'$
Fact 3 $A\boldsymbol{M}\alpha$ and $A \simeq A' \Rightarrow A'\boldsymbol{M}\alpha$.

These facts can be proved by $\boldsymbol{M}$ induction. Fact 1 means that all member pers are value respecting, and Fact 3 means that the type names are value respecting as well.

Pers for intensional type equality. We now want to define a per on type expressions which represents type equality and is value respecting. There is already a sensible equality that arises from $\boldsymbol{M}$, namely, $A = A'$ if $A\boldsymbol{M}\alpha, A'\boldsymbol{M}\alpha'$ and $\alpha = \alpha'$. This is an extensional equality. We want to model the structural equality of section 3.7, thus $A \times B = A' \times B'$ iff $A = A'$ and $B = B'$. Here is the appropriate definition of a binary relation $\boldsymbol{E}$ on terms.

$N\boldsymbol{E}N$
$A \times B\boldsymbol{E}A' \times B'$ if $A\boldsymbol{E}A'$ and $B\boldsymbol{E}B'$
$fun(A; xB)\boldsymbol{E}\ fun(A'; x'.\ B')$ if $A\boldsymbol{E}A'$ and $\exists\alpha\ A\boldsymbol{M}\alpha$ and $A'\boldsymbol{M}\alpha$ and $\forall a, a'.\ a\alpha a' \Rightarrow B[a/x]\boldsymbol{E}\ B'[a'/x']$
$A\boldsymbol{E}A'$ if $B\boldsymbol{E}B'$ and $A \downarrow B$ and $A' \downarrow B$.

We say that A is an intensional type expressions of model $\boldsymbol{M}$ iff $\exists\alpha(A\boldsymbol{M}\alpha$ and $A\boldsymbol{E}A)$. Clearly, the per representing type equality is value respecting. The relations $\boldsymbol{E}$ and $\boldsymbol{M}$ provide a model of our type theory fragment. The methods extend to Martin-Löf's '82 theory and to Nuprl 3 as Allen has shown.

We now summarize the approach described above, starting from $\boldsymbol{E}$ and following Harper's method of using least fixed points to present the inductive relations.

Summary of per semantics. Here is a summary of the per semantics along the lines developed by Harper [1992]. Let $\mathcal{T}$ be the collection of terms. We define on $\mathcal{T}$ a *partial equivalence relation* E intended to denote type equality. If aEa then we say that a is a type. If aEa' then a and a' are equal types. Let $|E| = \{t : \mathcal{T} \mid tEt\}$, called the *field* of E. Let $\mathcal{T}/E$ be the set of equivalence classes of terms; say $[t]_E = \{x : \mathcal{T} \mid xEt\}$. Let PER denote the set of all partial equivalence relations on $\mathcal{T}$.

Associated with each type is a membership equality, corresponding to $a = b$ in A; thus for each $a \in |E|$, there is a partial equivalence relation, $\boldsymbol{L}(a)$. So $\boldsymbol{L} \in \mathcal{T}/E \to$ PER.

We require of E and each $\boldsymbol{L}(a)$ that they respect evaluation, i.e. if $a_1 E a_2$ and $a_1 \downarrow a_1'$ and $a_2 \downarrow a_2'$, then $a_1' E a_2'$. Likewise for $\boldsymbol{L}(a)$ in place of E.

Now consider how we might define E and $\boldsymbol{L}$ mutually recursively to build a model of the type theory. For the sake of simplicity, we start with an extensional notion of type equality, as above. Call it Ext. We define Ext and $\boldsymbol{L}$ mutually recursively.

$$a \; Ext \; b \quad \text{iff} \quad \forall x, y.\ ((x\boldsymbol{L}(a)y) \Leftrightarrow (x\boldsymbol{L}(b)y))$$

$$s\boldsymbol{L}(a_1 \times a_2)t \quad \text{iff} \quad \exists s_1, s_2, t_1, t_2.\ s \downarrow pair(s_1; s_2) \;\&\; t \downarrow pair(t_1; t_2) \;\&\; s_1\boldsymbol{L}(a_1)t_1 \;\&\; s_2\boldsymbol{L}(a_2)t_2.$$

$$f\boldsymbol{L}(fun(a_1; x.\ a_2))f' \quad \text{iff} \quad \exists x, b, x', b'.\ f \downarrow \lambda(x.\ b) \;\&\; f' \downarrow \lambda(x'.\ b') \;\&\; \forall y, y' : |Ext(a_1)|.\ (y\boldsymbol{L}(a_1)y' \Rightarrow b[y/x]\boldsymbol{L}(a_2[y/x])b'[y'/x']).$$

This is a mutually recursive definition of Ext and $\boldsymbol{L}$, and it reflects our intuitive understanding, but the definition is not a standard positive (hence monotone) inductive definition because of the negative occurrence of $y\boldsymbol{L}(a_1)y'$ in the clause defining equal functions.

Allen calls these "half-positive" definitions; his method of using $\boldsymbol{K}$ and $\boldsymbol{M}$ as above shows how to replace this nonstandard definition with a standard positive induction which can be interpreted in either classical or constructive settings (for example, in ZF or IZF or in a theory of inductive definitions, see Troelstra [1973] and Feferman [1970]).

Definition. A *type system* τ is a pair $\langle E, \boldsymbol{L}\rangle$ where E is a value respecting per on $\mathcal{T}$ and for each $a \in |E|$, $\boldsymbol{L}(a)$ is a value respecting per. Given type systems $\tau = \langle E, \boldsymbol{L}\rangle$ and $\tau' = \langle E', \boldsymbol{L}'\rangle$, define $\tau \sqsubseteq \tau'$ iff $E \sqsubseteq E'$ and $\forall a : |E|.\ \boldsymbol{L}(a) = \boldsymbol{L}'(a)$; that is $\tau \sqsubseteq \tau'$ iff τ' has possibly more types, and on the types in τ it has the same equality.

Let TS be the collection of all type systems over $\mathcal{T}$ with evaluation $\downarrow$. It is easy to see that TS under $\sqsubseteq$ is a complete partially ordered set, a *cpo*. The relation

$\tau \sqsubseteq \tau'$ is a partial order on TS, and there is a least type system in this ordering, namely $\langle \phi, \phi \rangle$ where ϕ is the empty set. A non-empty subset D of TS is *directed* iff every pair of elements in D, say τ, τ', has an upper bound in D. Given any directed set D of type systems, say τ_i for $i \in I$, it has a least upper bound $\hat{\tau}$ where $\hat{E} = \cup E_i$ and $\boldsymbol{L}_\omega(a) = \boldsymbol{L}_i(a)$ if any $\boldsymbol{L}_i(a)$ is defined. If $\boldsymbol{L}_i(a)$ is defined, then since τ_i, τ_j for $i \neq j$ has an upper bound in D, say τ_k, we know that $\boldsymbol{L}_j(a) = \boldsymbol{L}_i(a)$ for $a \in E_i \cup E_j$, so the type system τ_a is well defined. Also τ_ω is least since if $\tau_i \sqsubseteq \tau'$ for all i, then $E_i \sqsubseteq E'$ for all i, so $\cup E_i \sqsubseteq E'$.

Theorem. *For any cpo D with order $\sqsubseteq$, if $F \in D \to D$ and F is monotone, i.e. $x \sqsubseteq y \Rightarrow F(x) \sqsubseteq F(y)$ for all x, y in D, then there exists a least fixed point of F in D, i.e. an element x_0 such that (i) $F(x_0) = x_0$, (ii) for all z such that $F(z) = z$ and $x_0 \sqsubseteq z$.*

We now define an operation $T \in TS \to TS$ which is monotone and whose least fixed point, $\hat{\tau}$, is a type system which models our rules.

Definition. Let $T(\langle E, \boldsymbol{L} \rangle) = \langle F^*, \boldsymbol{M} \rangle$ where

$$\begin{aligned} F = \{\langle N, N \rangle\} & \\ & \cup \{\langle a \times b, a' \times b' \rangle \mid aEa' \;\&\; bEb'\} \\ & \cup \{\langle \mathit{fun}(a; x.\, b), \mathit{fun}(a'; x'.\, b') \rangle \mid aEa' \;\& \\ & \qquad \forall y, y'.\; y\boldsymbol{L}(a)y' \Rightarrow b[y/x]Eb'[y'/x']\} \end{aligned}$$

$$\boldsymbol{M}(a) = \begin{cases} \mathit{Neq} \text{ if } a = N \\ \boldsymbol{L}(a_1) \otimes \boldsymbol{L}(a_2) \text{ if } a = a_1 \times a_2 \\ \Pi(\boldsymbol{L}(a_1), \lambda(x.\, \boldsymbol{L}(b))) \text{ if } a = \mathit{fun}(a_1; x.\, b) \;\& \\ a_1 \in |E| \;\&\; \forall y \colon |\, \boldsymbol{L}(a_1)|.\; b[y/x] \in |E| \end{cases}$$

Theorem. *T is monotone in $\sqsubseteq$ on TS.*

3.10. Using type systems to model type theories

Allen's techniques enable us to model a variety of type theories. Let us designate some models for the theories discussed earlier. We'll fix the terms and evaluation relation to include those of the richest theory; so the terms are: **0**, **1**, $\bullet$, **N**, 0, $\mathit{suc}(t)$, $\mathit{prd}(s)$, $\mathit{add}(s;t)$, $\mathit{mult}(s;t)$, $\mathit{exp}(s;t)$, $R(n;t;v.b;u,v,i.h)$, $s \times t$, $\mathit{pair}(s;t)$, $\mathit{prod}(s;x.t)$, $\mathit{fun}(s;x.t)$, $\lambda(x.t)$, $\mathit{list}(t)$, $(s.t)$, $L(s;a;v.b;h,t,v,i.g)$, $s + t$, $\mathit{inl}(s)$, $\mathit{inr}(t)$, and $\mathit{decide}(p;u.s;v.t)$.

There is also the evaluation relation s *evals_to* t which we abbreviate as $s \downarrow t$. We consider various mappings $\mathbf{T}_l : TS \to TS$ where l is a label such as N, G, ML, Nu, etc. The most elementary "theory" we will examine is a subtheory of arithmetic involving only equalities over **N** built from $0, \mathit{suc}(t), \mathit{prd}(s)$, and $\mathit{add}(s;t)$. This is modeled by $\mathbf{T}_N$. The input to $\mathbf{T}_N$ is any pair $\langle E, \boldsymbol{L} \rangle$ and the output is $\langle F, \boldsymbol{M} \rangle$ where the only type name is N, so NFN, thus $N \in |F|$, and the only type equality

is $\boldsymbol{M}(N)$ which is defined inductively. We have that $s\boldsymbol{M}(N)t$ is the least relation *Neq* such that

$$s\ Neq\ t\ \mathit{iff}\ (s \downarrow 0\ \&\ t \downarrow 0\ v\ \exists s', t'.\ s \downarrow suc(s')\ \&.\ t \downarrow suc(t') s' Neq\ t')\,.$$

The map T_N takes any $\langle E, \boldsymbol{L}\rangle$ to this $\langle F, \boldsymbol{M}\rangle$, so its least fixed point, $\mu(\boldsymbol{T}_N)$ is just $\langle F, \boldsymbol{M}\rangle$. This is the model for *successor arithmetic.* We see that in this model, N is a type, that $s = t$ in N iff s evaluates to a canonical natural number and t evaluates to the same canonical number.

The rules can be confirmed as follows. First, notice that evaluation is deterministic and idempotent on the terms. As we observed, the general equality rules hold in any type system (because $\boldsymbol{M}(N)$ is in equivalence relation on canonical numbers). This follows by showing inductively that $0, suc(0), suc(suc(0)), \ldots$ are in the relation $\boldsymbol{M}(N)$, i.e. in the field of the relation. The fact that $\boldsymbol{M}(N)$ respects evaluation validates the last equality rule.

$$\mu(T_N) \models 0 = 0 \text{ in } N$$
$$\mu(T_N) \models s = t \text{ in } N \text{ implies } \mu(T_N) \models suc(s) = suc(t) \text{ in } N.$$

The typing rule for successor is also confirmed by induction on *Neq*; namely, if $s'\boldsymbol{L}(N)t'$, then since $suc(s') \downarrow suc(s')$ and $suc(t') \downarrow suc(t')$, then we have $suc(s)\boldsymbol{M}(N)suc(t)$ as required for the typing rule.

In the case of N, the model $\mu(T_N)$, and the informal semantics are essentially the same. So the theory fragment for N can *stand on its own* with respect to the model. Even a set theoretic semantics for N will have the same essential ingredient of an inductive characterization. For instance, Frege's definition was that

$$N == \{x \mid \forall X.\ (0 \in X\ \&\ \forall y.\ (y \in X \Rightarrow s(y) \in X)) \Rightarrow x \in X\}$$

where $s(x)$ is $\{z \mid \exists u.\ (u \in z\ \&\ z - \{u\} = x)\}$. In ZF we can use the postulated infinite set, inf, and form $\omega == \{i : inf \mid \forall x.\ nat\_like(x) \Rightarrow i \in x\}$ where $nat\_like(x)$ *iff* $(0 \in x\ \&\ \forall y.\ (y \in x \Rightarrow suc(y) \in x))$ for $Suc(y) = y \cup \{y\}$. In both of these definitions, the inductive nature of N is expressed. But Frege's theory and ZF allow very general ways of using this inductive character. So far we have only used it for specifying the canonical values.

The same approach can be used to define a model for the type theory with cartesian products. In this case we denote the operator on type systems as ${T_N}^2$. Given ${T_N}^2(\langle E, \boldsymbol{L}\rangle) = \langle F, \boldsymbol{M}\rangle$, if $S \in |E|$ and $T \in |E|$ then $S \times T \in |F|$, and $\boldsymbol{M}(S \times T)$ is $\boldsymbol{L}(S) \otimes \boldsymbol{L}(T)$. For this system, ${T_N}^2$ is continuous, i. e. if $\tau_0 = \langle \phi, \phi\rangle$ and ${T_N}^2(\tau_i) = \tau_{i+1}$, then $\mu({T_N}^2) = \tau_\omega$.

