These documents explain the basic concepts and methods used for mathematical expression in Nuprl. The primitive operators as well as basic nonprimitives are discussed.

See Structure, Principles and Printed Copies of the Basic Documents {2}.

Below is a list of the main documents in a reasonable reading order. An approximate size in words is indicated for each, indicating a certain kind of emphasis. There is also a partial index to these documents by Emblematic Theorems {3}, some of which are quite interesting.

Main Documents Created as of September 2003 {62} - sfa:

Pairs {4}	210 words
Lists {5}	230 words
Boolean {6} (Deriving Bool {7})	290 (20) words
Unit {8} (Deriving Unit {9})	50 (60) words
Integers {10} (Negative Literals {11})	160 (160) words
Atoms {12}	290 words
Disjoint Union {13}	140 words
Comprehension Subtypes {14}	70 words
Types {15}	760 words
Type Expressions {16}	430 words
Intensional Types {17}	390 words
Universes {18}	390 words
Equality {19}	250 words
Intersection Types {20}	240 words
Quotient Types {21}	130 words
Void {22}	170 words
Functions {23}	590 words
Function Types {24}	380 words
Polymorphism {25}	490 words
Multi-Argument Functions {26}	280 words
Recursive Functions {27}	220 words
Recursion via the Y-combinator {28}	210 words
Y-combinator Continued {29}	340 words
Typing Y {30}	310 words
Typing Y Continued {31}	250 words
Multi-argument Functions with Y {32}	170 words
Recursive Types {33}	270 words
Propositions {34}	360 words
Higher-Order Propositions {35}	320 words
Guarded Types {36}	180 words
Functionality Sequents {37}	290 words
Type Functionality Sequents {38}	510 words
Prop Levels and Predicativity {39}	350 words
Russell's Orders {40}	330 words
Terms {41}	340 words

Computation {42}	360 words
Canonical Forms {43}	60 words
NonCanonical Forms {44}	140 words
Evaluation and Term Rewriting {45}	310 words
"Squiggle" Operator {46}	400 words
Operator Definition {47}	210 words
Substitution {48}	480 words
Op Def Restrictions {49}	420 words
Sequents {50}	430 words
Proofs {51}	530 words
ML {52}	70 words
Sequents with Type Conclusions {53}	280 words
Sequents with Restricted Declarations {54}	320 words
Proof Witness Extraction {55}	270 words
Propositions as Types {56}	520 words
Propositions as Types - standard defs {57}	160 words
Propositional Proof Witness Extraction {58}	340 words

Basic Nuprl Documentation: Structure and Printable Copies

The Nuprl Basics {1} documents explain the basic concepts and methods used for mathematical expression in Nuprl. The primitive operators as well as basic nonprimitives are discussed.

Structure.

These documents were generated according to three principles. First, they were generated progressively starting from the basic operators, and thereafter being generated as needed in order to continue the explanations. Second, concepts and notations that are standard were often used without first explaining them in type theoretic terms, in order not to clutter up the narrative; such elaboration is provided later in the explanations. Third, as what might be considered a corrolary of the second principle, the core theoretical connection between types and propositions was left unaddressed for as long as seemed practical. (Indeed, as of September 2003, the explanation of propositions-as-types is unfinished, but there is plenty explained independently of this.)

Thus, the overall effect is intended to be a progression from the ordinary and familiar towards more theoretically imbued explanations.

In addition to the links within pages to narratively "adjacent" ones there are, appended to each page, links to the documents explaining basic operators that occur on the page. These links are displayed as instances of those operators.

Nuprl Basics {1} contains a list of the main documents in a reasonable reading order. An approximate size in words is indicated for each, indicating a certain kind of emphasis.

Citations. I'll add them later.

Printed copies.
There is a print version (circa 70pp) (postscript) of the finished documents with cross-references, and a longer one (circa 120pp) (postscript) including mentioned proofs.

primitive_ops_emblem_thms {3}

This is a list of selected documents from Nuprl Basics {1} along with some theorems mentioned in them that are in some ways emblematic.

<table>
<tr><td>Pairs {4}</td><td>Thm* ∀B:(A→Type), p:(u:A×B(u)).
p = <p.1,p.2> ∈ a:A×B(a)</td></tr>
<tr><td>Boolean {6}</td><td>Thm* ∀p,q:𝔹. (p$\wedge_2$q) ⟺ p & q</td></tr>
<tr><td>Unit {8}</td><td>Thm* ∀a:Unit. a = ·</td></tr>
<tr><td>Disjoint Union {13}</td><td>Thm* ∀z:A+B. z = InjCase(z; u. inl(u); v. inr(v))</td></tr>
<tr><td>Equality {19}</td><td>Thm* (λx.x) = (λx.if x<0→ 0 ; x fi) ∈ ℕ→ℕ
Thm* (λx.x) ≠ (λx.if x<0→ 0 ; x fi) ∈ ℤ→ℤ</td></tr>
<tr><td>Intersection Types {20}</td><td>Thm* ∀B:(A→Type), a:A. (∩x:A. B(x)) ⊆ B(a)
Thm* ∀A:Type, B:Type(given A). A→B ∈ Type</td></tr>
<tr><td>Quotient Types {21}</td><td>Thm* 2 = 4 ∈ ℤ mod 2</td></tr>
<tr><td>Void {22}</td><td>Thm* (λx.whatever) = (λx.x) ∈ Void→A</td></tr>
<tr><td>Function Types {24}</td><td>Thm* ∀f:(A→B). (λx.f(x)) = f
Thm* ∀f,g:(A→B). f = g ⟺ (∀x:A. f(x) = g(x))</td></tr>
<tr><td>Polymorphism {25}</td><td>Thm* ∀f:((ℤ mod 2)→A). f ∈ ℤ→A
Thm* ∀T:Type, f:(∩D:Type. D→T). f(f) ∈ T
Thm* ∀f:(∩D:Type. D→T), A:Type, a:A, B:Type, b:B.
f(a) = f(b) ∈ T
Thm* ∀f:(∩C,D:Type. C×D→D×C), A,B:Type, a:A, b:B.
f(<a,b>) = <b,a> ∈ B×A
Thm* A ⊆ Top</td></tr>
<tr><td>Recursive Funs {27}</td><td>Thm* ∀x:ℕ. x! ∈ ℕ
Thm* mu ∈ {f:(ℕ→𝔹)| ∃x:ℕ. f(x) }→ℕ
Thm* ∀A:Type, n:ℕ. (A^n) ∈ Type</td></tr>
<tr><td>Recursive Types {33}</td><td>Thm* ∀A:Type. Sexpr(A) ∈ Type
Thm* ∀a,b,c:X.
Reverse(Cons(Inj(a);Cons(Inj(b);Inj(c))))
=
Cons(Cons(Inj(c);Inj(b));Inj(a))</td></tr>
<tr><td>Propositions {34}</td><td>Thm* ∀A,B:Prop. (A & B) ∈ Prop
Thm* ∀A:Type, P:(A→Prop). (∃x:A. P(x)) ∈ Prop</td></tr>
<tr><td>High-Order Props {35}</td><td>Thm* A ∨ B ⟺ (∀C:Prop. (A ⇒ C) ⇒ (B ⇒ C) ⇒ C)</td></tr>
<tr><td>Guarded Types {36}</td><td>Thm* ∀A:Prop, B:Prop(given A). (A & B) ∈ Prop</td></tr>
<tr><td>Type Functionality {38}</td><td>Thm* ∀A,B:Type{i}. A+B ∈ Type{i}
Thm* Type{i} ∈ Type{i'}</td></tr>
<tr><td>Prop Levels {39}</td><td>Thm* Prop{i} ∈ Type{i'}
Thm* Prop{i} ⊆ Prop{i'}
Thm* ∀A,B:Prop{i}.
(∀C:Prop{i}. (A ⇒ C) ⇒ (B ⇒ C) ⇒ C) ∈ Prop{i'}</td></tr>
<tr><td>Type Conclusions {53}</td><td>Thm* x:(ℤ List)→{y:ℤ| y greater-bounds x }</td></tr>
<tr><td>Restricted Decls {54}</td><td>Thm* ∩k:ℕ. x:(ℕk List)→{y:ℕ(k+1)| y greater-bounds x }</td></tr>
<tr><td>Extraction {55}</td><td>Thm* ext{sfa_doc_find_intlist_great_bound}
∈ x:(ℤ List)→{y:ℤ| y greater-bounds x }</td></tr>
<tr><td>Props as Types {56}</td><td>Thm* P ∨ Q ⇒ (∃x:ℕ. (x = 0 ⇒ P) & (x ≠ 0 ⇒ Q))
Thm* Dec(P) ⟺ (∃b:𝔹. P ⟺ b)</td></tr>
<tr><td>Prop Witnesses {58}</td><td>Thm* ∀x:ℕ. ∃y:ℕ. y·y≤x & x<(y+1)·(y+1)</td></tr>
</table>

(Oct 2001 {62} - sfa)

Nuprl's standard pair forming operation is <a,b>. The basic destructor for pairs, sometimes called spread, supplies a pair's components to a function, and is expressed as an operator binding two variables

```
p/x,y. f(x;y)
```

which evaluates thus:

```
<a,b>/x,y. f(x;y) *⇒ f(a;b)
```

The projection functions are special cases:

```
p.1 is p/x,y. x
p.2 is p/x,y. y

<a,b>.1 *⇒ a
<a,b>.2 *⇒ b
```

The "cartesian product" type A×B comprises pairs <a,b> where a ∈ A and b ∈ B. This is really just a special case of u:A×B(u) in which variable u does not occur free in the second type expression. The type u:A×B(u) comprisese pairs <a,b> where a ∈ A and b ∈ B(a). The expression u:A×B(u) denotes a type when A denotes a type and, further, B(u) denotes a type-valued function in u:A. Thus, if A denotes the empty type {22}, then u:A×B(u) does too, no matter what B(u) is (so long as no variables other than u are free in B(u)).
This type constructor is often called the "general sum" of a family of types, and indeed the most common notation for it is "**Σ**u:A. B(u)".

Here's a theorem about pairing projections, complementing the above computational facts about projecting off of pairs:

```
Thm* ∀B:(A→Type), p:(u:A×B(u)). p = <p.1,p.2> ∈ a:A×B(a)
```

Tupling.

We adopt the convention for forming standard tuples by iterating through the second component, so the triple <a,b,c> is a pair having <b,c> as its second component. The notation for spreading tuples is extended as well.

```
∀a:A, b:B, c:C. <a,b,c> ∈ A×B×C
```

or more generally,

```
∀A:Type, B:(A→Type), C:(x:A→B(x)→Type), a:A, b:B(a), c:C(a,b).
<a,b,c> ∈ u:A×v:B(u)×C(u,v)
```

The notation for spreading tuples is extended as well:

```
t/x,y,z. f(x,y,z) abbreviates t/x,#. #/y,z. f(x,y,z)
```

hence

<a,b,c>/x,y,z. f(x,y,z) *⇒ <b,c>/y,z. f(a,y,z) *⇒ f(a,b,c).

(Aug 2003 {62} - sfa)

Lists list_doc {5}

"A List" is a type of finite sequences of members of type A, employing the conventional list constructions from programming.

For a type A,

a.as ∈ A List is the list whose head is a ∈ A, and whose tail is as ∈ A List.

nil ∈ A List is the empty list.

Of course one often sees iterated list expressions, which we display like

[1+2; 0; 42]

If the last tail-nested expression is not the constant nil it is displayed like

[1+2; 0; 42/ f(x)]

Here is a destructor for lists:

Case of X; nil → e ; x.y → f(x,y)

It combines deciding emptiness with list splitting.

Case of nil; nil → e ; x.y → f(x,y) *⇒ e

Case of a.as; nil → e ; x.y → f(x,y) *⇒ f(a,as)

There are (derived) operations for treating lists as finite functions, namely a selector "xs[i]" and a list maker "f{ℕn}" from functions.

Thm* ∀n:ℕ, f:(ℕn→T), i:ℕn. (f{ℕn})[i] = f(i)

Thm* ∀as:T List. (λi.as[i]){ℕ||as||} = as

where ||as|| is list length.

Note on primitive recursion over lists.

There is a primitive, but little used, operation for doing recursion on lists. It is not much used because we tend to use more general methods for defining Recursive Functions {27} but this specific recursion operator has some historical and pedagogical value. Also, in some other type systems, general recursion is not expressible, and such structural recursion operations must be built in.

Case of X; nil → b ; x.y, rec:v → f(x;y;v) is the form of list recursion.

Case of nil; nil → b ; x.y, rec:v → f(x;y;v) *⇒ b

```
Case of a.as; nil → b ; x.y, rec:v → f(x;y;v)
*⇒
f(a;as;Case of as; nil → b ; x.y, rec:v → f(x;y;v))
```

Notice that there is a third argument to the list-splitting case; the application of the recursive function on the tail is passed in through it. Actually, our (Case of X; nil → e ; x.y → f(x,y)) operator is simply the special case of this recursion operator in which the recursion on the tail is ignored.

(Sept 2001 {62} - sfa)

The Type of Booleans bool_doc {6}

$\mathbb{B}$ is a standard two-valued type.

Thm* $\text{true}_2 \in \mathbb{B}$

Thm* $\text{false}_2 \in \mathbb{B}$

Thm* $\forall b{:}\mathbb{B}.\ b = \text{true}_2 \lor b = \text{false}_2$

There is a conditional expression whose execution branches on $\mathbb{B}$ values:

if X→ t else r fi

if true_2→ t else r fi *⇒ t

if false_2→ t else r fi *⇒ r

It happens that $\mathbb{B}$ is not primitive, but there is rarely need for users to know this. See Deriving Bool {7}.

A lot of Tactic {51} support has been developed for $\mathbb{B}$. It is intimately related to constructive decidability.

The Nuprl logic is "constructive" and does not assume that every proposition is either true or false, even when well-defined (see Propositions {34}).
More acutely, there is no method computationally for branching on the truth or falsity of an arbitrary proposition. So, where one would be inclined to branch on the truth value of a proposition, one instead attempts to formulate the concepts using the type $\mathbb{B}$ instead of the type Prop; this is not always possible.

To mediate between the type of booleans and the type of propositions, we employ a coercion operator,

Def ↑b == if b→ True else False fi

where True and False are standard true and false propositions.

Normally, except when showing definitions, we suppress the display of the coercion operator itself because it is usually clear whether a boolean or a propositional expression is needed, and it becomes noisy to read. The well-formedness theorem for this coercion is Thm* ∀b:𝔹. b ∈ Prop. The aptness of making the 𝔹→Prop coercion operator invisible is accentuated when we consider that we are in theory free to add $true_2$ and $false_2$ as new propositions themselves, which would put them directly into Prop and make 𝔹 a subtype of Prop, thus eliminating the need for defining a coercion operator at all.

Here is a sample theorem employing the (invisible) coercion operator to relate Propositional {34} conjunction to boolean conjunction (Def $p \wedge_2 q$ == if p→ q else $false_2$ fi):

Thm* ∀p,q:𝔹. ($p \wedge_2 q$) ⟺ p & q.

Here are the implicit 𝔹-to-Prop coercions in this theorem statement:

∀p,q:𝔹. ↑($p \wedge_2 q$) ⟺ ↑p & ↑q.

(May 2001 {62} - sfa)

The Derivation of Booleans - continued from Boolean {6} bool2_doc {7}

Def 𝔹 == Unit+Unit

where Unit is type with one value "·" (see Unit {8} and Disjoint Union {13}).

Def $true_2$ == inl(·)

Def $false_2$ == inr(·)

Def if b→ t else f fi == InjCase(b ; t; f)

(Feb 2001 {62} - sfa)

Unit unit_doc {8}

This type has one value.

Thm* · ∈ Unit

Thm* ∀a:Unit. a = ·

It happens to be derived rather than primitive. See Deriving Unit {9}.

There are many unit types of course, and this one has simply been adopted as a standard such type in by Nuprl users.

See Polymorphism {25} about another basic unit type, "Top".

(Aug 2001 {62} - sfa)

Unit is derived rather than primitive. unit2_doc {9}

This type has one value.

Thm* · ∈ Unit

Thm* ∀a:Unit. a = ·

There happen already to be other single element types. Propositions expressing equality and membership propositions are often represented as types that have a member iff they are true, and when they are true, the sole member is "*".
See Propositions {34}

Def Unit == 0 ∈ ℤ

Def · == *

Thm* ∀a:(0 ∈ ℤ). a = *

(Feb 2001 {62} - sfa)

Integers int_doc {10}

ℤ is the type of integers; e.g. 1 ∈ ℤ and -1 ∈ ℤ.

The usual arithmetic operators that are primitive in this system are these:

```
-a
a+b
ab
a·b
a ÷ b
a rem b
```

The following comparison operators happen also to be primitive in this system:

```
if a=b→ s ; t fi
if a<b→ s ; t fi
a<b
```

The two "conditionals" simply compare integers and take the appropriate branch. The "a<b" form represents the less-than relation on integers. See Propositions {34}.

```
if 0=0→ s ; t fi *⇒ s
if 0=1→ s ; t fi *⇒ t
if 0<1→ s ; t fi *⇒ s
if 0<0→ s ; t fi *⇒ t
```

There are also some often-used Boolean {6} expressions for these integer relations:

Def $i =_2 j$ == if i=j→ $true_2$; $false_2$ fi
Def $i <_2 j$ == if i<j→ $true_2$; $false_2$ fi

Def $i \leq_2 j$ == $\neg_2 j <_2 i$

Several Subtypes {14} of $\mathbb{Z}$ are in common use.

Def $\mathbb{Z}^{-0}$ == {i:$\mathbb{Z}$| i $\neq$ 0 }
Def $\mathbb{N}$ == {i:$\mathbb{Z}$| 0$\leq$i }
Def $\mathbb{N}^{+}$ == {i:$\mathbb{Z}$| 0<i }
Def {i...j} == {k:$\mathbb{Z}$| i$\leq$k & k$\leq$j }
Def {i..j^{-}} == {k:$\mathbb{Z}$| i $\leq$ k < j }

"$\mathbb{N}$k" is an abbreviated display for {0..k^{-}}, which has been adopted as Nuprl's standard finite type with k $\in$ $\mathbb{N}$ elements.

Integer Literals

In Nuprl the canonical form for an integer is atomic, rather than, say, bit strings or iteration of successor from zero. There is a small complication involving the implementation of negative literals, which will be of interest only to those interested in reconciling the implementation with theory. See Negative Literals {11}.

(March 2001 {62} - sfa)

Negative Integer Literals - continued from Integers {10}   int2_doc {11}

These matters are "below the radar" in practice, and of concern only to those interested in detailed theoretical grounding of the computation rules.

There is a complication with negative integer literals; they are not implemented.

