Innovations in Computational Type Theory using Nuprl

S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, E. Moran

Department of Computer Science, Cornell-University, Ithaca, NY 14853-7501
{sfa,markb,rc,eaton,kreitz,lolorigo,evan}@cs.cornell.edu

Abstract

For twenty years the Nuprl ("new pearl") system has been used to develop software systems and formal theories of computational mathematics. It has also been used to explore and implement computational type theory (CTT) – a formal theory of computation closely related to Martin-Löf's intuitionistic type theory (ITT) and to the calculus of inductive constructions (CIC) implemented in the Coq prover.

This article focuses on the theory and practice underpinning our use of Nuprl for much of the last decade. We discuss innovative elements of type theory, including new type constructors such as unions and dependent intersections, our theory of classes, and our theory of event structures.

We also discuss the innovative architecture of Nuprl as a distributed system and as a transactional database of formal mathematics using the notion of abstract object identifiers. The database has led to an independent project called the Formal Digital Library, FDL, now used as a repository for Nuprl results as well as selected results from HOL, MetaPRL, and PVS. We discuss Howe's set theoretic semantics that is used to relate such disparate theories and systems as those represented by these provers.

1 Introduction

One of the most profound contributions of computer science to intellectual history is the body of demonstration that computers can automate many intellectual processes (not only calculation), and that by having access to knowledge in digital form they become *indispensable partners in knowledge formation.* The first clear articulation of this idea was in our field, applied mathematical logic and automated reasoning, about fifty years ago.

In its full scope, the idea of automating intellectual processes is profound and applies to knowledge formation in every instance. In its full generality, we might call it computer-mediated knowledge formation, or *digital knowledge* for short. Like logic itself, this is a meta-idea, transforming the very concept of knowledge. It rests on the insights of Euclid, Aristotle, Leibniz, Boole, Kant, Frege, Russell, Brouwer, Cantor, Gödel –giants of human achievement – and legions of logicians, philosophers, mathematicians, and

Preprint submitted to Elsevier Science

computer scientists. It has transformed the methodology of the physical sciences and created the new methodology of the information sciences of which automated reasoning and formal knowledge management are a key part.

This article is about one ongoing theme in the above story. The PRL research group in applied logic and automated reasoning at Cornell University plays a role in developing computational logics, implementing them, and demonstrating their value in mathematics, theoretical computer science, and software system design and verification. We will highlight our latest contributions in each of these activities and put them into both historical and contemporary perspective, suggesting directions for promising future work. We start with a discussion of the major themes that have influenced our work.

Proofs and Rules of Thought Aristotle and Euclid gave us the idea of deductive proof as a means to establish truth. The Nuprl ("new pearl") proof development system is based on a formal account of deduction. Proofs are the main characters and are used not only to establish truth but also to denote evidence, including computational evidence in the form of programs (functional and distributed). The idea of a *proof term* is a key abstraction; it is a meaningful mathematical expression denoting evidence for truth. We advocate using propositions as a means of classifying the evidence; the structure of the proposition determines properties of the evidence [27].

Propositions and Logical Truth Propositions are abstract objects about which it is sensible to make truth judgments (among others). Some propositions are about abstract objects, some are about physical objects. Logic is concerned with those aspects of truth that are applicable to reasoning in any domain.

A great deal has been written justifying philosophically the laws of logic. The Platonic view is that there are immutable laws in the world of ideas. The Kantian view is that these truths are grounded in properties of the mind. Brouwer focused on the computational powers of the mind (and Chomsky on the brain). It is no surprise that computer scientists are drawn to computational justifications because they provide guidelines for implementing logics and for automating intellectual processes on digital computers.

The PRL group has focused on computational justifications in the spirit of Bishop [20] and Martin-Löf [90,91,92]. We have applied their ideas in the context of computer science, significantly extending them, and experimenting with them using the Nuprl prover.

Type Theories The elementary parts of logic, including Aristotle's discoveries are relatively unproblematic. For example, the propositional calculus is quite well understood and beautifully presented in classic books such as Smullyan [106], Church [24], and Kleene [69]. It can easily be given several philosophical justifications, including a classical (Boolean) one or the Brouwer/Heyting/Kolmogorov [108] one based on computation for the Constructive (or Intuitionist) fragment. The Gödel dialectica interpretation [108] also applied to the constructive fragment.

Serious and difficult issues are lurking one small step away from elementary logic. Suppose $Prop$ is the domain of discourse for propositional logic; so a tautology such as $P \Rightarrow P$ would be made into a closed proposition by writing $\forall p : Prop.(p \Rightarrow p)$. According to Russell [102], it is not a well defined proposition because giving the meaning of $\forall p : Prop.(p \Rightarrow p)$ requires already having defined what is in $Prop$. He

labelled such an act as impredicative concept formation, and created his *theory of types* to control it.

In a contemporary type theory, we would stratify Prop into *levels*, say $Prop_1, Prop_2$,.... An expression such as $\forall p : Prop_1.(p \Rightarrow p)$ would belong to $Prop_2$ but not to $Prop_1$. Each $Prop_i$ is regarded as a distinct type. This is a common feature of the type theories implemented in Coq and Nuprl.

Russell's ramified type theory has led quite directly to those used in several modern theorem provers, e.g., Church's Simple Theory of Types (STT) a direct descendent of Russell's *Principia Mathematica* logic (PM), the Calculus of Inductive Constructions (CIC) of Coquand, Huet, Paulin [15,40], Intuitionistic Type Theory (ITT) of Martin-Löf, and Computational Type Theory (CTT) of the Cornell group [31,29] (which was derived from one version of ITT, and will be discussed in detail).

Type theory based provers are natural for computer science because the idea of logical types helped organize the concept of a *data type* as used in programming languages (say Algol68, Pascal, ML, Haskell, Java, C++, etc.). The relationship between logical types and data types is explicitly treated in [29,105,59].

Foundations of Mathematics – Sets and Types *Principia Mathematica* [111] was intended to be a logical foundation for mathematics, but it was more complex and less intuitive than formalizations of set theory in first-order logic. Most mathematicians seem to prefer Zermelo Fraenkel Set Theory (ZF) with the Axiom of Choice (ZFC) as the default formalization. There are other essentially equivalent set theories such as Bernays Gödel (BG).

A distinct approach to foundations is through theories of *logical* type, in which (1) the notions of *function* and *type* are basic, a type being the domain of a function, and (2) relations and properties are functions over their possible arguments, and so the meaning of propositional expressions is treated explicitly in terms of how functions are denoted. Russell's and Church's type theories are obviously of this kind.

Among modern type theories used as theories of logical type are CIC, CTT, and ITT, which are adequate foundations (some persons further requiring the law of excluded middle to be added to them to achieve adequacy). Indeed, these three theories have a great deal in common and are in some sense equivalent and converging. If we call the common theory Computational Types (CT), then CT with Choice, (CTC), is a canonical foundation for mathematics as well. In ZFC, the familiar types such as integers ($\mathbb{Z}$), reals ($\mathbb{R}$), functions ($A \to B$), Cartesian products ($A \times B$) and so forth are defined in terms of sets. In CTC, sets are defined in terms of types.

Foundations of Computing Computational type theory (CTT) and the calculus of inductive constructions (CIC) are intended to be foundations for computing. This means that special attention is given in logic design to expressions that can express algorithms and data structures, designing rules adequate for reasoning about them with types. In practice, this means that these theories can provide a semantics for real programming languages and for the logics of programs associated with them. See also [10] for a fine exposition of such methods extending ITT.

For both CTT and CIC, the programming languages native to the theory are simple functional programming languages. In order to define and reason about side effects,

modules, and objects requires a great deal of work [53,54,41]. CTT has been extended to reason about events and distributed computing [17]. Here we call it CTT-E, and we describe it briefly later.

Proofs as Programs and Processes The constructive type theories, CIC, CTT, and ITT, express logic by reducing propositions to types, and so serve as theories of logical type. This reduction is often called the *propositions-as-types principle* [45,90,61]. The usual enunciation of this principle (Martin-Löf's innovation on it will be described below) is to associate with every proposition P a type, say $[P]$, consisting of objects representing proofs of P. The type $[A\&B]$ for a conjunction could be the cartesian product $[A] \times [B]$. For an implication $A \Rightarrow B$, the type $[A \Rightarrow B]$ is $[A] \rightarrow [B]$, the type of computable functions from the content of A into the content of B. Likewise, $[\forall x : T.B_x]$ consists of the dependent function space $x : [T] \rightarrow [B_x]$, where $[T]$ is the data type associated with the mathematical type T.[1] This topic is explained well in the literature [49,107,12,99].

When f is a proof of $A \Rightarrow B$, then its content, $[f]$, is an element of $[A] \rightarrow [B]$, that is a computable function. The proof object f is like a program in that its computational content is a computable function. Systems that implement CTT must have operations that *extract* the program from f, that is, operations to construct $[f]$ from f.

As we explain later, for expressions that use the type of events, $\mathbb{E}$, expressions of the form $\forall e : \mathbb{E}p.\exists e' : \mathbb{E}q.R(e, e')$ describe relations between events at one agent (or location) p and those at another q. A proof that this relation is realizable will denote a computing process that implements it.

Martin-Löf's innovation on the propositions-as-types principle was to use *computational content* of proofs of P rather than proofs as such. Thus, when explaining the meaning of proposition P as a type, only the computational content of all its possible proofs (i.e. the data available to functions implicit in proofs of implications from P) need be stipulated. The production of any non-computational epistemic content is deferred to producing an argument that $[P]$ has a member.

Under this version of the principle, one may be given a member e which has type T without being also provided a method for constructing from e *alone* an argument for this fact. Propositions that are difficult to prove, say having the form $\forall x : \mathbb{N}.\ f(x)=0$, can still have trivial computational content $\lambda x.\mathrm{r}$ where r is a trivial constant.

Automating Reasoning The Nuprl system is a tactic style theorem prover in the spirit of Edinburgh LCF [50]; moreover it introduced the notion of a *tactic-tree* so that users could interact directly with the proof object [37,52]. We take the approach that the basic elements of automated proof development can be seen as providing algorithms for formal metamathematical results, so it is computation in the *metalanguage* [33,71]. There isn't space in this article to discuss or illustrate the tactic mechanism, but the interested reader can see thousands of examples of tactic-tree proofs in our online library of formal mathematics at `http://www.nuprl.org`. In addition, the library

[1] This rather direct translation only works when any data computationally derivable from proofs of $t \in T$ are already computationally derivable from t itself. See [109] (chapter 1, section 3) and [6] (chapter 7) for discussion.

contains many examples of the basic tactics written in classical ML, the language of LCF which we adapted as the main programming language for Nuprl.

The first person to build substantial tactics for Nuprl was Douglas Howe, and he was able to use them to settle an important open problem about the typed lambda calculus, called the *Girard Paradox* [62]. Another person who solved an important open problem in mathematics was Chetan Murthy in generating a constructive proof of Higman's Lemma from its classical proof [96].

Over the years, many users have added to the Nuprl tactic library, sometimes by importing important algorithms from other provers, adapting packages from other provers such as HOL and Isabelle. We benefitted a great deal from the use of a version of Nuprl at Edinburgh for many years and have adapted many of the ideas from Alan Bundy's group there, especially their reconstruction of methods from the Boyer and Moore prover Nqthm [22]. Chetan Murthy worked with the Coq group at INRIA and this resulted in sharing other ideas, designs and code (in both directions). The MetaPRL [58,57] group has contributed many ideas and programs as well. Now when we are using Nuprl, we are acutely aware that its capabilities rest on the contributions of dozens of modern researchers and the legacy of the logicians, philosophers, mathematicians and computer scientists we mentioned before. We will discuss later the contribution of fully automatic proving, through JProver, to proof automation in Nuprl, MetaPRL, and Coq.

Formal Digital Libraries The important role of knowledge, and the information resources from which it arises, is clear from the behavior of the best system designers and programmers. They bring extensive knowledge to bear in their reasoning about designs and code, sharing knowledge and reasoning in a coordinated yet distributed manner. However, the tools used to help them create and check inferences have access to limited fragments of formal proofs. It is a terrible waste of time and huge cost to unnecessarily recreate knowledge. Several researchers have made significant strides to overcome the challenges that sharing formal mathematics create [2,9,23,73,72,112]. The community of Mathematical Knowledge Management (MKM) has emerged with these challenges at heart. The PRL research group has built a prototype Formal Digital Library in response to a MURI grant [30], and continues to focus efforts in this domain.

2 Computational Type Theory

Computational Type Theory (CTT) is a family of closely related theories designed to provide a foundation for reasoning about computation both in mathematics and computing practice. It is strongly influenced by the computer scientist's wish to have a universal programming language as an integral part of the theory. There are two implementations of this theory, the initial one in Nuprl and a newer one in MetaPRL. Any implementation of the theory will provide an implementation of this language. This requirement on a foundational theory for computing goes back to early work in computer science on programming logic, including work at Cornell in which PLC programs were the algorithms of the logic, which was called PLCV [38,36,35]. Interestingly, some of the reasoning methods from PLCV were directly recoded in the Nuprl system, the arithmetic decision procedure for example, presented in [35].

Computational Type Theory was also influenced by our experience trying to generalize the type system of PLCV to make it more like Algol68 and to integrate the assertions of the programming logic with the types. Our efforts in this direction were dramatically affected when we read the work of Per Martin-Löf [90]. We incorporated his ideas into our programming logic [39], and then decided with Joe Bates to adopt Martin-Löf's theory as the core of our type theory. We were deeply impressed by Martin-Löf's idea that an untyped lambda calculus underlies the semantics of the type theory, and that methods of describing types and values could be rich enough that one must sometimes carry out complex arguments that a purported expression actually succeeds at describing a type or a value of a certain type. That is, the *well-formedness* of expressions for types and typed values is not a mere decidable syntactic condition.

Our technical presentation of types will begin with Martin-Löf's Intuitionistic Type Theory as it appears in Nuprl. But first we describe some differences in the logic designed for the Nuprl system.

2.1 Differences in Logic Design

Martin-Löf's texts, especially [91], which informed our understanding of his methods, formalize arguments about types as natural deduction style proofs under hypotheses of form $x \in A$, for variable x and type expression A. A peculiarity of this style of proof was that any proof introducing such a hypothesis must be built by extending a proof comprising a proof of A being a well-formed type expression. Thus, a common form of a proof that $(\Pi x\!:\!A)B(x)$ denotes a type would be a proof that A denotes a type followed immediately by introducing hypothesis $x \in A$ and proving $B(x)$ denotes a type under that hypothesis, then finishing with an inference to the proof goal discharging the hypothesis.

