The Structure of Nuprl's Type Theory

Robert L. Constable
Cornell University
Ithaca, NY 14853

1 Introduction

1.1 Context

After my lectures on this topic were delivered in July 1995 at Marktoberdorf, my colleagues and I made available much more related material both at the Nuprl home page on the World Wide Web ("the Web") (www.cs.cornell.edu/Info/NuPrl/nuprl.html) and in publications [22], some soon to appear [11].

At the Web site the thesis of Jackson [31] and the article by Forester [20] are especially relevant to my lecture. Also, the 1986 Nuprl book is now available on line at the Web site as is the Nuprl 4 reference manual and a host of Nuprl *libraries*.

As a result of the ready access to this material and because the material in the Cornell Technical Report series is part of an expanding *digital library* whose longevity seems guaranteed, I designed this article to concentrate on the material not available elsewhere, e.g. a discussion of the Core Theory and its relation to Nuprl 4.

A naive account of Core Type Theory is especially simple, and I think it provides a bridge to understanding the more daunting axiomatization of the Nuprl type theory which is used in all of the on-line libraries. This article presents the Core Theory and relates it to a corresponding part of Nuprl. Comparisons are made to set theory as a way to motivate the concepts.

1.2 Types and Sets

The informal language of mathematics uses types and sets, but when mathematicians want to be rigorous about a concept, they tend to rely on the 100 year old tradition of reducing it to concepts in pure set theory. In this article, we will find the rigorous concepts in *type theory*. For pedagogical reasons, we will often mention the set theory version of a concept as well.

The idea of a type is built up inductively (as is the Zermelo hierarchical concept of set). We start with *primitive types* and *type constructors*.

The Core Theory needed for Nuprl involves only six type constructors: product, disjoint union, function space, inductive type, set type, and quotient type. We need some primitive types as well: void, unit, Type, and Prop. I have chosen to add co-inductive types as well although they are not in Nuprl 4.

2 The Core Theory

2.1 Primitive Types

void is a type with no elements
unit is a type with one element, denoted $\bullet$

There will be other primitive types introduced later. Notice, in set theory we usually have only one primitive set, some infinite set (usually ω). Sometimes the empty set, ϕ, is primitive as well, although it is definable by separation from ω.

Compound Types We build new types using *type constructors*. These tell us how to construct various kinds of objects. (In pure set theory, there is only one kind, *sets*).

The type constructors we choose are motivated both by mathematical and computational considerations. So we will see a tight relationship to the notion of *type* in programming languages. The notes by C.A.R. Hoare, *Notes on Data Structuring* [23], make the point well.

2.2 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 a product are pairs, $\langle a, b \rangle$. Specifically if a belongs to A and b belongs to B, then $\langle a, b \rangle$ belongs to $A \times B$. We abbreviate this by writing

$$\frac{a \in A \quad b \in B}{\langle a, b \rangle \in A \times B.}$$

In programming languages these types are generalized to n-ary products, say $A_1 \times A_2 \times \ldots \times A_n$. They are the basis for defining *records*.

We say that $\langle a, b\rangle = \langle c, d\rangle$ in $A \times B$ iff $a = c$ in A and $b = d$ in B.

In set theory, equality is uniform and built-in, but in type theory we define equality with each constructor, either built-in (as in Nurpl) or by definition as in this core theory.

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

2.3 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. These are given by rules which are defined for each a in A and which produce a *unique value*. We summarize by

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

The function notation we use informally comes from mathematics texts, e.g. Bourbaki's *Algebra*. 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 $(x \mapsto b)\ \epsilon\ A \to B$. We will also 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 phrase $x{:}A$ *declares* x to be of type A. 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 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 $a \epsilon A$, we write $f(a)$ for the value of the function.

2.4 Disjoint Unions (also called Discriminated 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 which type an element is in. 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. There is an operation called *decide* that discriminates on 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

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

2.5 Subtyping

Intuitively, A is a subtype of B iff every element of A is also an element of B; we write this relation as $A \subseteq B$. Clearly $\phi \subseteq A$ for any A. Notice that A is not a subtype of $A + B$ since the elements of A in $A + B$ have the form $inl(a)$. We have these properties however

$$\frac{A \subseteq A' \qquad B \subseteq B'}{\begin{array}{c} A \times B \subseteq A' \times B' \\ A + B \subseteq A' + B' \\ A' \to B \subseteq A \to B' \end{array}}$$

For $A \subseteq B$ we also require that $a = a'$ in A implies $a = a'$ in B.

2.6 Inductive Types

Defining types in terms of themselves is quite common in programming, often pointers are used in the definition (in Pascal and C for example), but in languages like ML, direct recursive definitions are possible. For example, a list of numbers, L can be introduced by a definition like

$$\textit{define type} \qquad L \;=\; \mathbb{N} + (\mathbb{N} \times L).$$

In due course, we will give conditions telling when such definitions are sensible, but for now let us understand how elements of such a type are created. Basically a type of this kind will be sensible just when we understand exactly what elements belong to the type. Clearly elements like $inl(0), inl(1), \ldots$, are elements. Given them it is clear that $inr(\langle 0, inl(0)\rangle), inr(\langle 1, inl(0)\rangle)$ and generally $inr(\langle n, inl(m)\rangle)$ are elements, and given these, we can also build

$$inr(\langle k, inr(\langle n, inl(m)\rangle)\rangle) \text{ and so forth.}$$

In general, we can build elements in any of the types

$$\begin{array}{l}\mathbb{N} + \mathbb{N} \times Y \\ \mathbb{N} + \mathbb{N} \times (\mathbb{N} + \mathbb{N} \times Y) \\ \mathbb{N} + \mathbb{N} \times (\mathbb{N} + \mathbb{N} \times (\mathbb{N} + \mathbb{N} \times Y))\end{array}$$

The key question about this process is whether we want to allow *anything else* in the type L. Our decision is **no**, we want only elements obtained by this finite process.

In set theory, we can understand similar inductive definitions, say a *set* L such that $L = \mathbb{N} \cup (\mathbb{N} \times L)$, as least fixed points of monotonic operators $F : Set \to Set$. In general, given such an operator F, we say that the set inductively defined by F is the least F*–closed* set, call it $I(F)$. We define it as

$$I(F) = \cap\{Y | F(Y) \subseteq Y\}.$$

We use set theory as a *guide* to justify recursive type definitions.

For the sake of defining monotonicity, we use the subtyping relation, $S \subseteq T$. This holds just when the elements of S are also elements of T, and the equality on S and T is the same. For example,

$$\text{if} \qquad S \subseteq T \qquad \text{then} \qquad \mathbb{N} + \mathbb{N} \times S \;\subseteq\; \mathbb{N} + \mathbb{N} \times T.$$

Def. Given a type expression $F : Type \to Type$ such that if $T_1 \subseteq T_2$ then $F(T_1) \subseteq F(T_2)$, then write $\mu X.F(X)$ as the *type inductively defined by F*.

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

if $t \in F(\mu X.F(X))$ then $t \in \mu X.F(X)$.

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

The power of recursive types comes from the fact that we can define total computable functions over them very elegantly, and we can prove properties of elements recursively. Recursive definitions are given by this term,

μ–$ind(a; f, z.b)$

called a *recursor* or *recursive-form*. It obeys the computation rule

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

(Note in this rule we use the notation $b[s/x, t/y]$ to mean that we substitute s for x and t for y in b.)

typing

The way we figure out a type for this form is given by a simple rule. We say that

$$\mu\text{–}ind(a; f, z.b) \quad \text{is in type } B$$

provided that $a \in \mu X.F(X)$, and if when Y is a Type, and $Y \subseteq \mu X.F(X)$, and z belongs to $F(Y)$, and f maps Y to B, then b is of type B.

induction

The principle of inductive reasoning over $\mu X.F(X)$ is just this.