In $\mu({T_N}^2)$ all the rules for the fragment of section 3.2 are true. Again the theory is so close to the semantics that it stands on its own. Notice that in confirming the rule for typing pairs, we rely on the fact that $\mu({T_N}^2)$ is a fixed point.

$$\mu({T_N}^2) \models s = s' \text{ in } S \text{ and } \mu({T_N}^2) \models (t = t' \text{ in } T) \text{ imply}$$
$$\mu({T_N}^2) \models pair(s; t) = pair(s'; t') \text{ in } S \times T.$$

Note, this fact would not be true in any fixed τ_i since $S \times T$ might be defined only in τ_{i+1}.

To provide a semantics for $fun(A; x.\ B)$ and $prod(A; x.\ B)$ we use the map T_{ML} defined in section 3.9. The model is $\mu(T_{ML})$. To prove the rules correct, we recall the meaning of sequents such as $x \in A \vdash B$ type and $x \in A \vdash s = t$ in T.

$$\mu(T_{ML}) \models (x \in A \vdash B\ type) \text{ implies } \mu(T_{ML}) \models fun(A; x.B)\ type$$
$$\mu(T_{ML}) \models (x \in A \vdash b \in B) \text{ implies } \mu(T_{ML}) \models \lambda(x.b) \text{ in } fun(A; x.B)$$

Modeling hypothetical judgments. The meaning of $x \in A \vdash b \in B$ is that A is a type and for any two elements, a, a' of A, $B[a/x]$ is a type and $B[a/x] = B[a'/x]$ (i.e. B is *type functional* in A), and moreover, $b[a/x] \in B[a/x]$ and $b[a/x] = b[a'/x]$ in $B[a/x]$. We have extended this notion to multiple hypotheses inductively to define $x_1 \in A_1, \ldots, x_n \in A_n \vdash b \in B$. This definition can be carried over to type systems.

3.11. A semantics of proofs

The discussion of proofs as objects and Heyting semantics in section 2 suggested treating proofs as objects and propositions as the types they inhabit. True propositions are those inhabited by proofs. But there were several questions left open in section 2.14 about the details of carrying out this idea.

The type theory of this section can answer these questions, and in so doing it provides a *semantics of proofs*. The basic idea is to consider a proposition as the type of all of its proofs and to take proof expressions to denote objects of these types. Based on Heyting's semantics we have a good idea of how to assign a type to compound propositions in terms of types assigned to the components. For atomic propositions there are several possibilities, but the simple one will turn out to provide a good semantics. The idea is to consider only those atomic propositions which can plausibly have *atomic proofs* and to denote the canonical atomic proofs by the term *axiom*. We will assign types to the compound propositions in such a way that the canonical elements will represent what we will call *canonical proofs*. Moreover, the reduction relation on the objects assigned to proof expressions will correspond to meaningful reductions on proofs. Proofs corresponding to noncanonical objects will be called noncanonical proofs. The correspondence will guarantee that noncanonical proofs p' of a proposition P will reduce to canonical proofs of P.

We now define the correspondence between propositions and types and between proofs and objects. Sometimes this correspondence is called the Curry-Howard isomorphism.

Curry-Howard isomorphism. For the sake of this definition, if P is a proposition, we let $[\![P]\!]$ be the corresponding type, and if p is a proof expression, we let $[\![p]\!]$ be the corresponding element of $[\![P]\!]$. We proceed to define $[\![\ ]\!]$ inductively on the structure of proposition P from section 2.5.

1. We consider only atomic propositions of the form $a = b$ in A. The type $[\![a = b \text{ in } A]\!]$ will have the atomic proof object *axiom* if the proposition is

axiomatically true.

If the proof expression e for $a = b$ in A evaluates to a canonical proof built only from equality rules, then we arrange that $e \downarrow axiom$. This is a simple form of correspondence that *ignores equality information.* For instance

$$[\![symmetry(3)]\!] \downarrow axiom \quad [\![transitivity(e_1, e_2)]\!] \downarrow axiom.$$

$$\frac{[\![e]\!] \downarrow e'}{[\![equality\_intro(e)]\!] \downarrow e'}$$

We also need these evaluation rules for the proof expressions for substitution and type equality.

$$\frac{[\![p]\!] \downarrow p'}{[\![subst(p; e)]\!] \downarrow p'} \quad \frac{[\![p]\!] \downarrow p'}{[\![eq(p; e)]\!] \downarrow p'}$$

2. $[\![P \,\&\, Q]\!] = [\![P]\!] \times [\![Q]\!]$ and
   $[\![\&R(e_1, e_2)]\!] = pair([\![e_1]\!]; [\![e_2]\!])$, and
   $[\![\&L(e_1; u, v.\ e_2)]\!] = [\![e_2]\!](1of([\![e_1]\!])/u, 2of([\![e_1]\!]/v))$.
3. $[\![P \vee Q]\!] = [\![P]\!] + [\![Q]\!]$,
   $[\![VRl(a)]\!] = inl([\![a]\!])$,
   $[\![VRr(b)]\!] = inr([\![b]\!])$,
   $[\![VL(d; u.\ e_1; v.\ e_2)]\!] = decide([\![d]\!]; u.[\![e_1]\!]; v.[\![e_2]\!])$.
4. $[\![P \Rightarrow Q]\!] = [\![P]\!] \rightarrow [\![Q]\!]$,
   $[\![\Rightarrow R(x.\ e)]\!] = \lambda(x.[\![e]\!])$,
   $[\![\Rightarrow L(f; p; y.\ q)]\!] = [\![g]\!][ap([\![f]\!]; [\![p]\!]/y]$.
5. $[\![\exists x{:}A.\ P[x]]\!] = prod(A; x.\ [\![P[x]]\!])$,
   $[\![\exists R(a; p)]\!] = pair(a; [\![p]\!])$,
   $[\![\exists L(p; u, v.\ g)]\!] = [\![g]\!](1of([\![p]\!])/u, 2of([\![p]\!]/v))$.
6. $[\![\forall x{:}A.\ P[x]]\!] = fun(A; x.\ [\![P[x]]\!])$,
   $[\![\forall R(x.\ e)]\!] = \lambda(x.[\![e]\!])$,
   $[\![\forall L(f; a; y.\ e)]\!] = [\![g]\!][ap([\![f]\!]; a)/y]$.

Sequents to typing judgments. We can now translate *deductions* of sequents $\bar{H} \vdash P$ *by* p to *derivations* of $[\![\bar{H}]\!] \vdash [\![p]\!] \in [\![P]\!]$. Given $\bar{H} = x_1 : H_1, \ldots, x_n : H_n$ we take $[\![\bar{H}]\!]$ to be $x_1 \in H'_1, \ldots, x_n \in H'_n$ where if H_i is a type then $H'_i = H_i$ and if H_i is a formula then $H'_i = [\![H_i]\!]$. In this case we treat the label x_i as a variable.

Now to translate a deduction tree to a derivation tree we work up from the leaves translating sequents as prescribed and changing the rule names. The proof system was designed in that we need not change the variable names.

Expressing well-formedness of formulas. The introduction of U_1 combined with the propositions-as-types interpretation allows us to express the pure proposition of typed logic more generally, and we can solve the small difficulty of insuring that $A + B$ is a type discussed at the end of section 3.5.

According to the propositions-as-types principle, U_1 represents the type of (small) propositions, and a function $P \in A \to U_1$ can be interpreted as a propositional function. When we want to stress this logical interpretation, we use the display form $Prop_1$ for U_1 and generally $Prop_i$ for U_i, and we call $Prop_i$ the proposition of $level_i$.

We can express general propositions in typed logic by quantifying over $Prop_i$ and U_i. Here are some examples from section 2.

1. $\forall A, B{:}U_1.\, \forall P{:}A \to Prop_1 \forall Q{:}B \to Prop_1.$
   $\forall x{:}A.\, \forall y{:}B.\, (P(x) \& Q(y)) \Leftrightarrow \forall x{:}A.\, P(x) \& \forall y{:}B.\, Q(y).$
2. $\forall A, B{:}U_1 . \forall R{:}A \times B \to Prop_1.\, (\exists y{:}B.\, \forall x{:}A.\, R(x,y) \Rightarrow \forall x{:}A.\, \exists y{:}B.\, R(x,y))\,.$

At this level of generality, we need to express the well-formedness of typed formulas in the logic rather than as preconditions on the formulas as we did in section 2. This can be accomplished easily using U_i and $Prop_i$. We incorporate into the rules the conditions necessary for well formedness. For example, in the rule

$$\begin{array}{l} \bar{H} \vdash P \Rightarrow Q \text{ by } \Rightarrow R \\ \quad \bar{H}, P \vdash Q \end{array}$$

We need to know that P and Q are propositions. We express this by additional well-formedness subgoals. A complete rule might be

$$\boxed{\begin{array}{l} \bar{H} \vdash P \Rightarrow Q \text{ by } \Rightarrow R \text{ at } i \\ \quad \bar{H}, P \vdash Q \\ \quad\quad \bar{H} \vdash P \in Prop_i \\ \quad\quad \bar{H} \vdash Q \in Prop_i \end{array}}$$

If we maintain the invariant that whenever we can prove $\bar{H} \vdash a \in A$ then we know A is in a U_i, and whenever we prove $\bar{H} \vdash P$ then we know P is in $Prop_i$, then we can simplify the rule to this

$$\boxed{\begin{array}{l} \bar{H} \vdash P \Rightarrow Q \text{ by } \Rightarrow R \text{ at } i \\ \quad \bar{H}, P \vdash Q \\ \quad\quad \bar{H} \vdash P \in Prop_i \end{array}}$$

We need to add well-formedness conditions to the following rules, $\vee R$, $\Rightarrow R$, $\forall R$, *Magic*. We already presented $\Rightarrow R$; here are the others.

$$\begin{array}{ll} VR & \bar{H} \vdash P \vee Q \text{ by } \vee R_l \text{ at } i \\ & \quad \bar{H} \vdash P \\ & \quad \bar{H} \vdash Q \in Prop_i \end{array}$$

The $\vee R_r$ case is similar.

$$\begin{array}{ll} \forall R & \bar{H} \vdash \forall x{:}A.\, P(x) \text{ by } \forall R \text{ at } i \\ & \bar{H}, x{:}A \vdash P(x) \\ & \bar{H} \quad \vdash A \in U_i \end{array}$$

$$\begin{array}{ll} Magic & \bar{H} \vdash P \vee \neg P \text{ by } Magic \text{ at } i \\ & \quad \bar{H} \vdash P \in Prop_i \end{array}$$

3.12. Proofs as programs

The type corresponding to a proposition of the form $(\forall x{:}A.\ \exists y{:}B.\ S[x,y])$ is the function space $x : A \to y : B \times [\![S[x,y]]\!]$. The proof expressions, say p, for this object denotes a canonical element of the type. That element is a function $\lambda(x.\ b)$ where for each $a \in A, b[a/x] \in y{:}B \times [\![S[a,y]]\!]$ and if $1of(b[a/x]) \in B$ and $2of(b[a/x]) \in [\![S[a, 1of(b[a/x])]\!]$. So the function $\lambda(x.\ 1of(b)) \in A \to B$ and let $f = \lambda(x.\ 1of(b))$, then $f \in A \to B$ and $\lambda(x.\ 2of(b))$ proves $\forall x{:}A.\ S[x, f(x)]$.

So we can see that the process of proving the "specification" $\forall x{:}A.\ \exists y{:}B.\ S[x,y]$ constructively creates a program f for solving the *programming task* given by the specification, and it simultaneously produces the verification $\lambda(x.\ 2of(b))$ that the program meets its specification (c.f. Constable [1972], Bates and Constable [1985] and Kreitz [n.d.]).

Refinement style programming. This style of programming provides a way to build the program and its justification hand-in-hand. It is possible to gradually refine these two objects, filling only as much detail as necessary for clarity. So for example, proof detail can be omitted for programming steps that are obvious. The extreme case of "unbridled" programming arises when we omit all proof steps except those that come automatically as part of the programming, e.g. certain "type checking steps" and the over all logical structure of the proof.

Explicit programming style. We can *program* a solution to $\forall x{:}A.\ \exists y{:}B.\ S[x,y]$ directly by writing a function $f \in A \to B$ and then proving $\forall x : A.\ S[x, f(x)]$. Christine Paulin-Mohring [1989] is studying how to use the program information to help drive the derivation of the proof.

4. Typed programming languages

4.1. Background

Programming at its "lowest level" involves communicating with specific digital hardware in "machine language," sequences of *bits* (0's and 1's). The particular machine model will classify sequences of bits into a fixed number of "types," say instructions, signals, addresses, and data; the data might be further classified as *floating point* or *integer* or *audio* or *video*, etc. Programming at this machine level or just above at *assembly language* level is generally regarded as "untyped" in part because everything is ultimately bits.

We are mainly concerned with so-called *higher-level programming* languages, and for the purpose of this discussion, higher-level languages will be classified into two groups as *typed* or essentially *untyped*. Two high level languages from the earliest period are still "alive," Fortran and Lisp. Fortran is considered typed (though minimally) as are more modern languages like Pascal, C^{++}, ML, and Java. Two of

the most historically significant typed languages were Algol 68 and Simula 67. Lisp is considered *untyped* as is its modern descendent Scheme. These languages have a notion of *run-time typing* in which data is tagged with type information during execution. Whereas Algol 68, ML, and Java, for example, are *statically typed* in that data and expressions are typed before execution (at "compile time").[41]

One of the major design debates in the computer science community over the years has been about the value of rich static typing, represented by Algol 68 and Simula, and "untyped" programming represented by Lisp and Scheme. There are formal languages that capture the essence of this distinction. Lisp and Scheme are represented by the *untyped lambda calculus* of Church [1960] (see Barendregt [1981], Stenlund [1972], Hindley, Lercher and Seldin [1972]) on which they were modeled, and ML by the *typed lambda calculus* (see Barendregt [1977], de Bruijn [1972]).