This did not suit the Nuprl design goal of organizing proofs by assertions to be proved, typically proving an assertion by proving independently provable assertions as premises. The meaning of Martin-Löf's "$B(x)$ type $(x \in A)$" *presupposes* that A is a type, and under that presupposition means that $B(x)$ describes a type-valued function in x over A. Hence proving "$B(x)$ type $(x \in A)$" requires proving A is a type. The alternative logic design choice made for Nuprl was to stipulate (and we use a different syntax here to avoid confusion) that "$x\!:\!A \vdash B(x)$ type " means that *if* A is a type then $B(x)$ describes a type-valued function in x over A. Consequently, the problem of showing that "$B(x)$ type $(x \in A)$" reduces to independently showing both "A type" and "$x\!:\!A \vdash B(x)$ type".

Another purpose of the Nuprl design was to automatically derive witnesses to claims that some type has a member, forestalling the actual construction of the witness as long as practical. This allows the user to carry out much of a typical proof as though it was an ordinary proof of a proposition; so a person might simply prove a theorem of the form "$\forall x : A.\ P(x)$" without always having to think about what functional expression is implicit in its construction. This provides an elegant solution to the project goal of implicitly extracting programs from proofs. Hence, "$\vdash T$ " was stipulated to mean that there is an expression t such that $t \in T$. And similarly, "$x\!:\!A \vdash B(x)$ " was stipulated to mean that if A is a type then $B(x)$ describes a type-valued function in x over A and there is an expression $t(x)$ that describes a $B(x)$-valued function in x over A.

By stipulating for each rule of inference how to automatically construct the witness for the conclusion from any witnesses for the premises, one has shown how to automatically construct a witness, often a program, from any complete proof.

This goal of constructing programs automatically from proofs that their specifications can be met was an original design goal for the PRL project[13], and Martin-Löf's theory provided an elegant basis for it[14], superior to the one suggested in [25].

2.2 Direct Computation Rules

A striking feature of the computationally type-free basis for the type theory as given in [91] is that *any* expression that evaluates to an expression of a given type is also an expression of that type and denotes the same value in that type. Thus, $(\lambda x.0)(a) = 0 \in \mathbb{N}$ no matter what (variable-free) untyped expression might be used for a. In the formal ITT systems provided by Martin-Löf there is no general rule exploiting this semantic feature. In contrast, such exploitation has become a hallmark of CTT methodology.

As reported in the 1986 book [31], standard Nuprl included *direct computation* rules, which allowed inference by various rewrites within a proof goal by steps of type-free computation. Thus, for example, showing the fact mentioned above reduces by one use of direct computation to showing that $0 = 0 \in \mathbb{N}$. Using these direct computation rules sometimes allows one to assign significant types to programs by induction over possible arguments. For example, a simple recursive unbounded search of a denumerable sequence of numbers for its first root, such as Kleene's minimization operation, can be typed $\{f : (\mathbb{N} \rightarrow \mathbb{N}) | \exists n : \mathbb{N}.\ f(n) = 0\} \rightarrow \mathbb{N}$ and shown to find the first root of any function in this domain.[2] Indeed, such a typing argument was one of the motives for including the direct computation rules in the system originally.

Users developed the practice of using the untyped Y-combinator to define functions recursively instead of using various primitive recursion combinators, and then proving they had the appropriate types by induction.[3] The original direct-computation rules as presented in [31], justified ultimately by methods in [5], involved unpalatable restrictions on contexts permitting rewrites, and were eventually simplified to allow type-free rewrites freely throughout a proof goal, justified by [63].

2.3 Intuitionistic Type Theory of Martin-Löf

Types are built from a few primitive types using basic *type constructors*. These provide templates for constructing various kinds of objects. The type constructors of ITT are motivated by mathematical considerations. However, they also correspond to data type constructors used in programming. The notes by C.A.R. Hoare, *Notes on Data Structuring*, also explain the ties between data types and logical types [59].

We start with the basic type constructors of ITT.

---

[2] For a CTT proof of similar facts see "Recursive Functions" at www.nuprl.org/NuprlBasics

[3] Indeed, the Y-combinator itself, not just its applications, can be assigned appropriate types if one uses the intersection type constructor of Section 2.5.1 below, and reformulates the function space constructor purely in terms of typing its applications to arguments, ignoring convergence. See "Typing Y" at `www.nuprl.org/NuprlBasics`

2.3.1 *Cartesian products*

If A and B are types, then so is their *product*, written $A \times B$. There will be many *formation rules* of this form, so we adopt a simple convention for stating them. We write

$$\frac{A \text{ is a } Type \quad B \text{ is a } Type}{A \times B \text{ is a } Type.}$$

The elements of this product are pairs, $< a, b >$. Specifically if a belongs to A and b belongs to B, then $< a, b >$ belongs to $A \times B$. We abbreviate this by writing

$$\frac{a \in A \quad b \in B}{< a, b > \ \in \ A \times B.}$$

In programming languages these types are generalized to n-ary products, say $A_1 \times A_2 \times \ldots \times A_{n2}$.

We define equality on the elements by

$$< a, b >=< c, d > \text{ in } A \times B \text{ iff } a = c \text{ in } A \text{ and } b = d \text{ in } B.$$

In set theory, equality is uniform and built-in, but in type theory we define equality with each constructor; in ITT these equalities are built-in.

There is essentially only one way to decompose pairs. We say things like, "take the first element of the pair P," symbolically we might say $first(P)$ or $1of(P)$. We can also "take the second element of P," $second(P)$ or $2of(P)$.

A uniform way to access both elements of a pair simultaneously is the *spread* operation. The formula $spread(P; x, y.t)$ expresses that every free occurrence of x in the expression t represents the first element of the pair P and that every free occurrence of y in t represents the second element of P. It is often displayed as $let < x, y >= P \ in \ t$.

2.3.2 *Function Space*

We use the words "function space" as well as "function type" for historical reasons. If A and B are types, then $A \to B$ is the type of *computable functions* from A to B. The formation rule is:

$$\frac{A \text{ is a } Type \quad B \text{ is a } Type}{A \to B \text{ is a } Type}$$

The informal notation we use for functions comes from mathematics texts, e.g. Bourbaki's *Algebra* [21]. We write expressions like $x \mapsto b$ or $x \overset{f}{\mapsto} b$; the latter gives a name to the function. For example, $x \mapsto x^2$ is the squaring function on numbers.

If b computes to an element of B when x has value a in A for each a, then we say $(x \mapsto b) \ \in \ A \to B$. Formally we use lambda notation, $\lambda(x.b)$ for $x \mapsto b$. The informal rule for typing a function $\lambda(x.b)$ is to say that $\lambda(x.b) \in A \to B$ provided that when x is of type A, b is of type B. We can express these *typing judgments* in the form $x : A \vdash b \in B$. The typing rule is then

$$\frac{x : A \vdash b \in B}{\vdash \lambda(x.b) \ \in \ A \to B}$$

If f, g are functions, we define their equality extensionally as

$$f = g \quad \text{iff } f(x) = g(x) \quad \text{for all} \quad x \text{ in } A.$$

If f is a function from A to B and $a \in A$, we write $f(a)$ for the value of the function.

2.3.3 Disjoint Unions

Forming the union of two sets, say $x \cup y$, is a basic operation in set theory. It is basic in type theory as well, *but* for computational purposes, we want to discriminate based on the type to which an element belongs. To accomplish this we put tags on the elements to keep them disjoint. Here we use *inl* and *inr* as the tags.

$$\frac{A \text{ is a } Type \quad B \text{ is a } Type}{A + B \text{ is a } Type}$$

The membership rules are

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

We say that $inl(a) = inl(a')$ iff $a = a'$ and likewise for $inr(b)$.

We can now use a case statement to detect the tags and use expressions like

$$\begin{array}{lll} \text{if } x = inl(z) & \text{then } \ldots & \text{some expression in } z \ldots \\ \text{if } x = inr(z) & \text{then } \ldots & \text{some expression in } z \ldots \end{array}$$

in defining other objects. The test for $inl(z)$ or $inr(z)$ is computable by an operation called *decide* that discriminates in the type tags. The typing rule and syntax for it are given in terms of a typing judgment of the form $E \vdash t \in T$ where is a list of declarations of the form $x_1 : A_1, \ldots, x_n : A_n$ called a *typing environment*. The A_i are types and x_i are variables declared to be of type A_i. The rule is

$$\frac{E \vdash d \in A + B \qquad E, u : A \vdash t_1 \in T \qquad E, v : B \vdash t_2 \in T}{E \vdash decide(d; u.t_1; v.t_2) \in T}$$

2.3.4 Dependent Product

Here is a simple example of the dependent product type constructor. Suppose you are writing an application needing a data type for the date.

$$\text{Month} = \{1,\ldots,12\},\ \text{Day} = \{1,\ldots,31\},\ \text{and Date} = \text{Month} \times \text{Day}.$$

We would need a way to check for valid dates. The pair, $2, 31$ is a perfectly legal member of $Date$, although it is not a valid date. One thing we can do is to define

$$\begin{aligned} Day(1) &= \{1, \ldots, 31\} \\ Day(2) &= \{1, \ldots, 29\} \\ &\vdots \\ Day(12) &= \{1, \ldots, 31\} \end{aligned}$$

and we will now write our data type as $Date = m : Month \times Day(m)$.

We mean by this that the second element of the pair belongs to the type indexed by the first element. Now, $2, 20$ is a legal date since $20 \in Day(2)$, and $2, 31$ is illegal because $31 \notin Day(2)$.

Many programming languages implement this or a similar concept in a limited way. An example is Pascal's *variant records*. While Pascal requires the indexing element to be of scalar type, we will allow it to be of any type.

One view of this is that we are generalizing the product type. It is very similar to $A \times B$. Let us call this type $prod(A, x.B)$. We can display this as $x : A \times B$. The typing rules are:

$$\frac{E \vdash a \in A \quad E \vdash b \in B[a/x]}{E \vdash pair(a,b) \in prod(A, x.B)} \qquad \frac{E \vdash p \in prod(A, x.B) \quad E, u : A, v : B[u/x] \vdash t \in T}{E \vdash spread(p; u, v.t) \in T}$$

Note that we haven't added any elements. We have just added some new typing rules.

2.3.5 Dependent Functions

If we allow B to be a family in the type $A \to B$, we get a new type, denoted by $fun(A; x.B)$, or $x : A \to B$, which generalizes the type $A \to B$. The rules are:

$$\frac{E, y : A \vdash b[y/x] \in B[y/x]}{E \vdash \lambda(x.b) \in fun(A; x.B)}\ new\ y \qquad \frac{E \vdash f \in fun(A; x.B) \quad E \vdash a \in A}{E \vdash f(a) \in B[a/x]}$$

Back to our example *Dates*. We see that $m : Month \to Day[m]$ is $fun(Month; m.Day)$, where Day is a family of twelve types. And $\lambda(x.maxday[x])$ is a term in it.

2.3.6 Universes

A key aspect of ITT is the ability to denote types and type-valued functions using all the generic methods of the theory, especially those for reasoning about (typed) functions. For example, one might choose to recursively define the types of n-place curried numeric functions, such that $\mathbb{N}^0 \to \mathbb{N} = \mathbb{N}$ and $\mathbb{N}^{n+1} \to \mathbb{N} = \mathbb{N} \to (\mathbb{N}^n \to \mathbb{N})$. A convenient explanation of how this defines a type would be to employ a generic form for iteration of functions which explains, for any type T, for base case $b \in T$, and any function $F \in T \to T$, how to define a function $g \in \mathbb{N} \to T$ such that $g(0) = b$ and $g(n+1) = F(g(n))$.

To apply this generic form of reasoning about functions, we would need to find an appropriate type T such that $\mathbb{N} \in T$ and $\lambda(A.\, \mathbb{N} \to A) \in T \to T$. The obvious first choice would be to introduce a type $\mathbb{U}$ of all types, but there cannot be one in ITT [60].

However, one can introduce a series of types $\mathbb{U}_n$ called *universes* [91], and one can stratify the explanation of how expressions denote types by taking whatever basic type construction methods one might have and iteratively introducing a sequence of type expressions $\mathbb{U}_n$ [5]. The base definition of type expressions is defined simply from the base construction methods. Then one defines the level $k+1$ system from the level k system by adding $\mathbb{U}_k$ as a new constant type expression whose members are the type expressions defined at level k. The union of this sequence of systems then assigns every type expression to a universe, assigning $\mathbb{U}_n \in \mathbb{U}_{n+1}$, and making the universes cumulative, i.e. $\mathbb{U}_n$ is a subtype of $\mathbb{U}_{n+1}$.

Returning to our example, for each k, adopting $\mathbb{U}_k$ as the type to be used as T in the generic iterated function pattern, we get $\lambda(n.\, \mathbb{N}^n \rightarrow \mathbb{N}) \in \mathbb{U}_k$. Similarly, when expressing and proving general facts about types, one quantifies over universes rather than a non-existent type of all types, similar to Russell's ramified logic.

Note that the *level variable* n in $\mathbb{U}_n$ is a *not* a variable over which one quantifies within the logic. Indeed, the most perspicuous formulation of a logic for type theory with universes would, like both Martin-Löf's and the original Nuprl logic of [31], require this universe level in any particular expression within the logic to be a constant, a numeral. But requiring users to produce these constants proved inconvenient; it too often happened that a lemma would be proved for level k that would later have been more useful had it been formulated at a higher level instead.

As a result, based upon [6], the Nuprl logic was reformulated to use more general level expressions for universes, including *level variables* (as letters), an operator for taking the *maximum* of two or more levels, and an operator for incrementing a level by a constant. A key rule was added that allowed the inference from any sequent to another gotten by uniformly instantiating all the level variables by any level-expressions. [4]

Hence, a proof of "$\forall A : \mathbb{U}_k.\ P(A)$" takes on the force of quantification over all defined types, and such a lemma could be applied to a type of any level by instantiating k with an appropriate level expression. As a notational matter, when all the level expressions throughout an expression of CTT are simply occurrences of the same level variable, we often write simply "*Type*" instead of $\mathbb{U}_i$, say.

2.3.7 Propositions as types

One of the key distinguishing features of Martin-Löf type theory [90,91,92] and the Nuprl type theory [31] is that propositions are considered to be types, and a proposition is true iff it is inhabited. The inhabitants of a proposition are mathematical objects that provide evidence for the truth of the proposition.