μ–induction

Let $R = \mu X.F(X)$ and assume that Y is a subtype of R and that for all x in Y, $P(x)$ is true. (This is the induction hypothesis.) Then if we can show $\forall z : F(Y).P(z)$, we can conclude

$$\forall x : R.P(x).$$

With this principle and the form μ–*ind*, we can write programs over recursive types and prove properties of them. The approach presented here is quite abstract, so it applies to a large variety of specific programming languages. It also stands on its own as a mathematical theory of types.

typing rules

We will write these informal rules as inference rules. To connect to the Nuprl account I will use its notation which is $rec(X.T)$ for $\mu X.T$ and $rec\_ind(a; f, z.b)$ for $\mu\_ind(a; f, z.b)$. In our Core Type Theory, recursive types are written as $\mu X.T$ where μ stands for the least fixed point operator. This is perhaps more standard notation; however, $rec(X.T)$ is the Nuprl syntax and is mnemonic.

Here is the rule which says $rec(X.T)$ is a type using a sequent style presentation of rules:

$$\frac{E, X : Type \vdash T \in Type}{E \vdash rec(X.T) \in Type}$$

For example, we can derive $rec(N.\mathbf{1} + \mathbf{1} \times N)$ is a type as follows:

$$\frac{\dfrac{\vdash \mathbf{1} \in Type \quad \dfrac{\vdash \mathbf{1} \in Type \quad E, N : Type \vdash N \in Type}{E \vdash \mathbf{1} \times N \in Type}}{E, N : Type \vdash (\mathbf{1} + \mathbf{1} \times N) \in Type}}{E \vdash rec(N.\mathbf{1} + \mathbf{1} \times N) \in Type}$$

Remember that $\mathbf{1}$ is a primitive type. This type is essentially "$\mathbf{1}$ list."

To introduce elements we use the following rule of unrolling:

$$\frac{E \vdash t \in T[rec(X.T)/X]}{E \vdash t \in rec(X.T)}$$

Here are some examples on in $rec(N.\mathbf{1} + \mathbf{1} \times N)$. Recall that type $\mathbf{1} = \{\bullet\}$.

The nil list is derived as:

$$\frac{\frac{\vdash \bullet \in \mathbf{1}}{N : Type \vdash inl(\bullet) \in \mathbf{1} + rec(N.\mathbf{1} + \mathbf{1} \times N)}}{\vdash inl(\bullet) \in rec(N.\mathbf{1} + \mathbf{1} \times N)}$$

Lists are derived as:

$$\frac{\frac{\frac{\vdash \bullet \in \mathbf{1} \quad \vdash inl(\bullet) \in rec(N.\mathbf{1} + \mathbf{1} \times N)}{\vdash pair(\bullet; inl(\bullet)) \in \mathbf{1} \times rec(N.\mathbf{1} + \mathbf{1} \times N)}}{\vdash inr(pair(\bullet; inl(\bullet))) \in \mathbf{1} + \mathbf{1} \times rec(N.\mathbf{1} + \mathbf{1} \times N)}}{\vdash inr(pair(\bullet; inl(\bullet))) \in rec(N.\mathbf{1} + \mathbf{1} \times N)}$$

To make a decision about, for example, whether l is nil or not, unroll on the left:

$$\frac{E, l : \mathbf{1} + \mathbf{1} \times rec(N.\mathbf{1} + \mathbf{1} \times N) \vdash g \in G}{E, l : rec(N.\mathbf{1} + \mathbf{1} \times N) \vdash g \in G}$$

recursion combinator typing

This version of recursive types allows "primitive recursion" over recursive types also called the *natural recursion*. The Nuprl syntax is

$$rec\_ind(t; f, z.g)$$

The usual convention for the syntax also applies here: t and g are two sub-terms and g is in the scope of two binding variables f and z. (We define the recursion combinator to be $\lambda(x.rec\_ind(x; f, z.g))$.)

For evaluation, we recall the computation rule:

$$rec\_ind(t; f, z.g) \rightarrow_\rho g[t/z, \lambda(w.rec\_ind(w; f, z.g/f))]$$

called ρ reduction. We don't evaluate t, but simply substitute it into g.

Typing of the recursion combinator is the key part:

$$\frac{E \vdash t \in rec(X.T) \quad E, X : Type, z : T, f : X \rightarrow G \vdash g \in G}{E \vdash rec\_ind(t; f, z.g) \in G}$$

Where is the induction? It's in the top right-hand side. For sanity check, unwind: $rec\_ind(t; f, z.g) \in G$ reduces to $g[t/z, \lambda(w.rec\_ind(w; f, z.g/f))] \in G$. As we expect, $g \in G$, z and y have the same type, f is a function from elements of recursive type to G. Then we bottom out on X. Thus X in $f : X \rightarrow G$ is the key part.

The longer you know this rule, the clearer it gets. It summarizes recursion in one rule.

definition of *ind* **using** *rec_ind*

Here is how to define a natural recursion combinator on $\mathbb{N}$, call it $ind(n; b; u, i.g)$, using $rec\_ind()$. Let's say how this behaves. The base case is:

$$ind(0; b; u, i.g) \rightarrow b$$

The inductive case is:

$$ind(succ(n); b; u, i.g) \rightarrow g[n/u, ind(n; b; u, i.g)/i]$$

We combine the base and the inductive cases into one rule as follows:

$$\frac{E \vdash n \in N \quad u : N, i : G[u/x] \vdash g \in G[succ(u)/x]}{E \vdash ind(n; b; u, i.g) \in G[n/x]}$$

To derive this rule for *ind* we can encode $\mathbb{N}$ into **1** list:

1. $\mathbf{1} = \{\bullet\}$
2. $0 = inl(\bullet)$
3. We can define the successor of m as $succ(m) = inr\langle\bullet, m\rangle$
4. **N** is isomorphic to $rec(N.\mathbf{1} + \mathbf{1} \times N)$

This presentation of N doesn't have any intrinsic value; it's just a way to look at N (for an alternative presentation try deriving *ind* from $rec(N.\ \mathbf{1}\ + N)$). It gives us an existence proof for the class of natural numbers.

Given the above setup, we want to derive the rule for `ind`:

$$\frac{E \vdash n \in \mathbb{N} \quad E \vdash b \in G[0/x] \quad E,\ u : \mathbb{N},\ i : G[u/x] \vdash g \in G[succ(u)/x]}{E \vdash ind(n; b; u, i.g) \in G[n/x]}$$

using the rule of *rec_ind*:

$$\frac{E \vdash t \in rec(X.T) \quad E,\ X : \text{Type},\ x : T,\ f : (y : X \rightarrow G[y/x]) \vdash g \in G}{E \vdash rec\_ind(t; f, x.g) \in G[t/x]}$$

Notice the following facts in the above two rules:

- In the rule for induction the first hypothesis denotes that we are doing induction over the natural nymbers, the second one is the base case of this induction (notice that we substitute x by zero) and the third is the induction hypothesis, by which we prove the fact for the successor of u assuming it for u.
- In the rule for recursive-induction x should be of type T (which is the body of the recursive type). This is because we want to be able to compute the predecessors of x.

If we want to indicate the new variables we use in the recursive induction rule we write the rule adding this information:

$$\frac{E \vdash t \in rec(X.T) \quad E,\ X : \text{Type},\ x : T,\ f : (y : X \to G[y/x]) \vdash g \in G}{E \vdash rec\_ind(t; f, x.g) \in G[t/x]}$$

This is an elegant induction form describing any possible recursive definition.

Lets see how this works in the case of building the *ind* rule. We constuct a $\hat{g}$ from g, so that if we use $\hat{g}$ in *rec_ind* we will derive $ind(n; b; u, i.g)$. Before we present $\hat{g}$, lets try to get a bit of the intuition. The most important part of the *ind* rule is its induction hypothesis. We want to specialise the *rec_ind* rule so that we can pull out and isolate this induction hypothesis.

The *idea* is the following:

> Let i be the function f applied to u, $i = f(u)$, where u is the predecessor of x we are looking at. We can get this predecessor u (as we have already noticed before) by using the hypothesis $x : T$ and by decomposing x.

