The Nearly Ultimate PEARL*

Robert L. Constable
Joseph L. Bates

TR 83-551
January 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

*Research supported in part by NSF grants MCS81-04018, MCS80-03349.

THE NEARLY ULTIMATE PEARL

R.L. Constable, J.L. Bates
Cornell University
Department of Computer Science
Ithaca, N.Y. 14853

Abstract

The PRL ("pearl") system is an environment providing computer assistance in the construction of formal proofs and programs in a particular formal theory, called the *object theory*. Certain proofs can in fact be considered as programs. The system embodies knowledge about programs in the form of rules of inference and in the form of facts stored in its library. Ultimately PRL may be regarded as an *intelligent system* for formal constructive problem solving in a large domain of mathematics.

The PRL system is evolving in stages. Since our report "The Definition of Micro-PRL" in October 1981, we have had the experience of designing, building and using a complete core version of the system (called λPRL).† We have also studied more deeply the theoretical issues raised in that report. We are now prepared to extend the core system closer to the "ultimate" PRL system envisioned earlier. This document describes the mathematical *theory of types* which is the object theory of that extension (called λPRL) and it is more or less self-contained. The type theory is defined in stages, starting from a

---

†The existing λPRL system consists of 33,000 lines of Franz Lisp code running under Berkeley 4.1c Unix on a VAX 780 plus a few thousand lines (3) of C code and about 15,000 lines of Franz Lisp to support ML. The implementation was accomplished principally by J.L. Bates, F. Corrado, J. Sasaki, J. Cremer with assistance from R. Harper, S. Allen, R. Stansifer, R. Constable and A. Demers. Certain decision procedures were taken from PL/CV2 [8] and from G. Nelson's thesis [17]. Improvements and extensions were made by M. Bromley and T. Knoblock based on the initial experience of J.L. Bates, R. Constable and N.P. Mendler using the system.

‡Research supported in part by NSF grants MCS81-04018, MCS80-03349.

constructive theory of integers and lists ("PRL). The development is a main feature of the paper.

Key Words and Phrases: automated logic, constructive mathematics, lambda calculus, programming logic, program verification.

CR Categories: F.3.1 [Logics of programs]
F.3.2 [Semantics]
F.4.1 [Computational logic]

General Terms: Languages, Theory, Verification

THE NEARLY ULTIMATE PRL

I. Introduction

1. Refinement Logics

The acronym PRL, for Program Refinement Logic or Proof Refinement Logic, denotes a computer system to help users derive constructive proofs of assertions. These assertions are of an extremely general sort (they are characterized precisely in section II.1). The derivations can be stored in a library, and certain forms of proof can be executed, in particular proofs of assertions of the following form: "for all x of type A we can find a y of type B satisfying Q." These proofs define functions from type A into B, and it is natural to think of them as *programs*. Thus PRL is a program development system. PRL is also an example of a simple *intelligent system* because it embodies knowledge of program structure and can accumulate knowledge about programming and mathematics in its library.

PRL encourages proof development in a refinement style - a goal is decomposed into subgoals. The proof is refined layer by layer; thus the name, "refinement logic" coined in [2]. Nevertheless, it is inevitable that other kinds of proof are possible, especially those built by assembling results from the library.

2. Programs As Constructive Proofs

Essential to the conception that *programs are executable proofs* is the notion that proofs are constructive. In PRL this is assured because the underlying language of the system has a computational semantics. Nonconstructive modes of thought can be represented in the language, but the *defining semantics* is computational. Indeed the resulting language is sufficiently expressive that virtually all sequential programming problems can be described in it. Those with nonconstructive proofs have no interesting execution.

To formulate a computational semantics for such an expressive language, we rely on the principles of constructive mathematics as they are formulated in the context of the predicate calculus. At a formal level this amounts to imposing Heyting's interpretation [22] of Brouwer's semantics [8] in Frege's predicate calculus [17]. In sections IV and V we will provide another semantic explanation of the system based on Per Martin-Löf's theory of types [26,27], but initially we will explain our concepts in terms of the predicate calculus and a constructive version of Frege's semantics.