This approach to propositions is related to the so-called *Curry-Howard isomorphism* [44,61] between propositions and types. But the correspondence is elevated in our type theory to the status of a principle. The Curry-Howard isomorphism explains why the definition is sensible for Heyting's formalization of Brouwer's constructive semantics for propositions. We briefly recall these semantics and state the Nuprl definitions for the logical operators. According to Brouwer the logical operators have these constructive meanings:

$\bot$	is never true (read $\bot$ as "false").
$A \,\&\, B$	is true (provable) iff we can prove A and we can prove B.
$A \vee B$	is true iff we can prove either A or B, and we say which is proved.
$A \Rightarrow B$	is true iff we can effectively transform any proof of A into a proof of B.
$\neg A$	holds iff $A \Rightarrow \bot$.
$\exists x{:}A.B$	is true iff we can construct an element a of type A and a proof of $B[a/x]$.
$\forall x{:}A.B$	is true iff we have an effective method to construct a proof of $B[a/x]$ for any a in A.

[4] Note that, therefore, since one can always change level variables, that there is no inferential significance of the same level variable occurring in different sequents of an inference.

For an atomic proposition, P, with no logical substructure, we say it is true exactly when we have evidence for it. The evidence can be taken to be atomic as well, e.g. unstructured. So we use the unit element, $\bullet$, as evidence.

We express Brouwer's semantics by these definitions of the logical operators:

Definition	$A \,\&\, B$	$\equiv$	$A \times B$
	$A \vee B$	$\equiv$	$A + B$
	$A \Rightarrow B$	$\equiv$	$A \rightarrow B$
	$\exists x : A.B$	$\equiv$	$x : A \times B$
	$\forall x : A.B$	$\equiv$	$x : A \rightarrow B$
	$\bot$	$\equiv$	$void$
	$\top$	$\equiv$	$\mathbf{1}$

We can also define other logical operators that express shades of meaning not commonly expressed. For example, $\cap x : A.B$ (Section 2.5.1) makes sense as a kind of universal quantifier.

We use $Prop_i$ as a synonym for $Type_i$.

2.4 Early Extension to ITT

2.4.1 Subset and Quotient Types

One of the first extensions we made to ITT type theory has been one of the most useful and pervasive, a type we called the *subset type* [26]. The idea seems obvious from set theory, but its computational interpretation is subtle. Given a type A and a propositional function on A, say $B : A \rightarrow Prop_i$, we form the type of those elements of A satisfying B, $\{x : A | B(x)\}$. For example, if $\mathbb{Z}$ is the type of integer, then the natural numbers can be defined as $\{z : \mathbb{Z} | z \geq 0\}$. This type is different from $z : \mathbb{Z} \times z \geq 0$ because the elements of $\{z : \mathbb{Z} | z \geq 0\}$ are simply numbers whereas the elements of the product are pairs $< z, p >$ where p is the content of the proposition $z \geq 0$. The subset type is very useful in a computational theory because it allows us to separate data from its logical properties.

Another key extension we introduced at the same time is the *quotient type*. Given a type A, it comes with an equality relation, $=_A$. We can form a new type by changing the equality. If E is an equivalence relation on A, then $A//E$ is a new type, the quotient of A by E. The elements of $A//E$ are the elements of A, but the equality on $A//E$ is E. Good examples are the congruences of integers. That is, if $E_n(x, y)$ is $x = y \ mod \ n$, then $\mathbb{Z}//E_n$ are the integers modulo n, often denoted $\mathbb{Z}_n$. In $\mathbb{Z}_2$, $0 = 2 = 4 = 6 = ...$

2.4.2 Inductive Types

For recursive types ITT uses W-types (well-founded trees, Brouwer ordinals). An alternative approach was used for the Nuprl system by which recursive types are defined as minimal fixed points of suitable operators on types [94]. For example, a list-like type could be defined as $(\mu X.\,\mathrm{Unit} + (A \times X))$, and Martin-Löf's type $(W x \in A) B(x)$ could be defined as $(\mu W.\, x{:}A \times (B(x) {\rightarrow} W))$.

We say that $t_1 = t_2 \ \in \ \mu X.F(X)$ iff $t_1 = t_2 \ \in \ F(\mu X.F(X))$.

To construct elements of $\mu X.F(X)$, we basically just *unwind* the definition. That is,

$$\text{if} \quad t \in F(\mu X.F(X)) \quad \text{then} \quad t \in \mu X.F(X).$$

There is a natural notion of *subtyping* in this theory. We say that $A \sqsubseteq B$ iff $a = a' \in A$ implies that $a = a' \in B$. Semantically, a recursive type defined by $(\mu X.\, F(X))$ is meant to be the $\sqsubseteq$-minimal transitive symmetric relation on terms that is closed under F.

Induction over this type can be expressed as a rule, similar to rules in [94,58],

$$\frac{H, x : (\mu Z.T), J, u : \mathbb{U}_i, u \sqsubseteq (\mu Z.T), w : (x : u \rightarrow G[x/x]), z : T[u/Z] \vdash G[z/x]}{H, x : (\mu Z.T), J \vdash G[x/x]}$$

where $A \sqsubseteq B$ is simply an operator notationally defined as $a : A \rightarrow (a \in B)$, which expresses subtyping, since there is a member iff equal expressions of type A are equal expressions of type B.

The power of recursive types comes from the fact that we can define total computable functions over them and we can prove properties of elements recursively. (These are the same idea.) Recursive definitions are given by this term, $\mu\text{–}ind(a; f, z.b)$ called a *recursor* or *recursive-form*. It obeys the computation rule

$\mu\text{–}ind(a; f, z.b)$ evaluates in one step to $b[a/z, (\lambda y.\, \mu\text{–}ind(y; f, z.b))/f]$.

It is rather clear what type $(\mu X.F(X))$ is when it is a type at all. It may be more controversial when $(\mu X.F(X))$ actually succeeds at describing a type. One widely acceptable criterion would be when $F(X)$ describes a $\sqsubseteq$-monotonic operation on types, which would be handily formulated by a rule

$$\frac{H, Z : \mathbb{U}_i \vdash T \in \mathbb{U}_i \quad H, X_1 : \mathbb{U}_i, X_2 : \mathbb{U}_i, (X_1 \sqsubseteq X_2) \vdash T[X_1/Z] \sqsubseteq T[X_2/Z]}{H \vdash (\mu Z.T) \in \mathbb{U}_i}$$

Some constructivists, however, might prefer a rule that imposes other constraints (perhaps intensional ones, involving positivity, say). One advantage of introducing particular kinds of recursive types, such as W-types and lists, instead of using this generic recursive type constructor is that the conditions for legitimately describing types would be quite narrow, and so more widely accepted.

2.5 Class Extensions

As we expanded our efforts from verifying moderately large programs (say up to one thousand lines [1]) to verifying systems, we were faced with formalizing some of the basic concepts from object-oriented programming. Jason Hickey took on this task at the start of his thesis work, and along with Karl Crary, Alexey Nogin, and Aleksey Kopylov, we made a great deal of progress in building a *theory of classes* in CTT. These ideas are reported in several publications, among them [32,58,42,43,75]. The MetaPRL prover was built using object-oriented concepts, even in the tactic mechanism.

The key concepts turned out to be the subtyping relation and a generalization of the intersection type to dependent intersections – an idea that has had a profound impact on both the theory and its applications to designing systems.

2.5.1 *Intersection types and the Top type*

Given two types A and B, it makes sense to consider the elements they have in common; we call that type the *intersection* of A and B, written $A \cap B$. We require that $(a = b \text{ in } A \cap B)$ iff $a = b$ in A and $a = b$ in B. To illustrate, it is clear that $void \cap A$ is $void$ for any type A and $A \cap A$ is A.

It might be a surprise that $(1 \to 1) \cap (void \to void)$ is not $void$ but contains the identity function, $\lambda(x.x)$. This is because the base objects of Nuprl are *untyped*, so $\lambda(x.x)$ is polymorphic, belonging to all types $A \to A$. It is clear that $\{x{:}A|P(x)\} \cap \{x{:}A|Q(x)\}$ is $\{x : A|P(x)\ \&\ Q(x)\}$.

The intersection type is defined for a family of types as well. Let $B(x)$ be a family of types for $x \in A$, then $\cap\, x : A.B(x)$ is their intersection; and $b \in \cap\, x : A.B(x)$ iff $b \in B(a)$ for all $a \in A$. also $b = b'$ in $\cap\, x : A.B(x)$ iff $b = b'$ in $B(a)$ for all $a \in A$. Notice that when A is empty, $\cap\, x : A.B(x)$ is not empty but has exactly one element. This element is denoted by any closed expression of the theory, e.g. $void = 1$ in $\cap\, x : void.B(x)$. Such equalities follow from the understanding that under the assumption that $x \in void$, we can infer anything.

Types such as $\cap\, x : void.B(x)$ are maximal in the sense that every type is a subtype and every closed term denotes the single member of the type. we pick one such type expression and call it the *top type*, denoted Top. For example, $\cap\, x : void.x$ can be Top.

Type equality follows the pattern for structural equality, namely

$$(\cap\, x : A.B(x) = \cap\, x : A'.B'(x)) \text{ iff } A = A' \text{ and } B(a) = B'(a)$$

Intersection types can express conditional typehood, which is useful in typing partial functions. If B does not depend on x, then $\cap\, x : A.B$ expresses the notion that B *is a type provided that* A *is inhabited*, as pointed out by Backhouse et. al [10]. We write this as B *given* A, following an analysis of this idea by Stuart Allen. Allen calls these *guarded types*.

Guarded types can specify the conditions under which $A \Rightarrow B$ is a type in Nuprl. Namely, if A is nonempty, the B must be a type. So the typing is

$$\lambda(A, B.A \to B) \in A : Type \to B : (\cap\, x : A.Type) \to Type.$$

Guarded types are also useful in expressing one-one correspondences between types. For example we know that mappings from pairs p in $x : A \times B(x)$ into $C(p)$ are isomorphic to curried mappings of the type $x{:}A \to y{:}B(x) \to C(< x, y >)$. Similarly, mappings $x : \{x{:}A \mid B(x)\} \to C(x)$ are isomorphic to those in $x{:}A \to \cap\, y{:}B(x).C(x)$.

2.5.2 *Subtyping*

As mentioned before, the natural notion of subtyping is to define $A \sqsubseteq B$ iff $a = a' \in A$ implies that $a = a' \in B$. For example, $\{x : A \mid P(x)\} \sqsubseteq A$ for any predicate $P(x)$. We clearly have $void \sqsubseteq A$ for any type A and $A \sqsubseteq Top$ for any type A, and $A \cap B \sqsubseteq A$ for any types A and B. It is easy to see that the following relationships hold.

If $A \sqsubseteq A'$ and $B \sqsubseteq B'$ then

1. $A \times B \sqsubseteq A' \times B'$
2. $A + B \sqsubseteq A' + B'$
3. $A' \to B \sqsubseteq A \to B'$

The relation in 3 holds because functions in type theory are polymorphic. Given $f \in A' \to B$, we see that it belongs to $A \to B'$ as well because on inputs restricted to A it will produce results in B. In set theory this relationship would not hold, as we see in examples from the discussion of classes.

2.5.3 Records

We follow the idea from algebra and programming languages that classes are like algebraic structures, sets with operators. But classes are defined using signatures, and these are large types. Without the signature definition, as a large type, we could not have the definition of a class as a small type. The precise type we start with is a record type, these will also be classes.

Let F be a discrete type, the field names, and define a signature over F as a function from F into types, say $Sig \in F \to \text{Types}$. A *record type on discrete type F with signature* $Sig \in F \to \text{Types}$ is the dependent type $i : F \to Sig(i)$. If F is a finite type, say $\{x_1, \ldots, x_n\}$, then we write the record as $\{x_1{:}Sig(x_1); \ldots; x_n{:}Sig(x_n)\}$. If we are given types $T_1, \ldots, T_n$ we can also write simply $\{x_1{:}T_1; \ldots; x_n{:}T_n\}$ and this implies that we built the function assigning type T_i to name x_i.

Notice that if F_1 and F_2 are discrete types such that $F_1 \sqsubseteq F_2$ and $Sig_1 : F_1 \to \text{Type}$, $Sig_2 : F_2 \to \text{Type}$ and $Sig_1(x) \sqsubseteq Sig_2(x)$ for $x \in F_1$ then

$$i : F_2 \to Sig_2(i) \quad \sqsubseteq \quad j : F_1 \to Sig_2(j).$$

An important example of these relationships will be used as a running illustration. We consider it here briefly. Given a type M as the carrier of algebraic structure, we get the structure of a monoid over M, $Monoid_M$, is $\{op : M \times M \to M; id : M\}$. We can extend this to a group structure, $Group_M$, as $\{op : M \times M \to M; op : M; inv : M \to M\}$. Notice that $Group_M \sqsubseteq Monoid_M$.

2.5.4 Uniform Records

Extending records is a basic operation, as in extending a Monoid structure to a Group structure. This can be done in a uniform way if we agree on a common set of names to be used as names of the components ("fields" as they are often called). Let us take *Label* as an infinite discrete type. Signatures *Sig* over *Label* are functions in $i : Label \to Type$, and records with this signature are $i : Label \to Sig(i)$.

We generally consider records in which only finitely many labels are names of significant components. If we map the insignificant components to the type Top, then intersection of arbitrary uniform records makes sense and is defined as

$$(i : Label \to Sig_1(i)) \ \cap \ (j : Label \to Sig_2(j)) \ = i : Label \to Sig_1(i) \ \cap \ Sig_2(i).$$

Record intersection is an operation that extends records as we see if we assume that *op*, *id* and *inv* are elements of *Label* and assume that $MonSig_M(i) = Top$ if $i \neq op$ and $i \neq id$. Generally when we write $\{x_1{:}Sig(x_1); \ldots; x_n{:}Sig(x_n)\}$ we will assume that $Sig(i) = Top$ for $i \neq x_j$. So now consider $\{op{:}M \times M \to M; id{:}M\} \ \cap \ \{inv{:}M \to M\}$.