We have seen the untyped lambda calculus in section 3.4. Its terms are variables, abstractions, and applications denoted respectively x_i, $\lambda(x.\ t)$, and $ap(s;t)$ for s and t terms. The typed calculus introduces some system of types T and requires that the variables are typed, x^T. Usually the types include the individuals, ι, and if α, β are types, then so is $(\alpha \to \beta)$. The untyped lambda calculus can express the full range of sequential control structures and hence the class of general recursive functions. For example, the Y combinator $\lambda(f.\ ap(\lambda(x.\ ap(f; ap(x;x)))); \lambda(x.\ ap(f; ap(x;x))))$ more commonly written $\lambda(f.\ f\lambda(x.\ xx))\ \lambda(x.\ xx)$ is used to define recursive functions. We have that $Y(\lambda(f.\ F[f])) = F[Y(\lambda(f.\ F[f]))]$ so that Y "solves" the recursive definition $f = F[f]$.

In the typed lambda calculus, Y is not typeable because the self-application $\lambda(x.\ ap(x;x))$ cannot be typed. This situation summarizes for "typeless programming devotees" the inherent limitations of typed programming; for them types "get in the way."

The debate about typed or untyped languages illustrates one of the many design issues that have been studied and debated over the years. Other topics include: functional *versus* imperative, lazy *versus* eager evaluation, manual *versus* automatic storage allocation, reflection or not, and so forth.

Many of these issues have been explored with theoretical models, and much is known about the design consequences. Indeed many programming language constructs arose first in the setting of formal logical theories, e.g. the lambda calculus, type systems, binding mechanisms, block structure, abstract data types (as algebraic structures) and modules. Just as assembling a good formal theory is high art, so is assembling a good programming language. Both are formal systems which can be processed by computers. But there is at least one major difference.

Good programming languages are widely used, perhaps by tens of thousands of people over their life times. Most logical theories are never implemented, and the best of those that are might be used by less than one hundred people over a lifetime.[42]

[41] A compiler translates high-level language programs into another language, typically a lower-level language such as assembly code or native code (machine language).

[42] We hope that the fact that Nuprl contains a programming language and that proofs are executable will attract a significant audience.

I believe that this fact has a major consequence for "theory designers," namely they must learn about programming language evolution.

We see from a history of programming languages what ideas "work", what combinations of features are most expressive, what constructs are heavily used. As with the evolution of natural languages, the speakers exert a force to mold the language to its purpose. One of the lessons of programming language history is that types are critical. A language's type system is its *most important component.* We also know that modularity mechanisms are critical, but this too is defined by the type system.

The evolutionary trend is toward ever richer type systems—from the fixed types of Fortran to the polymorphic recursive types of ML and the classes of Java. One might argue that *this development must eventually subsume the type systems of the mathematical theories.* I believe this is true, and our discussion of type systems will reveal why.

Role of types in programming. Let us examine the role of types in programming (see the excellent article by Hoare [1972] as well). Fortran used variable names beginning with i, j, k, l, m, n to denote integers (fixed point numbers), the other letters indicated reals (floating point numbers). This type distinction facilitated connection to mathematical practice where the same conventions were used, and it provided information to the compiler about how to translate expressions into assembly language which also made the distinction between fixed and floating numbers.

Another important type in Fortran and Algol was the *array.* Arrays represent sequences, matrices, tensors, etc. A typical specification (or declaration) of this type might be *real array*$[n, m]$, a two dimensional array (matrix) of reals. The declaration provides a link to important mathematical types such as sequences or matrices, and provides information to the compiler on how much memory needs to be allocated for this data.

The *record type* (or Algol structure) also provides links to mathematical types and provides information for the compiler. A typical record syntax is $record(a_1 : T_1; \ldots; a_n : T_n)$ where T_i are types and a_i are identifiers called *field selectors.* This type corresponds to a *cartesian product* $T_1 \times \ldots \times T_n$, and if t is an expression of this record type, then $t.a_i$ indicates the i-th component, which has type T_i. We discuss the field selectors in Section 4.4.

In this case the type declaration also introduces new identifiers (or names) into the language. This was a convenience not systematically used in mathematics. But it also led to some confusion about the status of these names a_i; are they bound variables or free? And if bound, what is their scope? Here a small "convenience" leads to interesting new questions about scope and naming in formal languages.

Algol 68 introduced a union type, $union(T_1, \ldots, T_n)$. This was an obvious attempt to link to mathematical types, but it created problems for efficient language translation since the compiler might have to reserve storage based on the type T_i needing the most memory. This type also brought language designers face to face with the problems of a "computable set theory." A programmer given data t in the type $union(A, B, C)$ will need to know which type it is in. So there must be an operation, like $decide(t)$ which will decide what type t belongs to. This operation

is not available as a computable operation in set theory, so new mathematics had to be worked out. Algol 68 was rich in a "computable mathematics of types," and its reference manual is a type theory which inspired both logician and computer scientist alike.

In Pascal the union type was considered to be a *variant* of the record type. The simplest such structure is essentially $record(x_1 : A_1; x_2 : A_2[x_1])$ which is thought of as a union indexed by the (necessarily finite) type A_1. This is a restricted version of our dependent product type $prod(A_1; x.\ A_2[x])$ from Section 3.7. The Pascal conception reveals both the computational way to treat unions, namely use *disjoint unions*, and reveals the implementation strategy (borrowed from set theory)—use elements from a type A_1 as tags on the data to keep track of the disjunct. So if the tag types are the booleans, $\mathbb{B}$, and $A_1(i) = \textit{if } i \textit{ then } S \textit{ else } T \textit{ fi}$ then $prod(\mathbb{B}; i.\ A_1(i))$ is the Algol 68 $union(S, T)$ and the Pascal variant record $record(i : \mathbb{B}; x : A_1[i])$.[43]

Algol 60 and Algol 68 considered the notion of higher order functions. Algol 68 essentially had the idea of the type $\mathsf{fun}(x : A)B$ as the type of function from A to B. But the implementation technology was not up to the task of returning functions as values. This type challenged the community to implement it correctly as done in Scheme and ML with *closures*.

The function space concept $\mathsf{fun}(x : A)B$ does not mean the same thing as the corresponding mathematical notion, $A \to B$ even in the constructive case. In computational mathematics the elements f of $A \to B$ are *total* functions; that is, on every element a of $A, f(a)$ converges to a value b in B. Whereas, the elements ψ of $\mathsf{fun}(x : A)B$ are *partial functions*, that is, $\psi(a)$ might diverge or abort without returning a value. This is a major difference between programming types and mathematical types.

There are two reactions to the difference. It is possible to give total function semantics to $\mathsf{fun}(x : A)B$ and claim that current implementations are just approximations to the idea. The full concept emerges in a programming logic with termination rules (Dijkstra [1968]). On the other hand, one can regard the partial function space as a new mathematical construct and try to work out axioms and models for it (Scott [1976], Plotkin [1977]). Both approaches have been pursued.

Notice that it is a simple manner to extend $\mathsf{fun}(x : A)B$ to dependent function types by allowing B to depend on x. This type is then closely related to $\mathsf{fun}(A; x.\ B)$ of Section 3.

A more modern addition to the type structure of programming languages is the *module* or *object* (or ADT or package or unit). This concept can be traced to Simula 67 and is well developed in Modula and SML. Among the interesting experimental languages for modules were Russell at Cornell (Demers and Donahue [1980], Boehm et al. [1986]), CLU at MIT (Liskov and Guttag [1986]), and Modula at DEC. The basic idea is that a module is a type, say D, and a collection of operations f_i

[43] The actual Pascal syntax is very baroque, and the so called free unions are a well known place for "breaking" the type discipline since the user must keep track of the dependency. Note the *if_then_else_fi* notation is the Algol 68 way of "bracketing" the conditional with delimiters *if*, *fi*.

on D and auxiliary types. This is the type of a *structure* in algebra (Bourbaki [1968a]) and model theory (Chang and Keisler [1990]). For example, we might have $\langle D, f, g, e\rangle$ where the *signature* of the module is list of types of the components, e.g. $D \in Type$, $f : D \times D \to D, g : D \to \mathbb{B}, e \in D$. A *group* would have signature $G \in Type, op : G \times G \to G, inv : G \to G, e \in G$, and then there would be axioms saying that *op* is associative, *inv* is an inverse and *e* an identity.

The module concept corresponds exactly to dependent types over Type. In Section 2 we would denote the type of groups (signature) as

$$G : Type \ \times op : (G \times G \to G) \times \ inv : (G \to G) \times e : G$$

Except for the fact that the function types in the programming type are partial and Type has less mathematical structure, the algebraic concept and the programming one are similar.

We will see that the notion of subtype and inheritance that is so critical to modern programming practice can be nicely captured in our type theory. This leads to a mathematical treatment of the central concepts in *object-oriented programming* (c.f. Meyer [1988]).

Looking over the types described above we discern these uses.

1. Types relate data in the machine to standard mathematical concepts.
2. Types express the domain of significance of a programming problem and impose constraints on the data for it to be "meaningful" in the sense that the computer will not "crash" (attempt to execute a meaningless instruction) and the data will not fail to represent mathematical objects.[44] Usually these constraints can be rapidly checked to provide some level of assurance that a program is sensible.
3. Types provide a notation for structuring a solution by decomposing a task into components (modules) and levels of abstraction.
4. Types provide an interface language for analyzing ("debugging") a computation.
5. Type information can be used to increase the performance of the compiled code.

There is a direct historical link from Russell and Church to languages like Algol and Lisp. Also we are seeing a close correspondence between mathematical types and data types: Cartesian products correspond to record types, unions to disjoint unions (or variant record types), function spaces to procedure types, inductive types to recursive data types, algebraic structures to modules (and superstructures correspond to subtypes). The integers are included in some programming languages as the data type "bignums", and real numbers are (badly) approximated by "floating point numbers". In a sense the system of data types provides a computational type theory capable of organizing and unifying programming problems and solutions in

[44] Crashing can mean a complete failure to respond or an unwanted response from the operating system ("bus error") or from the hardware ("segmentation fault").

the same way that type theory organizes and unifies computational (also constructive and intuitionistic) mathemataical problems and solutions. The continuing (rapid) evolution of programming languages will probably lead to data type theories that subsume mathematical type theories. There may be new data types appropriate for expressing the problems of *interaction* as well as those of "functional action" which now dominate.

Although the similarities between types and data types just enumerated is compelling and interesting, I think it is also important to understand the differences. These differences challenge us to find logical foundations for new types.

4.2. Type $\in$ type and domain theory

Given that programming types are not the same as mathematical ones, might it be sensible to allow a type of all types, precisely the notion that type theory was created to disallow in accordance with the vicious circle principle? One fact we know from the work of Meyer and Reinhold [1986] and Howe [1991,1989,1987,1996b] is that adding the typing rule $Type \in Type$ to the simply typed lambda calculus allows new terms to be typed among which are applications that fail to terminate. No such terms can be typed without this new rule. On the other hand, this rule would not cause the type system to "collapse" in the sense that every term could be typed or every term belongs to every type (as would happen if we added the rule $T_1 = T_2$ for any two types T_1 and T_2). Indeed, we know that such a type system has a nontrivial mathematical model (Cardelli [1994], Meyer [1988]).

The discovery of interesting mathematical models for programming language types is a flourishing topic in the field of programming language semantics. It has led directly to the rich subject of *domain theory* pioneered by Dana Scott [1970a,1970b,1972,1976] led early on by Gordon Plotkin [1975]. (The results of Plotkin [1981], Abramsky [1993], Reynolds [1981], Cardelli [1994], Mitchell [1996], Gunter [1994], Egli and Constable [1976], and Abadi and Cardelli [1996] are quite relevant to the work discussed here.)

One of the major early discoveries of domain theory is that there are referential or "denotational" mathematical models of partial function spaces, in particular, of the untyped lambda calculus in which function equality is extensional (see Scott [1976]). The challenge for domain theory has been to relate these models to the standard mathematical types and type theories. This remains an active area of research with especially promising recent results in analysis (Edalat [1994]).

Let us call types which allow diverging elements *partial types*. Given that there is a consistent theory of partial types allowing $Type \in Type$ and that this rule drastically simplifies the theory, we proceed to explore it.

One view of this theory is that it speaks about a domain. Another is that it is a "partial type theory" which will require refinement as more constraints are added, such as totality restrictions. But until we require totality, the vicious circle principle has no force since its consequence is merely a nonwell founded concept (nonterminating term). This approach to type theory permits a great deal of freedom—partial

objects are allowed, *illogical comprehension* is possible, e.g. $\{x : Type \mid x \in x\}$, negative recursive definitions are allowed (see Section 4.3), and concepts need not be referential since equality relations are not required. It will be left to the programming logics to impose more logical order on these "unruly" types.

One of the first benefits of this theory is that dependent products taken over Type provide a notion of *module*. The signature (or type) of a module is

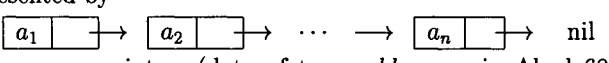
$$M : Type \times F(M)$$

where $F(M)$ is a type built from M such as $M \times M \to M$. By iterating this construct we get the general structure of a module

$$x_0 : Type \times x_1 : T_1(x_0) \times \cdots \times x_n : T_n(x_0, \ldots, x_{n-1}).$$

4.3. Recursive types

As we have seen, inductive definitions and principles of inductive reasoning lie at the heart of computational mathematics and logic. The inductive definition of the natural numbers, lists, and formulas come immediately to mind. The elements introduced inductively can be represented in computer memory by *linked data* structures constructed from *pointers*. For example, a list of elements of type A, say $(a_1, \ldots, a_n)$, would be represented by



where the arrows are pointers (data of type *address* or in Algol 68 terminology, *references* to A objects, thus of type $ref(A)$). A seminal discussion of these methods can be found in C.A.R. Hoare's article *Notes on Data Structuring* [1972].[45]

One of the most decisive uses of types in programming languages is in defining recursive data types at the same level of abstraction used in mathematics. This innovation was pioneered by Lisp and its treatment of lists without explicit mention of pointers. The pointer representation is managed by the run-time system of programming language, and a program called a *garbage collector* is used to dynamically manage the allocation and deallocation of memory for lists and other inductive structures.