3. Semantics of Constructive Logic

According to Brouwer [8], a *proposition* is specified by telling how to prove it. Thus propositions have proofs not truth values. The meaning of compound propositions is given in terms of rules for proving them from proofs of the components. We list the forms of compound propositions below with their intuitive meanings. This serves to introduce our logical notation. We assume that the concept of a term, in particular a variable x, occurring in a formula is clear. (It is in fact defined in II.1.) To know A&B we must know A and know B, so a proof of A&B will be a pair consisting of a proof of A and a proof of B. To know A|B (read as "A or B") we must either know A or know B. To know A⇒B we must provide a method which transforms a proof of A into a proof of B. To know some x:A.B we must produce an object a of type A and a proof that B holds with a substituted for x in B, written as B(a/x). Finally, to know all x:A.B we must have a method of producing for any object a of type A a proof of B(a/x).

The rules of logic given in section II, based on Heyting's [22] analysis of Brouwer, can be seen to be correct under this computational interpretation.

We will in section IV provide a so called "extract-form" of the rules which makes the computational meaning explicit. These extract-forms show for example how to build from a proof of A⇒B a computable function λx.b which takes proofs of A to proofs of B. In section IV the computational meaning of formulas is made explicit in the type structure of the logic using the so called propositions-as-types principle.

4. Type Structure

The λPRL logic [4] is built around exactly two types, int, the integers, and list, lists of integers. These types are not adequate to express all the computational problems of interest to us. Indeed we strive to express all the types needed to do computational mathematics. In particular, we need these type constructors from modern programming languages (such as Algol 68 [38] and ML [19]): Cartesian product of two types A and B, written A#B, disjoint union, written as A|B, the list of objects from A, written A list, and the function space (or exponentiation), written A→B. In the case of A→B we mean the type of computable total functions from A to B.

In addition we want the following types common to mathematics, infinite union, (or dependent product) written as x:A#B and infinite product (or dependent function space) written as x:A→B, and the set of all A such that B, written {x:A|B}. (We can in fact define #, ⇒ and | from these infinitary operators.)

We want to treat types as objects of computation, passing them as inputs to functions as a means of permitting polymorphic operations. To this end, we would like to treat the concept of type itself as a type. However, without restriction such a concept is paradoxical. So following Bertrand Russell [33], we introduce a layered concept of type, written type(i). This induces a layering structure on propositions as well. Propositions of level i are in the class prop(i).

With these basic types we can define various notions of recursive data type, so they are not explicitly added to the underlying type structure. We could indeed define the list constructor in this way.

With each type A we introduce a proposition defining equality on A, written $=_A$. In order for the user to provide explicitly defined equivalence relations on a type A, we provide the concept of a quotient type, written A//E where A is a type and E an equivalence relation. On this type the defined equality is the binary relation E.

5. Formulas and Judgements

The atomic formulas of this theory are the equalities $a =_A b$ where A is a type and the assertions $a < b$. In order for equality formulas to be well-formed, it must be that A is a type and that a and b are of type A. Because the theory allows the computation of types and because terms do not carry their types, there is no algorithm to decide whether these type constraints are met. Therefore, it is necessary to provide proofs which establish the well-formedness of types and formulas. The question then is how to express these constraints.

The fact that a is of type A is expressed by the type judgement $a \in A$. For the present we do not take this to be an assertion that a belongs to a type, but it is a form of judgement which must precede assertions such as $a =_A b$. Later in section IV.7 we shall see how to reduce this judgement to an assertion.

In order to make the type judgement $a \in A$, we must also be able to judge that A is a type, say of level i, which we write as $A \in \text{type}(i)$. (This is a _higher type_ judgement.)

To judge that an atomic formula $a =_A b$ is well-formed written $(a =_A b) \in \text{prop}(i)$, we need to judge

$$A \in \text{type}(i)$$
$$a \in A$$
$$b \in B$$

It is also necessary that we know that compound formulas are well-formed. For example to prove (A|B) we must either prove A or B, and we must know that each component is well-formed.† For compound formulas we also use the notation $A \in \text{prop}(i)$ to mean that _A is a proposition_ whose type components are of level i. This is another form of judgement, say a _formation judgement_. For example, we make the judgement that (some x:A.B) $\in$ prop(i) provided that $A \in \text{type}(i)$ and $B \in \text{prop}(i)$ under the condition that x is of type A. To show the bindings of variables, we use _declarations_ of the form x:A. The collection of these declarations forms an environment (much as in λPRL).