The result is $Group_M$, $\{op{:}M \times M \to M; id{:}M; inv{:}M \to M\}$ because in the structure $\{inv{:}M \to M\}$, the components for *op* and *id* are Top and in Monoid the component for *inv* is Top. Thus for example, the type at *op* in the intersected structure is

$$op{:}(M \times M \to M) \ \cap \ Top \ \text{ and } \ (M \times M \to M) \ \cap \ Top = M \times M \to M.$$

When two records share a common field, the intersected record has the intersection of the types, that is $\{x{:}T\} \cap \{x{:}S\} = \{x_1{:}T \cap S\}$ So generally,

$$\{x{:}T\} \cap \{y{:}S\} = \begin{cases} \{x{:}T, y{:}S\} & if\ x \neq y \\ \{x{:}T \cap S\} & if\ x = y \end{cases}$$

2.5.5 Dependent Records

Records can be seen as (Cartesian) products with labels for the components. Can we define the record analogue of dependent products? The right notation would be $\{x_1{:}T_1; x_2{:}T_2(x_1); \ldots; x_n{:}T_n(x_1, .., x_{n-1})\}$ where $T_i(x_1, .., x_{i-1})$ is a type depending on $x_1, .., x_{i-1}$. Can we define this structure in terms of a function $Sig : \{x_1, .., x_n\} \to Type$? We see that $Sig(x_1)$ is just a type, but $Sig(x_2)$ uses a function $F \in Sig(x_1) \to Type$ and it is F applied to x_1. We might think of writing $F(x_1)$, but x_1 is a name and F wants an element of $Sig(x_1)$. Basically we need to talk about an arbitrary element of the type we are building. That is if $r \in i : \{x_1, \ldots, X_n\} \to Sig(i)$, then $Sig(x_2)$ is $F(r(x_1))$. So to define Sig we need r, and to type r we need Sig.

We see that an element of a type such as $\{x_1 : T_1; \ldots; x_n : T_n(x_1, .., x_{n-1})\}$ has the property that its type depends on its "previous values". That is, the type of $r(x_1)$ is $Sig(x_1)$ but the type of $r(x_2)$ is $Sig(x_2)(r(x_1))$. We can encode this type using the *very-dependent function type* denoted $\{f \mid x : A \to B[f, x]\}$. We give Jason Hickey's original definition below. Now we use it to define the dependent records. We take A to be the finite type $\{0 \ldots (n-1)\}$. Let

$$T_f(n) \equiv f \not\to i \in \{0 \ldots (n-1)\} \begin{bmatrix} case(i)\, of \\ 0 & \Rightarrow T_0 \\ 1 & \Rightarrow T_1(f\,(0)) \\ & \vdots \\ n-1 & \Rightarrow T_{n-1}(f\,(0))(f\,(1)) \cdots (f\,(n-2)) \end{bmatrix}$$

If we have a function F that returns the values T_i for $0 \leq i < n$ given the index i, we can give this type a more concise form. Let $fix(f.b)$ define the fixed point of the program b. That is, $fix(f.b) \equiv Y\lambda f.b$. Given functions F and f, and an index i we can define a function *napply* that applies F to $f(0), \ldots, f(n-1)$:

$$napply \equiv \lambda f.\lambda i.fix(g.\lambda G.\lambda j.\ \mathsf{if}\ j = i\ \mathsf{then}\ G\ \mathsf{else}\ g(G(f(j)))(j+1))$$

The type for F is then $T_F(n) \equiv \{F | i : \{0 \ldots (n-1)\} \to napply(F)(i)(\lambda x.x)(0)\}$.

If we parameterize these types over n, we have the types

$$S_f \equiv n : \mathbb{N} \times T_f(n)$$
$$S_F \equiv n : \mathbb{N} \times T_F(n)$$

The type S_F specifies *descriptions* of dependent products of arbitrary finite arity, and the type S_f specifies the corresponding dependent product type. The inhabitants of S_f are pairs $< n, f >$ where n is the arity of the product, and f is a function with domain $\{0 \ldots (n-1)\}$ that serves as the projection function.

This finite arity dependent product type will be useful to define the type *Signature*. It can also be used to define the type of sequents in a constructive logic: the hypotheses

$$\frac{\Gamma \vdash A\,Type \qquad \Gamma; x : A \vdash B[x]\,Type}{\Gamma \vdash (x : A \bigcap B[x])\,Type} \qquad (TypeFormation)$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma; x : A \vdash B[x] = B'[x]}{\Gamma \vdash (x : A \bigcap B[x]) = (x : A' \bigcap B'[x])} \qquad (TypeEquality)$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash a \in B[a] \qquad \Gamma \vdash x : A \bigcap B[x]\,Type}{\Gamma \vdash a \in (x : A \bigcap B[x])} \qquad (Introduction)$$

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash a = a' \in B[a] \qquad \Gamma \vdash x : A \bigcap B[x]\,Type}{\Gamma \vdash a = a' \in (x : A \bigcap B[x])} \qquad (Equality)$$

$$\frac{\Gamma; u : (x : A \bigcap B[x]); \Delta[u]; x : A; y : B[x] \vdash C[x, y]}{\Gamma; u : (x : A \bigcap B[x]); \Delta[u] \vdash C[u; u]} \qquad (Elimination)$$

Table 1
Inference rules for dependent intersection

of the sequents are defined as a dependent product of finite arity, and the goal is a type that may depend on all hypotheses.

2.6 Later Extensions

2.6.1 Dependent intersection

The general subtyping of record types is captured exactly by the *contravariance* in function space subtyping: $A \sqsubseteq A'$ and $B \sqsubseteq B'$ implies $(A' \to B) \sqsubseteq (A \to B')$. This property makes it possible to express records as intersections of singletons, as illustrated in Section 2.5.4.

To express *dependent records* (see Section 2.5.5) in a similar fashion, a type constructor stronger than simple intersection is needed. Kopylov [74] elaborated an elegant definition of dependent records in terms of a new type constructor that he discovered: *dependent intersections*. Let A be a type and $B[x]$ be a type for all x of the type A. We define *dependent intersection* $x{:}A \bigcap B[x]$ as the type containing all elements a from A such that a is also in $B[a]$. This is a type that we might write as $\{x{:}A | x \in B_x\}$, but $x \in B_x$ is not a well-formed proposition of Nuprl unless it is true.

As an example, let $A = \mathbb{Z}$ and $B[x] = \{y{:}\mathbb{Z} | y > 2x\}$ (i.e. $B[x]$ is a type of all integers y, such that $y > 2x$). Then $x{:}A \bigcap B[x]$ is a set of all integers, such that $x > 2x$.

The ordinary binary intersection is just a special case of a dependent intersection with a constant second argument: $A \bigcap B = x{:}A \bigcap B$.

The dependent intersection is not the same as the intersection of a family of types $\bigcap_{x:A} B[x]$. The latter refers to an intersection of types $B[x]$ for all x in A. The difference between these two type constructors is similar to the one between dependent products $x{:}A \times B[x] = \Sigma_{x:A} B[x]$ and the product of a family of types $\Pi_{x:A} B[x] = x{:}A \to B[x]$.

Two elements, a and a', are equal in the dependent intersection $x{:}A \bigcap B[x]$ when they are equal both in A and $B[a]$. The rules for the new type appear in Table 1.

2.6.2 *Dependent record types as dependent intersections*

Now we will give a method for constructing record types and their elements using dependent intersection. This type is defined by other means in *Nuprl's Class Theory and its Applications* [32], and there are other quite different accounts [16,101].

Records Elements of record types are defined as in [32]. They map labels to the corresponding fields. We can build up these functions component by component starting with an empty record. We write the extension (or field update) as $r.x := a$; think of assigning a to $r.x$, where x is a label.

$$\{\} \equiv \lambda(l.l) \qquad \text{(the empty record)}$$

We could pick any function as a definition of an empty record.

$$(r.x := a) \equiv \lambda(l.\texttt{if } l = x \texttt{ then } a \texttt{ else } r\ l) \qquad \text{(field extension)}$$
$$r.x \equiv r(x) \qquad \text{(field extraction)}$$

Now we use a common notation for the elements of record types.

$$\{x_1 = a_1; \ldots; x_n = a_n\} \equiv \{\}.x_1 := a_1.x_2 := a_2. \ldots .x_n := a_n$$

These definitions provide that

$$\{x_1 = a_1; \ldots; x_n = a_n\} = \lambda l.\texttt{if } l = x_1 \texttt{ then } a_1 \texttt{ else}$$
$$\vdots$$
$$\texttt{if } l = x_n \texttt{ then } a_n$$

Record types We will show how to build finite record types by intersection. First we define a singleton record.

$$\{x{:}A\} \equiv \{x\} \rightarrow A \qquad \text{(single-field record type)}$$

where $\{x\} \equiv \{l{:}Label \mid l = x \in Label\}$ is a singleton set.

$$\{R_1; R_2\} \equiv R_1 \bigcap R_2 \qquad \text{(independent concatenation of record types)}$$

Thus $\{n{:}Nat;\ x{:}Real\}$ is $\{n{:}Nat\} \cap \{x{:}Real\}$.

Next we include dependency.

$$\{self{:}R_1; R_2[self]\} \equiv self{:}R_1 \bigcap R_2[self] \qquad \text{(dependent concatenation)}$$

Here *self* is a variable bound in R_2. We will usually use the name "self" for this variable and use the shortening $\{R_1; R_2[self]\}$ for this type. Further, we will omit "*self*." in the body of R_2, e.g. we will write just x for $self.x$, when such notation does not lead to misunderstanding. [5]

We assume that this concatenation is a left associative operation, and we will omit inner braces. For example, we will write $\{x{:}A;\ y{:}B[self];\ z{:}C[self]\}$ instead of $\{\{\{x{:}A\}; \{y{:}B[self]\}\}; \{z{:}C[self]\}\}$.

Note that in this expression there are two distinct bound variables *self*. The first one is bound in B, and refers to the record itself as a record of the type $\{x{:}A\}$. The second *self* is bound in C; it also refers to the same record, but it has type $\{x{:}A; y{:}B[self]\}$.

[5] Note that Allen's notational devices allow Nuprl to display these terms as written here [7].

The definition of independent concatenation is just a special case of dependent concatenation, when R_2 does not depend on *self*. We can also define *right associating records,* a less natural idea, but nevertheless expressible:

$$\{s{:}x{:}A; R[s]\} \equiv self{:}\{x{:}A\} \bigcap R[self.x]$$

Here x is a variable bound in R that represents a field x. Note that we can α-convert the variable s, but not the label x, e.g., $\{s{:}x{:}A; R[s]\} = \{y{:}x{:}A; R[y]\}$, but $\{s{:}x{:}A; R[s]\} \neq \{y{:}z{:}A; R[y]\}$. We will usually use the same name for labels and corresponding bound variables. This connection is right associative, e.g., $\{s : x{:}A; t : y{:}B[s]; z{:}C[s,t]\}$ stands for $\{s : x{:}A; \{t : y{:}B[s]; \{z{:}C[s,t]\}\}\}$.

These dependent records can be used to capture theories built in a context. For example, here is how a *monoid* could be defined:

$$\begin{array}{l}\{\ M{:}Type_i\ ;\\ \quad op{:}M \times M \to M;\\ \quad id{:}M;\\ \quad assocax : \forall x, y, z{:}M.\ op(x, op(y, z)) = op(op(x, y), z);\\ \quad idax : \forall x{:}M.\ (op(x, id) = x\ \&\ op(id, x) = x)\ \ \}.\end{array}$$

Here we use the fact that propositions are types — another key Automath contribution.

2.7 Unions, Singletons, and Mainstream Sets

Given the utility and elegance of the intersection operator, and given the fact that those two properties derive in large part from $\cap$'s position as the infimum of the subtyping order, it is natural to wonder what we could accomplish if we also had the dual. The union type operator has, in fact, recently been introduced into the CTT family ([58,76]). Guided and inspired not only by the supremum operation of order theory but also by mainstream set theory's union, the present implementation is a good compromise in the face of the infeasibility of getting the best of both those two worlds.

Part of the compromise is uneven compatibility with set-theoretic expectations, as exemplified by the fact that a union $A \cup B$ can have fewer distinct members than do its branches A and B. That fascinating information-hiding phenomenon is characteristic of general unions, but we will not pursue it further since the rest of this paper only requires restricted and unsurprising applications of $\cup$. Specifically, each collection of types that we will union together will be pairwise disjoint, and that special case of the operator is reliably like the familiar ZFC union, both in its behavior and in its versatility.

As hoped, these two new ways of combining types, intersection and union, do greatly enrich the foundational system, and the community is still in the early stages of exploiting them. Kopylov, for instance, is using unions and intersections to describe objects ([76]). Another ongoing effort is the use of the enhanced language to re-explain types in a way that creates a pervasive, bi-directional connection between CTT and ZFC.

In the limited space remaining, a preview of the latter project is presented. Our focus here is on one central aspect of the re-explanation: the fact that types are constructed from the bottom up, using unions and intersections as mortar and using singletons as stones. To see what is meant by that, consider the following unusual rendition of

$\mathbb{N}_2 \to \mathbb{N}_2$.[6] It has a superficial similarity to the analogous ZFC function space, $2 \to 2$, since it is the union of four (disjoint) things. If we look more closely, we see that the similarity to the native ZFC function space $2 \to 2$ goes much further than that; the two match all the way down to the leaves.

$$\bigcup \left[\begin{array}{ll} \bigcap \left[\begin{array}{l} \{0\} \to \{0\} \\ \{1\} \to \{1\} \end{array} \right. & \left\{ \begin{array}{l} (0,0) \\ (1,1) \end{array} \right. \\ \bigcap \left[\begin{array}{l} \{0\} \to \{0\} \\ \{1\} \to \{0\} \end{array} \right. & \left\{ \begin{array}{l} (0,0) \\ (1,0) \end{array} \right. \\ \bigcap \left[\begin{array}{l} \{0\} \to \{1\} \\ \{1\} \to \{0\} \end{array} \right. & \left\{ \begin{array}{l} (0,1) \\ (1,0) \end{array} \right. \\ \bigcap \left[\begin{array}{l} \{0\} \to \{1\} \\ \{1\} \to \{1\} \end{array} \right. & \left\{ \begin{array}{l} (0,1) \\ (1,1) \end{array} \right. \end{array} \right.$$

Thinking of them in this way, we could say that they're structurally identical, although the one on the left is admittedly much more heavily decorated.

To see why the left-hand type expression and $\mathbb{N}_2 \to \mathbb{N}_2$ are extensionally equal, consider the union's third branch. That intersection's top half, $\{0\} \to \{1\}$, says that 0 maps to 1, while its bottom half, $\{1\} \to \{0\}$, says that 1 maps to 0. It is thus a brute-force description of negation. The identity function and the two constant functions are visible in the remaining branches.