In programming these inductive types are called *recursive types* or recursive data structures by analogy with recursive programs. They include circular data structures, unfounded lists (or streams) and other "nonwell-founded" recursive data that would not be considered as properly "inductive."[46] The definition of such a

[45]The small book *Structured Programming*, Dahl, Dijkstra and Hoare [1972], is one of the gems of computer science. All three articles are closely related to the subject of this section.

[46]Perhaps the reason for the popularity of the term "recursive data type" comes from Hoare's evocative analogy: "There are certain close analogies between the methods used for structuring data and the methods for structuring a program which processes that data ... a discriminated union corresponds to a conditional ... arrays to *for* statements ... sequence structure ... to unfounded looping .... The question naturally arises whether the analogy can be to a data structure corresponding to recursive procedures."

type is disarmingly simple to paraphrase Hoare: "write the name of the type being defined inside its own definition." In his notation we write

$$type\ T = F[T]$$

where $F[X]$ is a type definition in X. If we use $+$ for disjoint union and $\mathbf{1}$ for the unit type and $\times$ for cartesian product, then here are the definitions for natural numbers and lists over a type A.

$$\begin{aligned} type\ N &= \mathbf{1} + N \\ list\ L &= \mathbf{1} + (A \times L). \end{aligned}$$

We will use a more compact notation, writing a single term with a binding construct. Our notations for these types are $\mu(N.\ \mathbf{1} + N), \mu(L.\ \mathbf{1} + A \times L)$ where N and L are bound variables. In general, if $F[T]$ is a type expression in T, then $\mu(T.\ F[T])$ denotes the *recursive type* used above to illustrate Hoare's notation. In giving the rules for recursive types, we will use $A \to B$ and $x{:}A \to B[x]$ for the programming type $fun(x{:}A)B$; so the elements are *partial functions.*

1. $H \vdash \mu\,(x.\ F[x]) \in Type \quad rec\_type\_def$
   $H, x{:}Type \vdash F[x] \in Type$
2. $H \vdash t \in \mu\,(x.\ F[x]) \quad rec\_type\_member$
   $H \vdash t \in F\,[\mu(x.\ F[x])]$
3. $\bar{H} \vdash \mu\,(t; f, y.\ g[f, y]) \in G \quad rec\_type\_elim$
   $\bar{H}, x{:}Type, f{:}x \to G, y{:}F[x] \vdash g[f, y] \in G$
   $\bar{H} \vdash t \in \mu(x.\ F[x])$

The term $\mu\,(t; f, y.\ g[f, y])$ is called a *recursion combinator.* It is the recursive program associated with the recursive definition. The evaluation rule is

$$\frac{g\,[\lambda\,(z.\ \mu\,(z; f, y.\ g[f, y]))\,/f, t/y] \downarrow a}{\mu\,(t; f, y.\ g[f, y]) \downarrow a}$$

The operational intuition behind these rules is this. A recursive type $type\ T = F[T]$ is well formed exactly when its "body" $F[T]$ is a type under the assumption that T is a type. This is "writing the name of the type being defined in its own definition." To construct a member of the type, build a member of $F[T]$, and if this construction requires an element of T, then apply the construction recursively (in the implementation, use a pointer to T and build recursively). The process may not terminate unless there is a "base case" which does not mention T, as in the left disjunct of $\mathbf{1} + T$ or of $\mathbf{1} + A \times T$. A definition like $\mu(X.\ X)$ is empty because no element can be created, likewise for $\mu(X.\ X + X)$ or $\mu(T.\ T \times T)$. Note however that $\mu(T.\ T \to T)$ will contain the element $\lambda(x.\ x)$ by this application of rules

$$\begin{aligned} \vdash \lambda(x.\ x) &\in \mu(T.\ T \to T) \\ T{:}Type \vdash \lambda(x.\ x) &\in T \to T \\ T.\ Type,\ x{:}T \vdash x &\in T. \end{aligned}$$

Associated with $\mu(x.\ F[x])$ is a method of *recursive computation* (as Hoare suggested and as we know from inductive definitions in mathematics). If the recursive type is "well-founded" then this procedure will terminate, otherwise it might not. The recursive procedure is the following. Given $t \in \mu(X.\ F[x])$, to compute an element of type G, use a program g that computes on elements of $F[x]$. This procedure g may decompose t into components t' of type $\mu(x.\ F[x])$. In this case, call the procedure recursively. To specify this we note that if we consider that t belongs to $F[x]$, then component t' will belong to X. The recursive call of the procedure is represented in the rule by the function variable f from X to G. We see from the evaluation rule that this is used exactly as a recursive call.

This method of organizing the rules comes from Constable and Mendler [1985] and Mendler [1988]; it can be made more expressive using the subtyping relation $S \sqsubseteq T$ and dependent function types and parameterized recursions. First, with dependent types we get

$$\bar{H}, u{:}\mu(X.\ F[X]) \vdash \mu(u; f, y.\ g) \in G[u]$$
$$\bar{H}, X{:}Type, f{:}\ (x{:}X \to G[x]), y{:}F[X] \vdash g \in G[y]$$

The parameterized form of recursive type allows the defined type to depend on a parameter of type A. The syntax is $\mu(X.\ F[x])\ @a$

1p. $\bar{H} \vdash \mu(X.\ F[X])\ @a \in Type$
$\bar{H}, X{:}A \to Type \vdash F[x] \in (A \to Type)$
$\bar{H} \vdash a \in A$

2p. $\bar{H} t \in \mu(X.\ F[x])\ @a$
$\bar{H} \vdash t \in F[\lambda(y.\ \mu(X.\ F[X]))\ @y]\ (a)$

3p. $\bar{H} \vdash \mu(a; t; f, u, y.\ g) \in G$
$\bar{H}, X{:}(A \to Type), \forall u{:}A.\ (X(u) \sqsubseteq \mu(X.\ F[X])@u)$
$\vdash g[f, u, y] \in G$
$\vdash a \in A$
$\vdash t \in \mu(X.\ F[x])@a$

4p. $$\frac{g[\lambda(u.\ \lambda(r.\ \mu(u; r; f, u, y.\ g)))/f, a/u, t/y] \downarrow c}{\mu(a; t; f, u, y.\ g) \downarrow c}$$

We can combine the parameterized form and the dependent form; such rules are given in Constable et al. [1986] and Mendler [1988], but we won't use this level of complexity here.

The parameterized recursive types can be used to define mutually recursive types since we can think of $\mu(X.\ F[x])@u$ as a family of simultaneously recursively defined types. With the propositions-as-types principle and restricting the recursive types to be well-founded, we get recursively defined relations. These have been exploited well in the Coq theorem prover (Coquand and Paulin-Mohring [1990], Coquand [1990], Paulin-Mohring and Werner [1993]).

With recursive types and disjoint unions and a unit type we can define natural numbers and lists as we have shown. Using record types we can define pairs of numbers which gives us integers and rational numbers. (Using function types we can define the *computable reals*; see Bishop [1967], Chirimar and Howe [1991], Forester

[1993].) Booleans can be defined as $\mathbf{1} + \mathbf{1}$. So the number of primitives for a rich type theory can be reduced to a very small set. We will examine some especially interesting reductions in the next section.

Example defining primitive recursion on N. To illustrate the workings of the recursion combinator $\mu()$, we use it to define primitive recursive functions from $\mathbf{N}$ to G. Suppose f is defined primitive recursively on $\mu(X.\ \mathbf{1} + X)$ to G by

$$\begin{aligned} &f(0) = b \\ &f(suc(u)) = h(n, f(n)). \end{aligned}$$

Then the corresponding combinator is $\mu\,(u; f, u.\ decide\,(u; v.\ b; v.\ h(v, f(v))))$ whose typing is seen from the judgment.

$$X{:}Type, f{:}X \to G, u{:}\mathbf{1} + X \vdash decide(u; v.\ b; v.\ h(v, f(v))) \in G.$$

Typing a fixed point combinator. While the recursion combinators are essential for inductive types, indeed they characterize them, in a rich enough partial type theory they can be defined. The idea is to use the richness of the recursive types to assign a type to a fixed point combinator, like Y. Recall that the Y combinator is abbreviated $\lambda\,(g.\ \lambda(x.\ g(xx))\lambda(x.\ g(xx)))$ or still further by letting $w = \lambda(x.\ g(xx))$ and writing Y as $\lambda(g.\ ww)$. We show that Y has type $(T \to T) \to T$ for any type T by using the auxiliary recursive type $S == \mu(X.\ X \to T)$. Here is the derivation.

The type of g will be $T \to T$, the type of w is $S \to T$. The "trick" is to type $ap(x; x)$ to be of type T. We examine the typing derivation for w.

$g{:}T \to T$	$\vdash \lambda(x.\ g(xx)) \in \mu(X.\ X \to T)$	by μ_membership
	$\vdash \lambda(x.\ g(xx)) \in S \to T$	by $\to R$
$g{:}T \to T, x{:}S$	$\vdash g(xx) \in T$	by $\to L$
	$\vdash xx \in T$	by ap
	$\vdash x \in S \to T$	by $unroll\ x$
	$\vdash x \in S$	by $hyp\ x$

Once we know that $w \in S \to T$ and $w \in S$, then $ww \in T$ and $g(ww) \in T$.

One corollary of this typing is that $Y(\lambda(x.\ x))$ belongs to the empty type $\mu(X.\ X)$ called *void*, since $\lambda(x.\ x) \in void \to void$. But $Y(\lambda(x.\ x))$ is a diverging term, so it is not a *value* belonging to *void*. Indeed, we can easily show that there are no values of type *void*.

Now we can use Y to define any partial recursive function, including the recursion combinators of type $\mu(X.\ F) \to G$. In general, $\lambda\,(x.\ \mu(x; f, u.\ g[f, u]))$ is just $Y\,(\lambda\,(f.\ \lambda(u.\ g[f, u])))$. The type of f is $(\mu\,(X.\ F) \to G) \to (\mu(X.\ F) \to G)$, and we observed that $g[f, u] \in G$ can be derived from this typing of f.

Applying this general construction to primitive recursion we get the term $Y(\lambda(f.\ \lambda(n.\ decide(u; v.\ b; v.\ h(v, f(v))))))$, which is R, the primitive recursion combinator, (with b and h as parameters).

Inductive types. Constable and Mendler [1985] and Mendler [1988] gave conditions needed to guarantee that recursive types $\mu(X.\ F)$ define only total objects. One such condition is that F be a monotone operation on types in the sense that $X \sqsubseteq Y \Rightarrow F[X] \sqsubseteq F[Y]$. We also studied conditions to guarantee that elements of these types are functional. The result is a set of rules used in Nuprl for inductive types (c.f. Constable et al. [1986], Hickey [1996a]).

When F is required to be monotone, then we cannot define the type $\mu(X.\ X \to T)$ used in typing Y. Indeed, it is not possible to type Y nor divergent elements. For this reason the $\mu(x; f, u.\ g)$ recursion forms are needed. They provide the structural induction rules for inductive types. In Nuprl these induction rules for recursive types can be used to prove that certain applications of the Y combinator, $Y(\lambda(f.\ b))$ are indeed total objects (see Constable et al. [1986]). So we get the advantages of general recursive programs without losing the logical structure of type theory.

4.4. Dependent records and very dependent types

We are aiming to exhibit a small core type system that will generate all of the types we have studied. The step in this direction that we take here is of considerable practical value—it builds record types from dependent function spaces.

Consider the record type $record(x_1 : A_1, \ldots, x_n : A_n)$. Let $N_n == \{1, \ldots, n\}$ be an n element enumeration type—it can simply be $\mathbf{1} + \cdots + \mathbf{1}$ taken n times. Define a function $B(i) = A_i$ from N_m to Type. Then the essential structure of the record is given by the dependent function space $e : N_n \to B(i)$. Given f in this type, $f(i)$ is the i-th component. We obtain a nice display form for record selection if we define $f.\ x_i == f(i)$.

This definition of records has nice subtyping properties. In a standard record calculus a record type, r_1, is a subytpe of record type r_2, written $r_1 \sqsubseteq r_2$, iff r_1 has additional fields. So a colored point is a subtype of a point or a group type is a subtype of monoid type, etc. Our definition provides this subtyping directly from the subtyping relation on function spaces. Recall that if $A_1 \sqsubseteq A_2, B_1 \sqsubseteq B_2$ then $A_2 \to B_1 \sqsubseteq A_1 \to B_2$. Also if $N_n \sqsubseteq N_m$, and $n \leq m$, and $B_1(i) = B_2(i)$ for $i \in N_n$ then

$$i : N_m \to B_1(i) \sqsubseteq i : N_n \to B_2(i).$$

Notice that $f \in (i : N_m \to B_1(i))$ is an element of $i : N_n \to B_2(i)$ simply by the polymorphic nature of functions (i.e. they are rules given by λ terms).

Encoding dependent records. The dependent product types, $x : A_1 \times A_2[x]$ offer a form of dependent record as mentioned above. The general form is $record(x_1 : A_1; x_2 : A_2[x_1]; \ldots; x_n : A_n[x_1, \ldots, x_{n-1}])$. Can we also define these records as dependent functions?