For example, if T is the type of **1** list, then $T = \mathbf{1} + \mathbf{1} \times N$. We do a case split and we get two cases; in the first case we get **1** which represents zero, and in the second case we get $\mathbf{1} \times N$ which is the successor. We can construct $\hat{g}$, given g of the `ind` form, by taking a close look at the structure of x:

- T is a disjunct (and, hence, $\hat{g}$ is a *decide*),
- T's *inl* side has the zero, so we need it as the base case (which is b) and
- T's *inr* side is a pair which we want to spread and use. In the spread, u will denote the predecessor of x.

It seems that the following form for $\hat{g}$ can give us all we need:

$$\hat{g} = decide(x; z.b; p.spread(p; \bullet, u.g[f(u)/i]))$$

As an **exercise** you can prove that $\hat{g} \in G$ (*hint*: use the fact that $b \in G[0/x]$).

We know that

$$b \in G[inl(\bullet)/x]$$

Also

$$y : X \text{ and } f : X \to G[y/x]$$

so

$$x = inr(pair(\bullet, u))$$

which means

$$x = succ(u).$$

So in fact we are doing the substitution $G[succ(u)/x]$ and, thus $g \in G[succ(u)/x]$.

We leave to the reader to carry out the simple type-checking (applying the rules for *decide* and *spread* to this definition). The interesting point is that in $\hat{g} \in G$, we are building up x either as $inl(\bullet)$ or as $inr(pair(\bullet, u))$; in one case this is $G[0/x]$ and in the other $G[succ(x)/x]$.

We have now shown that we have built the natural induction form for integers from the recursion combinator.

2.7 Co-inductive Types

The presentation of inductive types is from Nax Mendler's 1988 thesis (*Inductive Definitions in Type Theory*) and writings. Nax discovered a beautiful symmetry in the type definition process and used it to define a class of types that are sometimes called "lazy types' (see Thompson book, *Type Theory and Functional Programming* [43]). Essentially co-inductive types arise by taking maximal fixed points of monotone operators. The standard example is a *stream*, say of numbers

$$define\ type \qquad S = \mathbb{N} \times S.$$

The elements of this type are objects that *generate* unbounded lists of numbers according to a certain *generation law*. The generators are recursively defined and have the form

$$\nu\text{–}ind(a; f, z.g)$$

much like the recursors.

Here is how the types are defined.

Definition Given a monotone operator F on types, $\nu X.F(X)$ is a type, called the *co-inductive type generated by* F.

An element of $\nu X.F(X)$ is defined from the *generator* $\nu\text{–}ind(a; f, z.b)$. This is well defined under these conditions:

Given a type D, the seed type, if $d \epsilon D$ and Y is a type such that $\nu X.F(X) \subseteq Y$ with $f : D \rightarrow Y$ and $z \epsilon D$ then $b \epsilon F(Y)$,

then $\nu - ind(d; f, z.b)\ \epsilon\ \nu X.F(X)$.

For example, if we call $\nu Y.\mathbb{N} \times Y$ a Stream, then $\nu\text{–}ind(k; f, z.\langle z, f(z+1)\rangle)$ will generate an unbounded sequence of numbers starting at k.

In order to use the elements of a co-inductive type, we need some way to force the generator to produce an element. This is done with a form called *out*. It has this property:

If $t \in \nu X.F(X)$
then $out(t) \epsilon F(\nu X.F(X))$.

This form obeys the following *computation rule*.

out(ν–$ind(d; f, z.b)$)
evaluates to $b[d/z, (y \mapsto \nu\text{–}ind(y; f, z.b)/f]$.

2.8 Subset Types and Logic

One of the most basic and characteristic types of Nuprl is the so-called *set type* or *subset type*, written $\{x:A \mid P(x)\}$ and denoting the subtype of A consisting of exactly those elements satisfying condition P. This concept is closely related to the set theory notion written the same way and denoting the "subset" of A satisfying the predicate P. In axiomatic set theory the existence of this set is guaranteed by the *separation axiom*. The idea is that the predicate P separates a subset of A as in the example of say the prime numbers, $\{x:\mathbb{N} \mid prime(x)\}$.

To understand this type, we need to know something about predicates. In axiomatic set theory the predicates allowed are quite restrictive; they are built from the atomic membership predicate, $x \in y$ using the first order predicate calculus over the universe of sets. In type theory we allow a different class of predicates — those involving predicative higher-order logic in a sense. This topic is discussed in many articles and books on type theory [33, 13, 36, 43, 12, 11] and is beyond the scope of this article, so here we will just assume that the reader is familiar with one account of propositions-as-types or representing logic in type theory.

The Nuprl style is to use the type of propositions, denoted $Prop$. This concept is stratified into $Prop_i$ as in *Principia Mathematica*, and it is related by the propositions-as-types principle to the large types such as $Type$. $Prop_i$ are indeed considered to be a "large types." (See [11] for an extensive discussion of this notion.) For the work we do here we only need the notions of $Type$ and $Prop$ which we take to be $Type_1$ and $Prop_1$ in the full Nuprl theory.

The point of universes or large types is that they allow us to use expressions like $Type$, $A \to Type$, $A \times Type$, $Type \to Type$, etc. as if they were types except that $Type \in Type$ is not allowed. Instead, when we need such a notion we must attend to the level numbers used to stratify the notions of $Type$ and $Prop$. We can say $Type_i \in Type_j$ if $i < j$, and $Type_i \subseteq Type_j$ when $i < j$.

Thus the objects of mathematics we consider include *propositions*. For example, $0 = 0$ in N and $0 < 1$ are true propositions about the type N, so $0 < 1 \in Prop$ and $0 < 0 \in Prop$. We also consider *propositional forms* such as $x =_N y$ or $x < y$. These are sometimes called *predicates*.

Propositional functions on a type A are elements of the type $A \to Prop$.

We also need types that can be restricted by predicates such as $\{x = N | x = 0 \text{ or } x = 1\}$. This type behaves like the *Booleans*.

The general formation rule is this. If A is a type and $B : A \to Prop$, then $\{x : A | B(x)\}$ is a *subtype of* A.

The elements of $\{x : A | B(x)\}$ are those a in A for which $B(a)$ is true.

Sometimes we state the formation in terms of predicates, so if P is a proposition for any x in A, then $\{x : A | P\}$ is a subtype of A. We Clearly have $\{x : A | P\} \subseteq A$.

2.9 Dependent types and modules

We will be able to define modules and abstract data types by extending the existing types in a simple but very expressive way — using so-called *dependent types*.

dependent product

Suppose you are writing a business application and you wish to construct a type representing the date:

$$\begin{aligned} Month &= \{1, \ldots, 12\} \\ Day &= \{1, \ldots, 31\} \\ \\ Date &= Month \times Day \end{aligned}$$

We would need a way to check for valid dates. Currently, $\langle 2, 31 \rangle$ 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, $\langle 2, 20\rangle$ is a legal date since $20 \in Day(2)$, and $\langle 2, 31\rangle$ 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.

We can see that what we are doing is making a more general 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:

$$\frac{E \vdash a : A \; E \vdash b \in B[a/x]}{E \vdash pair(a, b) : prod(A, x.B)}$$

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

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

dependent functions

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

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

$$\frac{E \vdash f \in fun(A; x.B) \quad E \vdash a \in A}{E \vdash ap(f; a) \in B[a/x]}$$

Example 2 : Back to our example Dates. We see that $m : Month \rightarrow Day[m]$ is just $fun(Month; m.Day)$, where Day is a family of twelve types. And $\lambda(x.maxday[x])$ is a term in it.

3 Equality and Quotient Types

According to Martin-Löf's semantics, a mathematical type is created by specifying notation for its elements, called *canonical names*, and specifying our equality relation on these names. The equality relation can be an equivalence relation. This relation is one feature that distinguishes mere notation from the more abstract idea of an *object*, i.e. from notation with meaning.