Doug Howe, in [64,66,67], essentially showed that the connection illustrated above can be developed into a comprehensive system, one that encompasses not only finite function spaces like $\mathbb{N}_2 \to \mathbb{N}_2$ but also infinite function spaces, higher-order function spaces, dependent function spaces, dependent products, universes, W types, special cases of the intersection, union, and recursion operators, and even a variant of the quotient operator. The result underlies a novel set-theoretic semantics of a new, classical member of the CTT family.

The semantic framework discussed here is an extension and re-examination of his work, taking different approaches in many areas (in particular, by focusing on singletons instead of oracles) and aiming to ultimately explain a richer collection of types. The core ideas and methods are still the same, though. To get a glimpse of the mechanics, let's first note that the metatheory is ZFC + $\exists$infinitely many inaccessibles, abbreviated ZFCI$^\omega$, and that the meaning of a type T with κ equivalence classes is a set $[\![T]\!]$ with exactly κ elements.

Although entirely conventional up to this point, perfectly in line with [110, ch. 9] and [46], if we attempt to continue with a straightforward mapping we immediately encounter a snag. Namely, the set $[\![\{0\} \to \{0\}]\!]$ must contain exactly one element, but the type $\{0\} \to \{0\}$ contains terms such as $\lambda x.0$ and $\lambda x.x$ whose meanings must be distinct. It turns out that this counterexample is typical in the sense that the challenges that we face all boil down to difficulties with singletons. The essential problem is $[\![S]\!]$ when S is a singleton, and the counterexample shows that this can be perplexing even at the level of the oldest and simplest types.

[6] Throughout this section, we work modulo extensional type equality.

The solution is to abandon, at least initially, the idea that $[\![S]\!]$ will be a set of things that inhabit S. Instead, $[\![S]\!]$ is no more (and no less) than a description of how S can be built from other singletons. In a bit more detail, the unique member of $[\![S]\!]$ is a well-founded tree, the encoding of which is not particularly important. The tree's rank can be (and typically is) greater than several inaccessible cardinals. At each leaf, there is a code for either the singleton Top or a concrete singleton like $\{1\}$. At each internal node, there is a code for a type operator such as $\cdot \to \cdot$, $\cdot \times \cdot$, $\cap$, or $/\!/_{\mathsf{Unit}}$.

These trees, these blueprints for constructing singletons, have a number of useful and intriguing properties. First and foremost is the comprehensive, pervasive match that often exists between the blueprint and its native ZFCI$^\omega$ counterpart. This is true not only for function spaces, as illustrated with $\mathbb{N}_2 \to \mathbb{N}_2$ earlier, but also for products, disjoint unions, and so on, and it is the basis of Howe's justification of bi-directional sharing between STT and his classical CTT.

This singleton-based rendition of type theory is also remarkable for the fact that types are constructed bottom-up, building singletons from singletons via a fairly limited and orthogonal collection of type operators. It seems plausible that, with sufficient effort, this approach could be developed into a full-fledged, alternative formulation of a rich successor to Howe's classical CTT. Due to the percolation upward from mainstream set theory, we might aspire to a foundation that is compact, stable, and directly connected to ZFC.

2.8 A Logic of Events

As a result of working with system designers on the specification, design, and verification of distributed systems [19,85,56], we have developed a theory of event structures [17] that is the basis for extending CTT and the extending the method of correct-by-construction programming. We present distributed systems from the point of program synthesis that we helped create in the 80's. Logical synthesis of programs in constructive logic uses the *propositions-as-types principle*: a specification is a proposition and a constructive proof of the proposition yields a member of the corresponding type [14]. Because we can adequately characterize a sequential computation as a function from its input parameters to its output value, a computational theory of function types is a sufficient basis for logical synthesis of sequential programs. Specifications are propositions that correspond to function types and programs are members of these function types.

From Lamport's work [83,84], we know that to characterize a distributed computation we need a fundamental model of *messages*, i.e., information flow between agents. This naturally introduces abstract agents and temporal concepts into our logic since a message is sent from some agent at some time and is received by another agent at some, causally later, time. These aspects of distributed computation are not naturally modeled as functions and, hence, a distributed program is not merely a member of some function type. Instead, distributed algorithms are specified by describing the desired set S of possible runs, and a distributed program p satisfies the specification if all of its runs are in the set S. Thus a specification is a proposition about runs and a program is something that constrains the set of possible runs.

A *system* is a set of runs. A protocol p generates a system, namely the set of all runs

consistent with p. Again, following Lamport, we characterize a run of a program as a set of events. Each event is associated with a unique agent. When we want to speak more abstractly about *events happening* before the agent is fully defined, we talk about the events at a *location*. A location is an abstraction of the notion of an *agent* or a *process*. Formally, events are elements of a type E, and there is a function loc of type $E \rightarrow Loc$, where Loc is some set of locations. For each $i \in Loc$, the set of events e such that loc(e) = i is totally ordered. Intuitively, this set of events is the *history* of events at location i. If first(e) holds, then e is the first event in the history associated with loc(e); if not, then e has a predecessor pred(e). Events are further partitioned by their *kinds*. The kind(e) of event e is either rcv(l) – a receive event on *link* l, or else local(a) – a local event of kind a.[7] Every receive event has a *sender event* sender(e). The sender event of a message is the event when the message was sent.

In this setting we can define Lamport's [83] *causal order* on events as the transitive closure of the sender-receiver and predecessor relations. The local state of a process is represented as the values of a collection of state variables. Formally, state variables are just identifiers that are assigned a value at each event.[8]

There are binary functions when and after that describe the values of state variables before and after an event takes place. We typically write these functions using infix notation. Thus, if loc(e) = i then (x when e) describes the value of the state variable x at location i just before e, and (x after e) describes its value after e. Note that state variables are local variables and two locations may have state variables with the same name.

Every event e also has a *value* val(e). The value of a receive event is the message that is received, and the value of a local event represents a value (satisfying some constraints) chosen (non-deterministically) when the local event is generated. For example, if whenever a local event of kind a occurs an integer value is chosen and a message with twice that value is sent on link l, then events with kind(e)=rcv(l) and kind(sender(e))=local(a) will have val(e)= 2∗val(sender(e)).

Our six axioms of event structures say that an event is the sender of only a finite number of messages; the predecessor function is one-to-one; causal order is well-founded; the local predecessor of an event has the same location; the sender of an event has the location of the source of the link on which the message was received; and state variables change only at events, so that: (x after pred(e)) = (x when e).

Working in a powerful higher-order logic, we are free to define further derived concepts on top of the event structure model. An event structure is a rich enough structure that we can define various "history" operators that list or count previous events having certain properties. Because we can define operators like these we do not need to add "history variables" to the states in order to write specifications and prove them. For example, the basic history operator lists all the locally prior events at a location.

$$\mathsf{before}(e) = \ \mathsf{if}\ first(e)\ \ \mathsf{then}\ \ \mathsf{nil}\ \ \ \mathsf{else}\ pred(e) :: \ \mathsf{before}(pred(e))$$

[7] Receive events are further partitioned by a *tag* so that we can restrict the kinds of events sending messages on a given link with a given tag without restricting other uses of that link. To simplify the discussion in this paper, we have suppressed all mention of these tags.

[8] State variables are typed, but to simplify our discussion we suppress all type declarations.

We can also define useful notations for concepts such as "event e changes state variable x"

$$x\ \Delta\ e \equiv (x\ \ \mathsf{after}\ e \neq x\ \ \mathsf{when}\ e)$$

This language is a theoretical contribution that enables natural declarative descriptions of distributed systems, and it is a basis for formal verification. Now we need to examine the issue of code synthesis.

2.8.1 Worlds and Executable Code

In order to relate the event structure abstraction to executable code, we define a general model of the execution of a distributed system. We call instances of this general model *worlds*. Any concurrent programming language can be given a semantics in terms of its possible worlds, and every world gives rise to an event structure. We are now using one concurrent programming language that we call *message automata* (and their coding as Java programs). These automata are closely related to the IO-automata defined by Lynch [89] but have a built-in syntax for messages and a built-in mechanism, the frame conditions, to allow simple extension. We will summarize below how the semantics of message automata is defined in terms of possible worlds.

Definition of World A *world* is an idealized trace of the execution of a distributed system. It has locations, and links from a graph $\langle Loc, Lnk, src, dst : Lnk \to Loc\rangle$. In contrast to an event structure, in which time is only a partial order, a world has a notion of global, discrete time, modeled as the natural numbers $\mathbb{N}$. This global time is not expressible in the logic of events and is only used to construct these models. By observing the system at every location i and every time t, we have a *state* $s(i,t)$, an *action* $a(i,t)$, and a list of *messages* $m(i,t)$.

The state $s(i,t)$ is the state of the part of the system at location i at time t. The state at location i is a general record (a dependent function from names to values).

The action $a(i,t)$ is the action that was chosen by the system to be executed next at location i and time t. It may be a null action, indicating that no action was taken at $\langle i,t\rangle$. Other actions will be local actions and the action of receiving a message. Every action has a *kind* of one of these forms (null, local, or receive), and it also has a *value* whose type depends on the kind and location of the action.

The messages $m(i,t)$ are the list of messages sent from location i at time t. As in event structure, a message consists of a link, a tag, and a value.

Event System of a World If w is a world, then we can construct an event structure from w. The basic construction is to let the events E be the points $\langle i,t\rangle$ in spacetime such that the action $a(i,t)$ in world w is not null. So E is the set of points where an event occurred. We then define all the operations $<_{loc}$, $\prec$, $first$, $pred$, when, after, etc., to construct the event structure $Ev(w)$ of world w. *It is then a theorem that* $Ev(w)$ *satisfies all the axioms for event structures.*

Note, our logic is not about worlds. They are part of the meta-reasoning about execution. We reason about processes which we turn to next.

2.8.2 *Message Automata*

Event structures and worlds are infinite objects, but they arise from the behaviors of *finite* distributed programs. We call our representations of these finite programs *message automata*. A message automaton is a finite collection of declarations and clauses each of which has an assigned location. The *declarations* provide the names and types of state variables, local actions, input messages, and output messages. The *clauses* **initialize** variables, *state* **preconditions** on local actions, give the **effects** of actions on state variables, and give the *output* messages **sent** by actions. In addition, two further kinds of clauses, called *frame conditions*, restrict the kinds of actions that affect **only** a given state variable or send *only* on a given link. [9] For example, the following message automaton was automatically assembled from a proof of an algorithm for leader election in a ring.

state $me : \mathbb{N}$; initially $uid(i)$
state $done : \mathbb{B}$; initially $false$
state $x : \mathbb{B}$; initially $false$
action $vote$; precondition $\neg done$
  effect $done := true$
  sends $[msg(out(i), vote, me)]$
action $rcv_{in(i)}(vote)(v) : \mathbb{N}$;
  sends if $v > me$ then $[msg(out(i), vote, v)]$ else $[]$
  effect $x :=$ if $me = v$ then $true$ else x
action $leader$; precondition $x = true$
only $rcv_{in(i)}(vote)$ affects x
only $vote$ affects $done$
only $\{vote, rcv_{in(i)}(vote)\}$ sends $out(i), vote$

The semantics of a message automaton D is the set of possible worlds w that are consistent with it. To be consistent, w must be a fair-fifo world and respect the meanings of the clauses of the message automata at each location. We have a formal definition $Consistent(D, w)$ of this relation, and this defines the semantics of distributed systems of message automata.

Rules for Message Automata A message automaton constrains the possible worlds that can be executions of the system. We state these constraints as rules on the event structures that come from the possible worlds. A *rule* of the form $M : \psi$ means that:

$$\exists w : Consistent(M, w). \forall D : Dsys.\ \forall w : World.$$
$$Consistent(D, w) \ \wedge \ M \ \subseteq \ D \ \Rightarrow \ Ev(w) \models \psi$$

It says that there is at least one event structure of a possible world for M and the event structure of any possible world of any distributed system containing M will satisfy ψ. Since there are only six kinds of clauses in message automata, *we have a set of only six*

[9] Actually, send on a given link with a given tag.

basic rules and one composition rule. For example, the rules for initialization clauses and effect clauses are

$$@i \;\; \text{state } x : T; \;\; \text{initially } x = v :$$
$$\quad \forall e@i.\; first(e) \;\Rightarrow\; x \;\; \text{when } e \;=\; v$$
$$@i \;\; \text{state } x : T1; \;\; \text{action } k : T2;$$
$$\quad k(v) \;\; \text{effect } x := f(s, v) :$$
$$\quad \forall e@i.\; kind(e) = k \;\Rightarrow\; x \;\; \text{after } e \;=\; f(s \;\; \text{when } e, val(e))$$

2.8.3 Implementation of the Logic of Events and Extraction of Process Code

Our current Logical Programming Environments supports proof and program development by top down refinement of goals into subgoals and bottom up synthesis of programs from fragments of code derived from proofs of subgoals. We are extending this mechanism, called *program extraction*, to the synthesis of *processes* to support *process extraction*.

Our library of declarative knowledge about distributed systems will contain many theorems that state that some property ϕ of event structures is *realizable* (which we write as $\models \phi$.) A property ϕ is realizable if and only if there is a distributed system D that is both *feasible*—which implies that there is at least one world and, hence, at least one event structure, consistent with D—and realizes the property; every event structure consistent with D satisfies the property.

The basic rules for message automata provide initial knowledge of this kind—all the properties of single clauses of message automata are realizable. We add to our knowledge by proving that more properties are realizable. In these proofs, the system will automatically make use of the knowledge already in the library when we reach a subgoal that is known to be realizable.

To make this automated support possible, some new features, which we believe are unique to the Nuprl system, have been added to the system. In order to motivate a discussion of these features, let us examine in detail the steps a user takes in proving a new realizability result.

Suppose that we want to prove that ϕ is realizable, and we start a proof of the top-level goal $\models \phi$. From the form of the goal, the proof system knows that we must produce a feasible distributed system D that realizes ϕ so it adds a new abstraction $D(x, \ldots, z)$ to the library (where $x, \ldots, z$ are any parameters mentioned in ϕ). The new abstraction has no definition initially—that will be filled in automatically as the proof proceeds. This initial step leads to a goal where from the hypothesis that an event structure es is consistent with $D(x, \ldots, z)$ we must show the conclusion that $\phi(es)$, i.e., that es satisfies ϕ .

Now, suppose that we can prove a lemma stating that in any event structure, es, $\psi_1(es) \wedge \psi_2(es) \Rightarrow \phi(es)$ (the proof of this might be done automatically by a good decision procedure for event structures or interactively in the standard way). In this case, the user can refine the initial goal $\phi(es)$ by asserting the two subgoals $\psi_1(es)$ and $\psi_2(es)$ (and then finishing the proof of $\phi(es)$ using the lemma).