The existing dependent function space is not adequate for this task, but Jason Hickey [1996a] has discovered an extension that he calls *very dependent* function

spaces. The basic notation is $fun(A; f, x.\ B[f, x])$ as opposed to $fun(A; x.\ B[x])$. The idea is that the type B can depend not only on the argument to the function so that $g(a) \in B[a]$, but now the type of B can depend on "previous values" of g, so $g(a) \in B[g, a]$. To see how the idea works, let's use it to define the elements of $x_1 : A_1 \times x_2 : A_2(x_1)$. Note $A_2 : A_1 \to Type$, and an element is $\langle a_1, a_2 \rangle$ where $a_1 \in A_1, a_2 \in A_2(a_1)$. The encoding is based on $N_2 = \{1, 2\}$. Imagine that $B(1) = A_1$, and we want $B(2) = A_2(a_1)$ where $a_1 \in A_1$. We could say this if we had the element g such that $g(1) \in A_1$. So if we add g as a parameter to B we can say

$$\begin{aligned} B(g, 1) &= A_1 \\ B(g, 2) &= A_2(g(1)). \end{aligned}$$

This particular definition makes sense because at $B(g, 2)$, g is referenced only at previous arguments. Hickey takes this as the basis for defining the simplest *very dependent function space.* He requires a well-ordering on g as prerequisite to forming the type (see Hickey [1996a]). In a partial type theory we can get away with less. A particular computable function g will generate an ordering on values via its computation. So we can allow arbitrary expressions $B[g, x]$ in forming the type, but it will be empty unless there is a function satisfying the constraints of B. The (viciously circular) rules are:

1. $\bar{H} \vdash fun(A; f, x.\ B) \in Type$
   $\bar{H} \vdash A \in Type$
   $\bar{H}, x{:}A, f{:}fun(A; f, x.\ B) \vdash B \in Type$
2. $\bar{H} \vdash \lambda(x.\ b) \in fun(A; f, x.\ B)$
   $\bar{H}, x{:}A \vdash b \in B[\lambda(x.\ b)/f]$
3. $\bar{H} \vdash g(a) \in B[g/f, a/x]$ by *ap over* $fun(A; f, x.\ B)$
   $\bar{H} \vdash g \in fun(A; f, x.\ B)$
   $\bar{H} \vdash a \in A$

With this type we can define dependent products as

$$prod(A; x.\ B[x]) == fun(N_2; f, x.\ \text{if } x = 1 \text{ then } A \text{ else } B[f(1)]).$$

4.5. A very small type theory

The previous reductions show that we can define a very rich type theory using only three primitive type constructors and one primitive type, namely $Type$.

$$\begin{array}{lll} \text{types:} & Type \quad A + B & fun(A; f, x.\ B) \quad \mu(X.\ B) \\ \text{values:} & inl(a),\ inr(b),\ \lambda(x.\ b) & \\ \text{forms:} & decide(t; u.\ a; v.\ b) & ap(t; a) \end{array}$$

This language can be seen as a combination of the ideas from Constable and Mendler [1985], Mendler [1988], Hickey [1996a]; it is in the style of Mendler's thesis using Hickey's key reduction. The language FPC in Gunter's textbook [1992] considers the nondependent recursive types in a similar spirit.

5. Conclusion

In the main, this article is a snapshot of three subjects recently come into alignment. This conclusion addresses research dynamics driving these subjects.

Typed logic. Many standard topics in logic must be reworked for typed logic. We have already seen that its *deductive machinery* is different, so we need to ask about normalization results for natural deduction (as in Prawitz [1965]) or cut elimination for numerous variants of the sequent calculus (with structural rules or without, tableau style or bottom up, etc.) What properties of the normal syntax of proofs reflect their deeper semantic content? What symmetries of the sequent calculus reveal properties of evidence?

The emergence of automated deduction systems has introduced new issues and questions. For example, the notion of a *tactic-tree proof* (Allen et al. [1990]) illustrated here is a novel structure, and its use in *refinement logics* (Bates [1979], Bates and Constable [1985]) raises questions, such as, how is soundness and type correctness of the metalevel programming language for tactics related to the soundness of the logic?

The traditional questions about the relative "power" of logical theories can be posed for typed logics, and the various translation results such as the Kolmogorov and Gödel translations are being studied (Troelstra and Schwichtenberg [1996]). Chet Murthy [1990,1992] discovered remarkable results relating these translations to Plotkin's CPS translations, and he proved Friedman's [1978] theorem for a fragment of Nuprl as part of this work (see also Palmgren [1995a]). These results have been applied in interesting ways in program extraction by Murthy [1992] and Berger and Schwichtenberg [1996]. Friedman's program of "reverse mathematics" can be elaborated in this context as well, and now *programming logics* can be considered in a more uniform manner (Kozen [1977], Kozen and Tiuryn [1990]).

The subject of *applied logic* has emerged in the intersection of logic and computer science. This includes the study of specification languages such as Z (Spivey [1989]), a main topic in *formal methods.* The languages of typed logic (say in Coq, HOL, Nuprl, and PVS) provide alternative specification languages which seem to have advantages over Z in automation. These rich typed logics can accommodate special languages such as those needed in temporal logic and for hybrid systems (Nerode and Shore [1994], Henzinger and Ho [1994]).

The field of *automated deduction* is a flourishing part of applied logic. Presently, specialized tools such as *model checkers* (c.f. Clarke, Long and McMillan [1989], Burch et al. [1991], Henzinger and Ho [1994]), *type checkers* (c.f. Milner, Tofte and Harper [1991]), and arithmetic *decision procedures* are already used by industry in production. Integrated systems like Coq, HOL, Nuprl, and PVS are also valuable to industry.[47] The deployment of logic-based industrial systems has led to a wealth of research problems and challenges (Kreitz, Hayden and Hickey [n.d.]). For example,

[47]The late IBM Fellow, Harlam Mills, said in December 1984, "It is the kind of research that can change the course of industrial history."

it is becoming imperative to share libraries of mathematics between provers. Howe's work [1996a] with HOL libraries in Nuprl is one of the first examples of how this can be done. Practical deployment relies on several years of investigating the underlying semantic issues involved in translating between theories (Howe [1996b,1991]).

The need to share results between provers is only one example of a more general need to build more *open* theorem proving systems. These systems should be able to interface with several text and proof editors, with other provers, with programming languages to evaluate computable terms, and with metalanguages for managing proof planning and generation. Nuprl Version 5 is one such system. We discuss these problems in a wider context in Collaborative Mathematics Environments (Chew et al. [1996]).

Type theory. The research agenda in type theory is strongly tied to logic as this article illustrates, providing a new semantics. In addition, there are strong ties to pure and applied mathematics (Gallier [1993]). Indeed, Martin-Löf type theory arose as an attempt to find a foundational account of the practice of constructive mathematics, especially in the style of Bishop (Bishop [1967], Bishop and Bridges [1985], Mines, Richman and Ruitenburg [1988]). This constructive mathematics is more similar to the practice of computational mathematics than to Intuitionistic mathematics in that its results are consistent classically. Indeed, Bishop's book can be read as a piece of classical analysis or as computational or Intuitionistic mathematics. Nuprl, in fact, arose as an attempt to provide a foundation for computer science — numerical analysis, computer algebra, the theory of algorithms and computability. It was based on programming concepts (Constable [1972], Constable and Zlatin [1984]) and influenced by Algol68 and Simula, but we recognized in 1978 the power of Martin-Löf semantics to organize this activity, and in Constable and Zlatin [1984] used his semantics to improve our earlier design.

As computational mathematics has gained importance, more work has been done to systematize it. For example, the algebra underlying a computer algebra system such as AXIOM (Jenks and Sutor [1992]) is constructive: consider the definition of an integral domain; it provides a function, *div*, which will divide $a \star c$ by $c \neq o$. In general, in computer algebra, to claim that an object "exists" is to give an algorithm to construct it. A current active area of research is expressing the concepts of computer algebra in constructive type theory. It is especially promising that the work provides an orderly account of the types and domains used in algebra systems — for example, compare AXIOM (Jenks and Sutor [1992]) or Weyl (Zippel [1993]) to Jackson's account in Nuprl [1994b,1994a]. Peter Aczel is considering Galois theory in LEGO (Pollack [1995]), and more work of this sort will be done.

Another important topic in the same vein is the use of type theory to organize the foundations of numerical mathematics by Boehm et al. [1986], Chirimar and Howe [1991]. It will be interesting to see whether floating point numbers could be incorporated into a rigorous theory, perhaps even arranging that the notion of a constructive real number as a sequence of approximations each of which was a "floating point" number. It is intriguing to imagine that this work might extend to

a computational treatment of nonstandard analysis (see Nelson [1968], Wattenberg [1988]). This is potentially interesting because it is now realized since the work of Loeb that nonstandard accounts of probability applications can be significantly more intuitive than their classical counterparts.

Category theory can be seen as an abstract organization of type theory, and just as type theory provides an alternative and more general foundation for mathematics than set theory, so too, category theory provides such a foundation. The category called an elementary topos generalizes set theory.[48] It is possible to develop a predicative version of topos theory (the *Grothendieck topos*) in Martin-Löf type theory (Palmgren [1995a]). Likewise, category theory can provide models of type theory (Crole [1993], Seely [1987]). The categorical models allow new kinds of constructive completeness theorems for the Intuitionistic predicate calculus Palmgren [1995a], and from these it is possible to give a uniform computational interpretation to nonstandard analysis (Palmgren [1995a]).

Typed programming languages. The research agenda in programming languages is the most fast-paced of the three; like everything in computer science it is driven by curiosity, by technology, and by market forces. Research is put to use before the "ink is dry." Each small result seems to explode into an industry. Needs for secure mobile code will now be a major influence as code reuse and modularity were before.

Language research depends on a deeper understanding of the *design space* and on a range of semantic tools to rapidly validate experimental designs. Our approach of "partial types" is one of many attempts to provide this knowledge, domain theory, and theories of operational semantics (c.f. Plotkin [1981]) are others (see also Crary [1998]).

Acknowledgments. I want to thank Kate Ricks for preparing this manuscript and Stuart Allen for helpful comments on earlier drafts and for helping with a new account of his 1987 thesis work.

6. Appendix

6.1. Cantor's Theorem. Here is a Nuprl proof of Cantor's theorem from Section 2.9.

```
*T cantor
⊢ ∀A:U. (∃diff:A → A. ∀x:A. ¬(diff x = x))
            ⇒ (∀e:A → A → A. ∃d:A → A. ∀x:A. ¬(e x = d))
|
BY UnivCD THENW Auto
|
1. A: U
```

[48] "The startling aspect of topos theory is that it unifies two seemingly wholly distinct mathematical subjects: on the one hand, topology and algebraic geometry, and on the other hand, logic and set theory." MacLane and Moerdijk [1992,p.1]

```
2. ∃diff:A → A. ∀x:A. ¬(diff x = x)
3. e: A → A → A
⊢ ∃d:A → A. ∀x:A. ¬(e x = d)
|
BY D 2
|
2. diff: A → A
3. ∀x:A. ¬(diff x = x)
4. e: A → A → A
|
BY With ⌈λa.diff (e a a)⌉ (D 0) THENW Auto
|
⊢ ∀x:A. ¬(e x = (λa.diff (e a a)))
|
BY D 0 THENW Auto
|
5. x: A
⊢ ¬(e x = (λa.diff (e a a)))
|
BY D 0 THENW Auto
|
6. e x = (λa.diff (e a a))
⊢ False
|
BY With ⌈e x x⌉ (D 3) THENW Auto
|
3. e: A → A → A
4. x: A
5. e x = (λa.diff (e a a))
6. ¬(diff (e x x) = e x x)
|
BY D 6
|
⊢ diff (e x x) = e x x
|
BY RW (AddrC [3;1] (HypC 5)) 0 THENW Auto
|
⊢ diff (e x x) = (λa.diff (e a a)) x
|
BY Reduce 0 THEN Auto

*C cantor_end          ********************************
```

6.2. Stamps problem. Here is a complete Nuprl proof for a simple arithmetic problem. We show that any number greater than or equal to 8 can be written as a sum of 3's and 5's. We call this the "stamps" problem. When Sam Buss saw this theorem we discussed a generalization which is included in Section 6.3. Christoph

Kreitz proved the generalization following Sam's handwritten notes. It is interesting that Nuprl caught a missing case in this proof. The arguments seem sufficiently self-contained that we present them without further comment.

```
⊢ ∀i:{8...}. ∃m,n:N. 3 * m + 5 * n = i
|
BY D 0 THENA Auto·
|
1. i: {8...}
⊢ ∃m,n:N. 3 * m + 5 * n = i
|
BY NSubsetInd 1
| THEN Auto·
|\
| 1. i: Z
| 2. 0 < i
| 3. 8 = i
| |
1 BY DTerm ⌈1⌉ 0 THENM DTerm ⌈1⌉ 0 THEN Auto·
 \
 1. i: Z
 2. 8 < i
 3. ∃m,n:N. 3 * m + 5 * n = i - 1
 |
 BY D 3 THEN D 4·
 |
 3. m: N
 4. n: N
 5. 3 * m + 5 * n = i - 1
 |
 BY Decide ⌈n > 0⌉ THENA Auto·
 |\
 | 6. n > 0
 | |
 1 BY DTerm ⌈m + 2⌉ 0 THENM DTerm ⌈n - 1⌉ 0 THEN Auto·
  \
  6. ¬(n > 0)
  |
  BY DTerm ⌈m - 3⌉ 0 THENM DTerm ⌈n + 2⌉ 0 THEN Auto·
  |
  ⊢ 0 ≤ m - 3
  |
  BY SupInf THEN Auto
```

6.3. Generalized stamps problem

Lemmata from the Standard Nurpl Library.

```
*T mul_bounds_1a     ⊢ ∀a,b:N. 0 ≤ a*b
*T mul_bounds_1b     ⊢ ∀a,b:N+. 0 < a*b
*T mul_preserves_lt  ⊢ ∀a,b:Z. ∀n:N+. a < b ⇒ n*a < n*b
*T mul_preserves_le  ⊢ ∀a,b:Z. ∀n:N. a ≤ b ⇒ n*a ≤ n*b
*T multiply_functionality_wrt_le
              ⊢ ∀i1,i2,j1,j2:N. i1 ≤ j1 ⇒ i2 ≤ j2 ⇒ i1*i2 ≤ j1*j2

*T rem_bounds_1      ⊢ ∀a:N. ∀n:N+. 0 ≤ a rem n ∧ a rem n < n
*T div_rem_sum       ⊢ ∀a:Z. ∀n:Z-0. a = (a ÷ n)*n + a rem n
*A pm_equal          i = ± j
               == i = j ∨ i = -j

*A divides           b | a
               == ∃c:Z. a = b*c
*T divisor_bound     ⊢ ∀a:N. ∀b:N+. a | b ⇒ a ≤ b
```