For example, sets of ordered pairs of integers, $\langle x, y\rangle$ can be used as constants for rational numbers. In this case, the equality relation should equate $\langle 2, 4\rangle$ and $\langle 1, 2\rangle$, generally $\langle a, b\rangle = \langle c, d\rangle$ iff $a * d = b * c$. Similarly, let $\mathbb{Z}/mod\ n$ denote the congruence integers, i.e. the integers with equality taken $mod\ n$ so that $x = y$ mod n iff n divides $(x - y)$. The only difference between $\mathbb{Z}$ and $\mathbb{Z}/mod\ n$ is the equality relation on the integer constants.

Following Beeson [6] we might speak of the constants without an equality relation as a pre-type. Then a type arises by pairing an equality with a pre-type, say $\langle T, E\rangle$ where E is an equivalence relation on T. But when we think of functions on a type, say $f : \mathbb{Q} \to \mathbb{Q}$, we do not expect equality information to be included as part of the input to f. The "data" comes from T. This viewpoint is thus similar to Martin-Löf's as long as we provide a way to define new types by changing the equality relation on data and keeping the equality information "hidden" from operations on the type. This is accomplished by the *quotient type*.

To form a quotient type we need a type T and an equivalence relation E on T. The quotient of T by E is denoted $T//E$. (Nuprl uses a slightly more flexible notation as we see in section 3.) The elements of $T//E$ are those of T but equality is defined by E and denoted $x = y$ in $T//E$. For example the rational numbers can be taken to be $(\mathbb{Z} \times \mathbb{N}^+)//E$ where

$$E(\langle x, n\rangle, \langle y, m\rangle) \text{ iff } m * x = n * y.$$

It is noteworthy that because the equality information is "hidden" we cannot in general say

$$x = y \text{ in } T//E \text{ implies } E(x, y)$$

under the propositions-as-types interpretation of implication.

Type theory can express the logical idea that given $x = y$ in $T//E$ we know that $E(x, y)$ is "true" in the sense that there is a proof of $E(x, y)$ but we cannot access it. One way to say this is to replace the predicate $E(x, y)$ by the weaker type $\{\mathbf{1}|E(x, y)\}$. We call this the "squashed type." If $E(x, y)$ is true then this type, call it $sq(E(x, y))$ has $\bullet$ as member according to our rules for the set type. If $E(x, y)$ is not true, then $sq(E(x, y))$ is empty. We can thus say

$$x = y \text{ in } T//E \text{ implies } sq(E(x, y)).$$

4 A Nuprl Type Theory

In this section we look at some features of the Nuprl type theory. The first section is a discussion of the uniform syntax of Nuprl 4 terms. The second section considers Allen's semantics [4, 3] for Nuprl without recursive types. Mendler [35] provides a semantics for recursive types as well, but it is more involved than what we present here.

4.1 Nuprl Term Syntax

Following Frege, Church, and Martin-Löf, we take the basic unit of notation to be a *term*. A formula for a sentence, $\forall x : int.\ x^2 \geq 0$, is a term as is the expression for the square of a number, x^2.

We will distinguish the *concrete syntax* or the *display form* of a term from its abstract structure.

4.1.1 abstract structure of terms

What are the constituent parts of a term?

operator name

In our analysis a term is built from an *operator* and subterms. We choose to name the operator, so there is some means of finding an *operator_name*. This seems to be convenient for computer processing and helps in the conduct of mathematics as well. Informal practice sometimes settles for a glyph or symbol as the operator name, e.g. $\int$.

subterms

Given a term, there must be a finite number of subterms. These could be collected as a list or a multi-set (bag). In many cases, say $\frac{a}{b}$, it is not clear how to order the subterms a and b. But there must be a way to *address* (or locate) uniquely each subterm. We have chosen to *list* the subterms.

binding structure

We know that mathematical notation for sentences can be defined with *combinators* which do not introduce an idea of binding.

We adopt the analysis of notation based on *binding*. So with each operator there is a binding mechanism. Informal mathematics uses many different mechanisms, but we will attempt to analyze all of them as *first-order*. That is, the binding structure can be defined by designating a class of first order variables, i.e. just identifiers, as binding occurrences. The generic form of an operator with binding structure is

$$op_{x_1,\ldots,x_n}(t_1;\ldots;t_m)$$

where x_i are first order variables and t_j are terms. The binding structure is specified by saying which of the x_i is bound in which t_j. This could be done graphically as in

$$op_{x,y,z}(a_{x,y};b_{y,z};c_{x,z})$$

In Nuprl we have chosen to use the form

$$op(x,y.\ a_{x,y};y,z.\ b_{y,z};x,z.\ c_{x,z}).$$

because it is a simple way to represent the general binding structure described above.

4.1.2 operations on terms

The common operations on terms are these.

1. Create an operator.
2. Provide displays for terms built with an operator.
3. Navigate to subterms.
4. Replace a free variable with a term.
5. Rename bound variables.
6. Substitute a term for a free variable using renaming to avoid capture.

The behavior of substitution should satisfy a number of criteria beyond being mathematically correct. We expect any required renaming of bound variables to behave in a predictable manner. It should be possible to easily understand the renaming process in ways that affect computation and reasoning.

4.1.3 Nuprl term structure

The term structure of Nuprl 4 evolved from experience with Martin-Löf's type theory and Nuprl 3. We made the earlier term structure more uniform and more general.

terms

A term has the form

$$op(\bar{v}_1.\ t_1; \ldots; \bar{v}_n.\ t_n)$$

where $\bar{v}_i$ is a list of identifiers whose *scope* is the subterm t_i. These $\bar{v}_i$ are lists of *binders*. If $\bar{v}_i$ is empty then it and the dot separating binders from terms are not written. The component *op* is the *operator*; it is *not* a subterm.

If opa, opb, opc are operators and t, t_i are terms, then here are examples of terms.

$opa(x.\ t)$
$opb(x, y.\, t_1;\ y, z.\ t_2;\ x, z.\ t_3)$
$opc(t_1;\ x.\ t_2;\ x, y.\, t_3)$

operators

Operators are themselves compound objects. Their form is

$$opname\{i_1 : f_1; \ldots; i_n : f_n\}$$

These were introduced to express families of operators such as universes, U_i, and the injection of natural numbers. These are written as

$universe\{i : pos\}$
$natural\_number\{u : nat\}$
$variable\{v : string\}$.

We call the i_j an *index* and the f_i an *index family name.*

alternatives — λcalculus with operator names

Another representation of terms that is natural is based on using the lambda calculus to define all binding [30]. For example, $\forall x : A.\ B(x)$ would be represented as $all(\lambda x : A.\ B(x))$. And $spread(p; u, v.\ t)$ would be $spread(p, \lambda(u, v.\ t))$.

This approach is natural in Lisp where λterms are coded as S-expressions, but it creates notations with redexes such as $all((\lambda x : nat.\ \lambda y : A.\ B(x, y))(2))$. The presence of these redexes notation appears unnatural, and they complicate pattern matching. Moreover, because of redexes, there can be an arbitrary number of terms that represent a given informal notation such as $\forall y : A.\ B(2, y)$.

In the case of representing a term such as $op(x, y.\ a; y, z.\ b; x, z.\ c)$, the form $op(\lambda(x, y.\ a); \lambda(y, z.\ b); \lambda(x, z.\ c))$ will not represent the binding structure if we replace subterm by α-equivalent ones, e.g.

$$op(\lambda(x, y.\ a); \lambda(u, v.\ a[u/y, v/z]);\quad \lambda(y, x.\ c[y/x, x/z]))$$

alternatives — λcalculus with arities

Martin-Löf has introduced a variant of the approach discussed in section 3.3. The idea is that the lambda calculus with operator names is a description of syntax (he speaks of a *theory of expressions*). $\forall x : A.\ B(x)$ is represented as $all(A, \lambda x.\ B(x))$. The binding structure of the language is given by the *untyped* lambda calculus.