If ψ_1 is already known to be realizable, then there is a lemma $\models \psi_1$ in the library and,

since the proofs are constructive, there is a realizer A_1 for ψ_1. Thus to prove $\psi_1(es)$, it is enough to show that es is consistent with A_1, and since this follows from the fact that es is consistent with $D(x, \ldots, z)$ and that $A_1 \subset D(x, \ldots, z)$, the system will automatically refine the goal $\psi_1(es)$ to $A_1 \subset D(x, \ldots, z)$. If ψ_2 is also known to be realizable with realizer A_2 then the system produces the subgoal $A_2 \subset D(x, \ldots, z)$, and if not, the user uses other lemmas about event structures to refine this goal further.

Whenever the proof reaches a point where the only remaining subgoals are that $D(x, \ldots, z)$ is feasible or have the form $A_i \subset D(x, \ldots, z)$, then the proof can be completed automatically by defining $D(x, \ldots, z)$ to be the join of all the A_i. In this case, all the subgoals of the form $A_i \subset D(x, \ldots, z)$ are automatically proved, and only the feasibility proof remains. Since each of the realizers A_i is feasible, the feasibility of their join follows automatically from the pairwise compatibility of the A_i and the system will prove the pairwise compatibility of the realizers A_i automatically if they are indeed compatible, in which case the proof of the realizability of ϕ is complete and its realizer has been constructed and stored in the library.

How might realizer A_1 and A_2 be incompatible? For instance, A_1 might contain a clause that initializes a state variable x to true while A_2 contains a clause that initializes the same variable to false. Or, A_1 might declare that an action of kind k_1 has an effect on x while A_2 declares that only actions of kinds k_2 or k_3 may affect x.

If the variable x occurs explicitly in the top goal ϕ then the user has simply made incompatible design choices in his attempt to realize ϕ and must change the proof. However, if the variable x is not explicitly mentioned in ϕ then it is the case that x can be renamed to y without affecting ϕ. It is often the case that x can be renamed independently in the proofs of the subgoals ψ_1 and ψ_2 (say to y and z) and hence the realizers $A_1(y)$ and $A_2(z)$ will no longer be incompatible.

Incompatibilities such as these can arise when names for variables, local actions, links, locations, or message tags that may be chosen arbitrarily and independently, happen to clash. Managing all of these names is tedious and error prone, so we have added automatic support for managing these names.

By adding some restrictions to the definition mechanism, we are able to ensure that the names inherent in any term are always visible as explicit parameters. We have also defined the semantics of Nuprl in such a way that the *permutation rule* is valid. The permutation rule says that if proposition $\phi(x, y, \ldots, z)$ is true, where $x, y, \ldots, z$ are the names mentioned in ϕ, then proposition $\phi(x', y', \ldots, z')$ is true, where $x', y', \ldots, z'$ is the image of $x, y, \ldots, z$ under a permutation of all names.

Using the permutation rule, our automated proof assistant will always permute any names that occur in realizers brought in automatically as described above, so that any names that were chosen arbitrarily in the proof of the realizability lemma but were not explicit in that lemma will be renamed to fresh names that will not clash with any names chosen so far. This strategy is supported by the primitive rules of the logic and it guarantees that any incompatibility in the realizer built by the system was inherent in the user's design choices and was not just an unfortunate accident in the choice of names.

3 System Architecture

The architecture of the Nuprl [3,78,100] system is the product of many evolutions aimed at providing a theorem proving environment as rich and robust as its type theory. The resulting implementation composes a set of communicated processes, distributing tasks amongst the varied components. The independent communicating processes are centered around a common knowledge base, called the *library*. The library contains definitions, theorems, inference rules, and meta-level code (e.g. tactics), and serves as a transaction broker for the other processes. Those processes include inference engines (*refiners*), user interfaces (*editors*), rewrite engines (*evaluators*), and *translators*. Translators between the formal knowledge stored in the library and, for instance, programming languages like Java or Ocaml [79,80] allow the formal reasoning tools to supplement real-world software from various domains and thus provide a *logical programming environment* for the respective languages.

Multiple instances and kinds of processes can connect to the library, and can do so at any time during the user's session. This allows for multiple refiners in a session so that proofs can be refined in parallel, for example. At its core, the Nuprl system is based upon a transactional library, connected components, or clients, and library-client communications.

3.1 Transactional Library

The ways in which knowledge is stored, structured, changed, and in general managed, stem from desires to provide users with a reliable robust and sustainable system.

All content in the library is stored in the form of library objects. The library maintains a mapping of *abstract object identifiers* to *object content*. Abstract object identifiers prevent name collision problems, since their values cannot be imposed on the library. Instead, unique object identifiers are created by the library while mnemonic names chosen by users are included as simply part of an object's content, and thus need not be unique. When an object refers to another object, it contains that object's identifier. The mapping of object identifiers to object content can be viewed as a graph where an identifier occurring in some object constitutes an edge from this object to the identified object.

Apart from the object map and embedded abstract links to objects, the library does not impose any predefined organization. The absence of a predefined library organization is a prerequisite for integrating formal knowledge from other systems besides Nuprl without requiring these systems to change their representation structure. It is expected that contributors define their own organization of content using the simple but powerful primitives provided in the library. Translators of formal knowledge relating content of disparate systems would require specialized methods of accessing each systems content.

So that users need not worry about losing unsaved work, the library uses a transactional database to make objects persistent. All changes to object contents are committed to the database when a transaction ends, ensuring that knowledge doesn't get lost in case of a system failure.

In the intended use of the library to support mathematical knowledge management, it is expected that clients will be reading content repeatedly. To facilitate this use, the

library implements a *distributed lookup table* to allow clients to maintain local projections of the library content. A projection is a mapping of a subset of library content. Clients can tailor the content in their local tables by defining their own projections. The library maintains the consistency of the projection by broadcasting updates after modification of objects within the subset of the projection.

The library database is constructive in the sense that it never overwrites content. Instead the object identifier is rebound to newly constructed object content. The chain of bindings is preserved so that old versions may be accessed. This allows recovery of user data corrupted or destroyed erroneously. The original content is preserved until it is deleted during garbage collection of the database. Garbage collection must be explicitly invoked and retains a parameterizable number of generations of the library.

The ACID model is an old and important concept that sets prerequisites for the design of a reliable database. Drawing from this standard, we designed the knowledge base to be sure to provide the four ACID database properties.

Atomicity. Each object entry in the library table has three attributes. A *sticky bit* marks the object as a root object, which prevents it from being deleted during garbage collection. A *liveness bit* indicates whether an object can be included in a projection. The *content* is the term data associated with the object.

All persistent modifications to the library are performed via six fundamental *directives*: *Bind* and *unbind* modify the binding of object identifiers to content. *Allow* and *disallow* modify the sticky bit. *Activate* and *deactivate* modify the liveness bit.

There are many functions to manipulate properties and projections of content, but such modifications must be made persistent by unbinding the old content from the object identifier and binding the new content to it.

Having a small symmetric set of basic commands provides for easy implementation of atomicity. As modifications are performed within a transaction the library remembers which command was called (for an unbind it also remembers content prior to unbind). If a transaction fails, then modifications can be undone simply by performing the inverse operation in reverse order. If a transaction succeeds then the committed commands are written to a log.

Each deactivate or activate generates a broadcast of a conditional update. Each undo or commit will then generate a broadcast to either retract or confirm the update. Broadcasts are filtered and routed to clients depending on the projections they request. It is required that an object be inactive at bind or unbind. Thus the following sequence of actions is required to modify an object: deactivate, which generates a conditional delete broadcast; unbind old content; bind new content; activate, which generates a new conditional insert broadcast; and finally commit will generate appropriate confirm broadcasts for the delete and insert updates.

Consistency By extracting access list updates during the scope of a request, the *access list* is used to record dependencies between objects. For example, by extracting the accesses recorded during a refinement the lemma dependencies of the inference created can be recorded. Dependency accounting data are stored with the object content. If clients are robust in reporting such dependencies, dependency accounting can

be a powerful tool to convert ephemeral runtime dependencies into persistent links between objects.

Dependency links and object content time stamps provide the ability to detect stale dependencies. If a referenced object has a time stamp later than the referencing object, then some doubt is cast on the validity of the referencing object. Similar tests could be used to cause failure if such conditions are detected at modification of the referenced object. This would prevent stale dependencies, but it also requires more forethought when attempting modification of referenced objects. In the current implementation we have opted to allow inconsistent states, but have provided utilities for detecting such states and refreshing affected objects to resolve them. The concept of *certificates* (see Section 4.3.2) addresses similar concerns in a more rigorous manner.

Isolation Serializability is provided by locking objects. Current implementation uses only write locks and a transaction will fail if it attempts to gain a lock on an object held by another transaction. Read locks are not used but instead Nuprl supports multiple views. Modifications of objects by concurrent transactions are not visible to other transactions. At commit time, modified content is stamped. Each time an object is accessed the transaction remembers the stamp at access time. While Nuprl can record library lookup of objects, clients must also report lookups in distributed tables for the access list to be complete. After a transaction is complete, it can determine if any objects accessed were modified by a concurrent transaction which has since committed. Thus dirty reads are allowed, but can be detected. Given the nature of the expected content it is expected that transactions will read many objects and write few.

Durability The database provides durability for a library. A *library instance* is created from a log in the database. A log is a journal of committed directives. A user opens a library by choosing a log from the database, and a library process can then construct a library instance from the log. The content terms are stored separately from the log and thus each log entry consists of an indication of the operation, the object identifier, and possibly a pointer to the content. The log entries are represented as terms and the database is used to store logs as a sequence of terms. When a log is closed, an abbreviated copy is made containing only the entries needed to produce the library instance. If a library process fails, a new library instance can be produced from the raw log. In this manner durability is provided. When a new library instance is created, a new log is created as well. If the instance was created from a properly closed log, then the first entry will be a persistent pointer to the old log. Otherwise, the new log is initialized with the state of the library after the old log has been completely read.

3.2 Editing and Refining Proofs

Besides the library, the Nuprl system includes as native clients an editor and a Nuprl refiner for editing and refining proofs, respectively. While one process of each is the most standard configuration, multiple editor and refiner processes can be connected to a single library process. For example, multiple editors can allow for cooperative work on a shared library, and multiple refiners allow for parallel refinement

The Nuprl editor is a structure editor suited to manipulating terms. It provides many features to facilitate browsing and modification of library content and offers special support for editing terms, definitions, theorems, programs, proof tactics, etc. As a structure editor it enables the user to modify terms in a structured fashion by marking subterms and editing slots in the displayed term. Users may also follow hyperlinks between object identifiers contained in a term and the object they refer to.

The Nuprl refiner implements the formal system described in Section 2. It contains a *rule compiler* that translates rule objects contained in the library into inference mechanisms, which makes it possible to extend the logic in the future without re-implementing the refiner. The refiner can also extract programs from proofs and evaluate them.

JProver [81,82,104], a complete theorem prover for first-order intuitionistic and classical logic, is included as an extension to the Nuprl refiner. To ensure validity of a proof built from refiners of disparate logics, JProver is used as a heuristic, and the proof terms are translated and rerun in the Nuprl system. The communication between Nuprl and JProver utilizes the MathBus design [93].

3.3 Communication between the Components

The advantages of an open distributed system are vast, providing us not only with the robustness and flexibility we aimed for, but also with an excellent division of labor, particularly useful for targeting mathematical knowledge management. Still, managing the communication between the library and its clients required a significant evolution of earlier versions of the Nuprl system.

In our open architecture, refiner and editor processes will be repeatedly reading content. To facilitate this, the library has robust methods to push object content projections to clients. In our implementation, clients may subscribe to various projections and kinds of content. The library distributes lookup tables to the clients at the time of connection, and sends updates appropriately as changes are made. This minimizes load on the library and maintains the correct state for all processes.

All interprocess communication is done via the exchange of terms. Each process maintains a list of links to connected processes. Currently, term transfer is done over TCP sockets. The terms are encoded into a byte stream on output and then decoded back to a term on input.

Work is accomplished in Nuprl by evaluating requests. Requests must be evaluated in the scope of an *environment*. An environment is characterized by its subscriptions and what requests it can evaluate.

Just as an environment will process requests, it can make requests of other environments. Requests are sent by marshalling the request as a term and then routing the request to a connected environment.

Nuprl also supports a fixed API with predefined commands. The fixed API is suitable for clients which do not wish to support the distributed table features. Such clients could be used for reading Nuprl content or for batch imports of content into Nuprl. There are provisions in the API for a client to provide limited services to Nuprl.

To tightly integrate a system as a Nuprl client, several protocols need to be supported. The message protocol specifies the form and valid sequences of all messages. This protocol includes fields for relating messages to transaction scope and describes forms for delimiting the extent of transactions. The distributed table protocol specifies the form of broadcast messages used to update the distributed tables. The evaluation protocol specifies the form of messages used to make requests, return responses and report distributed table lookups. A client is not required to implement an evaluator in the library's native language (ML). Instead, the library can be extended to build client-specific commands to access client services provided the commands are expressible as terms.

4 A Formal Digital Library

It is very gratifying to researchers in automated reasoning that a great deal of formal mathematics has now been created using interactive theorem provers. There are probably as many as 50,000 formally proved theorems of interest among the files of a dozen major provers.

In the mid 90's, Doug Howe and his collaborators at Bell Labs who were using HOL and Nuprl on various projects decided that it was more efficient to carry out the basic theoretical and practical work needed to share mathematics across these provers than it was to rebuild the results separately. They worked to this end, showing how to embed HOL in Classical Nuprl [65,64] and using the combined library in their work [48,47]. We built on this work to also include PVS results in a library of formal mathematics [3]. Out of that work and our work on the Nuprl System, emerged our project on a Formal Digital Library (FDL). The FDL allows for multiple logics, and the integration of multiple support tools and services. discuss some of our results and contributions in this section. This is an on-going effort of our research group [30], and it ties us into the Mathematical Knowledge Management community.

4.1 FDL Design Goals

Mathematical Knowledge Management is concerned with providing access to mathematical knowledge in order to facilitate the creation or discovery of mathematics, facilitate mathematics education, and make feasible intelligent information retrieval in large bodies of digital mathematics. To better serve these goals, a digital library of formalized mathematical and algorithmic knowledge should meet the following objectives.