Although in theory (i.e., the theory in terms of which the semantics of the system is defined) each integer is represented as an atomic literal, we did not in fact implement negative literals. This complicates the rules for calculating with negative valued results somewhat.

The theoretical explanation of how to execute, e.g., m+n goes like this:

Execute the expressions m and n.
If the results are (atomic) integer literals, then return the integer literal corresponding to the sum.

The actual method of computation employed in proofs employs the non-canonical "-k" form, applied to a positive literal, in place of negative literals. Of course, the rules are still valid with respect to the theoretical execution method.

(Feb 2001 {62} - sfa)

"abc", "123", "d & $_2$"
These are samples of expressions that we consider, whenever they are used, to be denoting discrete atomic entities, despite the fact that we can see characters constituting the expressions. The use of such characters permits some mnemonic exploitation.
However, there are no constructors or destructors in the theory for these atoms, nor is there a "new atom" generator. There is only a method for checking identity. If one uniformly replaced all the atom constants throughout a statement in Nuprl, preserving identity and difference, the result would still be true (or false) if the original was.

"if a=b∈Atom→u; v fi" compares such atomic expressions a and b and continues by executing expression u or v accordingly.

if "abc"="abc"∈Atom→1; 2 fi *⇒ 1

if "abc"="def"∈Atom→1; 2 fi *⇒ 2

There is also an often-used Boolean {6} expression for atom equality:

Def x=$_2$y∈Atom == if x=y∈Atom→true$_2$; false$_2$ fi

Thm* ∀x,y:Atom. x=$_2$y∈Atom ∈ $\mathbb{B}$

Atom is the type of such atoms.

It's a bit subtle, in theory, but natural in practice.

There is no assumption pertaining to finiteness. One cannot prove that the type is finite or that it is not. However, for any number one can prove there are at least that many members of Atom, simply by showing that many examples chosen at will. This is odd.

Perhaps the best interpretation of Atom is to consider it not as a specific type, but simply as a variable of the semantics that, given a body of discourse, can be interpreted as any discrete type (i.e., a type with decidable equality) having at least as many elements as there are distinct atom constants actually used in the given body of discourse.

Another possible interpretation is as a primitive open-ended type whose members are introduced one-by-one as needed. Thus, one can always find a new member, but there's no systematic function for producing it. This seems as sensible as the intuitionistic notion of free choice sequence.

(Feb 2001 {62} - sfa)

A+B is called the disjoint union of type A with type B.
The members of A+B are gotten by tagging members from A and B. They are inl(a) for a ∈ A and inr(b) for b ∈ B.

inl(a) and inr(b) are sometimes called the left- and right-injections of a and b.

InjCase(x; u. e(u); v. f(v))
is the destructor for left- and right-injections.

InjCase(inl(a); u. e(u); v. f(v)) *⇒ e(a)
InjCase(inr(b); u. e(u); v. f(v)) *⇒ f(b)

Thm* ∀z:A+B. z = InjCase(z; u. inl(u); v. inr(v))

Some commonly used specializations of InjCase(x; u. e(u); v. f(v)) are:

Def isl(x) == InjCase(x; y. true$_2$; z. false$_2$)
Def outl(x) == InjCase(x; y. y; z. "???")
Def outr(x) == InjCase(x; y. "???"; z. z)

Thm* ∀A,B:Type, x:A+B. isl(x) ∈ $\mathbb{B}$
Thm* ∀A,B:Type, x:A+B. isl(x) ⇒ outl(x) ∈ A
Thm* ∀A,B:Type, x:A+B. $\neg_2$isl(x) ⇒ outr(x) ∈ B

outl(inl(a)) *⇒ a
outr(inr(b)) *⇒ b

A+A is essentially the type A with a bit added.

We treat A+B, inl(a), inr(b), and InjCase(x; u. e(u); v. f(v)) as primitives, but we COULD instead have defined them using the pairing operation to add a tag:

Let A+B be isleft:$\mathbb{B}$×if isleft→ A else B fi.

Let inl(a) be <true$_2$,a>.

Let inr(b) be <false$_2$,b>.

Let InjCase(x; u. e(u); v. f(v)) be

x/isleft,z. if isleft→ e(z) else f(z) fi.

(By $\mathbb{B}$ we mean a two-element Boolean {6} type.)

(Sept 2001 {62} - sfa)

Subtypes by Comprehension - sometimes called "set types"   set_type_doc {14}

{x:A| B(x) } is the subtype of A restricted to the Property {34} B(x) (x ∈ A).

It is the type of a ∈ A such that B(a).

Note that "a ∈ {x:A| B(x) }" is not equivalent to "B(a)", since "B(a)" can be assigned a meaning for any a ∈ A, but "a ∈ {x:A| B(x) }" is not assigned a meaning unless it is true, i.e., when "B(a)" is true.

See Polymorphism {25}, Equality {19}, Types {15} and Guarded Types {36}.

(Feb 2001 {62} - sfa)

Types: Ontic, Semantic, and Intensional   type_doc {15}

A type in computational type theory is essentially a class of values, such a class being the domain of some function. Further, since we adopt a constructive account of functions and classes, the type must be a method for notating such values, and in such a way that whatever computations one might apply to those values can be expressed as a computation on those notations.

Because of the intimate connection between values and notations in constructive mathematical notation, there are two manners, or "moods", of discourse involving types (ontic and semantic). There are two corresponding notions of "type", (ontic and semantic), plus a third notion of type, "intensional type", which is exploited in the formulation of Nuprl's logic (see Intensional Types {17}). The word "type" is used for any of these, as well as being used to mean an expression referring to a type in any of these senses (see Type Expressions {16}).

Ontic Types and Ontic Mood.

The ontic "mood" is that of a priori discourse about values; it is the one most like informal, pre-theoretical discourse. Thus, one talks about numbers, functions, pairs, etc. In this mood, a type is a class of values, and a type expression stands for such a class.
In the ontic mood, "∃x:ℤ. 0 = x·x ∈ ℤ" might be glossed as simply "Some integer multiplied by itself is 0". More simply, "1+2 = 3 ∈ ℤ" might be glossed as "the sum of integers 1 and 2 is 3".
It is the formalization of the ontic mood that the logic of computational type theory aims at, but to do this carefully requires us to engage in a form of semantic discourse.

Semantic Types and Semantic Mood.

The semantic mood is that of discourse about expressions and reference. For example, "1+2 = 3 ∈ ℤ" might be glossed as "the expressions 1+2 and 3 denote the same value of type ℤ"; or we might say "the expressions 1+2 and 3 satisfy the symmetric-transitive relation on terms stipulated as the meaning of type expression ℤ" if we want to emphasize the general form of the semantics of types used in computational type theory.

The general semantic form we adopt for types is to associate with each type expression a so-called partial equivalence relation, i.e. a symmetric transitive relation, on expressions; this relation is called the "equality" of the type. The semantics of computational type theory requires a notion of "expression" whose syntax and execution semantics is independent of the definition of types. Restrictions on candidates for an equality relation include that any expression satisfying the relation is closed, i.e. has no free variables, and that if $t \in T$ and t' is Kleene-equal to t, then $t = t' \in T$. Two expressions are Kleene-equal just when, if either one yields a value upon execution, the other yields the same value. See Computation {42}. This semantic form is directly exploited in Comprehension Subtypes {14} and Quotient Types {21}.

As a practical matter, Nuprl further requires that rewriting any subexpression of an expression, of whatever type, by a step of computation must result in an "equal" term of that type. It turns out that this constraint will follow from a few painless restrictions on the computation and type definition, which we do not further discuss here; this result is due to Doug Howe (see also "Squiggle" {46}). It is also required that change of bound variables results in an equal expression; while this must be considered at type definition time, it does not complicate type definition, and you'd have to go out of your way when designing the type system to violate it.

Intensional Types {17} are partial equivalence classes of Type Expressions {16}.

Ontic vs Semantic.

So, in the ontic mood, we may truly say that the boolean type $\mathbb{B}$ has exactly two members, whereas in the semantic mood, this would be false, since one can always make a larger expression that stands for a given value (if $t \in T$ then $t = (\lambda x.x)(t) \in T$, for example).
In the semantic mood, "$0 \in \mathbb{N}\rightarrow\mathbb{N}$" is false, whereas in the ontic mood, "$0 \in \mathbb{N}\rightarrow\mathbb{N}$" is assigned no sense at all. See Equality {19}.

One tends to use both moods since in order to be able to talk intuitionistically about a class of values, one must adopt a notation for those values that is computationally adequate.

Standard examples are natural numbers and functions.
A notation for natural numbers counts as constructively adequate just when one can effectively determine from any notation for a number whether that number is zero, and if it is not, one can effectively find a notation for its predecessor.
Let us suppose we have such a method for notating numbers, and call it $\mathbb{N}$.
A notation for a function, say from naturals to naturals, using the notational method $\mathbb{N}$, counts as constructively adequate just when, from any expression $n \in \mathbb{N}$ one can effectively find an expression of $\mathbb{N}$ for the value of that function on n.

(March 2001 {62} - sfa)

Type Expressions type_expr_doc {16}

In Nuprl, as in Martin-Lof's 1980 type system, our expressions for Types {15} are taken from the same general syntax of Expressions {41} used for other values in the theory. This means that type expressions may be freely mixed with others. When one defines a type, one also specifies an expression to denote it. We sometimes use "A =ext B" to express equality between extensional types, i.e. for type expressions A and B, "A =ext B" holds when they denote the same partial equivalence relation. That is, A =ext B iff for all terms r and s, r = s $\in$ A iff r = s $\in$ B. See also the Subtype Relation {59}.

Midway through Types {15} some syntactic and computational restrictions are given for membership of, and equality between, expressions in a type, involving symmetry and transitivity as well. The same restrictions apply to type expressions themselves, and equality between type expressions (whether we are discussing extensional or Intensional Types {17}).

The type expression $\mathbb{Z}$ is introduced as simply the class of integer literals, along with whatever terms evaluate to them.

But by far, the most common type definitions explain large classes of types given in terms of "constituent" types; these methods for specifying types are called "type constructors". For example, we stipulate that given a type A, there is a type "A List" whose members represent lists of members of type A, as described in Lists {5}. Naturally, our notation "A List" for a list type has as a component a notation "A" for its constituent type.

We use the notation "A+B" for a Disjoint Union {13} type when "A" and "B" denote types. This is a type constructor using two constituent types.

But it is also possible to define type constructors from a possibly infinite family of constituent types. For example, the type u:A$\times$B(u) is a type of pairs, as explained in Pairs {4}, given any type A and any family of types B(x) (for x $\in$ A). (It would be displayed as "A$\times$B" when B doesn't mention the binding variable.) One constituent type of this construction would be A, but also for each x $\in$ A, B(x) is a constituent. Notationally, "u:A$\times$B(u)" is a binding operator, in which the variable "u" binds any free occurrences "B(u)".

Here is an example of a type with infinitely many constituent types.

Consider this type-valued function A^n designed to represent n-tuples of A's (see Recursive Functions {27})

```
A^0 *⇒ Unit
A^1 *⇒ A
A^2 *⇒ A×A
A^3 *⇒ A×A×A
et cetera.
```

The type "n:$\mathbb{N}\times$A^n" has as its constituent types $\mathbb{N}$ and A^n for each n $\in \mathbb{N}$. This type represents all the finite tuples of A's each tagged with its tuple length.

The relation between a type and its constituent types is implicitly intended to be well-founded, i.e., the procedure of iteratively choosing a constituent type must always end.

(Sept 2001 {62} - sfa)

Type Expressions {16} denote partial equivalence relations between expressions (see Types {15}). Such a relation serves as a way of notating values constructively. This engenders a natural equivalence relation between type expressions themselves, namely extensional equality, i.e. whether they denote the same type. We sometimes express this equality as "A =ext B".

The designers of Nuprl chose to employ another partial equivalence relation between type expressions that is finer than "extensional" equality, and is therefore "intensional". This serves also to constrain construction of type expressions, since not every (extensional) type expression will be related even to itself by this relation.

When we talk about types we often mean type expressions modulo this intensional equality, that is, equivalence classes of type expressions under this intensional equality, restricted to those type expressions intensionally equal to themselves. An intensional type expression is a type expression that is intensionally equal to itself.

In particular, "x:A→B(x)" and "x:C→D(x)" are stipulated to be intensionally equal just when "A" and "C" are, and further when for all expressions r and s such that r = s ∈ A, "B(r)" and "D(s)" are intensionally equal. It is similar with most of the other type constructors; intensionally equal types have the same outermost operator, and intensionally equal corresponding components. An exception is equality between Quotient Types {21}.

This entails that there are extensional type expressions that are not intensional type expressions. The expressions "Void" and "ℤ×Void" both denote the empty type, and so are extensionally equal, but are not intensionally equal expressions due to their differing outermost structure. Let us use this to build an extensional type expression that is not an intensional type expression.

Consider the expression "x:(ℤ mod 2)→if x=0→ Void ; ℤ×Void fi".
This denotes a type extensionally since no matter what member of (ℤ mod 2) is substituted for x in "if x=0→ Void ; ℤ×Void fi" the result is an expression for the empty type, i.e. "if x=0→ Void ; ℤ×Void fi" denotes a(n extensional) type-valued function over (ℤ mod 2). But (0 = 2 ∈ ℤ mod 2), and even though "if 0=0→ Void ; ℤ×Void fi" and "if 2=0→ Void ; ℤ×Void fi" denote the same (extensional) type, they denote distinct intensional types, since the first evaluates to "Void" and the second evaluates to "ℤ×Void". Hence, "x:(ℤ mod 2)→if x=0→ Void ; ℤ×Void fi" is not intensionally equal to itself, and so is not an intensional type expression.

Universes {18} are examples of types whose members are intensional types rather than (extensional) types. Here is the argument, which also shows the sense in which this is so: Void is Type{1}×Void, extensionally, but Void is a member of Type{1} whereas Type{1}×Void is not.

(March 2001 {62} - sfa)

The language of type theory is largely aimed at reasoning about functions, a type being the domain of a function (see Types {15} and Functions {23}).

But often we build complex expressions for types out of operations on types themselves. To characterize such operations on types as functions the domain would need to include the types we wish to take as arguments. If there were a type of all types then we'd be done, but there are theoretical obstacles to construction of such a type, and indeed there are demonstrations of its impossibility.

So, we instead build large collections of types, each closed under all the usual type constructors; the first universe Type{1} is formed by collecting all the types that don't depend on building universes. The second universe Type{2} is built the same way except that we now add Type{1} as a new type. So, for example, $\mathbb{N}\rightarrow$Type{1} $\in$ Type{2}. The third universe Type{3} is the same except both Type{1} and Type{2} are made available as types.

The universes are "cumulative": Type{1} $\subseteq$ Type{2}, Type{2} $\subseteq$ Type{3}, et cetera.

Technically, the members of universes are types "in intension", or "intensional types" (see Intensional Types {17}).

Universes approximate the possibly desired but impossible type of all types.

One essential aspect of a family of universes is that every type inhabits one of the family. For this reason, the family of universes cannot be represented by a type-valued function over some type. For example, if there is a denumerable hierarchy of universes, then there can be no operator V(i) such that, for each i $\in$ $\mathbb{N}^+$, V(i) is the universe Type{i}; this is because i:$\mathbb{N}^+\rightarrow$V(i) would be a type but it would not inhabit any universe, contradicting the assumption that V(i) forms a universe hierarchy. And since using any type instead of $\mathbb{N}^+$ would fail the same way, it is not possible to internally index the universes with a type.

In Nuprl we have followed Martin-Lof 1980 making the expressions for universes atomic, i.e. they have no subterms, and no universe expression can be generated by evaluating expressions that don't already contain it. This means that a finite segment of the universe hierarchy is adequate for explaining any finite collection of type expressions, since only finitely many universe expressions can be used in it.

The inability to explicitly quantify over all types is somewhat mitigated in Nuprl by the use of universe level variables instead of constants.

See Type Functionality Sequents {38} and Prop Levels and Predicativity {39}.

(May 2001 {62} - sfa)

Given u ∈ A, v ∈ A, the expression "u = v ∈ A" represents the equality between u and v.

The primitive expression "u = v ∈ A" is given a meaning, whether or not it is true, only for u ∈ A and v ∈ A. See Propositions {34}.

The reason that the type is indicated as part of the equality is that a pair of notations may be used for distinct values in one type, and yet they may stand for the same value in another type, due to the radical type-polymorphism {25} of expressions employed in computational type theory.

For example, 2 = 4 ∈ ℤ mod 2, but 2 ≠ 4 ∈ ℤ.
Or again,

Thm* (**λ**x.x) = (**λ**x.if x<0→ 0 ; x fi) ∈ ℕ→ℕ,

since the first branch is never taken on ℕ, but

Thm* (**λ**x.x) ≠ (**λ**x.if x<0→ 0 ; x fi) ∈ ℤ→ℤ

since they take different integer values on -1.

Naturally, "equality" between notations is intimately connected to notations for Functions {23}.

Often there is a most likely type that can be discerned from context, and we elide the type from the display of an equality, though it's still in the term. For example, we may display "∀f:(ℤ→A). f(1) = f(2)" rather than "∀f:(ℤ→A). f(1) = f(2) ∈ A".

Expressing Membership: t ∈ T

The expression for membership in a type is simply defined as

Def t ∈ T == t = t ∈ T.

Thus, "t ∈ T" is sensible iff it is true, and is best regarded as expressing (in t) the everywhere-true predicate on type T. In particular, it is not useful as an ordinary predicate on values of some kind, since it cannot usefully serve as an antecedent nor be negated. It is useful only when used "positively" such as in assertions of well-formedness, for example in ∀x,y:ℕ. y≤x ⇒ xy ∈ ℕ.

For a more thorough discussion of types, denotation and equality see Types {15}.

(March 2001 {62} - sfa)

Intersecting a Family of Types isect_doc {20}

∩x:A. B(x) is the intersection of a family of types B(x) over x ∈ A.

More precisely, u = v ∈ (∩x:A. B(x)) when u = v ∈ B(x) for all x ∈ A (see Types {15}). Consequently,

```
  Thm* ∀B:(A→Type), a:A. (∩x:A. B(x)) ⊆ B(a)
```

If A is empty, then (∩x:A. B(x)) has exactly one member, and every (closed) expression denotes it (Thm* whatever1 = whatever2 ∈ (∩x:Void. Whatever(x))). See Types {15}.

The type (∩A:Type. (A List)→(A List)), for example, is the type of "polymorphic" {25} operations on lists that don't depend on the type of the elements. For example,

```
   (λas.rev(as)) ∈ (∩A:Type. (A List)→(A List))

   where rev(as) is the list reversing function,

   Def rev(as) == Case of as; nil → nil ; a.as' → rev(as') @ [a]  (recursive)
```

For more examples about intersection see Polymorphism {25}.