Martin-Löf then notes that the untyped lambda calculus is deficient as a theory of syntax because

- equality of terms is undecidable
- definitions are not eliminable (because of the Y combinator)

He proposes to fix this by an account of *arities.* These are explained in terms of saturated expressions versus expressions with "holes" in them. But we can also view the entire arity apparatus as a simply typed lambda calculus in which the base type,

o, stands for saturated expressions, such as $y + sin(y)$. The type $o \rightarrow o$ stands for expressions with one hole in them, say $\Box + sin(\Box)$. But he indicates these holes using the lambda mechanism, writing $\lambda y.\ y + sin(y)$ as an expression of arity $o \rightarrow o$. An operator such as the integral $\int_a^b y + sin(y)dy$ is analyzed as a *complex* that takes two saturated expressions and one with holes and produces a saturated expression. So it has arity $(o \times o \times (o \rightarrow o)) \rightarrow o$ where $\times$ is used to denote arities of compound expressions.

The arity calculus is the same thing as a simply typed lambda calculus whose types are built from o and the constructors $\rightarrow$ and $\times$. We know that equality is decidable and definitions are eliminable.

4.2 Semantics

The Nuprl semantics is a variation on that given by Martin-Löf for his type theory and formalized by Stuart Allen (see [4]). There are three stages in the semantic specification: the *computation system*, the *type system* and the so–called *judgement forms*. We shall specify a computation system consisting of *terms*, divided into *canonical* and *noncanonical*, and a procedure for *evaluating* terms which for a given term t returns at most one canonical term, called the *value* of t. In Nuprl whether a term is canonical depends only on the outermost form of the term, and there are terms which have no value. We shall write $s\ evals\_to\ t$ to mean that s has value t.

Next we shall abstract from the system of types and equality on types defined in Section 1. A *type* is a term T with which is associated a transitive, symmetric relation, $t = s \in T$, which respects evaluation in t and s; that is, if T is a type and $t\ evals\_to\ t'$ and $s\ evals\_to\ s'$, then $t' = s' \in T$ if and only if $t = s \in T$. We shall sometimes say "T type" to mean that T is a type. We say t is a *member* of T, or $t \in T$, if $t = t \in T$. Note that $t = s \in T$ is an equivalence relation (in t and s) when restricted to members of T.[1] Actually, $t = s \in T$ is a three–place relation on terms which respects evaluation in all three places. We also use a transitive, symmetric relation on terms, $T = S$, called *type equality*, which $t = s \in T$ respects in T; that is, if $T = S$ then $t = s \in T$ if and only if $t = s \in S$. The relation $T = S$ respects evaluation in T and S, and T is a type if and only if $T = T$. The restriction of $T = S$ to types is an equivalence relation.

For our purposes, then, a type system for a given computation system consists of a two–place relation $T = S$ and a three–place relation $t = s \in T$ on terms such that

$T = S$ is transitive and symmetric;
$T = S$ if and only if $\exists T'.\ T\ evals\_to\ T' \ \&\ T' = S$;
$t = s \in T$ is transitive and symmetric in t and s;
$t = s \in T$ if and only if $\exists t'.\ t\ evals\_to\ t' \ \&\ t' = s \in T$;

[1] There is a similarity between a type and Bishop's notion of *set*. Bishop says that to give a set, one gives a way to construct its members and gives an equivalence relation, called the *equality* on that set, on the members.

$T = T$ if $t = s \in T$;
$t = s \in T$ if $T = S$ & $t = s \in S$.

We define "T type" and "$t \in T$" by

T type if and only if $T = T$;
$t \in T$ if and only if $t = t \in T$.

Finally, so–called judgements will be explained. This requires consideration of terms with free variables because substitution of closed terms for free variables is central to judgements as presented here. In the description of semantics given so far "term" has meant a closed term, i.e., a term with no free variables. There is only one form of judgement in Nuprl, $x_1{:}T_1, \ldots, x_n{:}T_n \vdash S$ `[ext s]`, which in the case that n is 0 means $s \in S$. The explanation of the cases in which n is not 0 must wait.

substitution

For the purposes of giving the procedure for evaluation and explaining the semantics of judgements, we would only need to consider substitution of closed terms for free variables, and so we would not need to consider simultaneous substitution or capture. However, for the purpose of specifying inference rules later we want to have simultaneous substitution of terms with free variables for free variables. The result of such a substitution is indicated as in Section 1:

$$t[t_1, \ldots, t_n/x_1, \ldots, x_n]$$

where $0 \leq n$, $x_1, \ldots, x_n$ are variables (not necessarily distinct) and $t_1, \ldots, t_n$ are terms. It is handy to permit multiple occurrences of the same variable among the targets for replacements, all but the last of which are ignored. $t[t_1, \ldots, t_n/x_1, \ldots, x_n]$ is the result of replacing each free occurrence of x_i in t by s_i for $1 \leq i \leq n$, where s_i is t_j with j the greatest k such that x_i is x_k.

the computation system

Figure 1 shows the terms of Nuprl. Variables are terms, although since they are not closed they are not executable. Variables are written as identifiers, with distinct identifiers indicating distinct variables.[2] Nonnegative integers are written in standard decimal notation. There is no way to write a negative integer in Nuprl; the best one can do is to write a noncanonical term, such as `-5`, which evaluates to a negative integer. Atom constants are written as character strings enclosed in double quotes, with distinct strings indicating distinct atom constants.

The free occurrences of a variable x in a term t are the occurrences of x which either *are* t or are free in the immediate subterms of t, excepting those occurrences of x which become *bound* in t. In Figure 1 the variables written below the terms indicate which variable occurrences become bound; some examples are explained below.

[2] An identifier is any string of letters, digits, underscores or at–signs that starts with a letter. The only identifiers which cannot be used for variables are `term_of` and those which serve as operator names, such as `int` or `spread`.

x	n	`void`	
`int`	`atom`	`axiom`	`nil`
`U`k	`inl(`a`)`	`inr(`a`)`	`rec(`A`;` $\underset{x}{x}$`.` $\underset{x}{B}$`)`
$\lambda(\underset{x}{x}.\underset{x}{b})$		`<`a`,`b`>`	
$A \times B$	$\underset{x}{x}:A\times\underset{x}{B}$	A`->`B	$\underset{x}{x}:A$`->`$\underset{x}{B}$
A`+`B	A`//`B	$\underset{x}{x},\underset{y}{y}:A$`//`$\underset{\substack{x\\y}}{B}$	$\{A \mid B\}$
$\{\underset{x}{x}:A \mid \underset{x}{B}\}$	a `=` b `in` A		

canonical if closed

. . . . . .

noncanonical if closed

	`any(`a`)`	t`(`a`)`	a`+`b
	^	^	^ ^
a`-`b	a`*`b	a`/`b	a `mod` b
^ ^	^ ^	^ ^	^ ^

`spread(`a`;`$\underset{\substack{\wedge\\x}}{x}$`,`$\underset{y}{y}$`.`$\underset{\substack{x\\y}}{t}$`)`

`decide(`a`;`$\underset{\substack{\wedge\\x}}{x}$`.`$\underset{x}{s}$`;`$\underset{y}{y}$`.`$\underset{y}{t}$`)`

`rec_ind(a;f,x.g)`

`ind(`a`;`$\underset{\substack{\wedge\\x}}{x}$`,`$\underset{y}{y}$`.`$\underset{\substack{x\\y}}{s}$`;`$b$`;`$\underset{u}{u}$`,`$\underset{v}{v}$`.`$\underset{\substack{u\\v}}{t}$`)`

`less(`a`;`b`;`s`;`t`)`

`int_eq(`a`;`b`;`s`;`t`)`

x, y, u, v	range over variables.
a, b, s, t, A, B	range over terms.
n	ranges over integers.
k	ranges over positive integers.

Variables written below a term indicate where the variables become bound.
"^" indicates principal arguments.