Connectivity The FDL must be able to connect to multiple clients (proof tools, users, etc.) independently, asynchronously, and in parallel.

Usability Clients of the FDL must be able to browse library contents, search for information by a variety of search criteria, and contribute new knowledge to the library.

Interoperability The FDL shall support the cooperation of proof systems in the development of formal algorithmic knowledge. Different proof systems will be based on different formal theories and on different internal representations of knowledge. The

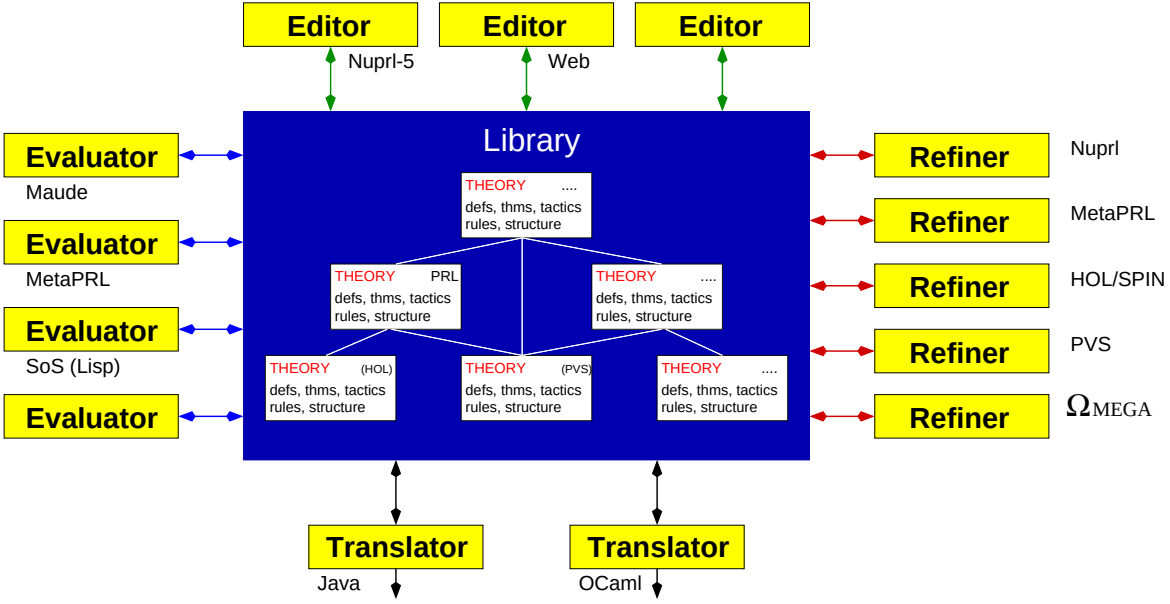

Fig. 1. FDL distributed open architecture

representation of knowledge in the FDL has to be generic, so that it can be translated into a large variety of formats when providing knowledge to clients or receiving formal knowledge from them.

Accountability The FDL needs to be able to account for the integrity of the formalized knowledge it contains. As it supports interoperability between very different proof tools, there cannot be an "absolute" notion of correctness. Instead, the FDL has to provide justifications for the validity of proofs, which will depend upon what rules and axioms are admitted and on the reliability of the inference engines employed. Furthermore, these justifications must be exposed to determine the extent to which one may rely upon the provided knowledge. We call these justifications *certificates*.

Information Preservation The FDL has to guarantee that existing knowledge and justifications cannot be destroyed or corrupted by clients or system crashes.

Archiving The FDL has to support the management of knowledge on a large scale such as merging separate developments of large theories and performing context-specific tasks. This requires the use of abstract references to knowledge objects, or *abstract identifiers*, as traditional naming schemes do not scale.

We have extended the Nuprl library described in Section 3.1 to a prototype FDL that interfaces with Nuprl refiners and editors and other tools and systems that support formalized mathematical reasoning and knowledge. The FDL system is depicted in Figure 1. Multiple logics are accommodated via translators, which can map subsets of one logic onto another [65,64,48,4]. Throughout this ongoing project, we have studied concepts, core functionalities and features of formal digital libraries, which we will discuss in the rest of this section. The latest version of the Nuprl library was in fact designed for the intended use of mathematical knowledge management. Constructing the FDL independently from the Nuprl Refiner and Editor as a fully autonomous system, and integrating many of the concepts discussed below provided us with an excellent prototype upon which to innovate further to meet the needs of mathematical

knowledge management. Furthermore, it allowed us to enhance the Nuprl system, because we can connect Nuprl refiners to it to get the benefits of the additional FDL functionalities. While concepts including abstract identifiers (Section 4.2.1) and general accounting mechanisms have been fully implemented into this prototype, closed maps (Section 4.2.2) and certificates have not yet been made available to the users, while their design is complete.

4.2 Organizing Concepts

4.2.1 Abstract Identifiers

The usual method of interaction with the FDL is to build and develop a *client work space*, i.e. a collection of named object contents that provide a specific view of the data and can be tailored to the specific needs and permissions of a client.

In a work space, *abstract object identifiers* [8] are linked to concrete names chosen by a user. This allows the user to organize objects in folders, to use the same name in different folders, and to establish "private" links between objects. The work space may also restrict a client's access to certain library objects. Most importantly, however, it protects internal identifiers and object contents from being modified without going through the FDL, helps preventing name collisions, and makes proof mechanisms independent of particular naming schemes.

The *library manager* provides clients with utilities for building, storing, and sharing collections of *session objects*. It maintains the work spaces and thus enforces a discipline for building named collections, thus preserving the *coherency* of the collections.

4.2.2 Closed Maps

Work spaces are represented by *maps* from a finite set of names to library objects. These maps have to be *closed* in the sense that the objects they refer to do not contain any references to objects that have no name in the map. Thus the basic model of interacting with the library is to maintain a *current closed map* as a part of state that is updated repeatedly as one works.

In general, a *closed map* is a function of type $D \rightarrow \mathrm{Term}(D)$, where D is a finite discrete type of indices and `Term(`D`)` is the type of terms whose subterms only contain abstract identifiers in D. Usually we identify objects in a closed map with their index (or *name*).

In practice the class D will be varied continually. For example, extending a closed map requires selecting a larger index class. Deleting members of a closed map requires a smaller index class. In both cases, we have to make sure that the resulting map remains closed.

If the restriction of a closed map $m \in D \rightarrow \mathrm{Term}(D)$ to a subclass $X \subseteq D$ is itself a closed map, then we call it a *submap* of m. Similarly a *supermap* of m is a closed extension of m to a class $Y \supseteq D$. Two closed maps m and m' are *equivalent*, if they are simply renamings of each other.

Closed maps are essential for defining the notion of *dependency*. Objects depend on others if they directly or indirectly refer to them. An expression $t \in$ `Term(`D`)` *refers* directly to an object (index) $x \in D$ if x occurs within a subterm of t. The notion

of dependency is the key to defining *correctness*. While it is possible to define useful notions of correctness with respect to state, the enduring ones can only be formulated in terms of closed maps: the correctness of an object should only depend on the correctness of the object it refers to but not on library objects that are not within the current closed map.

The library is a repository not of closed maps per se, but is rather a repository of data and instructions for building closed maps modulo choice of abstract identifiers. In a session the current closed map is initialized from the library, transformed through a sequence of operations, and then stored back into the library for later retrieval. Several operations can be defined on closed maps including operations on two maps such as *Merging*, in which objects can be identified, or operations that build a new map by shrinking or expanding a current map, or a subset of a current map, utilizing dependency information.

4.3 Accounting mechanisms

One of the central aspects of a formal digital library is to account for the integrity of its contents and to support arguments for claims of the following form:

> Because the library contains a proof of theorem T that refers to a given collection of proof rules and proof engines, theorem T is true if those rules are valid and those engines run correctly.

Accounting mechanisms determine how to execute *inferences* as specified by a proof tactic depending on the actual contents of the library and produce *certificates*, which attest that certain actions were taken at a certain time to account for the validity of an object. Accounting mechanisms are also needed to determine whether a proof built from a collection of certified inferences is acceptable for a given purpose. This would be trivial if the FDL were restricted to a single uniform logic and to a single inference engine. But inferences that may employ a variety of logics and inference engines cannot be simply combined. Instead, certain stipulations limiting proofs to a given set of rules, axioms, and other objects on which these may depend must be expressed and checked.

To account for the validity of library objects, the knowledge base supports *dependency tracking*. For this purpose a variety of information is stored together with an object, e.g. the logical rules and theorems on which it depends, the exact version of the refiners that were used to prove it, timestamps, etc. This information will help a user to validate theorems that rely on knowledge created by several systems, provided that the conditions for *hybrid validity* with respect to the underlying logics are well understood and stored in the library. For instance, a theorem referring to lemmata from Nuprl (constructive type theory) and HOL (classical higher order logic) would be marked as constructively valid if the HOL theorems involve only decidable predicates.

The library only provides methods to store and retrieve dependency data recorded for an object. Tools for examining the relationships between objects from disparate domains must be provided by developers of the systems being related.

Key units of the FDL accounting mechanisms include accounting at the level *inference*, and maintenance of *certificates*.

4.3.1 Inferences

One of the most fundamental mechanisms to account for the validity of library contents is the application of logical inferences from a finite number of premises to a conclusion. Inferences are represented by trees of *inference steps*, which in turn are represented as library objects. A library process checks or generates an inference step by applying *inference engines*, which create proofs in some formal calculus according to user specified methods.

The fact that an inference step has been verified by a given inference engine is represented by an external certificate that refers to the inference step. There may be multiple certificates for the same inference, certifying that the inference has been checked by different inference engines. Depending on the contents of the available certificates, the inference may be considered valid or not in a specific context.

Inference engines support the development of new formal knowledge by providing mechanisms for interactive, tactical, and fully automated reasoning in a specific formal language. As the formal digital library supports almost any formal language, it can be connected to a variety of inference engines that will provide justifications for its formal content.

4.3.2 Certificates

Certificates are the basis for logical accounting. They attest that certain library actions were taken at a certain time to validate the contents of, or identity between, objects. A certificate will be realized as an object, which can then be referenced and accessed like other objects save for certain constraints. A certificate cannot be created or modified except by the library process following a procedure specific to the kind of certificate in question.

Although certificate contents are expected to be rather compact, largely consisting of object references, they will often be expensive to establish. By realizing certificates as objects the library can build certificates that depend on others whose correctness is independently established. Thus one process of certification can contribute to many other certifications without having to be redone.

The paradigmatic certificates are those created to validate proofs. An *inference step certificate* attests to the fact that a specified inference engine accepted that a certain inference step is valid. It is built by applying the engine to the inference, and includes references to the inference step as well as to the instructions for building or deploying the inference engine. A proof is a rooted dag of inference steps. A *proof certificate* is created only when there is an inference certificate for the root inference, and there are already proof certificates for all the proofs of the premises of the root inference.

A certificate may fall into doubt when any object it refers to is modified and needs to be reconsidered and modified in this case. Certificates may also be reconsidered by explicit demand.

As certificates only attest to the fact that the objects they refer to satisfy certain policies for creating these certificates, they provide justifications that are more general than formal proofs in a specific target logic. Nevertheless, they are equally rigorous in the sense that they provide the exact reasons that were used for declaring an object valid.

This aspect is particularly interesting when one considers the possibility that certain inference engines may not be generally accepted. People who do trust a certain inference engine will accept the certificates produced by it while others will insist that the same inferences have to be checked by inference engines they trust. The connection between the FDL and JProver accounts for these two levels of trust: one may either trust that matrix proofs produced by JProver [81] are valid, or one may require that the algorithm for translating matrix proofs into sequent proofs [82] be executed and that the results will be checked with a proof checker for the intuitionistic sequent calculus.

4.4 Information Science Considerations

With the capabilities to merge vast collections of formal mathematics, to account for their correctness and origins, and to use these libraries in the further developments of formal mathematics, we see both a need and an opportunity to utilize and adapt information science techniques.

4.4.1 Publishing and Reading

One of the key services that a library must provide is an interface that makes formal algorithmic knowledge accessible to users. Efforts such as MoWGLI and the HELM project [55,95] attest to this importance. As we envision a variety of users who would benefit from being able to inspect the contents of a formal digital library, we have developed a *publication mechanism* that enables external users to access the logical library and to browse its contents without having to run a local copy of it.

In [98] we made a first step towards publishing our formal mathematics on the web. Our interface supports viewing formal contents at all levels of precision – from the rough sketch of a theory or proof down to the level of the underlying logic. We expect that the ability to unveil formal details on demand will have a significant educational value for teaching and understanding mathematical and algorithmic concepts.

We have built a large utility for converting related collections of objects to HTML, along with automatically generated structure-revealing documents and auxiliary annotations. When the utility is run, the various browsable pages and some print forms are generated, missing links are reported diagnostically, and a decision is rendered about whether the result is suitable for posting. The structure of the generated web material is thoroughly explained on the several documentation pages to which one may link from every page so created.

Currently, the publication mechanisms are capable of handling formal content created by Nuprl. Automatically showing new kinds of content in this way, such as pages about PVS libraries, requires appropriate modification and specialization. Not only might the presentation of each object need adjustment, but the relations to be revealed between documents must be determined and accommodated.

4.4.2 A Whole Greater than its Sum

From an information science perspective, the FDL is a dynamic information network, with library objects as its vertices and relationships between them as its arcs, not unlike web pages and hypertext. There can coexist multiple kinds of arcs, defined at

varying levels of detail, such as object to object or theory to theory, as well as multiple kinds of objects. The collection of these objects into such a network facilitates a deeper and richer understanding than instances of these objects, or even instances of these objects together with meta-properties. We have already seen this in numerous accounts for the web [70,103].

One meaningful and straightforward relationship in the FDL network is the logical dependency relationship. If we limit the links to logical dependencies, for example, then, unlike with hypertext on the web, we gain a well defined linking mechanism. A popular link analysis method to reveal relevancy in web search, HITS [70], was applied and adapted to the FDL dependency network in [88]. The algorithm finds popular nodes, which is useful for ranking query-based search results, and also finds communities, or clusters of related objects, which may be useful for automated organization of the library. Looking at the graph as a whole, it appears that our FDL mathematics collections are scale-free [11]. A scale-free network includes "very connected" nodes, which greatly shorten the likely distance between two nodes from that of a randomly connected graph with the same number of nodes and arcs. A characteristic depth and breadth of the graph was also revealed. Initial comparisons were made between the FDL network and the dependency network of a collection of Coq proofs, yielding a mix of similar and dissimilar patterns. Relationships to other objects, and the context within which a math object resides, can bring added support to the task of information retrieval in the math domain.