Guarded Types: T(given A)

In the special case of "∩x:A. B(x)" where some expression T not actually using x is used in place of B(x), then the intersection, denoted "T(given A)", is a way of "guarding" membership in T by whether A has a member.

If A has a member, then the type T(given A) is T (extensionally). If A is empty, then T(given A) is degenerate, having one member which is denoted by every closed expression (see Types {15}).

So for example, in Nuprl, in order for A→B to denote a type, A must also denote a type, but B need only denote a type if A has a member. If A is empty, then B may be any closed term, and A→B then denotes simply the one-element function space with empty domain. See Function Types {24}.

This well-formedness condition can be expressed using guarded types thus:

```
  Thm* ∀A:Type, B:Type(given A). A→B ∈ Type
```

The restriction of B to Type(given A) escapes the absolute condition of B simply being in Type.

See Guarded Types {36} for more on guarded types and propositions.

(Oct 2001 {62} - sfa)

Quotient Types (of equivalence classes) quotient_doc {21}

u,v:A//E(u;v) is the quotient of type A by equivalence relation E(u;v) for u $\in$ A, v $\in$ A. The members are equivalence classes. See Equality {19}.

However, in Nuprl we do not employ explicit coercions from members of A to equivalence classes under E(u;v). Rather, we just use the same notations for both members and equivalence classes containing them, which entails that we must sometimes specify which is meant (though often we can simply elide the type from display).

For example, Thm* 2 = 4 $\in$ $\mathbb{Z}$ mod 2, where

  Def $\mathbb{Z}$ mod k == x,y:$\mathbb{Z}$//($\exists$m:$\mathbb{Z}$. xy = m$\cdot$k).

The semantics of quotient types is, for x $\in$ A, y $\in$ A,
x = y $\in$ u,v:A//E(u;v) when E(x;y).

The Intensional Type Equality {17} between quotients is not strong in the equivalence relation. The types u,v:A//E(u;v) and u,v:B//R(u;v) are intensionally equal when A and B are, and also $\forall$u,v:A. E(u;v) $\iff$ R(u;v).

See also Polymorphism {25}.

(Sept 2001 {62} - sfa)

Void is an empty type, i.e., it has no members.

There are also other ways to express the empty type by means of other Nuprl primitives, such as,

Thm* Void =ext {x:$\mathbb{Z}$| x<x }

Thm* Void =ext rec(A.A)

Thm* Void =ext ($\cap$x:$\mathbb{Z}$. {x:$\mathbb{Z}$}) since these singleton types are disjoint.

And of course, Thm* Void $\subseteq$ A.

Here are a few examples of how Void relates to other type constructors.

Thm* Void$\times$A =ext Void

Thm* Void List =ext {nil:Void List}

Thm* Void$\rightarrow$A =ext {$\boldsymbol{\lambda}$x.x:Void$\rightarrow$A}

The last theorem above states that Void$\rightarrow$A has only the one member ($\boldsymbol{\lambda}$x.x). This is a degenerate function in that it cannot be applied to anything in Void. Indeed, the body of the function expression is irrelevant since it will never be evaluated as part of application to any argument of type Void.

Thm* ($\boldsymbol{\lambda}$x.whatever) = ($\boldsymbol{\lambda}$x.x) $\in$ Void$\rightarrow$A

So, although Void$\rightarrow$A has only one member, every (closed) lambda expression denotes that member. See Functions {23}.

Similarly, the intersection over the empty domain is a trivial one-element type whose single member is denoted by every expression that can denote at all:

Thm* whatever1 = whatever2 $\in$ Whatever(given Void)

Thm* whatever1 = whatever2 $\in$ ($\cap$x:Void. Whatever(x))

Sometimes this last type is denoted "Top" since:

Thm* A $\subseteq$ Top

where Def Top == Void(given Void).

See Polymorphism {25}.

(Aug 2001 {62} - sfa)

"Function" and "Type" are the fundamental a priori concepts of type theory. Nuprl's computational type theory is a constructive semantical theory about notations adequate for an intuitionistic account of functions, and the Nuprl logic emphasizes the computational aspect of notation.

Notations in Nuprl are "grounded" computationally. A class of computationally significant expressions is presupposed by the semantics of denotation (see Computation {42}). It should be emphasized that the relation of a computional expression to the result of executing it is NOT the same as that expression standing for an abstract value, even though we sometimes call executing an expression "evaluation".
One defines a method for notating a class of values, i.e. a "type" (see Types {15}), by specifying when two notations stand for the same value of that type. These notations are taken from the presupposed computational language, and it is required that whenever two expressions are "computationally equivalent" then if either stands for a value in some type, both stand for the same value.

For example, $0 \in \mathbb{Z}$, therefore 0 = if 1=2$\rightarrow$ 3 ; 0 fi $\in \mathbb{Z}$ since executing "if 1=2$\rightarrow$ 3 ; 0 fi" results in the expression "0".
Indeed, radically, "0 = if 1=2$\rightarrow$ X ; 0 fi $\in \mathbb{Z}$" is true for any closed expression X, such as the non-integer "<0,0>" or even the non-evaluable expression "(λx.x(x))(λx.x(x))". The fact that these are provable in Nuprl is an aspect of taking the computational basis for notation seriously.

As in normal practice, the most common way of expressing functions is by means of expressions containing variables. By substituting constants for those variables, the resulting expression stands for the value of the function on the values those constants stand for.
For example, "x+1" stands (in the variable x) for the successor function on $\mathbb{Z}$.
A basic constraint on the computational behavior of such an attempt to express a function is that if one replaces an argument expression by an "equal" one, then the resulting full expressions are also "equal". This "equality" of notations is dependent on type, on both the input and output types in the case of functions.
Consider the expression "if x=0$\rightarrow$ 1 ; x fi". Although it expresses (in x) a function from $\mathbb{Z}$ to $\mathbb{Z}$, it does not express a function from ($\mathbb{Z}$ mod 2) to $\mathbb{Z}$ because when applied to "0" it evaluates to "1", but when applied to "2", which stands for the same value of ($\mathbb{Z}$ mod 2) as "0", it evaluates to "2", which is not equal to "1" in $\mathbb{Z}$, of course. Another, trivial, example of such an expression is simply "x" which obviously expresses the identity function, in x, over any type, but does not express a function from ($\mathbb{Z}$ mod 2) to $\mathbb{Z}$ since 0 = 2 $\in \mathbb{Z}$ mod 2 but $0 \neq 2 \in \mathbb{Z}$.

The "Abstraction" Operator: λx.b(x)

It is notationally quite handy to have a form for expressing functions with closed expressions, i.e. without free variables, and a notation for applying functions to arguments in which the notation for the function can be separated from the argument. We can then build a type of such functional expressions for any given input and output types.

Our two-place notation for function application is "f(t)" where "f" indicates the place for the function expression, and "t" indicates the place for the argument expression. We mediate the two methods for expressing functions (viz.

expressions with free variables, versus closed function expressions applied via "<fun>(<arg>)") with the so-called "abstraction" operator "λx.b(x)".
For example, the doubling function on integers expressed as "x+x" using free variables as arguments, may be expressed in closed form as "λx.x+x".
Naturally, the expression "(λx.x+x)(2)" evaluates according to the rewrite sequence:

(λx.x+x)(2) $*\!\Rightarrow$ 2+2 $*\!\Rightarrow$ 4

Generally, "(λx.b(x))(t)" is evaluated by evaluating "b(t)".

Function Types {24} are collections of these closed forms expressing functions.

See also Multi-Argument Functions {26} and Recursive Functions {27}.

Functionality Sequents {37} are forms of assertion about the open forms for expressing functions using free variables for argument places.

(May 2001 {62} - sfa)

---

functionality2_doc {24}

Function Types - continued from Functions {23} and Types {15}.

The type $\mathbb{Z}\to\mathbb{Z}$ represents the class of one-place integer-valued functions of integers; here's how:

For any (closed) expressions t1 and t2, t1 = t2 $\in$ $\mathbb{Z}\to\mathbb{Z}$ just when

for any expressions r1 and r2, if r1 = r2 $\in$ $\mathbb{Z}$ then t1(r1) = t2(r2) $\in$ $\mathbb{Z}$.

Specializing this condition variously,

if t $\in$ $\mathbb{Z}\to\mathbb{Z}$ and r $\in$ $\mathbb{Z}$ then t(r) $\in$ $\mathbb{Z}$

if t $\in$ $\mathbb{Z}\to\mathbb{Z}$ and r1 = r2 $\in$ $\mathbb{Z}$ then t(r1) = t(r2) $\in$ $\mathbb{Z}$

if t1 = t2 $\in$ $\mathbb{Z}\to\mathbb{Z}$ and r $\in$ $\mathbb{Z}$ then t1(r) = t2(r) $\in$ $\mathbb{Z}$

Generalizing away from domain and result types, the type A$\to$B represents the class of one-place B-valued functions of arguments from A. Here's how:
t1 = t2 $\in$ A$\to$B just when

t1 and t2 execute giving result expressions "λx1.b1(x1)" and "λx2.b2(x2)", and
for any expressions r1 and r2, if r1 = r2 $\in$ A then t1(r1) = t2(r2) $\in$ B.

This extra condition about the execution of t1 and t2 themselves evaluating to "λx.<*>" forms is not theoretically essential, but it is standard. In the case of $\mathbb{Z}\to\mathbb{Z}$ above we omitted mentioning it for simplicity, and because it follows from the way we happen to define evaluation of "f(r)" and the fact that $\mathbb{Z}$ has members.

Note: We could have chosen to define A$\to$B without this condition on its members with the result that, when A was empty, every closed expression would denote the single degenerate member of A$\to$B. It is degenerate because it cannot be applied to anything in the domain A.

Identity on functions is extensional, consequently,

Thm* $\forall$f:(A$\rightarrow$B). (λx.f(x)) = f

Thm* $\forall$f,g:(A$\rightarrow$B). f = g $\iff$ ($\forall$x:A. f(x) = g(x))

Dependent Function Types: x:A$\rightarrow$C(x)

Actually, A$\rightarrow$B is a special form of the more general x:A$\rightarrow$C(x). When A is a type and C(x) is a type-valued function (in x) over domain A, then a member of x:A$\rightarrow$C(x) is a function which for an argument a $\in$ A takes a value in the type C(a). That is, one can specify the TYPE of the output as a function of the VALUE of the input. So, t1 = t2 $\in$ x:A$\rightarrow$C(x) just when

t1 and t2 execute giving result expressions "λx1.b1(x1)" and "λx2.b2(x2)", and for any expressions r1 and r2, if r1 = r2 $\in$ A then t1(r1) = t2(r2) $\in$ C(r1).

Since C(x) must be a type-valued function in x $\in$ A, C(r1) and C(r2) must denote the same type, so we might as well have said "t1(r1) = t2(r2) $\in$ C(r2)" above.

This type constructor is also called the "general product" of a family of types, and indeed the most common notation for it is "Πx:A. B(x)".

When the expression for the result type does not use the input variable, we get the special form for display A$\rightarrow$B.

Multi-Argument Functions {26}

(Sept 2001 {62} - sfa)

Subtyping and Polymorphism — polymorphism_doc {25}

As explained in Equality {19}, a single notation can be used to denote values in multiple types, and a pair of notations can be used to denote distinct values of one type and yet denote a single value in another type.
(One consequence is that one cannot always simply "read off" an intended type from an expression.)

Subtyping: Void ⊆ A; {x:A| P(x) } ⊆ A; A ⊆ (x,y:A//E(x,y)); (∩x:A. B(x)) ⊆ B(a); A ⊆ Top

Recall that Functions {23} are denoted by Computationally {42} significant expressions that operate on notations for the arguments. It often happens that functional notations working on notations of one type "T" will also "automatically" work on all notations of some other type "S". Or equivalently, the notations of type "S" are also notations of type "T". Then we may say

```
Def S ⊆ T == ∀x:S. x ∈ T,
```

and consequently, when S ⊆ T, then (T→A) ⊆ (S→A), i.e. ∀f:(T→A). f ∈ S→A.
See also Subtype Relation {59} for an alternative to this definition.
One obvious way for S ⊆ T to happen is simply for S to be a subclass of T, e.g.

```
ℕ ⊆ ℤ, or more generally,
{x:A| P(x) } ⊆ A (see Comprehension Subtypes {14}), and as an extreme case,
Void ⊆ A.
```

But there is also a subtler aspect of subtyping. Recall from Functions {23} that the core criterion for being a notation for a function over a domain type D is that its application to any notations of D denoting the same input value must yield notations for the same output value. Thus,

```
Thm* ∀f:((ℤ mod 2)→A). f ∈ ℤ→A
```

since the notations of "ℤ mod k" are just those of ℤ with certain distinct integer notations becoming identified, namely those x and y such that ∃m:ℤ. xy = m·k. Thus,

```
ℤ ⊆ (ℤ mod k), or more generally,
T ⊆ (x,y:T//E(x,y)) (see Quotient Types {21})
```

Another type constructor intimately related to expressing polymorphism is Intersection {20}, ∩x:A. B(x). Here we give six other facts about functions that are polymorphic over all input types. The first says that a function expression completely polymorphic in the domain type may be used as both function and argument in a single application. The second says that if the output type does not depend on the input type, then the function is constant. The third says that identity is the only polymorphic function mapping the domain into itself. The fourth and fifth together say that a function generating a (D List) from a D value, ignoring the choice of D, must generate a list whose length is independent of the input, and all of whose members simply are the input value. The sixth says that the swap function is the only polymorphic C×D→D×C function.

```
Thm* ∀T:Type, f:(∩D:Type. D→T). f(f) ∈ T
Thm* ∀f:(∩D:Type. D→T), A:Type, a:A, B:Type, b:B. f(a) = f(b) ∈ T
Thm* ∀f:(∩D:Type. D→D), A:Type, a:A. f(a) = a
Thm* ∀f:(∩D:Type. D→(D List)), A:Type, a:A, B:Type, b:B.
     ||f(a)|| = ||f(b)|| ∈ ℤ
Thm* ∀f:(∩D:Type. D→(D List)), A:Type, a,x:A. x ∈list f(a) ⇒ x = a
Thm* ∀f:(∩C,D:Type. C×D→D×C), A,B:Type, a:A, b:B. f(<a,b>) = <b,a> ∈ B×A
```

See Typing Y {30} for another example.

The type Top

Finally, there is a unit type "Top" whose sole member is denoted by every closed expression. It is called "Top" because Thm* $A \subseteq Top$.

NOTE that "Top" is NOT a type of all values, even though every (closed) expression has type "Top" (thus "whatever $\in$ Top"); it is just that the semantics is so trivial as to exclude no possible notation (the other extreme of triviality is Void, whose semantics trivially excludes every notation). In particular even though (Top $\in$ Top) is true, this does not mean that the TYPE Top is a member of Top, since equality in Top has nothing to do with equality between types.

(Rather than building Top in as a primitive it is defined as
Def Top == Void(given Void).)

(Nov 2002 {62} - sfa)

Multi-Argument Functions - continued from Function Types {24} and Functions {23}

Functions of more than one argument can be presented in various ways. The most common, of course, is to use an expression with free variables for the arguments.
But again, it is highly convenient to have closed forms of expression. We have not included a primitive type constructor for multi-argument functions. Instead, we use the method called "currying" in which $A \to B \to C$ is considered as a type of C-valued functions of two arguments, the first from type A and the second from type B. Note that fully parenthesized, "$A \to B \to C$" would be "$A \to (B \to C)$" and not "$(A \to B) \to C$".

To apply $f \in A \to B \to C$ to two arguments, say $a \in A$ and $b \in B$, first apply it to a, giving $f(a) \in B \to C$, then apply that resulting function to b, giving $f(a)(b) \in C$. We display this form of iterated application more conventionally as $f(a,b) \in C$.

Naturally, the method generalizes to more argument places. If $g \in A \to B \to C \to D$ then $g(a,b,c) \in D$, for $a \in A$, $b \in B$, $c \in C$.

Corresponding to the iteration of application, we can iterate the abstraction operation. Doing so assigns an order to the arguments.
For example, "$\lambda x.\lambda y.xy+2$", which we may display as "$\lambda x,y.\ xy+2$", works like this when applied to a couple of arguments:

$$(\lambda x,y.\ xy+2)(3,6) \ast\!\!\Rightarrow (\lambda y.3y+2)(6) \ast\!\!\Rightarrow 36+2 \ast\!\!\Rightarrow -1$$

Or, we could assign a different order to the arguments when converting the free variable form of expression to the closed form:

$$(\lambda y,x.\ xy+2)(3,6) \ast\!\!\Rightarrow (\lambda x.x3+2)(6) \ast\!\!\Rightarrow 63+2 \ast\!\!\Rightarrow 5$$

We have been looking only at types for multi-argument functions where the output type is constant. The general form for the type of three-place functions, say, in which the output type may depend on the input values, and indeed, each input type may depend on values of earlier arguments, is
"$x{:}A \to y{:}B(x) \to z{:}C(x;y) \to D(x;y;z)$".
Suppose $a \in A$, $b \in B(a)$, $c \in C(a;b)$, and $f \in x{:}A \to y{:}B(x) \to z{:}C(x;y) \to D(x;y;z)$.
Then

$$f(a) \in y{:}B(a) \to z{:}C(a;y) \to D(a;y;z)$$

$$f(a,b) \in z{:}C(a;b) \to D(a;b;z)$$

$$f(a,b,c) \in D(a;b;c)$$

Functions {23}

(Feb 2001 {62} - sfa)

Unlike most type theory based logics, Nuprl's computational type theory admits computation by general recursion. This is possible because how expressions are to be executed is prior to (independent of) type formation, and admits untyped methods of computation on expressions. For detail on how this is done, see Recursion via the Y-combinator {28}.
Consider the factorial function.

Def x! == if x=$_2$0→ 1 else x·(x1)! fi (recursive)

where ?=$_2$? is a 𝔹-valued test of integer equality.

Here's an example of how it evaluates:

3! *⇒ 3·(31)! *⇒ 3·2·(311)! *⇒ 3·2·1·(3111)! *⇒ 3·2·1·1 *⇒ 6

Observe that execution (-1)! would never complete.

(-1)! *⇒ (-1)·(-11)! *⇒ (-1)·(-11)·(-111)!, et cetera.

The argument that Thm* ∀x:ℕ. x! ∈ ℕ is carried out by induction over ℕ.
The potential and convenience of general recursion is better demonstrated with the following variant of factorial that has a result only on non-negative even inputs.