Newly Introduced Notions and Lemmata.

```
STAMPS
*T pm_equal_nat      ⊢ ∀a:N+. a = ± 1 ⇒ a = 1
*T divisor_of_sub    ⊢ ∀a,b,c:Z. a | b ⇒ a | c ⇒ a | b - c
*T divisor_of_sub_self ⊢ ∀a,b:Z. a | b ⇒ a | b - a
*A even              a is even == 2 | a
*A odd               a is odd == 2 | a + 1
*T odd_mul           ⊢ ∀b,m:Z. m*b is odd ⇒ b is odd
*T odd_or_even       ⊢ ∀z:Z. z is even ∨ z is odd
*A stampproperty     a and b are useful stamp values
               == ∀i:{a + b...}. ∃n,m:N. i = n*a + m*b
```

Proof of the 'Induction' Step.

```
*T stamp_pre ⊢ ∀a,b:N+. a < b
          ⇒ (∀i:{(a + b)..(2*a + b)-}. ∃n,m:N. i = n*a + m*b)
           ⇒ a and b are useful stamp values
BY    simple_prover
|
1. a: N+
2. b: N+
3. a < b
4. ∀i:{(a + b)..(2*a + b)-}. ∃n,m:N. i = n*a + m*b
5. i: {a + b...}
```

```
⊢ ∃n,m:N. i = n*a + m*b
BY Cases [⌈i < 2*a + b⌉;⌈i ≥ 2*a + b ⌉]....
|\
| 6. i < 2*a + b
1 BY allE 4 ⌈i⌉....
|
 \
 6. i ≥ 2*a + b
 BY     Assert ⌈(a + b) + (i - b) rem a ∈ {(a + b)..(2*a + b)⁻}⌉
 | THENL [Id; allE 4 ⌈(a + b) + (i - b) rem a⌉]
 |\
 | ⊢ (a + b) + (i - b) rem a ∈ {(a + b)..(2*a + b)⁻}
 1 BY FLemmaOn 'rem_bounds_1' [⌈i - b⌉;⌈a⌉] THEN SupInf....
 |
  \
  7. x: (a + b) + (i - b) rem a ∈ {(a + b)..(2*a + b)⁻}
  8. ∃n,m:N. (a + b) + (i - b) rem a = n*a + m*b
  BY    thin 7
  | THEN Repeat existentialE
  | THEN exI ⌈((i - b) ÷ a - 1) + n⌉
  |\
  | 4. i:{(a + b)..(2*a + b)⁻} → (n:N × m:N × (i = n*a + m*b))
  | 6. 2*a + b ≤ i
  | 7. n: N
  | 8. m: N
  | 9. (a + b) + (i - b) rem a = n*a + m*b
  | ⊢ 0 ≤ ((i - b) ÷ a - 1) + n
  1 BY Assert ⌈2*a ≤ i - b⌉
  |   THENL [ SupInfWf; Assert ⌈2 ≤ (i - b) ÷ a⌉ THEN SupInfWf ]
   \
   7. n: N
   8. m: N
   9. (a + b) + (i - b) rem a = n*a + m*b
   ⊢ ∃m:N. i = (((i - b) ÷ a - 1) + n)*a + m*b
   BY exI ⌈m⌉
   |
   ⊢ i = (((i - b) ÷ a - 1) + n)*a + m*b
   BY    SubstInConcl ⌈(((i - b) ÷ a - 1) + n)*a + m*b
                 = (((i - b) ÷ a)*a - a) + n*a + m*b⌉....
     THEN RevHypSubst 9 0 ...
     THEN FLemmaOn 'div_rem_sum' [⌈i - b⌉;⌈a⌉]
     THEN SupInf....
```

Main Theorem.

```
*T StampThm ⊢ ∀a,b:N⁺. a < b
          ⇒ (a and b are useful stamp values
            ⇐⇒  a = 1
              ∨ (a = 2 ∧ b is odd)
              ∨ (a = 3 ∧ b = 4)
              ∨ (a = 3 ∧ b = 5))
```

References

M. ABADI AND L. CARDELLI
[1996] *A Theory of Objects*, Springer-Verlag, Berlin.

S. ABRAMSKY
[1993] Computational interpretations of linear logic, *Theoretical Computer Science*, 111, pp. 3–57.

P. H. G. ACZEL
[1977] An introduction to inductive definitions, in: *Handbook of Mathematical Logic*, J. Barwise, ed., North-Holland, Amsterdam, pp. 739–782.
[1986] The type theoretic interpretation of constructive set theory, in: *Logic, Methodology and Philosophy of Science VII*, R. B. Marcus, G. J. W. Dorn, and P. Weingartner, eds., Elsevier Science Publishers, Amsterdam, pp. 17–49.

S. F. ALLEN
[1987a] A non-type-theoretic definition of Martin-Löf's types, in: *Proceedings of the Second Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, D.C., pp. 215–224.
[1987b] *A non-type-theoretic semantics for type-theoretic language*, PhD thesis, Cornell University.

S. F. ALLEN, R. L. CONSTABLE, D. J. HOWE, AND W. AITKEN
[1990] The semantics of reflected proof, in: *Proceedings of the Fifth Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, D.C., pp. 95–197.

R. C. BACKHOUSE
[1989] Constructive type theory–an introduction, in: *Constructive Methods in Computer Science, NATO ASI Series, Vol. F55: Computer & System Sciences*, M. Broy, ed., Springer-Verlag, Berlin, pp. 6–92.

H. P. BARENDREGT
[1977] The typed lambda calculus, in: *Handbook of Mathematical Logic*, J. Barwise, ed., North-Holland, Amsterdam, pp. 1091–1132.
[1981] *The Lambda Calculus: Its Syntax and Symantics*, North-Holland, Amsterdam.

J. BARWISE AND J. ETCHEMENDY
[1991] *The Language of First-Order Logic*, Lecture Notes Number 23, Center for the Study of Language and Information, Stanford University, second ed.

D. BASIN AND R. L. CONSTABLE
[1993] Metalogical frameworks, in: *Logical Environments*, G. Huet and G. Plotkin, eds., Cambridge University Press, ch. 1, pp. 1–29.

D. BASIN AND M. KAUFMANN
[1991] The Boyer-Moore prover and Nuprl: An experimental comparison, in: *Logical Frameworks*, G. Huet and G. Plotkin, eds., Cambridge University Press, pp. 89–119.

J. L. BATES
[1979] *A Logic for Correct Program Development*, PhD thesis, Cornell University.

J. L. BATES AND R. L. CONSTABLE
[1985] Proofs as programs, *ACM Transactions on Programming Languages and Systems*, 7, pp. 53–71.

M. J. BEESON
[1985] *Foundations of Constructive Mathematics*, Springer-Verlag, Berlin.

J. L. BELL
[1988] *Toposes and Local Set Theories*, Oxford Logic Guides #14, Oxford University Press.

U. BERGER AND H. SCHWICHTENBERG
[1996] The greatest common divisor: a case study for program extraction from classical proofs, in: *Types for Proofs and Programs: International Workshop TYPES'95*, S. Berardi and M. Coppo, eds., Lecture Notes in Computer Science #1158, Springer Verlag, Berlin, pp. 36–46.

P. BERNAYS
[1958] *Axiomatic Set Theory*, North-Holland, Amsterdam. With an introduction by A.A. Fraenkel

E. BISHOP
[1967] *Foundations of Constructive Analysis*, McGraw Hill, New York.

E. BISHOP AND D. BRIDGES
[1985] *Constructive Analysis*, Springer-Verlag, Berlin.

N. BJØRNER, A. BROWN, E. CHANG, M. COLÓN, A. KAPUR, Z. MANNA, H. B. SIPMA, AND T. E. URIBE
[1996] STep: Deductive-algorithmic verification of reactive and real-time systems, in: *CAV'96: Proceedings of the Eighth International Conference on Computer Aided Verification*, R. Alur and T. A. Henzinger, eds., Lecture Notes in Computer Science #1102, Springer-Verlag, Berlin, pp. 415–418.

W. W. BLEDSOE
[1975] A new method for proving certain Presburger formulas, in: *Proceedings of the Fourth International Joint Conference on Artificial Intelligence, held in Tbilisi, Georgia, USSR.*

H.-J. BOEHM, R. CARTWRIGHT, M. RIGGLE, AND M. J. O'DONNELL
[1986] Exact real arithmetic: A case study in higher order programming, in: *ACM Symposium on LISP and Functional Programming*, Association for Computing Machinery, New York.

N. BOURBAKI
[1968a] *Elements of Mathematics, Algebra, Volume 1*, Addison-Wesley, Reading, Massachusetts.
[1968b] *Elements of Mathematics, Theory of Sets*, Addison-Wesley, Reading, Massachusetts.

R. S. BOYER AND J. S. MOORE
[1979] *A Computational Logic*, Academic Press, New York.
[1988] Integrating recursive procedures into hueristic theorem provers: A case study in linear arithmetic, in: *Machine Intelligence II*, Oxford University Press.

L. E. J. BROUWER
[1975] *Collected Works*, vol. 1, North-Holland, Amsterdam. Edited by A. Heyting.

N. G. DE BRUIJN
[1970] The mathematical language Automath, its usage and some of its extensions, in: *Symposium on Automatic Demonstration*, M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, eds., Lecture Notes in Mathematics #125, Springer-Verlag, Berlin, pp. 29–61.
[1972] Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, *Indagationes Mathematicae*, 34, pp. 381–392.
[1980] A survey of the project Automath., in: *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley, eds., Academic Press, New York, pp. 589–606.

A. BUNDY

[1991] The use of proof plans for normalization, in: *Essays in Honor of Woody Bledsoe*, R. S. Boyer, ed., Kluwer Academic Publishers, Dordrecht, Boston, pp. 149–166.

J. R. BURCH, E. CLARKE, K. L. MCMILLAN, D. L. DILL, AND L. J. HWANG

[1991] Symbolic model checking: 10^{20} states and beyond, in: *Proceedings of the Fifth Symposium on Logic in Computer Science*, IEEE Computer Society, Los Alamitos, California, pp. 428–439.

S. R. BUSS

[1986] The polynomial hierarchy and intuitionistic bounded arithmetic, in: *Structure in Complexity Theory*, A. L. Selman, ed., Lecture Notes in Computer Science #223, Springer-Verlag, Berlin, pp. 77–103.

L. CARDELLI

[1994] Extensible records in a pure calculus of subtyping, in: *Theoretical Aspects of object-oriented Programming: Types, Semantics and Language Design*, C. A. Gunter and J. C. Mitchell, eds., MIT Press, Cambridge, Massachusetts.

T. CHAN

[1982] An algorithm for checking PL/CV arithmetic inferences, in: *An Introduction to the PL/CV2 Programming Logic*, R. L. Constable, S. D. Johnson, and C. D. Eichenlaub, eds., Lecture Notes in Computer Science #135, Springer-Verlag, Berlin, pp. 227–264.

C. C. CHANG AND H. J. KEISLER

[1990] *Model Theory*, vol. 73 of Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, 3rd ed.

L. P. CHEW, R. L. CONSTABLE, S. VAVASIS, K. PINGALI, AND R. ZIPPEL

[1996] *Collaborative Mathematics Environments*. www@cs.cornell.edu/Info/Projects/Nuprl.

J. CHIRIMAR AND D. J. HOWE

[1991] Implementing constructive real analysis: a preliminary report, in: *Symposium on Constructivity in Computer Science*, J. P. Myers and M. J. O'Donnell, eds., Lecture Notes in Computer Science #613, Springer-Verlag, Berlin, pp. 165–178.

N. CHOMSKY

[1988] *Language and Problems of Knowledge: Managua Lectures*, MIT Press, Cambridge, Massachusetts.

A. CHURCH

[1940] A formulation of the simple theory of types, *Journal of Symbolic Logic*, 5, pp. 55–68.

[1956] *Introduction to Mathematical Logic, Vol I*, Princeton University Press.

[1960] Application of recursive arithmetic to the problem of circuit synthesis,, in: *Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University 1957*, Institute for Defense Analyses, Princeton, pp. 3–50.

E. CLARKE, D. E. LONG, AND K. L. MCMILLAN

[1989] Compositional model checking, in: *Proceedings of the Fourth Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, D.C.

R. L. CONSTABLE

[1972] Constructive mathematics and automatic program writers, in: *Proceedings of Information Processing 71 (IFIP)*, C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, eds., North-Holland, Amsterdam, pp. 229–233.

[1989] Assigning meaning to proofs: a semantic basis for problem solving environments, in: *Constructive Methods in Computer Science, NATO ASI Series, Vol. F55: Computer & System Sciences*, M. Broy, ed., Springer-Verlag, Berlin, pp. 63–91.

[1991] Type theory as a foundation for computer science, in: *Theoretical Aspects of Computer Software, International Conference TACS '91*, T. Ito and A. R. Meyer, eds., Lecture Notes in Computer Science #526, Springer-Verlag, Berlin, pp. 226–243.

R. L. CONSTABLE, S. F. ALLEN, H. M. BROMLEY, W. R. CLEAVELAND, J. F. CREMER, R. HARPER, D. J. HOWE, T. B. KNOBLOCK, N. P. MENDLER, P. PANANGADEN, J. T. SASAKI, AND S. F. SMITH

[1986] *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall, Englewood Cliffs, New Jersey.

R. L. CONSTABLE, P. B. JACKSON, P. NAUMOV, AND J. URIBE

[1998] Constructively formalizing automata theory I: Finite automata, in: *Robin Milner Festschrift*, Springer-Verlag, Berlin. To appear.