Figure 1: Terms

Lower Precedence

`=`,`in`	left associative
$\times$,`->`, `+`, `//`	right associative
`<` (as in a`<`b)	left associative
`+`,`-` (infix)	left associative
`*`,`/`,`mod`	left associative
`inl`,`inr`,`-` (prefix)	—
. (as in $a\,.\,b$)	right associative
$\lambda x.$	—
`(`a`)` (as in t`(`a`)`)	—

Higher Precedence

Figure 2: Operator Precedence in Abbreviations

- In $x : A \times B$ the x in front of the colon becomes bound and any free occurrences of x in B become bound. The free occurrences of variables in $x : A \times B$ are all the free occurrences of variables in A and all the free occurrences of variables in B except for x.
- In `<`a`,`b`>` no variable occurrences become bound; hence, the free occurrences of variables in `<`a`,`b`>` are those of a and those of b.
- In `spread(`s`;`x`,`y`.`t`)` the x and y in front of the dot and any free occurrences of x or y in t become bound.

Parentheses may be used freely around terms and often must be used to resolve ambiguous notations correctly. Figure 1 gives the relative precedences and associativities of Nuprl operators.

The closed terms above the dotted line in Figure 1 are the canonical terms, while the closed terms below it are the noncanonical terms. Note that carets appear below most of the noncanonical forms; these indicate the *principal argument places* of those terms. This notion is used in the evaluation procedure below. Certain terms are designated as *redices*, and each redex has a unique *contractum*. Figure 1 shows all redices and their contracta.

The evaluation procedure is as follows. Given a (closed) term t,

If t is canonical then the procedure terminates with result t.

Redex	**Contractum**
$\lambda(x\,.\,b)(a)$	$b[a/x]$
`spread(<`a`,`b`>;`x`,`y`.`t`)`	$t[a,b/x,y]$
`decide(inl (`a`) ;`x`.`s`;`y`.`t`)`	$s[a/x]$
`decide(inr (`b`) ;`x`.`s`;`y`.`t`)`	$t[b/y]$
`rec_ind(`a`;`f`,`z`.`b`)`	$b[a/z, \lambda(y\,.$ `rec_ind(`y`;`f`,`z`.`b`))/f]$
`int_eq(`m`;`n`;`s`;`t`)`	s if m is n; t otherwise
`less(`m`;`n`;`s`;`t`)`	s if m is less than n; t otherwise
`-`n	the negation of n
m`+`n	the sum of m and n
m`-`n	the difference
m`*`n	the product
m`/`n	`0` if n is `0`; otherwise, the floor of the obvious rational.
m `mod` n	`0` if n is `0`; otherwise, the positive integer nearest `0` that differs from m by a multiple of n.
`ind(`m`;`x`,`y`.`s`;`b`;`u`,`v`.`t`)`	b if m is `0`; $t[m,$`ind(`$m-1$`;`x`,`y`.`s`;`b`;`u`,`v`.`t`)`$/u,v]$ if m is positive; $s[m,$`ind(`$m+1$`;`x`,`y`.`s`;`b`;`u`,`v`.`t`)`$/x,y]$ if m is negative.

a, b, s, t	range over terms.
x, y, u, v	range over variables.
m, n	range over integers.

Figure 3: Redices and Contracta

Otherwise, execute the evaluation procedure on each principal argument of t, and if each has a value, replace the principal arguments of t by their respective values; call this term s.

If s is a redex then the procedure for evaluating t is continued by evaluating the contractum of s.

If s is not a redex then the procedure is terminated without result; t has no value.

4.3 The Type System

For convenience we shall extend the relation s *evals_to* t to possibly open terms. If s or t contain free variables then s *evals_to* t does not hold; otherwise, it is true if and only if s has value t.

Recall that the members of a type are its canonical members and the terms which have those members as values. The integers are the canonical members of the type `int`. The type `void` is empty. The type A`+`B is a disjoint union of types A and B. The terms `inl(`a`)` and `inr(`b`)` are canonical members so long as $a \in A$ and $b \in B$; a and b need not be canonical. The canonical members of $x : A \times B$ are the terms `<`a`,`b`>` with $a \in A$ and $b \in B[a/x]$, a and b not necessarily canonical. Note that the type from which the second component is selected may depend on the value of the first component.

A term of the form t`(`a`)` is called an *application* of t to a, and a is called its *argument*. The members of x`:`A`->`B are called *functions*, and each canonical member is a *lambda term*, $\lambda(x.\, b)$, whose application to any $a \in A$ is a member of $B[a/x]$. It is required that applications to equal members of type A be equal. Clearly, t`(`a`)`$\in B[a/x]$ if $t \in$ x`:`A`->`B and $a \in A$.

The significance of some constructors derives from the representation of propositions as types. A proposition represented by a type is true if and only if the type is inhabited. The type a`<`b is inhabited if and only if the value of a is less than the value of b. The type (`(`a`=`b `in` A`)`) is inhabited if and only if $a = b \in A$. Obviously, the type (`(`a`=`a `in` A`)`) is inhabited if and only if $a \in A$, so "a `in` A" has been adopted as a notation for this type. The members of $\{x$`:`A`|`$B\}$ are the members a of A such that $B[a/x]$ is inhabited. Types of the form $\{x$`:`A`|`$B\}$ are called *set types*. The set constructor provides a device for specifying subtypes; for example, {`x:int|0<x`} has just the positive integers as canonical members.

The members of x`,`y`:`A`//`B are the members of A. The difference between this type and A is equality. $a = a' \in x$`,`y`:`A`//`B if and only if a and a' are members of A and $B[a, a'/x, y]$ is inhabited. Types of this form are called *quotient types*. The relation $\exists b.\, b \in B[a, a'/x, y]$ is an equivalence relation over A in a and a'; this is built into the criteria for x`,`y`:`A`//`B being a type.

Now consider equality on the other types already discussed. (Recall that terms are equal in a given type if and only if they evaluate to canonical terms equal in that

type. Recall also that $a = a' \in A$ is an equivalence relation in a and a' when restricted to members of A.) Members of `int` are equal (in `int`) if and only if they have the same value. Canonical members of A`+`B ($x : A \times B$) are equal if and only if they have the same outermost operator and their corresponding immediate subterms are equal (in the corresponding types). Members of x:A`->`B are equal if and only if their applications to any member a of A are equal in $B[a/x]$. We say equality on x:A`->`B is *extensional*. The types a`<`b and (`(`a`=`b `in` A`)`) have at most one canonical member, `axiom`. Equality in $\{x : A \mid B\}$ is just the restriction of equality in A to $\{x : A \mid B\}$.

We must now consider the notion of *functionality*. A term B is *type–functional in* x:A if and only if A is a type and $B[a/x] = B[a'/x]$ for any a and a' such that $a = a' \in A$. A term b is *B–functional in* $x : A$ if and only if B is type–functional in $x : A$ and $b[a/x] = b[a'/x] \in B[a/x]$ for any a and a' such that $a = a' \in A$. There are restrictions on type formation involving type–functionality. These can be seen in the type formation clauses for $x : A \times B$, x:A`->`B, and $\{x$:A`+`$B\}$. In each of these B must be type–functional in x:A.[3] We may now say that the members of x:A`->`B are the lambda terms $\lambda(x.b)$ such that b is B–functional in x:A. In the type x`,`y:A`//`B, that B must be type–functional in both x,y:A follows from the fact that x:A`->`y:A`->`B`->`B must be a type. There are also constraints on the typehood of x`,`y:A`//`B which guarantee that the relation $\exists b.\, b \in B[a, a'/x, y]$ is an equivalence relation on members of A and respects equality in A. It should be noted that if A is empty then every term is type–functional in its free variables over A. Hence, x:`void` $\times$ `3` is a type (with no members) even though `3` is not a type.