Def x!$_2$ == if x=$_2$0→ 1 else x·(x2)!$_2$ fi (recursive)

Thm* ∀x:{x:ℕ| (x rem 2) = 0 }. x!$_2$ ∈ ℕ

Indeed, one can even type Kleene's paradigmatic unbounded search function,

Def mu(f) == if f(0)→ 0 else 1+mu(**λ**x.f(1+x)) fi (recursive)

Thm* mu ∈ {f:(ℕ→𝔹)| ∃x:ℕ. f(x) }→ℕ
Thm* ∀f:{f:(ℕ→𝔹)| ∃x:ℕ. f(x) }. f(mu(f))
Thm* ∀f:{f:(ℕ→𝔹)| ∃x:ℕ. f(x) }, i:ℕ. i<mu(f) ⇒ ¬f(i)

(See also Typing Y {30}.)

One can also define type-valued functions and predicates by general recursion on the domain. Here is a definition of n-tuples defined by iterating cartesian product. We include a degenerate singleton case for 0.
Note: this is not a recursive type, but a recursive definition of a family of types. See Recursive Types {33}.

Def A^n == if n=$_2$0→ Unit ; n=$_2$1→ A else A×(A^(n1)) fi (recursive)

Thm* ∀A:Type, n:ℕ. (A^n) ∈ Type

Now let us compute:

A^0 *⇒ Unit
A^1 *⇒ A
A^2 *⇒ A×A

```
A^3 *⇒ A×A×A
```

Here's a predicate on n-tuples to the effect that a tuple includes a value satisfying some property of the element type (see Propositions {34}).

```
Def ∃ u in X: A^n. P(u)
    == n = 1 ∈ ℕ & P(X)
       ∨ n≥2 & (X/a,rest. P(a) ∨ (∃ u in rest: A^(n1). P(u)))
    (recursive)

Thm* ∀A:Type, P:(A→Prop), n:ℕ, X:(A^n). (∃ u in X: A^n. P(u)) ∈ Prop

Example of use:
Thm* ∃ i in <1,(-1),0>: ℤ^3. i<0
```

(March 2001 {62} - sfa)

recursive_function2_doc {28}

Recursion via the Y-combinator - from Recursive Functions {27}

Consider the definition of factorial:

  Def x! == if x$=_2$0$\rightarrow$ 1 else x·(x1)! fi (recursive)

Observe that the operator "n!" occurs on both sides of the definition, connoting a recursive definition. There is no need for further explanation for the typical user's needs, but this form of operator definition is not built into the system, and here we explain how such definitions are implemented on top of the real primitives of Nuprl.

The computational basis of Nuprl's semantics is an untyped calculus, which includes the "λx.b(x)" and "f(a)" operators.
Thus, we can build the computational fixed-point "combinator" Y:

  Def Y(f) == (λx.f(x(x)))(λx.f(x(x)))

Now, observe how (λx.f(x(x)))(λx.f(x(x))) computes:

  (λx.f(x(x)))(λx.f(x(x))) $*\Rightarrow$ f((λx.f(x(x)))(λx.f(x(x))))

One step of computation stuffs the whole original term into the argument place of an application of f; thus, whatever f is, (λx.f(x(x)))(λx.f(x(x))) is a fixed-point of evaluation. If we abstract from the variable f we get λf.(λx.f(x(x)))(λx.f(x(x))) which can then be applied to various functions we want to take fixed-points of. This function abstraction is abbreviated as "Y":

  Y == λf.(λx.f(x(x)))(λx.f(x(x)))

Normally, we display such definitions using applications to the variable on the lefthand side instead of using the abstraction operator on the right hand side, which comports better with standard mathematical practice:
Def Y(f) == (λx.f(x(x)))(λx.f(x(x))).

With this definition in place, we can rewrite Y(f) thus:

  Y(f) $*\Rightarrow$ (λx.f(x(x)))(λx.f(x(x))) $*\Rightarrow$ f((λx.f(x(x)))(λx.f(x(x)))) $*\Rightarrow$ f(Y(f))

Eliding intermediate steps pointedly demonstrates the fixed-pointedness
Y(f) $*\Rightarrow$ f(Y(f)).

Y-combinator Continued {29}

(Feb 2001 {62} - sfa)

---

recursive_function3_doc {29}

Y-combinator Continued - from Recursion via the Y-combinator {28}

Now, let's see how to exploit Y(f) $*\Rightarrow$ f(Y(f)) to express recursive function definitions, which are not primitive in Nuprl. A general form of a one-place recursive function definition would be f(z) == G(z; x.f(x))

which indicates that in evaluating f(z), not only does the function G(z; x.f(x)) have access to the argument z that was supplied in f(z), but also it may further apply function f(x) to any new arguments at all; thus, recursive calls to f(x) may be made, for various values of x, as part of evaluating f(z).

However, in Nuprl, such recursive definitions are not built-in, they are not primitives. The primitive Operator Definitions {47}s of Nuprl must not use the defined operator in its own expansion. So, how do we implement "f(z) == G(z; x.f(x))"?

f(z) == Y((λh,z. G(z; x.(h(x)))),z)

Notice that in this definition, the operator f(?) does not occur on the right hand side, so the definition is an ordinary one, rather than a recursive one. The recursive "behavior" is a result of using the Y-combinator, as illustrated below. Note that we're not just looking at Y applied to one argument "λh,z. G(z; x.(h(x)))" but its further application to another argument u in order to see how f(u) would be evaluated (recall that "g(x,y)" means "(g(x))(y)"; see Multi-Argument Functions {26}).

Y((λh,z. G(z; x.(h(x)))),u)
*⇒
(λh,z. G(z; x.(h(x))))(Y(λh,z. G(z; x.(h(x)))),u)
*⇒
G(u; x.(Y((λh,z. G(z; x.(h(x)))),x)))

Further evaluation depends upon how G(?; x.?) is evaluated. Observe that "Y(λh,z. G(z; x.(h(x))))" has been replicated down inside of the instance of G(?; x.?), and that how it gets further applied is up to G(?; x.?).

Returning to our definition of factorial, introduced in Recursive Functions {27},

Def x! == if x$=_2$0→ 1 else x·(x1)! fi (recursive)

is actually an improved representation of the actual content of the definition, which is:

x! == Y((λsfa_doc_factorial,x. if x$=_2$0→ 1 else x·sfa_doc_factorial(x1) fi),x)

The recursive form was entered by the user, and this primitive form was automatically generated and stored as the "official" definition. The variable name "sfa_doc_factorial" was selected automatically by the system, using the name assigned to this factorial operator by its creator (sfa).

Note: Although we type applications of Y to functions, as we did for Thm* ∀x:ℕ. x! ∈ ℕ, we don't assign Y itself a function type. For a discussion of how this could be accomplished see Typing Y {30}.

Multi-argument Functions with Y {32}

(Feb 2001 {62} - sfa)

ycomb_typing_doc {30}

Typing the Y-combinator - Polymorphism and Functions on Void

We could type the Y-combinator in Nuprl with various natural types IF we were to add a variation of the rule for function extensionality, which would be justified by liberalizing the primitive definition of the function type to admit arbitrary notations when the domain is empty.

Here we discuss these variations, and consequences.

Our sample typing will be Y ∈ (∩k:ℕ. (ℕk→T)→ℕ(k+1)→T)→ℕ→T, which could be used to type functions defined by primitive recursion over ℕ as we shall exemplify here.
See Polymorphism {25}, Recursive Functions {27} and Void {22}.

Consider this definition of factorial and the use of the Y combinator underlying it.

  Def x! == if x=$_2$0→ 1 else x·(x1)! fi (recursive)

  z! *⇒ Y((λf,x. if x=$_2$0→ 1 else x·f(x1) fi),z)

So factorial is a fixed point of

  Thm* (λf,x. if x=$_2$0→ 1 else x·f(x1) fi) ∈ (∩k:ℕ. (ℕk→ℕ)→ℕ(k+1)→ℕ)

This typing theorem says that this function takes a function with finite domain, and generates a function over a certain one-element extension to that domain.
From the above typing for Y,

  Thm* Y(λf,x. if x=$_2$0→ 1 else x·f(x1) fi) ∈ ℕ→ℕ

would follow trivially from the two typings of Y and the given function.

Now, as indicated by the theorem above, we can still prove this application of Y has type ℕ→ℕ, but examination of the proof actually given would reveal it to be a complex proof involving induction and rewriting, rather than the simple type inference we desire, exploiting the type of Y itself.

How might we prove Y ∈ (∩k:ℕ. (ℕk→T)→ℕ(k+1)→T)→ℕ→T? Most of a proof is given HERE {61}; there is only one unproved subgoal:

   1. T : Type
   2. F : ∩k:ℕ. (ℕk→T)→ℕ(k+1)→T
   ⊢ Y(F) ∈ ℕ0→T

The type ℕ0→T is the one-element type of functions over the empty domain (see Void {22}). Our current (standard) primitive definition of a type A→B requires that each term of that type must reduce to a "λ[var].<*>" form. Indeed, if one plugs in an expression of form "λf,x. b(f;x)" in place of F, one can simply rewrite

  Y(λf,x. b(f;x)) *⇒ λx.b(Y(λf,x. b(f;x));x)

then easily type this in ℕ0→T. But we do not have this syntactic form in "Y(F)" and so cannot proceed this way.

A direct solution would be to modify the Function Extensionality rule.

CONTINUE {31}

(Sept 2001 {62} - sfa)

Typing the Y-combinator - continued from Typing Y {30}   ycomb_typing2_doc {31}

It is shown in HERE {61} that typing the Y-combinator as "(∩k:ℕ. (ℕk→T)→ℕ(k+1)→T)→ℕ→T" reduces to showing "Y(F) ∈ ℕ0→T".

The difficulty of doing so is indicated in Typing Y {30}, where it was also suggested that we might modify the Function Extensionality rule to achieve this typing.

The current Function Extensionality rule permits inferences of the form:

⊢ f = g ∈ A→B

from
1. x : A ⊢ f(x) = g(x) ∈ B,

⊢ A ∈ Type, ⊢ f ∈ C→D, and ⊢ g ∈ E→F

The last two subgoals serve merely to require that f and g reduce to "**λ**[var].<*>" forms.

Suppose we were to use a different primitive definition of A→B, stipulating for type A→B, t1 = t2 ∈ A→B just when

for any expressions r1 and r2, if r1 = r2 ∈ A then t1(r1) = t2(r2) ∈ B.

This definition was taken from Function Types {24} and simplified. Then in the case that A is empty, every closed term has type A→B. (See Void {22}.)

The only obstacle to adopting this definition would appear to be the a priori desire that every value of every type should evaluate to some canonical form {43}. It should be noted that users of standard Nuprl have already implicitly abandoned this presumption by adopting the Intersection Types {20}. All the current standard rules would remain valid under this new primitive definition of function types (how to generalize it to dependent types is obvious). In addition a new, more liberal Function Extensionality rule should then be adopted, allowing inference to

⊢ f = g ∈ A→B

simply from from

1. x : A ⊢ f(x) = g(x) ∈ B, and

⊢ A ∈ Type.

The proof of {61} $Y \in (\cap k{:}\mathbb{N}.\ (\mathbb{N}k \to T) \to \mathbb{N}(k+1) \to T)$ would then be quite simple to complete by proving the sole remaining subgoal

1. T : Type
2. $F : \cap k{:}\mathbb{N}.\ (\mathbb{N}k \to T) \to \mathbb{N}(k+1) \to T$

$\vdash\ Y(F) \in \mathbb{N}0 \to T$

using the new rule, since

1. T : Type
2. $F : \cap k{:}\mathbb{N}.\ (\mathbb{N}k \to T) \to \mathbb{N}(k+1) \to T$
3. $x : \mathbb{N}0$

$\vdash\ Y(F,x) \in T$ by Auto

and

1. T : Type
2. $\cap k{:}\mathbb{N}.\ (\mathbb{N}k \to T) \to \mathbb{N}(k+1) \to T$

$\vdash\ \mathbb{N}0 \in$ Type by Auto

(Sept 2001 {62} - sfa)

recursive_function4_doc {32}

Multi-argument Recursive Functions - continued from Y-combinator Continued {29}

We have seen how one-argument recursive functions can be defined via the Y-combinator {29}.

Consider now the form:

```
f(x; y; z) == G(x; y; z; u,v.f(u; y; v))
```

Here we have introduced a couple of complexities. First, there are three arguments, but second, notice that the middle argument is held constant during recursion.

The system converts this recursive definition form to the following explicit form of definition using the Y-combinator:

```
f(a; b; c) == Y((λh,x,z. G(x; b; z; u,v.(h(u,v)))),a,c)
```

Notice that the function given as argument to Y has a place h for the recursion, and places x and z holding places for new values of the first and third arguments on recursive calls. But there is no way for the second argument to be changed on recursion -- it is held at b, and so is not supplied as an argument to Y(λh,x,z. G(x; b; z; u,v.(h(u,v)))) on the right hand side like a and c are. Here's how f(a; b; c) evaluates after expanding the above definition (recall "g(r,s,t)" is really "g(r)(s)(t)"):

```
Y((λh,x,z. G(x; b; z; u,v.(h(u,v)))),a,c)
*⇒
(λh,x,z. G(x; b; z; u,v.(h(u,v))))(Y(λh,x,z. G(x; b; z; u,v.(h(u,v)))),a,c)
*⇒
G(a; b; c; u,v.(Y((λh,x,z. G(x; b; z; u,v.(h(u,v)))),u,v)))
```

The fixed-point Y(λh,x,z. G(x; b; z; u,v.(h(u,v)))) gets replicated during evaluation.

(Feb 2001 {62} - sfa)

Recursive Types: rec(X.F(X)) rec_type_doc {33}

The Nuprl list types of form "A List" are primitive recursive types, since from any $x \in A$ and $y \in$ A List one can build a larger list $x.y \in$ A List. See Lists {5}

A more general form for recursive types in Nuprl is rec(X.F(X)), where F(X) is a type-valued function of types (in X), with certain restrictions. We don't give detail on them here, but the main feature of F(X) that follows from these restrictions is that F(X) will be monotonic in type argument X, i.e.,

```
if A ⊆ B then F(A) ⊆ F(B), where Def S ⊆ T == ∀x:S. x ∈ T
```

For example, we could have defined a non-primitive list type as rec(X.Unit+(A×X)), and interpreted inl(·) as the null list, and inr(<a,y>) as the list with head $a \in A$ and tail y. (See Disjoint Union {13}, Unit {8} and Pairs {4})

For the normal user, it should suffice that obviously monotonic functions on types will normally meet the criteria for forming a recursive type. (One exception is that rec(X.(X→ℕ)→ℕ) is does not meet the criteria stipulated for forming a type even though (X→ℕ)→ℕ is monotonic in X.)

A type rec(X.F(X)) is (extensionally) equal to its "unrolling" F(rec(X.F(X))), and is the smallest such type. However, unrolling rec(X.F(X)) will usually not result in an intensionally equal type. For example, rec(X.Unit+(A×X)) is not intensionally equal to Unit+(A×rec(X.Unit+(A×X))) since they have different outermost type constructors, namely "<Type>+<Type>" and "rec([Var].<Type>)". On the other hand the type rec(X.X) does indeed unroll to itself (but Thm* Void =ext rec(A.A)).

Example: S-expressions
Here is a type definition inspired by lisp's S-expressions, which are binary trees with "atoms" at the leaves.

```
   Def Sexpr(A) == rec(T.(T×T)+A)
   Def Cons(s1;s2) == inl(<s1,s2>)
   Def Inj(a) == inr(a)
   Def Case of s :  Inj(x) → g(x) ; Cons(s1;s2) → f(s1;s2)
       == InjCase(s; s1s2. s1s2/s1,s2. f(s1;s2); x. g(x))

   Thm* ∀A:Type. Sexpr(A) ∈ Type
   Thm* ∀A:Type. Sexpr(A) =ext (Sexpr(A)×Sexpr(A))+A
   Thm* ∀A:Type, s1,s2:Sexpr(A). Cons(s1;s2) ∈ Sexpr(A)
   Thm* ∀A:Type, a:A. Inj(a) ∈ Sexpr(A)
```

Here is a recursive function that counts the number of leaves:

```
    Def Size(s) == Case of s :  Inj(x) → 1 ; Cons(s1;s2) → Size(s1)+Size(s2)
        (recursive)
    Thm* ∀A:Type, s:Sexpr(A). Size(s) ∈ ℕ+

    Thm* Size(Cons(Inj(1);Cons(Inj(2);Inj(1)))) = 3 ∈ ℕ+
```

Here is a recursive function that reverses an s-expression:

```
    Def Reverse(e)
        == Case of e
             Inj(x) → Inj(x)
             Cons(s1;s2) → Cons(Reverse(s2);Reverse(s1))
        (recursive)
    Thm* ∀A:Type, e:Sexpr(A). Reverse(e) ∈ Sexpr(A)

    Thm* ∀a,b,c:X.
         Reverse(Cons(Inj(a);Cons(Inj(b);Inj(c))))
         =
         Cons(Cons(Inj(c);Inj(b));Inj(a))
```

Functions over recursive types are usually themselves recursive. See Recursive Functions {27}.

(Jan 2003 {62} - sfa)

The topic of how propositions are treated in type theory has many aspects, some rather subtle. We start with superficial aspects, which may suffice for most users. Here are some of the usual predicates and connectives.

$P \Rightarrow Q$	[if P then Q]
$P \iff Q$	[P if and only if Q]
P & Q	[P and Q]
$P \vee Q$	[P or Q]
$\neg P$	[not P]
False	[a canonical false proposition]
True	[a canonical true proposition]
$\forall x{:}A.\ P(x)$	[for every $x \in A$, P(x)]
$\exists x{:}A.\ P(x)$	[there is some $x \in A$ such that P(x)]
$a = b \in A$	[a is the same member of A as b; the type may be elided]
$a \neq b \in A$	
$a<b \quad a\leq b \quad a>b \quad a\geq b$	[relations between integers]

Rather than interpret propositional formulas as having Boolean {6} values, i.e. as being either true or false, we use a type "Prop" of "propositional values". (This is because users of Nuprl usually want their results to be constructively or intuitionistically meaningful, which precludes the classical Boolean-valued interpretation of propositions.) For example,

Thm* $\forall i,j{:}\mathbb{Z}.\ (i<j) \in$ Prop
Thm* $\forall T{:}$Type, $a,b{:}T.\ (a = b \in T) \in$ Prop
Thm* False $\in$ Prop
Thm* $\forall A{:}$Prop. $(\neg A) \in$ Prop
Thm* $\forall A,B{:}$Prop. $(A \vee B) \in$ Prop
Thm* $\forall A,B{:}$Prop. (A & B) $\in$ Prop
Thm* $\forall A,B{:}$Prop. $(A \Rightarrow B) \in$ Prop
Thm* $\forall A{:}$Type, $P{:}(A\rightarrow$Prop$).\ (\exists x{:}A.\ P(x)) \in$ Prop
Thm* $\forall A{:}$Type, $P{:}(A\rightarrow$Prop$).\ (\forall x{:}A.\ P(x)) \in$ Prop

Observe that we treat properties, relations, connectives and quantifiers as Prop-valued Functions {23}. Propositional connectives take Prop's as arguments. Quantifiers such as "$\exists x{:}A.\ P(x)$" and "$\forall x{:}A.\ P(x)$" take two arguments, one being the "domain of quantification", and the other being a property (i.e. Prop-valued function) defined on that domain. As indicated by the forms of these theorems, one can quantify over (some) propositions and properties, thus, the Nuprl language is higher-order. See Higher-Order Propositions {35}.