R. L. CONSTABLE, S. D. JOHNSON, AND C. D. EICHENLAUB

[1982] Introduction to the PL/CV2 progamming logic, in: *An Introduction to the PL/CV2 Programming Logic*, R. L. Constable, S. D. Johnson, and C. D. Eichenlaub, eds., Lecture Notes in Computer Science #135, Springer-Verlag, Berlin, pp. 165–178.

R. L. CONSTABLE AND N. P. MENDLER

[1985] Recursive Definitions in Type Theory, in: *Logics of Programs*, R. Parikh, ed., Lecture Notes in Computer Science #193, Springer-Verlag, Berlin, pp. 61–78.

R. L. CONSTABLE AND S. F. SMITH

[1993] Computational foundations of basic recursive function theory, *Theoretical Computer Science*, 121, pp. 89–112.

R. L. CONSTABLE AND D. R. ZLATIN

[1984] The type theory of PL/CV3, *ACM Transactions on Programming Languages and Systems*, 6, pp. 94–117.

T. COQUAND

[1990] Metamathematical investigations of a calculus of constructions, in: *Logic and Computer Science*, P. Odifreddi, ed., Academic Press, London, pp. 91–122.

T. COQUAND AND G. HUET

[1988] The Calculus of Constructions, *Information and Computation*, 76, pp. 95–120.

T. COQUAND AND C. PAULIN-MOHRING

[1990] Inductively defined types, preliminary version, in: *COLOG '88, International Conference on Computer Logic*, P. Martin-Löf and G. E. Mints, eds., Lecture Notes in Computer Science #417, Springer-Verlag, Berlin, pp. 50–66.

K. CRARY

[1998] *Programming Language Semantics in Foundational Type Theory*. To appear in *IFIP Working Conference on Programming Concepts and Methods, New York, June 1998*.

R. L. CROLE

[1993] *Categories for Types*, Cambridge University Press, Cambridge.

H. B. CURRY AND R. FEYS

[1958] *Combinatory Logic, Volume I*, North-Holland, Amsterdam.

O.-J. DAHL, E. W. DIJKSTRA, AND C. A. R. HOARE

[1972] *Structured Programming*, Academic Press, London, New York.

A. J. DEMERS AND J. DONAHUE

[1980] Type-completeness as a language principle., in: *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, New York, pp. 234–244.

E. W. DIJKSTRA
[1968] A constructive approach to the problem of software correctness, *BIT*, 8, pp. 174–186.

M. A. E. DUMMETT
[1981] *Frege: Philosophy of Mathematics*, Harvard University Press, Cambridge and Duckworth, London, 2nd ed.

P. DYBJER
[1994] Inductive families, *Formal Aspects of Computing*, 6, pp. 1–26.

A. EDALAT
[1994] Domain theory and integration, in: *Proceedings of the Ninth Annual Symposium on Logic in Computer Science*, IEEE Computer Society, Los Alamitos, California, pp. 115–124.

H. EGLI AND R. L. CONSTABLE
[1976] Computability concepts for programming language semantics, *Theoretical Computer Science*, 2, pp. 133–145.

W. M. FARMER
[1990] A partial functions version of Church's simple theory of types, *Journal of Symbolic Logic*, 55, pp. 1269–1291.

W. M. FARMER, J. D. GUTTMAN, AND F. J. THAYER
[1991] *IMPS: an interactive mathematical proof system*, Tech. Rep. M90-19, The MITRE Corp., Bedford, Massachusetts.

S. FEFERMAN
[1970] Formal theories for transfinite iterations of generalized inductive definitions and some subsystems of analysis, in: *Intuitionism and Proof Theory: Proceedings of the Summer Conference at Buffalo N.Y. 1968*, A. Kino, J. Myhill, and R. E. Vesley, eds., North-Holland, Amsterdam, pp. 303–326.
[1975] A language and axioms for explicit mathematics, in: *Algebra and Logic*, J. N. Crossley, ed., Lecture Notes in Mathematics #450, Springer-Verlag, Berlin, pp. 87–139.

M. B. FORESTER
[1993] *Formalizing Constructive Real Analysis*, Tech. Rep. TR93-1382, Computer Science Dept., Cornell University, Ithaca, New York.

G. FREGE
[1879] *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*, Halle. English translation in van Heijenoort [1967], pp.1-82.
[1903] *Grundgesetze der Arithmetik, Begriffsschriftlich Abgeleitet*, Pohle, Jena. Reprinted 1962, Olms, Hildesheim.

H. M. FRIEDMAN
[1978] Classically and intuitionistically provably recursive functions, in: *Higher Set Theory*, D. S. Scott and G. H. Muller, eds., Lecture Notes in Mathematics #669, Springer-Verlag, Berlin, pp. 21–28.

H. M. FRIEDMAN AND A. SCEDROV
[1983] Set existence property for intuitionistic theories with countable choice, *Annals of Pure and Applied Logic*, 25, pp. 129–140.

J. H. GALLIER
[1993] Constructive logics. Part I: A tutorial on proof systems and typed λ-calculi, *Theoretical Computer Science*, 110, pp. 249–339.

G. GENTZEN
[1935] Untersuchungen über das logische Schliessen, *Mathematische Zeitschrift*, 39, pp. 176–210, 405–431. English translation in: Gentzen [1969], pp.68-131.

[1969] *Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam. Edited by M. E. Szabo.

J.-Y. GIRARD

[1987] *Proof Theory and Logical Complexity, Volume 1*, Bibliopolis, Napoli.

[1991] A new constructive logic: classical logic, *Mathematical Structures in Computer Science*, 1, pp. 255–296.

J.-Y. GIRARD, P. TAYLOR, AND Y. LAFONT

[1989] *Proofs and Types*, Cambridge Tracts in Computer Science, Vol. 7, Cambridge University Press.

K. GÖDEL

[1931] Über formal unentscheidbar Sätze der *principa mathematica* und verwandter Systeme i, *Monatshefte für Mathematik und Physik*, 38, pp. 173–198. Reprinted and translated in Gödel [1986,pp.144-195].

[1933] Zur intuitionistisched Arithmetic und Zahlentheorie, *Ergebnisse eines Mathematischen Kolloquiums*, 4, pp. 34–38. Reprinted and translated in Gödel [1986,pp.282-295].

[1986] *Collected Works, Volume 1*, Oxford University Press. Edited by S. Feferman, J. W. Dawson., S. C. Kleene, G. H. Moore, R. M. Solovay, J. van Heijenoort.

D. GOOD

[1985] Mechanical proofs about computer programs, in: *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, eds., Prentice-Hall, Englewood Cliffs, New Jersey, pp. 55–75.

M. GORDON AND T. MELHAM

[1993] *Introduction to HOL*, University Press, Cambridge.

M. GORDON, R. MILNER, AND C. WADSWORTH

[1979] *Edinburgh LCF: a mechanized logic of computation*, Lecture Notes in Computer Science #78, Springer-Verlag, Berlin.

T. G. GRIFFIN

[1988a] Notational definition - a formal account, in: *Proceedings of the Third Annual Symposium on Logic in Computer Science*, IEEE Computer Society, Los Alamitos, California, pp. 372–383.

[1988b] *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems*, PhD thesis, Cornell University.

C. A. GUNTER

[1992] *Semantics of Programming Languages: Structures and Techniques*, Foundations of Computing Series, MIT Press, Cambridge, Massachusetts.

E. GUNTER

[1994] *Studying the ML Module System in HOL90.* Presented at Logic & Computation Seminar, Univ. of Pennsylvania, December.

J. GUTTAG, J. HORNING, AND J. M. WING

[1985] *Larch in five easy pieces*, Tech. Rep. 5, Digital Equipment Corporation, Systems Research Center, July.

R. HARPER

[1992] Constructing type systems over an operational semantics, *Journal of Symbolic Computing*, 14, pp. 71–84.

R. HARPER, F. HONSELL, AND G. PLOTKIN

[1993] A framework for defining logics, *Journal of the Association for Computing Machinery*, 40, pp. 143–184.

J. HARTMANIS AND R. STEARNS

[1965] On the computational complexity of algorithms, *Transactions of the American Mathematics Society*, 117, pp. 285–306.

J. VAN HEIJENOORT

[1967] *From Frege to Gödel: A sourcebook in mathematical logic, 1879-1931*, Harvard University Press.

L. HELMINK

[1992] *Tools for Proofs and Programs*, PhD thesis, Universiteit van Amsterdam, The Netherlands.

T. A. HENZINGER AND P.-H. HO

[1994] Model checking strategies for hybrid systems, in: *Seventh International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, F. D. Anger, R. V. Rodriguez, and M. Ali, eds., Gordon and Breach, Langhorne, Pennsylvania.

J. J. HICKEY

[1996a] *Formal Abstract Data Types*. unpublished manuscript.

[1996b] Objects and theories as very dependent types, in: *Proceedings of FOOL 3*, July.

[1997] Nuprl-Light: An implementation framework for higher-order logics, in: *CADE-14: 14th International Conference on Automated Deduction*, W. McCune, ed., Lecture Notes in Computer Science #1249, Springer-Verlag, Berlin, pp. 395–399.

D. HILBERT

[1926] Über das Unendliche, *Mathematische Annalen*, 95, pp. 161–190.

J. R. HINDLEY, B. LERCHER, AND J. P. SELDIN

[1972] *Introduction to Combinatory Logic*, Cambridge University Press, London.

C. A. R. HOARE

[1972] Notes on data structuring, in: *Structured Programming*, Academic Press, New York.

J. E. HOPCROFT AND J. D. ULLMAN

[1969] *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Massachusetts.

D. J. HOWE

[1987] The computational behaviour of Girard's paradox, in: *Proceedings of the Second Annual Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, D.C., pp. 205–214.

[1989] Equality in lazy computation systems, in: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, D.C., pp. 198–203.

[1991] On computational open-endedness in Martin-Löf's type theory, in: *Proceedings of the Sixth Symposium on Logic in Computer Sciience*, IEEE Computer Society, Los Alamitos, California, pp. 162–172.

[1996a] Importing mathematics from HOL into Nuprl, in: *Proceedings of the Ninth International Conference on Theorem Proving in Higher Order Logics*, J. von Wright, J. Grundy, and J. Harrison, eds., Lecture Notes in Computer Science #1125, Springer-Verlag, Berlin, pp. 267–281.

[1996b] Semantic foundations for embedding HOL in Nuprl, in: *Proceedings the Fifth International Conference on Algebraic Methodology and Software Technology (AMAST'96)*, M. Wirsing and M. Nivat, eds., Lecture Notes in Computer Science #1101, Springer-Verlag, Berlin, pp. 85–101.

G. HUET AND B. LANG

[1978] Proving and applying program transformations expressed with second-order patterns, *Acta Informatica*, 11, pp. 31–55.

J. M. E. HYLAND

[1982] The effective topos, in: *The Brouwer Centenary Symposium*, A. S. Troelstra and D. van Dalen, eds., North-Holland, Amsterdam, pp. 165–216.

S. IGARASHI, R. LONDON, AND D. LUCKHAM

[1975] Automatic program verification I: a logical basis and its implementation., *Acta Informatica*, 4, pp. 145–82.

P. B. JACKSON

[1994a] *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*, PhD thesis, Cornell University, August. Forthcoming.

[1994b] Exploring abstract algebra in constructive type theory, in: *CADE-12: International Conference on Automated Deduction*, A. Bundy, ed., Lecture Notes in Computer Science #814 (Lecture Notes in Artificial Intelligence), Springer-Verlag, Berlin, June, pp. 590–604.

[1994c] *The Nuprl Proof Development System, Version 4.1 Reference Manual and User's Guide*, Cornell University, Ithaca, New York, February.

R. D. JENKS AND R. S. SUTOR

[1992] *Axiom: The Scientific Computation System*, Springer-Verlag, Berlin.

A. JOYAL AND I. MOERDIJK

[1995] *Algebraic Set Theory*, Cambridge University Press, Cambridge.

A. KENNY

[1995] *Frege*, Penguin Books, London.

D. E. KNUTH

[1984] *The TEXbook*, Addison-Wesley, Reading, Massachusetts.

D. KOZEN

[1977] *Complexity of Finitely Presented Algebras*, PhD thesis, Computer Science Department, Cornell University, Ithaca, New York.

D. KOZEN AND J. TIURYN

[1990] Logics of programs, in: *Handbook of Theoretical Computer Science, Volume B*, J. van Leeuwen, ed., North Holland, Amsterdam, pp. 789–840.

G. KREISEL

[1959] Interpretation of analysis by means of constructive functionals of finite type, in: *Constructivity in Mathematics: Proceedings of the Colloquium held at Amsterdam, 1957*, A. Heyting, ed., North-Holland, Amsterdam, pp. 101–128.

[1981] Neglected possibilities of processing assertions and proofs mechanically: choice of problems and data, in: *University-Level Computer-Assisted Instruction at Stanford: 1968-1980*, P. Suppes, ed., Institute for Mathematical Studies in the Social Sciences, Stanford University, pp. 131–147.

C. KREITZ

[n.d.] Program synthesis, in: *Automated Deduction – A Basis for Applications*, W. Bibel and P. Schmitt, eds., Kluwer Academic Publishers, Dordrecht, Boston, ch. III.2.10. To appear.

C. KREITZ, M. HAYDEN, AND J. J. HICKEY

[n.d.] *A Proof Environment for the Development of Group Communication Systems.* to appear.

S. KRIPKE

[1965] Semantical analysis of intuitionistic logic I, in: *Formal Systems and Recursive Functions*, J. N. Crossley and M. A. E. Dummett, eds., North-Holland, Amsterdam, pp. 92–130.

J. LAMBEK AND P. J. SCOTT

[1986] *Introduction to Higher-Order Categorical Logic*, vol. 7 of Cambridge Studies in Advanced Mathematics, Cambridge University Press, Cambridge, UK.