Equal types have the same membership and equality, but not conversely. Type, etc. The relations that must hold between the respective immediate subterms are seen easily enough in the definition of type equality. It should be noted that in contrast to equality between types of the form $x : A \times B$ or $x : A -> B$, much less is required for $\{x : A \mid B\}$=$\{x : A \mid B'\}$ than type–functional equality of B and B' in x:A. All that is required is the existence of functions which for each $a \in A$ evaluate to functions mapping back and forth between $B[a/x]$ and $B'[a/x]$. Equality between quotient types is defined similarly. If x does not occur free in B then $A \times B$=$x : A \times B$, A`->`B=x:A`->`B if x and y do not occur free in B then A`//`B=x`,`y:A`//`B. As a result there is no need for clauses in the type system description giving the criteria for $t = t' \in A \times B$ and the others explicitly.

Now consider the so–called *universes*, U_k (k positive). The members of U_k are types. The universes are cumulative; that is, if j is less than k then membership and equality in U_j are just restrictions of membership and equality in U_k. Universe U_k is closed under all the type–forming operations except formation of U_i for i greater than or equal to k. Equality (hence membership) in U_k is similar to type equality as defined previously except that equality (membership) *in* U_k is required wherever type equality (typehood) was formerly required, and although all universes are types, only those `U`i such that i is less than k are in U_k. Equality in U_k is the restriction of type equality to members of U_k.

[3] In the formation of these as members of U_k, B must be U_k–functional in x:A.

So far the only noncanonical form explicitly mentioned in connection with the type system is application. We shall elaborate here on a couple of forms, and it should then be easy to see how to treat the others. The `spread` form is used for computational analysis of pairs. The pair of components is *spread* apart so that the components can be used separately.

$$\begin{array}{l}\texttt{spread}(e\,;x\,,y\,.\,t)\in T[e/z] \text{ if}\\ \qquad e \in x\!:\!A \times B\\ \qquad \&\ T \text{ is type functional in } z\!:\!(x\!:\!A \times B)\\ \qquad \&\ \forall a, b.\ t[a, b/x, y] \in T[\texttt{<}a\,,b\texttt{>}/z]\\ \qquad\qquad \text{if } a \in A \text{ and } b \in B[a/x]\end{array}$$

Since $e \in x\!:\!A \times B$, then for some a and b $\texttt{<}a\,,b\texttt{>}\leftarrow e$ where $a \in A$, and $b \in B[a/x]$. Hence $\texttt{spread}(e\,;x\,,y\,.\,t)$ and $t[a, b/x, y]$ have the same value, so it is enough that $t[a, b/x, y] \in T[e/z]$. But from our hypotheses it follows that $t[a, b/x, y] \in T[\texttt{<}a\,,b\texttt{>}/z]$, so it is enough that $T[e/z] = T[\texttt{<}a\,,b\texttt{>}/z]$. Now $e = \texttt{<}a\,,b\texttt{>} \in x\!:\!A \times B$ since $e \in x\!:\!A \times B$ and equality respects evaluation; therefore $T[e/z] = T[\texttt{<}a\,,b\texttt{>}/z]$ in light of T's functionality in $z\!:\!(x\!:\!A \times B)$.

judgments

The significance of judgments lies in the fact that they express the claims of a proof. They are the units of assertion. The judgments of Nuprl have the form

$$x_1\!:\!T_1\,,\ldots,x_n\!:\!T_n \ \vdash\ S\ \ \texttt{[ext } s\texttt{]}$$

where $x_1, \ldots, x_n$ are distinct variables and $T_1, \ldots, T_n, S, s$ are terms (n may be 0), every free variable of T_i is one of $x_1, \ldots, x_{i-1}$, and every free variable of S or of s is one of $x_1, \ldots, x_n$. The list $x_1\!:\!T_1\,,\ldots\,,x_n\!:\!T_n$ is called the *hypothesis list* or *assumption list*, each $x_i\!:\!T_i$ is called a *declaration* (of x_i), each T_i is called a *hypothesis* or *assumption*, S is called the *consequent* or *conclusion*, s the *extract term* (the reason will be seen later), and the whole thing is called a *sequent*.

Before explaining the conditions which make a Nuprl sequent true we shall define a relation $H@l$, where H is a hypothesis list and l is a list of terms, and we shall define what it is for a sequent to be *true at* a list of terms. Allen [3] calls this pointwise functionality.

$$\begin{array}{l}x_1\!:\!T_1\,,\ldots,x_n\!:\!T_n\ @\ t_1, \ldots, t_n \text{ if and only if}\\ \qquad \forall j < n.\ t_{j+1} \in T_{j+1}[t_1, \ldots, t_j/x_1, \ldots, x_j]\\ \qquad\quad \&\ \forall t'_1, \ldots, t'_j.\ T_{j+1}[t_1, \ldots, t_j/x_1, \ldots, x_j] =\\ \qquad\qquad\quad T_{j+1}[t'_1, \ldots, t'_j/x_1, \ldots, x_j]\\ \qquad\qquad\quad \text{if } \forall i < j.\ t_{i+1} = t'_{i+1} \in T_{i+1}[t_1, \ldots, t_i/x_1, \ldots, x_i]\end{array}$$

The sequent

$$x_1\!:\!T_1\,,\ldots,x_n\!:\!T_n \ \vdash\ S\ \ \texttt{[ext } s\texttt{]}$$

is true at $t_1, \ldots, t_n$ if and only if

$$\forall t'_1, \ldots, t'_n.\ (\ S[t_1, \ldots, t_n/x_1, \ldots, x_n] = S[t'_1, \ldots, t'_n/x_1, \ldots, x_n]$$
$$\&\ s[t_1, \ldots, t_n/x_1, \ldots, x_n] =$$
$$s[t'_1, \ldots, t'_n/x_1, \ldots, x_n] \in S[t_1, \ldots, t_n/x_1, \ldots, x_n]\ )$$
$$\text{if } x_1{:}T_1\texttt{,}\ldots\texttt{,}x_n{:}T_n\ @\ t_1, \ldots, t_n$$
$$\&\ \forall i < n.\ t_{i+1} = t'_{i+1} \in T_{i+1}[t_1, \ldots, t_i/x_1, \ldots, x_i]$$

Equivalently, we can say that s is S–functional in $x_1 : \hat{T}_1, \ldots, x_n : \hat{T}_n$ if $x_1{:}T_1\texttt{,}\ldots\texttt{,}x_n{:}T_n\ @\ t_1, \ldots, t_n$. The sequent

$x_1{:}T_1\texttt{,}\ldots\texttt{,}x_n{:}T_n \vdash S$ `[ext s]` is true if and only if
$\forall t_1, \ldots, t_n.\ x_1{:}T_1\texttt{,}\ldots\texttt{,}x_n{:}T_n \vdash S$ `[ext s]` is true at $t_1, \ldots, t_n$

The connection between functionality and the truth of sequents lies in the fact that S is type–functional (or s is S–functional) in $x : T$ if and only if T is a type and for each member t of T, S is type–functional (s is S–functional) in $x :\{x{:}T \mid x\texttt{=}t$ `in` $T\}$. Therefore, s is S–functional in $x : T$ if and only if T is a type and the sequent $x{:}T \vdash S$ `[ext s]` is true.

It is not possible in Nuprl for the user to enter a complete sequent directly; the extract term *must be* omitted. A sequent is never displayed with its extract term. The system has been designed so that upon completion of a proof, a component called the *extractor* automatically provides, or *extracts*, the extract term. This is because in the standard mode of use, the user tries to prove that a certain type is inhabited without regard to the identity of any member. In this mode the user thinks of the type (that is to be shown inhabited) as a proposition, and that it is merely the truth of this proposition that the user wants to show. When one does wish to show explicitly that $a \in A$, one instead shows the type $(a$ `in` $A)$ to be inhabited.

Also, the system can often extract a term from an incomplete proof when the extraction is independent of the extract terms of any unproven claims within the proof body. Of course, such unproven claims may still contribute to the truth of the proof's main claim. For example, it is possible to provide an incomplete proof of the untrue sequent $\vdash$ `1<1 [ext axiom]`, the extract term `axiom` being provided automatically.