Two type constructors having propositional components are {x:A| B(x) } and T(given P), which are discussed in Comprehension Subtypes {14} and Guarded Types {36}. There is, however, no "place" in computational type theory for a "choice" operator such as David Hilbert's, which takes any type having a member and denotes a member of it, because, as explained in Types {15}, the method for computing any such purported choice operations would have to have been already defined independently of the types.

The heart of intuitionistic type theory, such as Martin-Lof's or Nuprl's, is to account for some "computational content" of propositions, expressed as types of such content. Thus, there is a sense in which one can represent Propositions as Types {56}.

Finally, there is another distinct notion of proposition, namely the content of what one may assert in Nuprl Proofs {51}. Those forms of assertion used are codified as Sequents {50}, and are designed to help us reason about types and about propositions in the above sense. In type theory one normally reserves the nomenclature "Proposition" for the notion employed above, and adopts some other term such as "possible judgement" or "form of judgement" for the other; in Nuprl we tend simply to use "sequent" since it is the form we have adopted for such forms of assertion in Proofs {51}.

(Sept 2003 {62} - sfa)

Higher-Order Propositions higher_order_prop_doc {35}

In Propositions {34} several theorems about typing the usual logical connectives and quantifiers were stated, such as Thm* ∀A,B:Prop. (A ∨ B) ∈ Prop, which itself quantifies over propositions, and presents the connective as an operation on propositions.

Here we shall consider various more substantial examples of such "higher-order" propositions. First, observe that the when quantification over propositions and properties (proposition valued functions) is permitted, then the other standard connectives and equality can all be characterized purely in terms of universal quantification and implication:

```
Thm* A & B ⟺ (∀C:Prop. (A ⇒ B ⇒ C) ⇒ C)
Thm* A ∨ B ⟺ (∀C:Prop. (A ⇒ C) ⇒ (B ⇒ C) ⇒ C)
Thm* False ⟺ (∀C:Prop. C)
Thm* ∀B:(A→Prop). (∃x:A. B(x)) ⟺ (∀C:Prop. (∀x:A. B(x) ⇒ C) ⇒ C)
Thm* ∀a,b:A. a = b ⟺ (∀P:(A→Prop). P(a) ⇒ P(b))

Thm* ¬A ⟺ (∀C:Prop. A ⇒ C)
Thm* True ⟺ (∀C:Prop. C ⇒ C)
Thm* (A ⟺ B) ⟺ (∀C:Prop. ((A ⇒ B) ⇒ (B ⇒ A) ⇒ C) ⇒ C)
```

The simplicity of these characterizations leads one to consider simply adopting them as definitions in the first place, and indeed it is not unusual to do so in standard developments of higher-order logic. But we cannot do so because we have constrained ourselves to a so-called "predicative" logic (a term of Bertrand Russell's of uncertain etymology), which requires layering propositions into a hierarchy in a way that precludes us from giving the desired definitions. See Prop Levels and Predicativity {39}.

Here's another example asserting the definability of properties by structural recursion on lists, given base case Q ∈ Prop and step method R ∈ A→(A List)→Prop→Prop:

```
Thm* ∀R:(A→(A List)→Prop→Prop).
       ∃P:((A List)→Prop).
```

$(P(nil) \iff Q)$ & $(\forall a{:}A,\ x{:}A\ List.\ P(a.x) \iff R(a,x,P(x)))$

Equality on the type Prop.

Just as Type is a type of Intensional Types {17}, so members of Prop are treated intensionally. One natural alternative would be to identify propositional expressions with their truth-values, which is to identify any two propositions P and Q such that $P \iff Q$. At the other extreme, one might identify propositions strictly by their syntactic identity, or perhaps modulo change of bound variables. In Nuprl, the "equality" on Prop is analogous to that on Type. For the most part, propositional expressions are equal when their corresponding constituent expressions are equal in the appropriate types. And of course, like any other Nuprl type, expanding an Operator Definition {47} or rewriting by Computation {42} steps results in an equal expression.

One case of particular note is that $(a = b \in T) = (a' = b' \in T') \in Prop$ just when $T = T' \in Type$, and $a = a' \in T$, and $b = b' \in T$.
Thus, $(a \in T) = (a' \in T') \in Prop$ just when $T = T' \in Type$, and $a = a' \in T$.

See Prop Levels and Predicativity {39}.

(Feb 2001 {62} - sfa)

Sometimes when describing the intended types of arguments to functions, we want to hedge our typing with some further condition.

For example, the conjunction operator "A & B" of Nuprl is actually what is often called a "cand" or "conditional and", in which the second conjunct is required to be sensible only when the first conjunct is true. Consider the theorem:

Thm* $\forall$k:$\mathbb{N}$. 0<k & (2 $\div$ k)<1 $\Rightarrow$ 2<k

Even if the expression "2 $\div$ k" is assigned no truth value for k = 0, it can occur in a context entailing k $\neq$ 0, such as in "0<k & (2 $\div$ k)<1" in this case.

The well-formedness of (conditional) "and" can be expressed thus:

Thm* $\forall$A:Prop, B:Prop(given A). (A & B) $\in$ Prop

Similarly, "A $\Rightarrow$ B" and even "B(given A)" itself are of the same kind.

Thm* $\forall$A:Prop, B:Prop(given A). (A $\Rightarrow$ B) $\in$ Prop

Thm* $\forall$A:Prop, B:Type(given A). B(given A) $\in$ Type

Thus, one can naturally express dependent propositions such as "$\forall$k:$\mathbb{N}$. 0<k $\Rightarrow$ (2 $\div$ k)<1", where the truth of the antecedent would entail that the consequent makes sense.

More specifically, the membership of a type "T(given A)" has these properties: for any (closed) expression "e", if "A" is true then e $\in$ T(given A) just when e $\in$ T. If "A" is false then e $\in$ T(given A) unconditionally.

There is a special connection between guarded types and Comprehension Subtypes {14}:

Thm* $\forall$P:(A$\rightarrow$Prop), B:(x:A$\rightarrow$Type(given P(x))).
x:{u:A| P(u) }$\rightarrow$B(x) =ext x:A$\rightarrow$B(x)(given P(x))

See also Intersection Types {20} and Propositions {34}.

(Feb 2001 {62} - sfa)

Functionality Sequents and Well-formed Formulas
- from Functions {23} and Sequents {50}

In Nuprl, the basic form for making assertions about the functional behavior of expressions with free variables is the Functionality Sequent, a special case of Sequents {50}. In the degenerate form such assertions merely assert the well-formedness of expressions, indicating a type for the expression, such as "⊢ (λx.x) ∈ ℕ→ℕ" or "⊢ 1+2 ∈ {x:ℤ| 0<x }". (Note that an expression can have many types.)
More commonly, a sequent asserts the well-formedness of an expression in free variables or under assumptions. For example, the following sequent expresses the fact that "x+x" denotes a ℤ-valued function in x ∈ ℤ:

```
  1. x : ℤ
  ⊢  x+x ∈ ℤ
```

Or, considering a multi-argument function:

```
  1. x : ℤ
  2. y : ℤ
  ⊢  x+x(y+2) ∈ ℤ
```

Proofs of sequents often proceed by progressive decomposition of parts. See EXAMPLE {60}

Equivalent assertions using closed forms for these function expressions would be these:

```
  ⊢  (λx.x+x) ∈ ℤ→ℤ

  ⊢  (λx,y. x+x(y+2)) ∈ ℤ→ℤ→ℤ
```

For any type A, and any type-valued function B(x) over x ∈ A, the following assertions are equivalent:

```
  ⊢  (λx.b(x)) ∈ x:A→B(x)

  1. x : A
  ⊢  b(x) ∈ B(x)
```

However, the sequent form generally is more expressive, because the sequent form

```
   1. x : A
   ⊢  b(x) ∈ B(x)
```

means that IF the expression A denotes a type, then B(x) denotes a type-valued function over x ∈ A, and b(x) denotes a B(x)-valued function over A.
The purpose for this conditional is to allow us in proofs to separate the problem of showing that A denotes a type from the problem of showing other proof obligations. Suppose we want to show that "⊢ (λx.b(x)) ∈ x:A→B(x)". This can be broken into two independent problems of showing other sequents, namely,

⊢ A ∈ Type, which suffices to show that A denotes a type (see Type Functionality Sequents {38}) and

1. x : A
⊢ b(x) ∈ B(x).

The second of these assertions assumes for the sake of argument that A is a type, while the first of these assertions says that A is a type. Combining these justifies our original goal, namely, "⊢ (λx.b(x)) ∈ x:A→B(x)".

See Type Functionality Sequents {38} and Sequents {50}.

(Feb 2001 {62} - sfa)

---

type_functionality_sequents_doc {38}

Type Functionality Sequents and Level Expressions
- from Functionality Sequents {37} and Universes {18}

Discussions in type theory often refer to expressions being "functional" on some input types and with respect to some output type (see Functions {23}). But also we use the concept of an expression being "type-functional" on some input types, meaning equal inputs give output expressions denoting the same type. As was mentioned in Universes {18}, we cannot assimilate the concept of type-functionality to functionality with respect to an output type because there is no type TYPE comprising all types; if there were, then type-functionality would simply be functionality with respect to output type TYPE.

Although reasoning about type functionality via a type of all types is impossible, one might still reason about it by adding a form of assertion about it, as Martin-Lof did in his 1980 type system from which Nuprl stems. Despite the advantages of reasoning about type-functionality explicitly, the original designers of Nuprl hoped to simplify their design by not doing so (however, such rules are likely to be incorporated eventually); rather, the primitive rules of Nuprl simply approximate most reasoning about "all types" by reasoning instead about a hierarchy of types Type{i} called "universes", which together cover all types (see Universes {18}).

So, rather than having a rule saying, for example, that A+B is a type if A and B are, there is a rule that allows us to infer that A+B ∈ Type{i} given A ∈ Type{i} and B ∈ Type{i}, whatever i might be. Indeed one can prove Thm* ∀A,B:Type{i}. A+B ∈ Type{i}.

Notice that in the statement of Thm* ∀A,B:Type{i}. A+B ∈ Type{i}, the universe level is left as a variable rather than a constant. In a Nuprl, the meaning of a sequent is always general with respect to choices of levels that can be assigned to letters used inside universe level expressions.

Consider Thm* ∀A:Type{i}, B:Type{j}. A+B ∈ Type{[i | j]}.
This says that the disjoint union of two types from any two universe levels is a type in the universe whose level is the maximum of the two levels. So, there is a "max" operation you can use in universe level expressions.

There is also a successor operation: Thm* Type{i} ∈ Type{i'}.
As a convenience, there is also an offset operation that adds a constant to a

level: Thm* Type{i'} ∈ Type{i 2}.

Although there is no need to do so in a theorem statement, one can also use constant level expressions: Thm* ℤ ∈ Type{3}.

The forms of level expression may be freely combined.

Universe Polymorphism.

The reason it is practical to use these variables in level expressions (but note that they are not like other ordinary variables of the logic -- you cannot bind them) is that any true sequent remains true if all the level variables throughout it are uniformly replaced by level expressions of your choice.
So from "⊢ ∀A:Type{i}, B:Type{j}. A+B ∈ Type{[i | j]}" it follows that "⊢ ∀A:Type{3}, B:Type{k'}. A+B ∈ Type{[3 | k']}".
It also follows that "⊢ ∀A,B:Type{i}. A+B ∈ Type{[i | i]}" which is equivalent to "⊢ ∀A,B:Type{i}. A+B ∈ Type{i}".

In a typical sequent, it turns out, all the universe level expressions are the same and consist of simply a level variable. When this is so, we normally choose to display the universe expression simply as "Type" rather than in full, such as "Type{i}". Thus, one would normally display the sequent "⊢ ∀A,B:Type{i}. A+B ∈ Type{i}" simply as "⊢ ∀A,B:Type. A+B ∈ Type".

The same level expressions are used as indices to Prop{[level]}.
See Prop Levels and Predicativity {39}.

(Feb 2001 {62} - sfa)

Propositional Levels and Predicativity prop_levels_doc {39}

As was mentioned in Higher-Order Propositions {35}, even though one could characterize, say, disjunction or equality with a higher-order formula involving only implication and quantification over propositions, thus

```
Thm* A ∨ B ⟺ (∀C:Prop. (A ⇒ C) ⇒ (B ⇒ C) ⇒ C)
Thm* ∀a,b:A. a = b ⟺ (∀P:(A→Prop). P(a) ⇒ P(b))
```

we cannot adopt these as definitions, although it is not unusual to do so in some other accounts of higher-order logic. The reason is that we have adopted a "predicative" semantics (a term of Bertrand Russell's of uncertain etymology) which avoids defining a proposition by quantifying over a domain that includes the proposition to be defined. Basically, in a predicative logic, to assign meaning to a quantificational expression one must "already" have given meaning to all the expressions that might be used as witnesses. So, assigning a meaning to "∀C:Prop. (A ⇒ C) ⇒ (B ⇒ C) ⇒ C" requires already having assigned a meaning to every member of Prop, which therefore could not include "∀C:Prop. (A ⇒ C) ⇒ (B ⇒ C) ⇒ C" itself as a member. Thus, there is no type of all propositions in predicative type theory.

The solution adopted here is to stratify the propositions into a hierarchy Prop{i} where "i" is some positive number. Then

Thm* ∀A,B:Prop{i}. (∀C:Prop{i}. (A ⇒ C) ⇒ (B ⇒ C) ⇒ C) ∈ Prop{i'}
Thm* ∀A:Type{i}, a,b:A. (∀P:(A→Prop{i}). P(a) ⇒ P(b)) ∈ Prop{i'}

where Prop{i'} is the next level of propositions above Prop{i}. The propositional levels are treated identically to the type levels explained in Universes {18} and in Type Functionality Sequents {38}. But disjunction and equality do not raise the propositional level, so the "iffs" of their higher-order characterizations cannot be converted to definitions:

Thm* ∀A,B:Prop{i}. (A ∨ B) ∈ Prop{i}
Thm* ∀T:Type{i}, a,b:T. (a = b ∈ T) ∈ Prop{i}

As with Universe Levels {38} we normally elide the proposition level from display when it just a single level variable used throughout the expression. See Propositions {34} for examples.

Quantification over types raises the propositional level, just as does quantification over propositions, indeed, "Type{[level]}" and "Prop{[level]}" are interchangeable as far as determining levels is concerned.

Thm* Prop{i} ∈ Type{i'}
Thm* Prop{i} ⊆ Prop{i'}

Because the universes and propositional types are cumulative, one can often state theorems with a single level variable. For example,

Thm* ∀A:Type{k}, P:(A→Prop{j}). (∃x:A. P(x)) ∈ Prop{[k | j]}

which says that the existential is a proposition at the maximum of the two levels of its domain and predicate, follows simply from the at-first-sight less general

Thm* ∀A:Type{i}, P:(A→Prop{i}). (∃x:A. P(x)) ∈ Prop{i}

since if A ∈ Type{k} and P ∈ A→Prop{j} then A ∈ Type{[k | j]} and P ∈ A→Prop{[k | j]}, putting (∃x:A. P(x)) ∈ Prop{[k | j]}. Of course, eliding the single level variable from the display of the last theorem gives us our usual

Thm* ∀A:Type, P:(A→Prop). (∃x:A. P(x)) ∈ Prop

See also Russell's Orders {40}.

(May 2001 {62} - sfa)

Type levels and propositional levels are explained in Universes {18} and Prop Levels and Predicativity {39}.

Bertrand Russell's "ramified" theory of types is the principal ancestor to intuitionistic type theory. Russell conceives of a type as a "range of significance" for a propositional function. In an impredicative logic, such as Frege's, the notion of type is "extensional" in that the type of a functional expression is determined wholly by its input and output types. For example, in Frege's logic, the types are basically these: individuals; one-place functions of individuals; one-place functions of one-place functions of individuals; two-place functions on individuals; two-place functions taking one individual and one one-place function of individuals; and so forth.

In contrast, in addition to this simple extensional type structure, there is an "intensional" aspect to Russell's types, hence the nomenclature "ramified" (meaning "branching"). Russell assigns "orders": individuals are the lowest order entities; then come propositions that don't involve quantification over propositions; then propositions that involve quantification over propositions of the previous sort; and so forth with various mixtures.

Here is Nuprl's approximation of order in ramified type theory.

In Nuprl's type theory the members of Type{1} are themselves types of individuals; call them first-order types. The members of Prop{1} are propositions involving at most quantification over first-order types of individuals; call them first-order propositions. The members of Type{2} may be totalities definable given all of Type{1} and Prop{1} themselves as classes of entities; call them second-order types (thus Type{1} and Prop{1} are second-order types). The members of Prop{2} are propositions involving at most quantification over second-order types including Type{1} and Prop{1}. And so forth.

Observe that we have included lower-order types among higher-order types, and lower-order propositions among higher-order propositions. This has a certain convenience, but is actually critical in intuitionistic computational type theory since one cannot effectively recognize when a second-order type or proposition, say, is actually first-order. For example,

Thm* ∀f:(ℕ→ℕ). (∀n:ℕ. if f(n)=$_2$0→ False else ∀C:Prop{1}. C fi) ∈ Prop{2}

But whether (∀n:ℕ. if f(n)=0→ False ; ∀C:Prop{1}. C fi) ∈ Prop{1} depends on whether f is everywhere zero, which is not in general recognizable.

Thm* ∀f:(ℕ→ℕ).
 (∀n:ℕ. f(n) = 0)
 ⇒
 (∀n:ℕ. if f(n)=$_2$0→ False else ∀C:Prop{1}. C fi) ∈ Prop{1}

(March 2001 {62} - sfa)

Terms (Expressions) terms_doc {41}

Nuprl employs a generic form of expression, or term, for both primitive and defined {47} operators. We often prefer the nomenclature "term" because "expression" connotes a meaningfulness that is not a priori assigned to these terms.