5 Conclusion

The Nuprl system, its Computational Type Theory, and the design of a formal digital library are the result of more than 20 years of research and practical experiments with mathematical proof assistants. In addition to aspects of theoretical developments and issues of implementing state-of-the-art proof environments there have been numerous applications ranging from mathematics and formal courseware to hardware and software design, verification, and optimization [62,97,1,68,64,28,80,56,77,85,34,19,86,18,79]. Much of our current research supports the interaction between Nuprl and other mathematical assistants, proof engines, and publication mechanisms – both from the technological and from the logical perspective – which will lead to a greater acceptance of these systems in the practice of mathematics and software development.

Acknowledgments This material is based upon work supported by the National Science Foundation under Grants No. 0204193 (Proof Automation in Constructive Type Theory) and 0208536 (Innovative Programming Technology for Embedded Systems). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Mark Aagaard and Miriam Leeser. Verifying a logic synthesis tool in Nuprl. In Gregor Bochmann and David Probst, editors, *Proceedings of Workshop on Computer-Aided Verification*, LNCS 663, pages 72–83. Springer-Verlag, 1993.

[2] J. Abbott, A. van Leeuwan, and A. Strotmann. Openmath: Communicating mathematical information between co-operating agents in a knowledge network. *Journal of Intelligent Systems*, 1998.

[3] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In D. McAllester, editor, *17th Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[4] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, and Christoph Kreitz. A Nuprl-PVS connection: Integrating libraries of formal mathematics. Technical report, Cornell University. Department of Computer Science, 2003.

[5] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In Gries [51], pages 215–224.

[6] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[7] Stuart F. Allen. From dy/dx to [ ]p: a matter of notation. In *Proceedings of the Conference on User Interfaces for Theorem Provers*, Eindhoven, The Netherlands, 1998.

[8] Stuart F. Allen. Abstract identifiers, intertextual reference and a computational basis for recordkeeping. *First Monday*, 9(2), 2004.

[9] A. Asperti, L. Padovani, C. Sacerdoti Coen, F. Guidi, and I. Schena. Mathematical knowledge management in HELM. volume 38, Issue 1-3, pages 27–46. Kluwer Academic Publishers, 2003.

[10] R. C. Backhouse, P. Chisholm, and G. Malcolm. Do-it-yourself type theory (part II). *EATCS Bulletin*, 35:205–245, 1988.

[11] A.L. Barabsi and R. Albert. Emergence of scaling in random networks. pages 509–512.

[12] Henk P. Barendregt. *Handbook of Logic in Computer Science*, volume 2, chapter Lambda Calculi with Types, pages 118–310. Oxford University Press, 1992.

[13] J. L. Bates. *A Logic for Correct Program Development*. PhD thesis, Cornell University, 1979.

[14] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

[15] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[16] Gustavo Betarte and Alvaro Tasistro. Extension of Martin-Löf's type theory with record types and subtyping. In *Twenty-Five Years of Constructive Type Theory*, pages 21–39. Oxford University Press, 1998.

[17] Mark Bickford and Robert L. Constable. A logic of events. Technical Report TR2003-1893, Cornell University, 2003.

[18] Mark Bickford, Christoph Kreitz, Robbert van Renesse, and Xiaoming Liu. Proving hybrid protocols correct. In Richard Boulton and Paul Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics*, LNCS 2152, pages 105–120. Springer Verlag, 2001.

[19] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition*, pages 149–160. IEEE Computer Society Press, 2000.

[20] E. Bishop. *Foundations of Constructive Analysis*. McGraw Hill, NY, 1967.

[21] N. Bourbaki. *Elements of Mathematics, Algebra*, volume 1. Addison-Wesley, Reading, MA, 1968.

[22] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[23] D. Carlisle, P. Ion, R. Minder, and N. Poppelier. Mathematical markup language (MathML) 2.0 w3c recommendation. February 2001. URL: www.w3.org/TR/2001/REC-MAthML2-20010221/.

[24] Alonzo Church. *Introduction to Mathematical Logic*, volume I. Princeton University Press, 1956.

[25] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland, 1971.

[26] Robert L. Constable. Mathematics as programming. In *Proceedings of the Workshop on Programming and Logics*, LNCS 164, pages 116–128. Springer-Verlag, 1983.

[27] Robert L. Constable. The semantics of evidence. Technical Report TR85-684, Cornell University, 1985.

[28] Robert L. Constable. Creating and evaluating interactive formal courseware for mathematics and computing. In Magdy F. Iskander, Mario J. Gonzalez, Gerald L. Engel, Craig K. Rushforth, Mark A. Yoder, Richard W. Grow, and Carl H. Durney, editors, *Frontiers in Education*. IEEE, 1996.

[29] Robert L. Constable. Types in logic, mathematics and programming. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter X, pages 683–786. Elsevier Science B.V., 1998.

[30] Robert L. Constable, Stuart Allen, Mark Bickford, James Caldwell, Jason Hickey, and Christoph Kreitz. *Steps Toward a World Wide Digital Library of Formal Algorithmic Knowledge*, volume 1. 2003.

[31] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[32] Robert L. Constable and Jason Hickey. Nuprl's class theory and its applications. In Friedrich L. Bauer and Ralf Steinbrueggen, editors, *Foundations of Secure Computation*, NATO ASI Series, Series F: Computer & System Sciences, pages 91–116. IOS Press, 2000.

[33] Robert L. Constable and Douglas J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R. B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book*, pages 45–76. Elsevier Science Publishers, 1990.

[34] Robert L. Constable, Paul B. Jackson, Pavel Naumov, and Juan Uribe. Constructively formalizing automata. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 213–238. MIT Press, Cambridge, 2000.

[35] Robert L. Constable, S. Johnson, and C. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*, LNCS 135. Springer-Verlag, 1982.

[36] Robert L. Constable and S. D. Johnson. A PL/CV précis. In *Principles of Programming Languages*, pages 7–20. ACM, NY, 1979.

[37] Robert L. Constable, T. Knoblock, and J. L. Bates. Writing programs that construct proofs. *J. Automated Reasoning*, 1(3):285–326, 1984.

[38] Robert L. Constable and Michael J. O'Donnell. *A Programming Logic*. Winthrop, Mass., 1978.

[39] Robert L. Constable and D. R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, January 1984.

[40] Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, Christine Paulin-Mohring, César Muñoz, Chetan Murthy, Catherine Parent, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant reference manual. Technical report, INRIA, 1995.

[41] Karl Crary. Foundations for the implementation of higher-order subtyping. In *1997 ACM SIGPLAN International Conference on Functional Programming*, 1997.

[42] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report TR98-1675, Department of Computer Science, Cornell University, April 1998.

[43] Karl Crary. *Type–Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, August 1998.

[44] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic, Volume I*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.

[45] N. G. de Bruijn. The mathematical language Automath: its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration*, LNM 125, pages 29–61. Springer-Verlag, 1970.

[46] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[47] Amy Felty and Frank Stomp. A correctness proof of a cache coherence protocol. In *11th Annual Conference on Computer Assurance*, pages 128–141, 1996.

[48] Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In William McCune, editor, *Proceedings of the 14^{th} International Conference on Automated Deduction*, LNAI 1249, pages 351–365. Springer, 1997.

[49] J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*, volume 7 of *Cambridge Tracts in Computer Science*. Cambridge University Press, 1989.

[50] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, LNCS 78. Springer-Verlag, 1979.

[51] D. Gries, editor. *Proceedings of the 2^{nd} IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1987.

[52] T. G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems*. PhD thesis, Cornell University, 1988.

[53] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[54] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[55] HELM: Hypertextual electronic library of mathematics. `http://www.cs.unibo.it/~asperti/HELM`.

[56] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In R. Cleaveland, ed., *5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, pages 119–133. Springer Verlag, 1999.

[57] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, LNCS 2758, pages 287–303. Springer-Verlag, 2003.

[58] Jason J. Hickey. *The MetaPRL Logical Programming Environment.* PhD thesis, Cornell University, Ithaca, NY, January 2001.

[59] C. A. R. Hoare. Notes on data structuring. In *Structured Programming.* Academic Press, New York, 1972.

[60] James G. Hook and Douglas J. Howe. Impredicative strong existential equivalent to type:type. Technical Report TR 86-760, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, June 1986.

[61] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

[62] Douglas J. Howe. The computational behaviour of Girard's paradox. In David Gries, editor, *LICS-87 – Second Annual Symposium on Logic in Computer Science*, pages 205–214. IEEE Computer Society Press, 1987.

[63] Douglas J. Howe. Equality in lazy computation systems. In *4th IEEE Symposium on Logic in Computer Science*, pages 198–203, IEEE Computer Society Press.

[64] Douglas J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, LNCS 1125, pages 267–282. Springer-Verlag,1996.

[65] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, LNCS 1101, pages 85–101. Springer-Verlag, 1996.

[66] Douglas J. Howe. A classical set-theoretic model of polymorphic extensional type theory. Unpublished manuscript, 1997. Available at ftp://ftp.research.bell-labs.com/dist/howe/set-nuprl.ps.gz.

[67] Douglas J. Howe. Toward sharing libraries of mathematics between theorem provers. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 161–176. Research Studies Press/Wiley, 2000.

[68] Paul B. Jackson. Exploring abstract algebra in constructive type theory. In Alan Bundy, editor, *12th Conference on Automated Deduction*, LNAI 814, Springer Verlag, 1994.

[69] S. C. Kleene. *Introduction to Metamathematics.* D. Van Nostrand, Princeton, 1952.

[70] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the Ninth ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[71] T. Knoblock. *Metamathematical Extensibility in Type Theory.* PhD thesis, Cornell University, 1987.

[72] M. Kohlhase. Omdoc: An infrastructure for openmath content dictionary information. In *Bulletin of the ACM Special Interest Group for Algorithmic Mathematics SIGSAM*, 2000.

[73] M. Kohlhase. Omdoc: Towards an internet standard for the administration, distribution and teaching of mathematical knowledge. In *Proceedings of Artificial Intelligence and Symbolic Computation*, Lecture Notes in Computer Science. Springer, 2000.

[74] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. Department of Computer Science TR2000-1809, Cornell University, Ithaca, New York, 2000.

[75] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *18th IEEE Symposium on Logic in Computer Science*, pages 86–95, 2003.

[76] Alexei Pavlovich Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects*. PhD thesis, Cornell University, January 2004.

[77] Christoph Kreitz. Automated fast-track reconfiguration of group communication systems. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, pages 104–118. Springer Verlag, 1999.

[78] Christoph Kreitz. *The Nuprl Proof Development System, Version 5: Reference Manual and User's Guide*. Cornell University. Department of Computer Science, 2002.

[79] Christoph Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *Journal of Functional Programming*, 14(1):21–68, 2004.

[80] Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communication systems. In C. Kirchner and H. Kirchner, editors, *15th Conference on Automated Deduction*, LNAI 1421, pages 317–331. Springer Verlag, 1998.

[81] Christoph Kreitz and Jens Otten. Connection-based theorem proving in classical and non-classical logics. *Journal of Universal Computer Science*, 5(3):88–112, 1999.

[82] Christoph Kreitz and Stephan Schmitt. A uniform procedure for converting matrix proofs into sequent-style systems. *Journal of Information and Computation*, 162(1–2):226–254, 2000.

[83] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, 21(7):558–65, 1978.

[84] Leslie Lamport. The mutual exclusion problem. part ii: statement and solutions. *J. ACM*, 33(2):327–348, 1986.

[85] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, *Operating Systems Review* 34:80–92, 1999.

[86] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta-properties. In Luis Rodrigues and Michel Raynal, editors, *International Workshop on Applied Reliable Group Communication (WARGC 2001)*, pages 37–42. IEEE Computer Society Press, 2001.

[87] Lori Lorigo. Building proofs in parallel: A collaborative tutorial for the nuprl logical programming environment (LPE). Masters of Engineering thesis, January 2001.

[88] Lori Lorigo, Jon Kleinberg, Rich Eaton, and Robert Constable. A graph-based approach towards discerning inherent structures in a digital library of formal mathematics. In *Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, 2004.

[89] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[90] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.

[91] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[92] Per Martin-Löf. *Intuitionistic Type Theory.* Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.

[93] The MathBus Term Structure. `www.nuprl.org/mathbus/mathbusTOC.htm`.

[94] P.F. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, Ithaca, NY, 1988.

[95] MoWGLI: Mathematic on the Web: Get it by Logics and Interfaces. `http://mowgli.cs.unibo.it`.

[96] Chetan Murthy. A computational analysis of Girard's translation and LC. In *Proceedings of Seventh Symposium on Logic in Comp. Sci.*, pages 90–101, 1992.

[97] Chetan R. Murthy and James R. Russell. A constructive proof of Higman's lemma. In John C. Mitchell, editor, *Fifth Annual Symposium on Logic in Computer Science, 1990*, pages 257–267. IEEE Computer Society Press, 1990.

[98] Pavel Naumov. Publishing formal mathematics on the web. Technical Report TR98-1689, Cornell University. Department of Computer Science, 1998.

[99] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Sciences Publication, Oxford, 1990.

[100] Nuprl home page. `http://www.nuprl.org`.

[101] Robert Pollack. Dependently typed records for representing mathematical structure. In J. Harrison and M. Aagaard, editors, *: 13^{th} International Conference on Theorem Proving in Higher Order Logics*, LNCS 1869, pages 461–478. Springer-Verlag, 2000.

[102] Bertrand Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.

[103] L. Page S. Brin. The anatomy of a large-scale hypertextual (web) search engine. In *Proceedings of the Seventh International World Wide Web Conference*, 1998.

[104] Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Alexey Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In R. Gore, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning*, LNAI 2083, pages 421–426. Springer Verlag, 2001.

[105] D. Scott. Data types as lattices. *SIAM J. Comput.*, 5:522–87, 1976.

[106] R. M. Smullyan. *First–Order Logic.* Springer-Verlag, 1968.

[107] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[108] Anne Sjerp Troelstra. *Metamathematical Investigation of Intuitionistic Mathematics*, LNM 344. Springer-Verlag, 1973.

[109] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction*, volume I, II. North-Holland, 1988.

[110] University of Cambridge, Computer Laboratory. *The HOL System: DESCRIPTION*, March 1994.

[111] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925–27.

[112] J. Zimmer and M. Kohlhase. The Mathweb Software Bus for distributed mathematical reasoning. In *18th Int'l Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2002.