Although the term extracted from a proof of a sequent is not displayed in the sequent, the term is accessible by other means through the name assigned to the proof in the user's library.

4.4 Rules

The Nuprl system has been designed to accommodate the top–down construction of proofs by refinement. In this style one proves a judgement (i.e., a *goal*) by applying a *refinement rule*, obtaining a set of judgements called *subgoals*, and then proving each of the subgoals. In this section we will describe the rules. The actual rules are

available at the Nuprl web page. First we give some general comments regarding the rules and then proceed to give a description of each rule.

4.4.1 the form of a rule

To accommodate the top–down style of the proofs the rules of the logic are presented in the following *refinement* style.

$$
\begin{array}{l}
H \vdash T \ \texttt{ext}\ t \ \texttt{by}\ rule \\
\quad H_1 \vdash T_1 \ \texttt{ext}\ t_1 \\
\quad\ \vdots \\
\quad H_k \vdash T_k \ \texttt{ext}\ t_k
\end{array}
$$

The goal is shown at the top, and each subgoal is shown indented underneath. The rules are defined so that if every subgoal is true then one can show the truth of the goal, where the truth of a judgement is to be understood as defined above. If there are no subgoals ($k = 0$) then the truth of the goal is axiomatic.

The rules have the property that each subgoal can be constructed from the information in the rule and from the goal, exclusive of the extraction term. As a result some of the more complicated rules require certain terms as parameters.

Implicit in showing a judgement to be true is showing that the conclusion of the judgement is in fact a type. We cannot directly judge a term to be a type; rather, we show that it inhabits a universe. An examination of the semantic definition will reveal that this is sufficient. Due to the rich type structure of the system it is not possible in general to decide algorithmically if a given term denotes an element of a universe, so this is something which will require proof. The logic has been arranged so the proof that the conclusion of a goal is a type can be conducted *simultaneously* with the proof that the type is inhabited. In many cases this causes no great overhead, but some rules have subgoals whose only purpose is to establish that the goal is a type, that is, that it is *well–formed*. These subgoals all have the form $H \vdash T \ \texttt{in}\ \texttt{U}i$ and are referred to as *well–formedness* subgoals.

4.4.2 web access

The complete set of Nuprl rules is available on the Web under 4.2 Libraries. The explanation given here should make them understandable.

5 Conclusion

At the summer school I presented the details of Max Forester's constructive proof of the Intermediate Value Theorem [20] which was taken from Bishop and Bridges [9]. I also discussed the stamps problem from the Nuprl 4.2 library. I related this to Sam Buss' account of feasible arithmetic by using the efficient induction tactic

(complete_nat_ind_with_y at the end of int_1) [31]. All of this material is now on the Web, and this article should help make it more accessible.

6 Acknowledgments

I want to thank Kate Ricks for preparing this manuscript and Karla Consroe and Kate for helping with the Nuprl Web page.

References

[1] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.

[3] Stuart F. Allen. *A non-type-theoretic semantics for type-theoretic language*. PhD thesis, Cornell University, 1987.

[4] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 215–224. IEEE, June 1987.

[5] R. C. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory (part I). *Formal Aspects of Computing*, 1:19–84, 1989.

[6] M. J. Beeson. Formalizing constructive mathematics: Why and how? In F. Richman, editor, *Constructive Mathematics,* Lecture Notes in Mathematics, Vol. 873, pages 146–90. Springer, Berlin, 1981.

[7] M.J. Beeson. *Foundations of Constructive Mathematics*. Springer Berlin, 1985.

[8] U. Berger and H. Schwichtenberg. Program extraction from classical proofs. In Daniel Leivant, editor, *Logic and Computational Complexity*, pages 77–97. Springer, Berlin, 1994.

[9] E. Bishop and D. Bridges. *Constructive Analysis*. Springer, New York, 1985.

[10] S. Buss. The polynomial hierarchy and intuitionistic bounded arithmetic. In *Structure in Complexity Theory,* Lecture Notes in Computer Science. 223, pages 77–103. Springer, Berlin, 1986.

[11] Robert L. Constable. Types in Logic, Mathematics and Programming. In S. Buss, editor, *Handbook of Proof Theory*, North Holland, 1998.

[12] Robert L. Constable. Using reflection to explain and enhance type theory. In Helmut Schwichtenberg, editor, *Proof and Computation*, pages 65–100, Berlin, 1994. NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20-August 1, 1995, NATO Series F, Vol. 139, Springer, Berlin.

[13] Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[14] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, London, 1990.

[15] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic,* Lecture Notes in Computer Science, Vol. 417, pages 50–66. Springer, Berlin, 1990.

[16] N. G. deBruijn. A survey of the project Automath. In *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[17] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Proc. of the First Annual Workshop on Logical Frameworks*, pages 280–306, Sophia-Antipolis, France, June 1990. Programming Methodology Group, Chamers University of Technology and University of Göteborg.

[18] Peter 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.

[19] Solomon Feferman. A language and axioms for explicit mathematics. In J. N. Crossley, editor, *Algebra and Logic,* Lecture Notes in Mathematics, Vol. 480, pages 87–139. Springer, Berlin, 1975.

[20] Max B. Forester. Formalizing constructive real analysis. Technical Report TR93-1382, Computer Science Dept., Cornell University, Ithaca, NY, 1993.

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

[22] Jason J. Hickey. Objects and theories as very dependent types. In *Proceedings of FOOL 3*, July 1996.

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

[24] 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.

[25] Douglas J. Howe. Equality in lazy computation systems. In *Proc. of Fourth Symp. on Logic in Comp. Sci.*, pages 198–203. IEEE Computer Society, June 1989.

[26] Douglas J. Howe. Reasoning about functional programs in Nuprl. *Functional Programming, Concurrency, Simulation and Automated Reasoning,* Lecture Notes in Computer Science, Vol. 693, Springer, Berlin, 1993.

[27] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In *Proceedings of AMAST'96*, 1996. To appear.

[28] Douglas J. Howe and Scott D. Stoller. An operational approach to combining classical set theory and functional programming languages. In and J. C. Mitchell M. Hahiya, editor, Lecture Notes in Computer Science, Vol. 789, pages 36–55, New York, April 1994. International Symposium TACS '94, Springer, Berlin. Theoretical Aspects of Computer Software.

[29] G. Huet. A uniform approach to type theory. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 337–398. Addison-Wesley, 1990.

[30] G. P. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[31] Paul B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, Ithaca, NY, January 1995.

[32] Daniel Leivant. Intrinsic theories and computational complexity. In Daniel Leivant, editor, *Logic and Computational Complexity*, pages 177–194. Springer, Berlin, 1994.

[33] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982.

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

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

[36] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Sciences Publication, Oxford, 1990.

[37] Erik Palmgren. *On Fixed Point Operators, Inductive Definitions and Universes in Martin-Löf's Type Theory.* PhD thesis, Uppsala University, Thunbergsvägen 3, S-752, Uppsala, Sweden, March 1991.

[38] A. M. Pitts. *Operationally-Based Theories of Program Equivalence.* University of Cambridge, Cambridge, UK, 1995.

[39] Helmut Schwichtenberg. *Computational Content of Proofs.* Mathematisches Institut, Universität München, München, Germany, 1995. Working material for Marktoberdorf lecture.

[40] D. Scott. Constructive validity. In D. Lacombe M. Laudelt, editor, *Symposium on Automatic Demonstration,*Lecture Notes in Mathematics, Vol. 5 #3, pages 237–275, New York, 1970. Springer-Verlag.

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

[42] Anton Setzer. *Proof theoretical strength of Martin-Löf Type Theory with W-type and one universe.* PhD thesis, Ludwig-Maximilians-Universität, München, September 1993.

[43] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[44] S.S. Wainer. The hierarchy of terminating recursive programs over N. In Daniel Leivant, editor, *Logic and Computational Complexity,* Lecture Notes in Computer Science, Vol. 959, pages 281–299. Springer, Berlin, 1994.