A term consists of: zero or more immediate subterms; for each subterm's place, a list of zero or more binding variables; and a nonempty list of non-term constituents, such as strings or numbers, which we call "operator parameters" or "op-parms" for reasons to be indicated below. For example, the term "∀x:A. 0≤x", has two immediate subterms, "A" and "0≤x", the variable "x" becoming bound in "0≤x" and no variables becoming bound in "A"; the only op-parm is the string "all", chosen for mnemonic purposes.

Normally one can infer the relevant term structure from the definitions provided for operators, or when the notation is standard or familiar. When we want to show the form of a term explicitly for some reason, we employ a uniform notation we have adopted for that purpose. For example, the form of "∀x:A. 0≤x" is

```
all{}(A; x.0≤x)
```

which shows that the first subterm is "A", with no variables becoming bound there, the second subterm is "0≤x", with the variable "x" becoming bound there, and there is just one op-parm "all". This first op-parm is called the "op-id".

The general form for this way of showing term structure is:

```
<opid>{<parms>}(<bterms>)
```

where <opid> is the first op-parm, usually thought of as (but not necessarily) identifying the operator; <parms> is a possibly empty sequence of op-parms; <bterms> is a sequence of "bound-terms" each consisting of a term and a list of binding variable names. When the <parms> or <bterms> is empty we often suppress the brackets surrounding them. Here we list several terms indicating both the structure revealing form, and the normal display.

```
nat                         ℕ
natural_number{4:n}         4
token{abc:t}                "abc"
variable{x:v}               x
variable{a:v}(x)            a(x)      (second-order variable instance)
variable{a:v}(x; y)         a(x;y)    (second-order variable instance)
add(a; b)                   a+b
lambda(x.b)                 λx.b
all(A; x.B)                 ∀x:A. B
let(a; x.b)                 let x = a in b
sym(A; x,y.B)               Sym x,y:A. B
decide(e; x.a; y.b)         InjCase(e; x. a; y. b)
wellfounded{i:l}(A; x,y.B)  WellFnd{i}(A;x,y.B)
```

Any combination of term components forms a term. Note that there are several kinds of op-parm values, indicated after the colon in the above forms: [nat]:n [token]:t [string]:s [var]:v [level]:l.

See Computation {42}.

(March 2001 {62} - sfa)

Terms {41} in Nuprl provide the syntactic entities used for expressions. In Nuprl, one might say, terms are adopted as expressions for mathematical objects when Types {15} are defined. The terms become notations for the values of that type, such as 1+2 $\in \mathbb{Z}$, and sometimes notations for the types themselves, such as "$\mathbb{Z}$" itself.

We follow a method given by Per Martin-Lof.

Prior to, that is independently of, using terms to denote mathematical values in this way, one defines the raw, untyped computational behavior of terms. This behavior is then used in part to explain how mathematical values of various types shall be denoted.

All that the semantics of types requires is that we know how to execute, or "evaluate", closed terms. (A closed term is one with no free variables.) If such execution produces a result, that result is sometimes called the "value" of the term; beware that, in this sense, the value of "1+2" is not the number 3, but is instead a term "3", i.e., a numeral.

All closed terms are classified as having "canonical" or "non-canonical" form, and whether a term is in canonical form depends only on its outermost form. Specifically, whether a term is in canonical form is preserved by any change of binding variables or subterms.

Any closed term in canonical form simply evaluates to itself. Any result of evaluating a term must be closed and have canonical form. A key part of introducing a new primitive operator into the semantics is to explain how to evaluate it, and this explanation must be independent of type definition. A term can have at most one value.

Here is an example.

> The pair constructor <a,b> is in canonical form.
> A closed expression of form w/x,y. e(x;y) has a value R iff w has some value <a,b> and e(a;b) has value R. See Pairs {4}.

And another.

> The left- and right-injectors inl(a) and inr(b) are in canonical form.
> A closed expression of form InjCase(w; x. e(x); y. f(y)) has a value R iff either w has some value inl(a) and e(a) has value R or else w has some value inr(b) and f(b) has value R. See Disjoint Union {13}.

See Canonical Forms {43} and NonCanonical Forms {44} for lists and descriptions of various primitive operators.

See Evaluation and Term Rewriting {45}.

(Dec 2001 {62} - sfa)

Canonical Forms - continued from Computation {42} canonical_forms_doc {43}

Here we list the primitive operators having canonical form, which means that their (closed) instances simply evaluate to themselves.

First, we list those operators that happen to be used to denote type constructors, although so classifying operators is not part of how they evaluate.

```
x:A→B(x)
u:A×B(u)
Type{i}         (where i is a Level Expression)
Atom
Void
ℤ
a<b
A+B
u,v:A//E(u;v)
{x:A| B(x) }
u = v ∈ A
∩x:A. B(x)
A List
rec(A.F(A))
a ∼ b
```

Here are the remainder of the canonical forms:

```
*
inl(a)
inr(b)
nil
a.as
<a,b>
λx.b(x)
"$tok"    (where $tok is a string of characters)
$n        (where $n is one or more digits)
```

See NonCanonical Forms {44}.

(Feb 2001 {62} - sfa)

NonCanonical Forms - continued from Computation {42}

Here we list the NonCanonical forms and indicate how they evaluate, except for forms that operate specifically on numeric literals or token literals; for those, see Integers {10} and Atoms {12}.

Destructors are operators that break down complex expressions having canonical form, and continue computing on those parts. Here are our primitive destructors:

```
InjCase(inl(a); u. c(u); v. d(v)) *⇒ c(a)

InjCase(inr(b); u. c(u); v. d(v)) *⇒ d(b)

Case of nil; nil → b ; x.xs, rec:v → f(x;xs;v) *⇒ b

Case of a.as; nil → b ; x.xs, rec:v → f(x;xs;v)
*⇒
f(a;as;Case of as; nil → b ; x.xs, rec:v → f(x;xs;v))

<a,b>/c,d. f(c;d) *⇒ f(a;b)

(λx.b(x))(a) *⇒ b(a)
```

Each of these rewrites has the following significance. For any closed instance of the rewrite, to evaluate the left-hand side, one evaluates the right hand side.

These forms on the left-hand sides are called "redexes" and are said to "reduce" to their "contractums" shown on the right-hand side. In these destructor forms, the place filled by the term to be analyzed is called a "principal argument" place, which means that it is evaluated in place before the resulting redex is evaluated.

It is also possible to have operators that compute regardless of the form of the arguments.

```
recind(a; f,x. g(f;x)) *⇒ g(λx.recind(x; f,x. g(f;x));a)
```

There is also a rather degenerate and unnecessary noncanonical form that ignores its argument and has no value at all, namely "any(x)". It is a vestige of another, older type system.

See Canonical Forms {43}.

(March 2001 {62} - sfa)

As mentioned in Types {15} and Type Expressions {16}, an expression will denote the same value or type if any part of it is rewritten by a step of Computation {42}, or bound variables are uniformly changed. And naturally, as mentioned in Operator Definition {47}, expanding the definition of an operator occurring in an expression preserves the meaning.

Since equality between terms, either with respect to a type or between extensional or intensional type expressions themselves, is symmetric and transitive, any chain of these alterations, in either direction, preserves the denotation of an expression. We shall usually indicate such rewrites like this:

<a,b,c>/x,y,z. f(x;y;z) *⇒ <b,c>/y,z. f(a;y;z)

and they may be chained

<a,b,c>/x,y,z. f(x;y;z) *⇒ <b,c>/y,z. f(a;y;z) *⇒ f(a;b;c)

The "<pre term> *⇒ <post term>" form indicates the direction of the intended rewrite, though a rewrite can always be reversed as well. The reason a direction is indicated is that these notations are used in Proofs {51} to indicate rewrites to be performed during steps of inference.

It is possible to indicate subparts of a term to be rewritten, and how to rewrite it. As will be exemplified below, the ⌈<term>⌉ operator indicates which part the term is to be rewritten. One may always simply indicate that the whole term is rewritten, leaving it to the reader whether subparts are rewritten, but sometimes it helps to focus attention. Also, since these rewrite instructions are created interactively by the user pointing at parts to rewrite, and how to do the rewrite, these ⌈<term>⌉ indications are actually in the rewrite specification, and are simply being suppressed for improved readability. For example,

if true_2→ 1+1 else 2+2 fi *⇒ InjCase(true_2 ; 2; 2+2) *⇒ 2

was actually derived from the following rewrite specification, and implicitly contains the same information about how to do the rewrite:

⌈if true_2→ ⌈1+1⌉ *⇒ 2 else 2+2 fi⌉
*⇒
⌈InjCase(true_2 ; 2; 2+2)⌉
*⇒
⌈InjCase(inl(·) ; 2; 2+2)⌉
*⇒
2

Note that there are more general descriptive, non-schematic methods for specifying rewrites in proofs, and there are methods for performing rewrites that are justified by semantics of equality beyond simply evaluation and definition. These are not discussed here.

(March 2001 {62} - sfa)

As was mentioned in Types {15}, rewriting any subexpression by a step of symbolic computation will produce an expression equal to it in any type of which the original expression is a member. See [Howe LICS 1989].

This congruence relation between terms could only be directly represented in the Nuprl language itself if terms were. And while reflecting term structure is possible, it has not been implemented in the extant system. (Some theoretical work on reflection may be found in [Allen,Constable,Howe and Aitken 1987] (postscript). Barzilay is working on newer practical methods; see [Barzilay and Allen 2002])

However, a partial reflection of the squiggle relation between terms can be easily implemented, and exploited to allow limited reasoning about rewriting expressions in a few types that are used a lot in Nuprl. Because of this limited applicability, it is not clear how theoretically significant this partial reflection is.

The type "r ∼ t" is defined semantically to be a proposition for any closed terms r and t, and is true just when the two terms are related by Howe's "squiggle" relation. One interpretation one might make is that this is equality for the type of closed terms modulo Howe's "squiggle" relation, and in the expression "r ∼ t", each of r and t denotes its own equivalence class under "squiggle".

Then various rewrites can be justifed by arguments about r ∼ t, such as

```
  inferring  "1. x : H  ⊢  B(x; r(x))"
  from       "1. x : H  ⊢  B(x; t(x))"
  and        "1. x : H  ⊢  r(x) ∼ t(x)"
```

or

```
  inferring  "1. x : H  2. B(x; r(x))  ⊢  C(x)"
  from       "1. x : H  2. B(x; t(x))  ⊢  C(x)"
  and        "1. x : H  2. B(x; r(x))  ⊢  r(x) ∼ t(x)".
```

Of course, we need to get our "r ∼ t" premises from somewhere, and "⊢ b ∼ b" can get us started, but the real leverage is to exploit that for certain types commonly use in Nuprl, equal expressions in those types are always "squiggle" related. These include ℤ, ℕ, 𝔹, Unit, and Atom.

```
   Def SQType(T) == ∀x,y:T. x = y ⇒ {x ∼ y}

   Thm* SQType(ℤ)
   Thm* SQType(ℕ)
   Thm* SQType(𝔹)
   Thm* SQType(Unit)
   Thm* SQType(Atom)
```

But there are common types that do not have this property:

   Thm* (**λ**n.n$<_2$0) = (**λ**n.false$_2$) ∈ ℕ→𝔹

Thm* $\neg((\boldsymbol{\lambda}n.n<_2 0) \sim (\boldsymbol{\lambda}n.\mathrm{false}_2))$

And many ways one might chooose to represent classes of values lead to types that violate this property. For example, you might choose to represent k-tuples of numbers as $\{1...k\}\rightarrow\mathbb{Z}$, or use "Top" rather than "Unit" as your standard 1-element type, or Top+Top instead of $\mathbb{B}$. You might want to define the rational numbers as equivalence classes of integer pairs rather than, say, reduced canonical fractions. In any of these scenarios the resulting types will have non-squiggle-related but equal (i.e. co-denoting) notations.

So perhaps it is best to consider the use "r $\sim$ t" in the logic as a mere expedient.

(August 2001 {62} - sfa)

Operator Definitions opdef_doc {47}

An operator definition has the form

<defined op instance> == <expansion>

which stipulates that the left-hand side "expands" to the right-hand side. Or more generally, for any "instantiation" of the definition, its left-hand side expands to its right-hand side. As a convenience it is also permitted to use function applications on the left-hand-side instead of lambda operators on the right-hand-side; for example the definition of function composition Def (f o g)(x) == f(g(x)) is equivalent to f o g == $\boldsymbol{\lambda}$x.f(g(x)).

This leaves three things to be explained:
1. So what if one term expands to another?
2. What is an instance of a definition?
3. What restrictions are there on legitimate definitions?

1. Part of the meaning of an assertion in Nuprl is that it is true iff expanding all the defined operators within it (until none remain) results in a true assertion. Thus, a term can be regarded as meaning whatever it fully expands to.

Below we shall employ the concepts about the form of expressions explained in Terms {41}.

2. Instantiating a definition consists of possible change of binding variables, first- and second-order substitution for free variables throughout the definition, and uniform replacement of op-parm variables by op-parm values or other suitable expressions. See Substitution {48}. Op-parm variables are displayed like "$x" within terms, except for level-expression variables, which are simply identifiers. Here are examples:

$\mathbb{N}$ expands to $\{i:\mathbb{Z}|\ 0\leq i\ \}$ according to Def $\mathbb{N}$ == $\{i:\mathbb{Z}|\ 0\leq i\ \}$

1+2$\leq$x expands to $\neg$x<1+2 according to Def A$\leq$B == $\neg$B<A

x.12 expands to <"x",12> according to Def $abc.$n == <"$abc",$n>

"Induction for x:$\mathbb{N}$. 0<x"

expands to
"0<0 & (∀x:ℕ. 0<x ⇒ 0<x+1) ⇒ (∀x:ℕ. 0<x)"
according to
Def Induction for x:ℕ. P(x) == P(0) & (∀x:ℕ. P(x) ⇒ P(x+1)) ⇒ (∀x:ℕ. P(x))

Weird(3; u,v. v·vu) expands to 3·3(0·03)
according to
Def Weird(x; u,v. F(u;v)) == F(F(x;0);x)

HigherConsequence{k'}(0<x) expands to {X:Prop{k''}| 0<x ⇒ X }
according to
Def HigherConsequence{i}(P) == {X:Prop{i'}| P ⇒ X }

(Prop{[level]} takes a level-expression; see
Prop Levels and Predicativity {39}.)

3. See Op Def Restrictions {49}.

(November 2002 {62} - sfa)

Substitution (second-order) subst_doc {48}

Substitution is used to give an Operator Definition {47} in Nuprl, and showing how operators can be expanded according to definitions seems the clearest way to illustrate how substitution works.

First-order substitution is a method of inserting expressions into expressions, perhaps changing binding variables to avoid capture. For example according to the definition

Def {i...j} == {k:ℤ| i≤k & k≤j }

the expression "{3...x+2}" expands to "{k:ℤ| 3≤k & k≤x+2 }", the expressions "3" and "x+2" being substituted for "i" and "j" in "{k:ℤ| i≤k & k≤j }". But expression "{3...k+2}" expands to "{k@0:ℤ| 3≤k@0 & k@0≤k+2 }", avoiding a capture of "k".

Generally, one specifies a first order substitution by giving the "matrix", i.e. the expression into which one substitutes, and one gives a list of "target" variables to be replaced, and for each target one gives its "replacement expression". If we consider the replacement expressions as arguments, then the matrix and target variables together specify a "first-order K-place substitution function" if there are K targets given.

Second-order substitution is a method for inserting instances of a "pattern" into an expression, again changing binding variables as need be to avoid capture, where the pattern is itself simply instantiated by first order substitution. Consider the definition of symmetry for a relation.

Def Sym x,y:T. E(x;y) == ∀a,b:T. E(a;b) ⇒ E(b;a)

According to this definition, "Sym x,y:ℤ. y<x+1" expands to "∀a,b:ℤ. b<a+1 ⇒ a<b+1". It works like this. Of course, T is replaced by ℤ, but the "second-order" variable instances "E(a;b)" and "E(b;a)" are replaced by

performing first-order substitution into "y<x+1". It is not necessary to use the same binding variables -- they will be changed as needed. Thus, "Sym u,v:ℤ. v<u+1" also expands to "∀a,b:ℤ. b<a+1 ⇒ a<b+1".

A second-order K-place variable takes expressions, K of them, as arguments in any given "instance" of that variable. In the above example, "E(?;?)" is a second-order 2-place variable, which is to say its instances are the expressions generated by filling in its 2 argument places with expressions. In the definition of symmetry there are three instances of this second-order variable, namely E(x;y), E(a;b) and E(b;a).

Generally, one specifies a second-order substitution by giving the matrix, a list of target variables which may now include second-order as well as first-order variables, and for each target variable, if it is a first-order variable then one gives its replacement expression, but if it is a second-order variable then one gives its "replacement function" which is a first-order substitution function (see above) with as many places as the second-order variable takes.

It is easy to see where the replacement functions for second order variables come from when expanding an operator instance according to an Operator Definition {47}. The immediate subterms of a definiendum (left side) are taken as the target variables, and the subterms of the operator instance, in those places that have binding variables, are taken together with those binding variables to determine the replacement function (which is a first-order substitution function).

As a last example illustrating more complex substitution consider this definition:

    Def Weird(x; u,v. F(u;v)) == F(F(x;0);x)

    Weird(3; u,v. v·vu) expands to 3·3(0·03)

    Weird(3; v,u. v·vu) expands to (3·30)·(3·30)3

    Weird(3; u,v. u) and Weird(3; v,u. u) both expand to 3

Finally, observe that first-order variables can be construed simply as 0-place second-order variables, and then first-order substitution becomes a special case of second-order substitution.

(Feb 2001 {62} - sfa)

Operator Definitions: Restrictions - continued from Operator Definition {47}

Here are the restrictions we place on definitions.

First we consider some restrictions on a collection of definitions as a whole. Expansion must be well-founded, and no two definitions can be instantiated in such a way that the resulting left-hand sides are the same. This way there is no ambiguity about how to expand a term and repeated expansion must come to an end.

The left-hand side, the definiendum, must be "general" in the following ways. A subterm in a position where no variables can become bound must be filled by a (first-order) variable, as in Def A≤B == ¬B<A. The left-hand side of a definition could not be "1≤B" because the subterm "1" is not a variable. A subterm in a position having binding variables must be an "instance" of a second-order variable {48} whose subterms are the binding variables, and there are no duplicates among the binding variables or subterms of the second-order variable instance, as in Def Sym x,y:T. E(x;y) == ∀a,b:T. E(a;b) ⇒ E(b;a). It would not be possible to make a definition with left-hand side "Sym x,x:T. E(x;x)" or "Sym x,y:T. E(x;y;z)" or "Sym x,y:T. E(x)". Distinct subterm positions of the left-hand side must be filled by distinct variables.