H. LÄUCHLI

[1970] An abstract notion of realizability for which intuitionistic predicate calculus is complete, in: *Intuitionism and Proof Theory: Proceedings of the Summer Conference at Buffalo N.Y. 1968*, A. Kino, J. Myhill, and R. E. Vesley, eds., North-Holland, Amsterdam, pp. 227–34.

D. LEIVANT

[1994a] A foundational delineation of poly-time, *Information and Computation*, 110, pp. 391–420.

[1994b] Predicative recurrence in finite type, in: *Logical Foundations of Computer Science*, A. Nerode and Yu. V. Matijacevič, eds., Lecture Notes in Computer Science #813, Springer-Verlag, Berlin, pp. 227–239.

[1995] Ramified recurrence and computational complexity I: Word recurrence and polynomial time, in: *Feasible Mathematics II, Perspectives in Computer Science*, P. Clote and J. B. Remmel, eds., Birkhäuser, Boston.

B. LISKOV AND J. GUTTAG

[1986] *Abstraction and Specification in Program Development*, MIT Press, Cambridge, Massachusetts.

C. L. LIU

[1985] *Elements of Discrete Mathematics*, McGraw-Hill, New York, 2nd ed.

Z. LUO

[1994] *Computation and Reasoning, A Type Theory for Computer Science*, Oxford University Press.

D. MACKENZIE

[1995] The automation of proof: A historical and sociological exploration, *IEEE Annals of the History of Computing*, 17, pp. 7–29.

S. MACLANE AND I. MOERDIJK

[1992] *Sheaves in Geometry and Logic, a First Introduction to Topos Theory*, Springer-Verlag, Berlin.

A. A. MARKOV

[1949] On the representation of recursive functions (Russian), *Izvestiya Akad. Nauk SSSR. Ser. Mat.*, 13, pp. 417–424. English translation: American Mathematical Society Translation 54 (1950) 13 pp.

P. MARTIN-LÖF

[1982] Constructive mathematics and computer programming., in: *Proceedings of the Sixth International Congress on Logic, Methodology, and Philosophy of Science*, L. J. Cohen, J. Łoś, H. Pfeiffer, and K.-P. Podewski, eds., North-Holland, Amsterdam, pp. 153–75.

[1983] *On the Meaning of the Logical Constants and the Justification of the Logical Laws.* Lectures in Siena.

P. MARTIN-LÖF

[1984] *Intuitionistic Type Theory, Studies in Proof Theory, Lecture Notes*, Bibliopolis, Napoli.

D. A. MCALLESTER

[1989] *ONTIC: A Knowledge Representation System for Mathematics*, MIT Press, Cambridge, Massachusetts.

J. McCarthy
[1963] A basis for a mathematical theory of computation, in: *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, eds., North-Holland, Amsterdam, pp. 33–70.

J. McCarthy et al.
[1962] *Lisp 1.5 Users Manual*, Mit Press, Cambridge, Massachusetts.

N. P. Mendler, R. L. Constable, and P. Panangaden
[1986] Infinite Objects in Type Theory, in: *Proceedings of the First Annual Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, D.C., pp. 249–257.

P. F. Mendler
[1988] *Inductive Definition in Type Theory*, PhD thesis, Cornell University, Ithaca, New York.

A. R. Meyer and M. B. Reinhold
[1986] Type is not a type, in: *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, New York, pp. 287–295.

B. Meyer
[1988] *Object-oriented software construction*, Prentice-Hall, Englewood Cliffs, New Jersey.

R. Milner, M. Tofte, and R. Harper
[1991] *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts.

R. Mines, F. Richman, and W. Ruitenburg
[1988] *A Course in Constructive Algebra*, Springer-Verlag, Berlin.

J. C. Mitchell
[1996] *Foundations for Programming Languages*, MIT Press, Cambridge, Massachusetts.

I. Moerdijk and G. E. Reyes
[1991] *Models for Smooth Infinitesimal Analysis*, Springer-Verlag, Berlin.

C. Murthy
[1990] *Extracting Constructive Content for Classical Proofs*, PhD thesis, Cornell University, Dept. of Computer Science. (TR 89-1151).
[1991] An evaluation semantics for classical proofs, in: *Proceedings of the Sixth Symposium on Logic in Computer Science*, IEEE Computer Society, Los Alamitos, California, pp. 96–109.
[1992] A computational analysis of Girard's translation and LC, in: *Proceedings of the Seventh Symposium on Logic in Computer Science*, IEEE Computer Society, Los Alamitos, California, pp. 90–101.

R. P. Nederpelt, J. H. Geuvers, and R. C. D. Vrijer
[1994] eds., *Selected Papers in Automath*, Studies in Logic and the Foundations of Mathematics #133, Elsevier Science Publishers, Amsterdam.

G. Nelson and D. Oppen
[1979] Simplification by cooperating decision procedures, *ACM Transactions on Programming Languages and Systems*, 1, pp. 245–257.

R. J. Nelson
[1968] *Introduction to Automata*, John Wiley & Sons, Inc., New York.

A. Nerode and R. A. Shore
[1994] *Logic for Applications*, Springer-Verlag, Berlin.

B. Nordstrom, K. Petersson, and J. M. Smith
[1990] *Programming in Martin-Löf's Type Theory*, Clarendon Press, Oxford University Press.

J. O'Leary, M. Leeser, J. J. Hickey, and M. Aagaard

[1995] Non-restoring integer square root: A case study in design by principled optimization, in: *Theorem Provers in Circuit Design: Second International Conference (TPCD '94)*, T. Kropf and R. Kumar, eds., Lecture Notes in Computer Science #901, Springer-Verlag, Berlin, pp. 52–71.

S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas

[1996] PVS: Combining specification, proof checking, and model checking, in: *CAV'96: Proceedings of the Eighth International Conference on Computer Aided Verification*, R. Alur and T. A. Henzinger, eds., Lecture Notes in Computer Science #1102, Springer-Verlag, Berlin, pp. 411–414.

S. Owre, J. M. Rushby, and N. Shankar

[1992] PVS: A prototype verification system, in: *CADE-11: International Conference on Automated Deduction*, D. Kapur, ed., Lecture Notes in Computer Science #607 (Lecture Notes in Artificial Intelligence), Springer-Verlag, Berlin, pp. 748–752.

E. Palmgren

[1991] *On Fixed Point Operators, Inductive Definitions and Universes in Martin-Löf's Type Theory*, PhD thesis, Uppsala University, March.

[1995a] The Friedman translation for Martin-Löf's type theory, *Mathematical Logic Quarterly*, 41, pp. 314–326.

[1995b] *A sheaf-theoretic foundation for nonstandard analysis*, Tech. Rep. 1995:43, Department of Mathematics, Uppsala University.

C. Paulin-Mohring

[1989] Extracting $F'_w s$ programs from proofs in the calculus of constructions, in: *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, New York, pp. 89–104.

C. Paulin-Mohring and B. Werner

[1993] Synthesis of ML programs in the system Coq, *Journal of Symbolic Computation*, 15, pp. 607–640.

L. C. Paulson

[1993] Set theory for verification: I from foundations to functions, *Journal of Automated Reasoning*, 11, pp. 353–389.

[1994] *Isabelle: A Generic Theorem Prover*, Lecture Notes in Computer Science #828, Springer-Verlag, Berlin.

A. M. Pitts

[1987] Polymorphism is set-theoretic, constructively, in: *Category Theory and Computer Science*, D. H. Pitt, A. Poigne, and D. E. Rydeheard, eds., Lecture Notes in Computer Science #283, Springer-Verlag, Berlin, pp. 12–39.

G. Plotkin

[1975] Call-by-name, call-by-value, and the λ-calculus, *Theoretical Computer Science*, pp. 125–59.

[1977] LCF considered as a programming language, *Theoretical Computer Science*, 5, pp. 223–255.

[1981] *A Structural Approach to Operational Semantics*, Tech. Rep. DAIMI-FN-19, Computer Science Department, Aarhus University.

E. Poll

[1994] *A Programming Logic Based on Type Theory*, PhD thesis, Technische Universiteit Eindhoven.

R. Pollack

[1995] *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*, PhD thesis, Department of Computer Science, University of Edinburgh, April.

D. PRAWITZ
[1965] *Natural Deduction*, Almquist and Wiksell, Stockholm.

W. QUINE
[1960] *Word and Object*, MIT Press, Cambridge, Massachusetts.

T. REPS
[1982] *Generating Language-Based Environments*, PhD thesis, Cornell University, Ithaca, New York.

T. REPS AND T. TEITELBAUM
[1988] *The Synthesizer Generator Reference Manual*, Springer-Verlag, Berlin, third ed.

J. C. REYNOLDS
[1974] Towards a theory of type structure., in: *Programming Symposium*, B. Robinet, ed., Lecture Notes in Computer Science #19, Springer-Verlag, Berlin, pp. 408–425.
[1981] The essence of Algol, in: *International Symposium on Algorithmic Languages*, J. de Bakker and J. van Vliet, eds., North-Holland, Amsterdam, pp. 345–372.

A. REZUS
[1985] *Semantics of Constructive Type Theory*, Tech. Rep. 70, Informatics Dept., Nijmegen University, September.

H. ROGERS
[1967] *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York.

B. RUSSELL
[1908] *The Principles of Mathematics*, Cambridge University Press, Cambridge.

D. S. SCOTT
[1970a] Constructive validity, in: *Symposium on Automatic Demonstration*, M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, eds., Lecture Notes in Mathematics #125, Springer-Verlag, Berlin, pp. 237–275.
[1970b] Outline of a mathematical theory of computation, in: *Proc. 4th Annual Princeton Conf. on Information Sciences & Systems*, Princeton, New Jersey, pp. 169–176.
[1972] Lattice theoretic models for various type-free calculi., in: *Proceedings of the Fourth International Congress on Logic and Methodology and Philosophy of Science*, R. J. Bogdan and I. Niiniluoto, eds., D. Reidel, Dordrecht, pp. 157–87.
[1976] Data types as lattices., *SIAM Journal on Computing*, 5, pp. 522–87.

R. A. G. SEELY
[1987] Categorical semantics for higher order polymorphic lambda calculus, *Journal of Symbolic Logic*, 52, pp. 969–989.

A. SETZER
[1993] *Proof theoretical strength of Martin-Löf Type Theory with W-type and one universe*, PhD thesis, Ludwig-Maximilians-Universität, München, September.

N. SHANKAR
[1994] *Metamathematics, Machines, and Gödel's Proof*, Cambridge University Press, Cambridge, Massachusetts.

R. E. SHOSTAK
[1979] A practical decision procedure for arithmetic with function symbols, *Journal of the Association for Computing Machinery*, 26, pp. 351–360.

J. SIEKMANN AND G. WRIGHTSON
[1983] *Automation of Reasoning*, vol. 1, Springer-Verlag, Berlin.

B. C. SMITH
[1984] Reflection and semantics in LISP, in: *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, Washington, D.C., pp. 23–35.

J. M. SPIVEY
[1989] *The Z Notation*, Prentice-Hall, Englewood Cliffs, New Jersey.

S. STENLUND
[1972] *Combinators, λ-Terms, and Proof Theory.*, D. Reidel, Dordrechte.

W. P. VAN STIGT
[1990] *Brouwer's Intuitionism*, North-Holland, Amsterdam.

W. W. TAIT
[1967] Intensional interpretation of functionals of finite type, *Journal of Symbolic Logic*, 32, pp. 189–212.
[1983] Against intuitionism: Constructive mathematics is part of classical mathematics, *Journal of Philosophical Logic*, 12, pp. 173–195.

A. TARSKI
[1956] The concept of truth in formalized languages, Clarendon Press, Oxford, pp. 152–278. Translation of 1933 paper in Polish.

S. THOMPSON
[1991] *Type Theory and Functional Programming*, Addison-Wesley, Reading, Massachusetts.

A. S. TROELSTRA
[1973] *Metamathematical Investigation of Intuitionistic Mathematics*, Lecture Notes in Mathematics #344, Springer-Verlag, Berlin.

A. S. TROELSTRA AND D. VAN DALEN
[1988] *Constructivism in Mathematics, An Introduction, Vol. I,II*, North-Holland, Amsterdam.

A. S. TROELSTRA AND H. SCHWICHTENBERG
[1996] *Basic Proof Theory*, Cambridge University Press.

A. TRYBULEC
[1983] On a system of computer-aided instruction of logic, *Bulletin of the Section of Logic*, 12, pp. 214–220. Polish Academy of Science.

A. M. UNGAR
[1992] *Normalization, Cut-elimination and the theory of proofs*, Lecture Notes 28, CLSI, Stanford.

F. WATTENBERG
[1988] Nonstandard analysis and constructivism, *Studia Logica*, 47, pp. 303–309.

R. WEYRAUCH
[1980] Prolegomena to a theory of formal reasoning, *Artificial Intelligence*, 13, pp. 133–170.

A. N. WHITEHEAD AND B. RUSSELL
[1925-27] *Principia Mathematica*, vol. 1, 2, 3, Cambridge University Press, 2nd ed.

L. WITTGENSTEIN
[1922] *Tractatus Logico-Philosophicus*, Routledge & Kegan Paul, London.
[1953] *Philosophical Investigations*, Basil Blackwell, Oxford.

S. WOLFRAM
[1988] *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Reading, Massachusetts.

L. Wos, R. Overbeek, E. Lusk, and J. Boyle
[1984] *Automated Reasoning*, Prentice–Hall, Englewood Cliffs, New Jersey.

R. Zippel
[1993] *Effective Polynomial Computation*, Kluwer Academic Publishers, Dordrecht, Boston.

J. I. Zucker
[1974] The correspondence between cut-elimination and normalization, *Annals of Mathematical Logic*, 7, pp. 1–112. Errata *ibid* 7, p.156.
[1977] Formalization of classical mathematics in Automath, in: *Colloque International de Logique*, Colloques Internationaux du Centre National de la Recherche Scientifique, CNRS, Paris, pp. 135–145.