Note that this condition on definienda together with the well-foundedness of definition expansion entail that an instance of the left-hand side of a definition cannot occur on the right-hand side of the definition, since that would cause an expansion cycle. Recursive Functions {27} and Recursive Types {33} are defined by other means.

On the left-hand side an op-parm place (see Terms {41}) may be filled either by an op-parm variable (see Operator Definition {47}) as in Def $abc.$n == <"$abc",$n>, or by a constant. That is, one is not permitted to use a complex op-parm expression on the left if it contains a variable, so for example, "HigherConsequence{k'}(P)" cannot be the left side of a definition since the level expression is a complex level-expression containing a variable. Note that one can use op-parm constants to distinguish methods of expansion. For example, the following can coexist:

```
Def sfa_doc_oparm_sample{1:n}(A) == ℕ→A

Def sfa_doc_oparm_sample{2:n}(A) == A×ℕ
```

The right-hand side (definiens) of a definition is constrained by the left-hand side (definiendum). All the first order variables occurring free, all the second-order variables occurring, and all the op-parm variables occurring in the right-hand side must occur in the left-hand side. So, all the (free) variables in a term resulting from expanding definitions must come from the original term, and cannot have been introduced by the definitions.

Thus, one never has to examine the contents of definitions in order to know what the free variables or the level variables of a term are, which is key since as a result one can usually recognize instances of inference rules independently of operator definitions.

(May 2001 {62} - sfa)

Sequents (assertions in Proofs {51}) sequents_doc {50}

There is a form of assertion used in Proofs {51}, designed to facilitate reasoning about Types {15}, Functions {23}, and Propositions {34}. These are called "sequents" (after a similar form introduced by Gentzen). A sequent typically expresses some assertion under hypotheses and under declarations of variables for hypothetical values of specified types. An example is:

1. $x : \mathbb{Z}$
2. $y : \mathbb{Z}$
3. $x \neq y$

$\vdash$ $x<y \vee y<x$

Hypotheses and declarations are normally labelled in the display of a sequent for convenient reference. The place in the sequent filled by "$x<y \vee y<x$" in this case is called the conclusion of the sequent; sometimes the word "hypothesis" is used broadly to include declarations as well. Some sequent calculuses permit multiple conclusions in a sequent, but Nuprl sequents always have one conclusion. Further, all free variables in the conclusion, or in a hypothesis, or in the type of a declaration, must be declared earlier in the sequent, and no two declarations in a sequent can declare the same variable.

Please, read Functionality Sequents {37} for an introduction to sequents, which is restricted to a special form of sequent, namely, those whose conclusion is of the form "$a \in A$". Here we describe sequents more generally, and with graduated precision.

Version 1.
Roughly, the standard meaning of a Nuprl sequent is that whatever closed expressions one chooses for the declared variables of the sequent, if each resulting (purported) type-expression is actually a type including the value chosen for its declared variable, and each hypothesis is a true proposition, then the conclusion is a true proposition.

Version 2.
Ordinarily one does not need to keep the following complication in mind. It pertains to assumptions as claims about expressions respecting equality, i.e., about Functionality {23}.
The "if" part of the sequent is a little stronger than simply the type expressions being types and having the appropriate chosen values as members, and the hypotheses simply being true. What is further assumed in a sequent is that using any other equal expressions (with respect to the declared types) in place of the chosen expressions results in an equal type for each declaration, and an equal proposition for each hypothesis. Besides the "if" part of the sequent being a little stronger, so is the "then" part: the conclusion must also yield an Equal Proposition {35} (or Equal Type {17}) if different expressions are chosen for the sequent variables.

Version 3.
It is also permitted to use a type expression, say "T", as the conclusion of a sequent. In this case, the sequent is true just when there is some "witness" expression, say "???", that would make the sequent true if the conclusion were replaced by "??? $\in$ T".

This form is especially pertinent to Proof Witness Extraction {55}, and gains further significance when considering Propositions as Types {56}.

For further explanation see Sequents with Type Conclusions {53}.

(March 2001 {62} - sfa)

Proofs proof_doc {51}

Nuprl employs a "sequent style" logic for making arguments about Types {15} and Propositions {34}. The assertions at each proof step must be in the form of Sequents {50}, which usually involve variable declarations for arbitrary hypothetical values and assumptions made for the sake of argument. A proof is a tree of sequents labelled with "justifications"; the sequent at the root of a subtree is called the "goal", and the sequents of the children are called the "subgoals". The Nuprl system actually generates the subgoals from the goal and its justification, and such subgoal generation is called "refinement" since one often thinks of this step as refining a problem specification into sufficient subproblems. Such a problem is completely solved, or a proof is "complete", when there are no remaining subgoals.

A goal, a justification, and the resultant subgoals constitute a "proof step", sometimes called an "inference step"; the goal and subgoals are often called the "conclusion" and "premises" of the inference. A principle aim of the Nuprl system design is to assist in the relatively easy generation of "valid" inferences, i.e., inferences whose conclusions are true if their premises are, and yet to permit these inferences within a proof to be appropriate to a human reader, i.e., rather high-level.

Although ultimately inferences are justified by low level rules, one major style of proof is to formulate each step in a manner that seems evident, or at least plausible, to a human reader, and to make the "granularity" coarse enough so that the whole proof is not too tedious to read or comprehend as a whole. This is technically possible because of programs called "tactics".

Usually, for the reader of a proof, the most interesting thing about tactic code of a justification is which lemmas, definitions, or hypotheses it refers to, and what witnesses are provided (Note that hypotheses are often referred by number, negative numbers indicating counting backwards from the conclusion as zero). Sometimes the code is elided in favor of a user-created summary comment which is displayed instead (of course the tactic code is still there somewhere).

Tactics

In Nuprl, a justification is "tactic" code. Given a collection of prespecified "primitive" inference forms, a tactic is a program for reducing a desired proof goal to premises by composing primitive inferences. A tactic is essentially a program for constructing such an inference tree, and one chooses which tactics to apply according to how you want to generate subgoals from the goal. The execution of the tactic gives rise to an inference step, the premises being all the unproved leaf premises of the primitive proof tree. Further, to count as a tactic, although its execution might not terminate or might raise an exception, if it does terminate without exception, then it must be guaranteed to generate subgoals justifiable by primitive inferences.

In order to freely write heuristic tactics, one must have some practical criterion for recognizing tactics. In ML {52}, this criterion is type checking a program to the type "tactic".

The labor of justifying an inference is divided between ascertaining that the program generating the subgoals is a tactic, and checking the primitive inference rules, which are presumably formulated to make such verification feasible; primitive inference forms tend to be expressed schematically. In contrast, the justification of complex forms of inference via tactics is in terms of the generic criterion for being a tactic (i.e. that the result of execution determines a primitive proof tree), and is not schematically described.

See also Sequents {50}.

(Sept 2003 {62} - sfa)

The ML programming language of Nuprl whatis_ml_doc {52}

Executable code for Tactics {51} (as well as a lot of editor code) is written in a dialect of ML described in the ML manual (postscript). It is older than Standard ML. The term "ML" connotes the fact that it was a "meta-language" of computation for an earlier tactic prover at Edinburgh, namely LCF, as it is now for Nuprl. Some new primitive types, including "term" and "proof", were added.

(Feb 2001 {62} - sfa)

---

sequents2_doc {53}

Sequents with Type Conclusions - continued from Sequents {50}

We continue elaborating on the meaning of sequents.

A type expression may be used as the conclusion of a sequent, and the sequent is considered true just when there is some "witness" expression, say "???", that would make the sequent true if the conclusion, say "T", were replaced by "$??? \in T$". For example,

1. $x : \mathbb{Z}$
2. $y : \mathbb{Z}$
3. $x+1<y$

$\vdash \{z:\mathbb{Z}|\ x<z \ \&\ z<y \}$

is true because, for example,

1. $x : \mathbb{Z}$
2. $y : \mathbb{Z}$
3. $x+1<y$

$\vdash y1 \in \{z:\mathbb{Z}|\ x<z \ \&\ z<y \}$

While there is a proof rule for providing such a witness explicitly to justify such a sequent, it is common to defer or avoid explicitly supplying such witnesses. For example, one may infer

```
⊢  x:A→B(x) from

⊢  A ∈ Type, and

1. x : A  ⊢  B(x)
```

since we can construct a witness "λx.b" for the goal from a witness "b" for the last subgoal. Similarly, we can infer "⊢ A×B" from "⊢ A" and "⊢ B", since a witness <a,b> can be constructed from witnesses for the subgoals. An example of a proof that defers witness specification is

```
Thm* x:(ℤ List)→{y:ℤ| y greater-bounds x }
```

The proof proceeds by induction on lists, specifying witnesses explicitly at three key points. It provides -3 as a value on the empty list, and for the (u.v) case (under the inductive hypothesis) it specifies using either a value w assumed to work for sublist v, or u+2, depending on whether u<w.

The proof does NOT explicitly show a member of x:(ℤ List)→{y:ℤ| y greater-bounds x }. It is simply argued that such a member can be constructed.

There are three lines of further elaboration at this point, and there is some interaction between them.

Proof Witness Extraction {55} is a method for actually extracting a witness from the proof of a type-conclusioned sequent.

Sequents with Restricted Declarations {54} are introduced to allow us to assert the independence of witnesses from certain variables. (...)

Propositional Proof Witness Extraction {58} is extracting from proofs of propositions certain kinds of witnesses inherent in their constructive meaning, this being effected by representing Propositions as Types {56}

(March 2001 {62} - sfa)

Sequents with Restricted Declarations   sequents3_doc {54}
- continued from Sequents with Type Conclusions {53}

We continue elaborating on sequents having types as conclusions. A declaration may be annotated as the second one is in the following sequent:

```
1. x : ℤ
2. y : ℤ  [not for witness]
3. x+1<y
⊢  {z:ℤ| x<z & z<y }
```

The sequent above claims that there is some "witness" expression ???, NOT containing y free, such that

```
1. x : ℤ
2. y : ℤ
```

```
3. x+1<y
⊢  ??? ∈ {z:ℤ| x<z & z<y }
```

in which case the witness y1 $\in$ {z:$\mathbb{Z}$| x<z & z<y } would not suffice; however, the witness expression "x+1" would suffice since it does not use y.

Now, why do we care about restricting the use of declared variables from witnesses?
Recalling {10} that $\mathbb{N}$k (i.e. {i:$\mathbb{Z}$| 0 $\leq$ i < k }) is our standard finite set of size k $\in$ $\mathbb{N}$, consider the difference between the following two types of functions.

```
Thm* k:ℕ→x:(ℕk List)→{y:ℕ(k+1)| y greater-bounds x }
Thm* ∩k:ℕ. x:(ℕk List)→{y:ℕ(k+1)| y greater-bounds x }

where Def y greater-bounds x == ∀i:ℕ||x||. x[i]<y,
and ||x|| denotes the length of a list,
and x[i] denotes the member at position i ∈ ℕ||x|| of list x,
the head of a list being considered position 0.
```

The first is the type of functions which, applied to two arguments {26}, a number k $\in$ $\mathbb{N}$ and a list of numbers less than k, find a number no bigger than k, but bigger than every member of the list. It is of course quite trivial to find such a function-- just use k, as suggested in the proof.
The second, an Intersection Type {20}, is the type of functions of one argument, a list of any type ($\mathbb{N}$k List) for any k $\in$ $\mathbb{N}$, that return such an upper bound, but do not take k itself as an argument. Such functions must examine the list to come up with a bound.

The proofs given for the above two theorems begin similarly, producing the following subgoals respectively:

```
1. k : ℕ
2. x : ℕk List
⊢  {y:ℕ(k+1)| y greater-bounds x }

1. k : ℕ  [not for witness]
2. x : ℕk List
⊢  {y:ℕ(k+1)| y greater-bounds x }
```

the difference being simply that the upper sequent can be satisfied by witness "k", but the second requires a witness not involving "k", and indeed the proof can be seen to proceed by induction on list structure. Incidentally, note that the inference to 1. x : A List $\vdash$ B(x) from $\vdash$ B(nil) and
1. u : A 2. v : A List 3. B(v) $\vdash$ B((u.v)) is justified by constructing the witness

```
Case of x; nil → p ; u.v, rec:r → q(u; v; r)
```

from witnesses "p" and "q(u; v; r)" for the two premise sequents.

Representing Propositions as Types {56} affords
Further Uses of Restricted Declarations {58}.

(March 2001 {62} - sfa)

As explained in Sequents with Type Conclusions {53}, when the conclusion of a sequent is a type expression, the sequent is construed as claiming the existence of a "witness" expression. In the simple case of a sequent having no hypotheses or declarations, such as "⊢ T", it simply claims the type to have a member "t ∈ T". The use of these type-conclusioned sequents permits us forestall the explicit production of such witness expressions, sometimes completely, in a proof. And yet in Nuprl, when the proof is completed, one may "extract" from the proof such a witness expression.

For example, from the proof named "sfa_doc_find_intlist_great_bound" given for Thm* x:(ℤ List)→{y:ℤ| y greater-bounds x }, which claims the existence of a function finding an integer bigger than all members of an integer list, the Nuprl system extracts:

```
λx.Case of x
     nil → -3
     u.v, rec:% → (λ%1.(λ%3.InjCase(%3; %. %1; %4. u+2))
                        ((λ%3.%3(u,%1))(ext{decidable__lt})))
                 (%)
```

which is Computationally Equivalent {45}, after a a bit of simplification and variable renaming, to the expression

```
λx.Case of x
     nil → -3
     u.v, rec:w → InjCase(if u<w→ inl(*) ; inr(λa.*) fi; b. w; c. u+2)
```

The term ext{decidable__lt} was defined by the system to expand to the term extracted from the theorem named "decidable__lt", which was used as a lemma in the proof sfa_doc_find_intlist_great_bound, and expanded to λi,j. if i<j→ inl(*) ; inr(λ.*) fi. Indeed, there is also a term ext{sfa_doc_find_intlist_great_bound} which expands in one step to the proof extract, which is how I found the above extract expression for this proof in the first place.

```
Thm* ext{sfa_doc_find_intlist_great_bound}
     ∈ x:(ℤ List)→{y:ℤ| y greater-bounds x }

ext{sfa_doc_find_intlist_great_bound}([1; 4; 2; -1]) *⇒ 6

ext{sfa_doc_find_intlist_great_bound}([1; 2; -1]) *⇒ 4

ext{sfa_doc_find_intlist_great_bound}(nil) *⇒ -3
```

The reason the system is able to find such witnesses to extract is that each rule for inferring a type-conclusioned sequent stipulates how to build a witness, perhaps using witnesses from proofs of its subgoals.

Let's look at another example. In Sequents with Restricted Declarations {54} there was discussion of sequents with variables barred from use in witnesses. The extracts from the theorems contrasted there are:

```
Thm* (λk,x. k)
     =
     ext{sfa_doc_find_greater_bound_stupidly}
     ∈ k:ℕ→x:(ℕk List)→{y:ℕ(k+1)| y greater-bounds x }

Thm* (λx.Case of x
           nil → 0
           u.v, rec:r → InjCase(if u<r→ inl(*) ; inr(λ.*) fi; a. r; b. u+1))
     =
     ext{sfa_doc_find_small_greater_bound}
     ∈ (∩k:ℕ. x:(ℕk List)→{y:ℕ(k+1)| y greater-bounds x })
```

Representing Propositions as Types {56} enables
Propositional Proof Witness Extraction {58} of constructive content.

(March 2001 {62} - sfa)

Constructive type theorists attempt to account for certain key features of propositions. The basic type theoretic claim is that the "computational content" of a proposition P can be expressed as a type T, this content being what is available to the procedure that is implicit, according to constructivists, in a proof of any implication "P $\Rightarrow$?".

It should be noted that the operators we have been displaying as the logical operators are actually defined as type-valued functions. See Propositions as Types - standard defs {57}. The connection between the usual logical operators and these type operators requires explanation.

We shall focus on a few aspects of constructive mathematical language: effective notations, computational content, and existential proof witnesses.

Effective Notation

The core of computational type theory is a semantic method for notating values and types that is intended to be adequate for constructivist mathematical expression. See Types {15}.

A paradigm for constructively adequate notation is notating natural numbers. From a(n adequate) notation for a number, one can effectively determine whether it denotes zero, and if it does not, then one can effectively derive a notation for its predecessor. Similarly, from a notation for an ordered pair one must be able to derive a notation for each component.
Unless otherwise stipulated, we'll use the word "notation" to refer to an effective notation in this sense.

Notations for functions inherit criteria for effectiveness from their output types, and can exploit effectiveness of notations for their arguments. A notation for a (one-place) function is, at least in part, a way of building a notation for its result for any argument from a notation for that argument. So, from notations for a natural-number-valued function and for an argument, one can build a notation for the function's value and ascertain whether it denotes zero, et cetera.

Computational Content and Constructive Choice

Constructivists maintain that constructively understanding what a proposition means requires a conception of "computational content" of possible proofs of that proposition. Further, if one understands a proof then one can come up with its computational content. In particular, the computational content of a proof of an implication "P $\Rightarrow$ Q" is taken to be an effective method for deriving from the computational content of any proof of P the computational content of a proof of Q. (It should be noted that there is no general acceptance among constructivists that Church's thesis is correct with regard to the identity between effective procedures and recursive enumerability, even when restricted to numeric functions.)

Similarly, the computational content of a proof that "$\forall$x:A. Q(x)" is taken to be a method which from any effective notation "r" referring to a member of type A, together with any further computational content there may be in a proof that r $\in$ A, will construct the computational content of a proof of the proposition Q(r). Constructivists typically also presuppose that these methods are deterministic, making this interpretation of universal quantification a sort of choice principal.

Constructive type theory claims that the computational content of all the possible proofs of a proposition constitutes a type, and that giving a complete explanation of the meaning of a proposition requires explaining what that corresponding type of computational content is. Thus, the type of computational content for proofs of "P $\Rightarrow$ Q" is "A$\rightarrow$B" if A and B are the types of computational content for proofs of P and of Q. The computational type theory of Nuprl has no explicit logic of propositions, and is really just a logic of functions and types. The interpretation of types as the propositions whose computational content they describe is an informal one, albeit one encouraged by the practice of adopting standard propositional notations but defining them as type notations. See Propositions as Types - standard defs {57}.

Existential Proof Witnesses

From a (constructive) proof of an existential proposition "$\exists$x:A. P(x)" one can find a notation "r" referring to a member of A, and can also produce a proof of "P(r)".
Consequently, for example, since
Thm* P $\vee$ Q $\Rightarrow$ ($\exists$x:$\mathbb{N}$. (x = 0 $\Rightarrow$ P) & (x $\neq$ 0 $\Rightarrow$ Q)), then from any proof of "P $\vee$ Q" one can produce a proof of "P" or a proof of "Q". Here's how.
Use modus ponens on the above theorem to get a proof of "$\exists$x:$\mathbb{N}$. (x = 0 $\Rightarrow$ P) & (x $\neq$ 0 $\Rightarrow$ Q)" and find a witness r $\in$ $\mathbb{N}$ and a proof of "(r = 0 $\Rightarrow$ P) & (r $\neq$ 0 $\Rightarrow$ Q)". Decide whether "r" denotes zero and if so use modus ponens on "r = 0 $\Rightarrow$ P" to get a proof of "P"; otherwise, use modus ponens on "r $\neq$ 0 $\Rightarrow$ Q" to get a proof of "Q".

This example leads to the famous intuitionist rejection of the "law of the excluded middle", $\forall$A:Prop. A $\vee$ $\neg$A. It is quite implausible that one could always effectively determine whether an arbitrary proposition were true. The above argument depends critically on the requirement that the intermediate existential witness be an effective notation for a number. In a non-constructive language one might simply use the trivial witness expression "if P is true then 0, otherwise 1", but this would not count as an effective notation unless one already had a way of determining "P" as true or as false.

Thus, constructively, one may not always perform a case split on whether a proposition is true. One may do so for a proposition P when "P $\vee$ $\neg$P", which we abbreviate by Def Dec(P) == P $\vee$ $\neg$P.

Thm* Dec(P) $\iff$ ($\exists$b:$\mathbb{B}$. P $\iff$ b) [there is a coercion from $\mathbb{B}$ to Prop]

It should be noted that we do not mean by "proof" here a particular formalization of proof, such as Nuprl Proofs {51}, but the full informal notion of constructive proof, which is conceived of as open-ended.

(Sept 2003 {62} - sfa)

Some Type Constructors used to represent Propositions

The following forms of expression were mentioned in Propositions {34}.

$P \Rightarrow Q$
$P \;\&\; Q$
$P \vee Q$
False
True
$\forall x{:}A.\ P(x)$
$\exists x{:}A.\ P(x)$
$a = b \in T$
$a<b$

As discussed in Propositions as Types {56} we use types to represent propositions. The standard logical symbols are not primitive in Nuprl and we actually use them for the following types and type constructors, which approximate their constructive content. The "weak" correspondence between a type and a proposition consists of the type's having a member just when the proposition is true. The "strong" correspondence between a type and a proposition consists of the type's expressing the computational content of the proposition.

The expression "$r = s \in T$" is a primitive that is defined to be a type given that $r \in T$ and $s \in T$. It has a member "*" iff r and s are equal expressions of T. Further,

Thm* $\forall r,s{:}T,\ x{:}(r = s \in T).\ x = * \in (r = s \in T)$

The expression "$a<b$" is treated the same way for $a \in \mathbb{Z}$, $b \in \mathbb{Z}$.

Def Prop == Type

Def $P \Rightarrow Q$ == $P \rightarrow Q$
Def $P \;\&\; Q$ == $P \times Q$
Def $P \vee Q$ == $P+Q$
Def False == Void
Def True == $0 \in \mathbb{Z}$
Def $\forall x{:}A.\ B(x)$ == $x{:}A \rightarrow B(x)$
Def $\exists x{:}A.\ B(x)$ == $x{:}A \times B(x)$

A member of "$\forall x{:}\mathbb{N}.\ \exists y{:}\mathbb{N}.\ P(x;y)$" is some function in "$x{:}\mathbb{N} \rightarrow y{:}\mathbb{N} \times P(x;y)$".

(Sept 2001 {62} - sfa)

proof_extraction2_doc {58}

Extracting Witnesses from Proofs {51} of Propositions {34}

As is pointed out in Propositions as Types {56}, the notations used in Nuprl for propositions actually denote types which are inhabited just when the corresponding propositions are true. See Propositions as Types - standard defs {57}. Indeed, Def Prop == Type. Thus, all Nuprl Sequents {50} are Sequents with Type Conclusions {53}, hypotheses of sequents are declarations, and Proof Witness Extraction {55} can be effected for all sequents.

For example, because "∃x:ℤ. x·x = 4" expands by definition to "x:ℤ×x·x = 4" from a proof of it one can extract an expression containing a witness (see Propositions as Types {56}). Here's what we get from this particular proof, named sfa_doc_simple_existential_1 incidentally:

```
ext{sfa_doc_simple_existential_1} *⇒ <-2,*>
```

The first component -2 is the witness, and the second component "*" is the trivial member of any Equality Type {57} when the equality actually holds. The equality type is empty when it doesn't hold, so one could not extract from any proof of ∃x:ℤ. x·x = 4 (i.e. x:ℤ×x·x = 4) a pair where the square of the first component was NOT 4.

Consider a more complex example. The expression "∃x,y:ℤ. x<y & (∃z:ℤ. z<x ∨ y<z)" expands by definition to the type expression "x:ℤ×y:ℤ×(x<y)×(z:ℤ×(z<x)+(y<z))". Here is an extract from a proof sfa_doc_simple_existential_2 of this proposition:

```
ext{sfa_doc_simple_existential_2} *⇒ <4,6,*,(-1),inl(*)>
```

Like equality types, a Less-than Type {57} is defined as a primitive type that has a (trivial) member just if the relation holds between the specified integers. Observe that the last component of the tuple, inl(*) ∈ (-1<4)+(6<-1) informs us that the left disjunct that is inhabited by *.

Now we move to a more interesting case. Consider Thm* ∀x:ℕ. ∃y:ℕ. y·y≤x & x<(y+1)·(y+1), which claims we can find a certain integer approximation of the square root of any natural number. This expression expands to "x:ℕ→y:ℕ×((x<y·y)→Void)×(x<(y+1)·(y+1))", expressing the part of constructive content to the effect that there is a function for producing the witness y for each instance x of the universal. Thus, the extract of this theorem, ext{sfa_doc_nat_sqrt}, must be a function producing for each input a pair with a witness as the first component.

```
ext{sfa_doc_nat_sqrt}
*⇒
λx.ind(x; zzj,%.λ%2.*; λ%.<0,(λ.*),*>; zzj,%.λ%2.%(λ.*)/y,%3.
   %3/%4,%5.
   InjCase(if zzj<((y+1)·(y+1))→ inl(*) ; inr(λ.*) fi
         ; %. <y,(λ.*),*>
         ; %7. <(y+1),(λ.*),*>))
   (λ.*)
```

Pretty ugly, but automatically extracted, and applicable:

```
ext{sfa_doc_nat_sqrt}(5) *⇒ <2,(λ.*),*>.
```

So, 2 approximates the square root of 5. Let's evaluate this function for several values, mapping over a list, and pairing each input with the witness gotten by projecting off the result of applying ext{sfa_doc_nat_sqrt}.

```
Def map(f;as) == Case of as; nil → nil ; a.as' → f(a).map(f;as')  (recursive)

   map(λx.<x,ext{sfa_doc_nat_sqrt}(x).1>;[0; 1; 2; 4; 5; 7; 9; 11])
   *⇒
   [<0,0>; <1,1>; <2,1>; <4,2>; <5,2>; <7,2>; <9,3>; <11,3>]
```

(Sept 2001 {62} - sfa)

As was mentioned in Polymorphism {25}, we have defined

Def S ⊆ T == ∀x:S. x ∈ T

with the consquence that S ⊆ T does not express a general relation between types (even restricted to a Universe {18}). This is because, as explained in Equality {19}, "t ∈ T" is assigned a meaning only when it is true, and therefore also S ⊆ T is assigned a meaning only when it is true. Thus, it is not true that "∀S,T:Type. S ⊆ T ∈ Prop", and so one cannot effectively use S ⊆ T as an assumption is propositions.

This is quite unfortunate since there are many natural properties we would like to state about a subtyping relation between types, such as

∀S,T:Type. S ⊆ T ⇒ (S List) ⊆ (T List)

∀A,A',B,B':Type. A ⊆ A' ⇒ B ⊆ B' ⇒ (A'→B) ⊆ (A→B')

but these are not even well-formed propositions.
Similarly, there are properties about extensional equality {16} between types that we would like to express, and we would be able to once we defined an extensional equality relation in Nuprl as symmetric closure of subtyping.
Then we would be able to express properly theorems like

∀S,T:Type. S =ext T ⇒ S List =ext T List

∀S:Type, x:S, T:Type. S =ext T ⇒ x ∈ T

which would be quite handy as lemmas.

A better approach, which we intend to adopt later, is to add a new primitive operator that is stipulated to be well formed whenever its two arguments are types, and is true just when the one type is in fact a subtype of the other, i.e., when every pair of terms equal in the first type are also equal in the second.

Note that these two concepts can technically coexist, so extant theorems using the old form of subtyping would survive. Karl Crary has worked out a suitable set of rules for the new subtype relation which we need to incorporate into the standard tactics {51}.

(August 2001 {62} - sfa)

```
1. x : ℤ                                              sfa_doc_decomp_example {60}
2. y : ℤ
⊢  x+x(y+2) ∈ ℤ  by Analyze
|
|\
| ⊢  x+x ∈ ℤ  by Analyze
| |
| |\
| | ⊢  x ∈ ℤ  by Trivial
| |
| .\
| . ⊢  x ∈ ℤ  by Trivial
|
.\
. ⊢  y+2 ∈ ℤ  by Analyze
. |
. |\
. | ⊢  y ∈ ℤ  by Trivial
. |
. .\
. . ⊢  2 ∈ ℤ  by Auto
```

```
⊢  ∀T:Type. Y ∈ (∩k:ℕ. (ℕk→T)→ℕ(k+1)→T)→ℕ→T  by Analyze ...w
|
1. T : Type
⊢  Y ∈ (∩k:ℕ. (ℕk→T)→ℕ(k+1)→T)→ℕ→T  by New:F FunExtensionality
|
|\
| .....wf .....
| ⊢  Y ∈ Void→Void  by Def ....
|
.\
. 2. F : ∩k:ℕ. (ℕk→T)→ℕ(k+1)→T
. ⊢  Y(F) ∈ ℕ→T  by ∀n:ℕ. Y(F) ∈ ℕn→T  Asserted
. |
. |\
. | .....assertion .....
. | ⊢  ∀n:ℕ. Y(F) ∈ ℕn→T  by NatInd Concl
. | |
. | |\
. | | .....basecase .....
. | | ⊢  Y(F) ∈ ℕ0→T  by <TACTIC>
. | |
. | .\
. | . .....upcase .....
. | . 3. n : ℤ
. | . 4. 0<n
. | . 5. Y(F) ∈ ℕ(n1)→T
. | . ⊢  Y(F) ∈ ℕn→T  by FunExtensionality
. | . |
. | . 6. x : ℕn
. | . ⊢  Y(F,x) ∈ T  by Compute Y(F) *⇒ F(Y(F))
. | . |
. | . ⊢  F(Y(F),x) ∈ T
. | . | by F ∈ (∩k:ℕ. (ℕk→T)→ℕ(k+1)→T)  Asserted THEN Witness-1: n1 ...w
. | . |
. | . 7. F ∈ (∩k:ℕ. (ℕk→T)→ℕ(k+1)→T)
. | . 8. F ∈ (ℕ(n1)→T)→ℕ(n1+1)→T
. | . ⊢" F(Y(F),x) ∈ T  by Auto
. |
. .\
. . 3. ∀n:ℕ. Y(F) ∈ ℕn→T
. . ⊢" Y(F) ∈ ℕ→T
. . | by With ⌈ℕ0→T⌉ FunExtensionality THENW (BackThru: Hyp:3 ...w)
. . |
. . 4. x : ℕ
. . ⊢  Y(F,x) ∈ T  by Witness3: x+1 ....
```

primitive_op_doc_HISTORY {62}

These documents {1} were written by sfa as a supplement to the Nuprl Library. When reading the library in progressively greater detail, one rather quickly hits upon Nuprl primitives and expressive methods that should be explained. These documents were generated by documenting the primitive type constructors, which engendered further descriptions progressively down to type theory. The design is to forestall explanation of essentially type-theoretic issues as long as possible, to emphasize the higher level ordinary expression of mathematics.

September 2003

Added to Proofs {51} a remark about negative hyp number references.
Added some paraphrases for the standard propositional ops in Propositions {34}.
Reworked Propositions as Types {56} to incorporate a brief treatment of computational content and dropped its references to the merely planned pages about it; also adjusted Propositions as Types - standard defs {57}, Propositions {34}, and Propositional Proof Witness Extraction {58} similarly.

August 2003

Began reformatting documents to make the more suitable as printable pages.

January 2003

Added a document Subtyping Relation about advance adoption of the subtype relation primitive.
Added a new extensional fixedpoint theorem
Thm* $\forall$A:Type. Sexpr(A) =ext (Sexpr(A)$\times$Sexpr(A))+A
to Recursive Types {33} (a note from Kreitz prompted this).

November 2002

Added to Operator Definition {47} indication that function applications, "<fun>(<arg>)" can be used on the lhs of an opdef instead of lambdas, "$\boldsymbol{\lambda}$[var].<*>", on the rhs.
Added a new theorem about polymorphic functions
Thm* $\forall$f:($\cap$D:Type. D$\rightarrow$T), A:Type, a:A, B:Type, b:B. f(a) = f(b) $\in$ T
to Polymorphism {25} (Barzilay suggested adding this).

December 2001

Corrected an erroneous inl(b) to inr(b) in Computation {42}.
Indicated rec(X.(X$\rightarrow\mathbb{N}$)$\rightarrow\mathbb{N}$) as an example failure of type formation in Recursive Types {33}.

October 2001

Added a reference to Thm* whatever1 = whatever2 $\in$ ($\cap$x:Void. Whatever(x)) in Intersection Types {20}.
Added a mention of Dec(P) to Propositions as Types {56}.
Added Thm* Dec(P) $\iff$ ($\exists$b:$\mathbb{B}$. P $\iff$ b) to $\rightarrow$primitive_ops_emblem_thms {3} for Props as Types {56}.

September 2001

Corrected some small errors in Polymorphism {25}, and added more thms as examples.
Added a thm citation to Pairs {4}.
Added some thm citations to Lists {5}.
Added a thm citation and some material about outl,inl,isl to Disjoint Union {13}.
Added a thm citation to Intersection Types {20}
Added a thm citation to Quotient Types {21}.
Added some thm citations about extensionalty to Function Types {24}.
Added Typing Y {30} and Typing Y Continued {31} about how to type the Y combinator if we extend the Function Extensionality rule.
Added →primitive_ops_emblem_thms {3} (partial index to documents by emblamatic theorem).

August 2001

Moved the reference to Void {22} in Nuprl Basics {1}.
Added much more material to Void {22}.
Added Polymorphism {25}.
Added references to Top to Unit {8}, Intersection Types {20}, Quotient Types {21}.
Added Subtype Relation {59}.
Added reference to Subtype Relation {59} to Type Expressions {16}.
Added "Squiggle" {46} and linked to it from Types {15}.

May 2001

In Boolean {6} added a demonstration of where implicit $\mathbb{B}$-to-Prop coercions occurred in Thm* $\forall$p,q:$\mathbb{B}$. (p$\wedge_2$q) $\iff$ p & q.
In Prop Levels and Predicativity {39} changed "predicative logic" to "predicative semantics".
I changed the suggested reading order of Substitution {48} and Op Def Restrictions {49} in Nuprl Basics {1}.
In Op Def Restrictions {49}, I eliminated a true but distracting parenthetical comment about the left hand side not being too general. I also glossed "second-order variable {48}" as a link.
Added a caveat to Functions {23} to not identify computational evaluation with an expressions standing for a value.
Added a missing condition to Op Def Restrictions {49}, namely "Distinct subterm positions of the left-hand side must be filled by distinct variables."
Corrected Universes {18} to use $\mathbb{N}^+$ as domain of V(i) instead of $\mathbb{N}$.

April 2001

Added Propositional Proof Witness Extraction {58}.

Feb-March 2001 {63}

CONTINUED from →primitive_op_doc_HISTORY {62} primitive_op_doc_HISTORY1 {63}

March 2001

Completed Proof Witness Extraction {55}.
Added Sequents with Type Conclusions {53},
Sequents with Restricted Declarations {54}.
Reworked Sequents {50} a bit punting to Sequents with Type Conclusions {53}.
Mentioned polymorphism and gave an example in Intersection Types {20}
Added Propositions as Types - standard defs {57} which really needs support from further explanations.
Alluded to Comprehension Subtypes {14} and Quotient Types {21} in Types {15}.
Described equality between quotient type expressions in Quotient Types {21} and alluded to it in Intensional Types {17}.
Reordered entries in VIEW→ primitive_op_doc_INDEX {1} to make it closer to order of understanding.
Made some of the structure of the Operator Definition {47} document explicit.
Improved the wording in Evaluation and Term Rewriting {45} about marking subterms for rewriting.
In NonCanonical Forms {44}, removed the ⌈<term>⌉ marks, and added mention of principal argument place.
In Terms {41}, added an example of 2-place snd-o var instance, and deleted line about creation of new expressions.
Reworded Russell's Orders {40} to make it clear that a sequence of orders was being described.
Eliminated from Equality {19} references to props as types.
Added some detailed reminders to previous info Type Expressions {16}.
Added mention of typing Kleene's mu operator to Recursive Functions {27}.
Some subtypes of $\mathbb{Z}$ were mentioned in Integers {10}.

Feb 2001

The bulk of the documentation, down to but excluding computational content of propositions and choice principles, written in February 2001. Citations are to be added later. Here are the documents then considered complete:

Pairs {4}	Boolean {6}
Atoms {12}	Negative Literals {11}
Disjoint Union {13}	Deriving Bool {7}
Lists {5}	Unit {8}
Void {22}	Deriving Unit {9}
Integers {10}	Recursive Functions {27}
Quotient Types {21}	Recursion via the Y-combinator {28}
Comprehension Subtypes {14}	Y-combinator Continued {29}
Equality {19}	Multi-argument Functions with Y {32}
Intersection Types {20}	Recursive Types {33}
Guarded Types {36}	Terms {41}
Types {15}	Operator Definition {47}
Intensional Types {17}	Op Def Restrictions {49}
Computation {42}	Substitution {48}
Evaluation and Term Rewriting {45}	Canonical Forms {43}
Universes {18}	NonCanonical Forms {44}
Propositions {34}	Type Expressions {16}
Functions {23}	Proofs {51}

Function Types {24}
Multi-Argument Functions {26}
Functionality Sequents {37}
Sequents {50}
Type Functionality Sequents {38}
Propositions as Types {56}
Prop Levels and Predicativity {39}
Higher-Order Propositions {35}
ML {52}
Russell's Orders {40}

The incomplete documents these refer to are Choice(incomplete) and Computational Content(incomplete).