6. Deductive Apparatus

Since Frege [17] it has been common to organize the deductive structure of a logic around _assertions_ that a formula A is provable. Frege wrote $\vdash A$ to mean that A is true or provable. Assertion is another form of judgement. For typographical and video display reasons we write this judgement as >> A. We are also interested in _hypothetical assertions_, these are denoted by syntactic objects called (after Gentzen [18]) _sequents_. They have the shape $A_1,\ldots,A_n$

---

†One can consider a logic without the property that all subformulas of a well-formed formula are well-formed. Such a logic has certain advantages, but we have not pursued them in this report.

>> A where A_i are called hypotheses. They are either formulas or declarations. This sequent means that the judgement A is made from $A_1,\ldots,A_n$. In other words, given proofs of the propositions among $A_1,\ldots,A_n$ we claim that we can build a proof of A. We allow more generally hypothetical judgements $A_1,\ldots,A_n$ >> J where J is any "epsilon" form of judgement, e.g. $t \in T$.

The pattern of deduction in PRL is that the user states a goal, a formula or judgement to be proved under certain assumptions and in a certain environment, and then issues a command to the system telling how the proof should proceed. The command is either a primitive rule or an entire proof method (see section VI). The system responds by decomposing the goal into subgoals based on this command. The user then considers the subgoals, proceeding iteratively until the proof attempt succeeds or is abandoned.

To accommodate this top-down form of reasoning, the rules are presented in the refinement style [2] shown below where H, the hypothesis list, is a sequence of formulas and declarations, and T, the conclusion, is a formula or a judgement.

$$
\begin{array}{l}
H \gg T \\
\quad H_1 \gg T_1 \\
\qquad \vdots \\
\quad H_n \gg T_n
\end{array}
$$

This pattern means that to know the sequent H >> T, called the goal, it is sufficient to know all of the $H_i \gg T_i$, the subgoals. If a rule has no subgoals (n=0) as in H >> $0 \in$ int, then it is axiomatic.

The rules will display level tags which are positive integers or letters written to the right of the conclusion, as in H >> A&B i where i is a level letter. These tags are used in the treatment of higher level and higher order logic (sections II.4, II.7). In a type judgement a ϵ A i the tag indicates that A ϵ type(i) is being claimed, and in an assertion A&B i the tag indicates that A&B ϵ prop(i) is being claimed. The formulas and declarations of the hypothesis list all have levels associated with them as well, but they are not displayed in the rule, instead they are mentioned in provisos as necessary. The level tags can be ignored for the λPRL subtheory.

This top down approach to proof is also used to establish that a formula is well-formed, and the process of showing well-formedness can proceed simultaneously with the proof of truth. This means that the user can state goals which are not *a priori* known to be well-formed. These goals need only be *readable*, that is they must satisfy certain context free constraints which allow them to be decomposed based on an abstract syntactic structure.

The logic is designed to guarantee that if a goal A is provable, then we know that it is in fact well-formed. (See [21] for a proof of this fact.)

The top-down checking of well-formedness is applied not only to formulas but to terms of the logic as well.

7. *Organization of the Rules*

First we present all the rules of the λPRL core logic which include the predicate calculus, equality, arithmetic, and list theory. These rules include level numbers and well-formedness conditions (marked with *), which were not needed in λPRL, but otherwise they are the same as the λPRL version. It would be sensible to include here the rules for void and atoms, but they are not in fact types of λPRL so they are introduced later.

Next we present rules for the new type constructors. It is significant that these rules can also be presented in the introduction and elimination format. In addition to these two forms, there are also rules for *type formation* and rules for equality of objects in types as well as equality among types.

The presentation is organized to introduce features of the logic in order of their similarity to the well-known concepts from first order logic, say [24] and set theory. So we start with the predicate calculus over the types integer and list of integers. Then we introduce higher types such as cartesian products, disjoint unions and functions spaces. Then sets and quotients appear. Then we consider higher order logic and the notion of universes. In section III we show how the logic can be simplified and yet made more powerful by adopting the propositions-as-types principle. We also show how to reduce all the diverse forms of judgement either to assertion or to type judgements. This leads to two compact versions of the logic from which the highly irredundant version given first can be derived - by inverting the reduction steps.

II. LOGICAL RULES

1. *Syntactic Issues*

The rules that follow will define the concept of a *well-formed formula*, a *well-formed expression*, and the concept of a *provable formula*. As a preliminary notion we need the concept of a *readable formula* and a *readable expression*. These are character strings which have the surface structure of a formula or an expression. They are defined by a simple context free grammar given here. The words "form", "exp", and "id" are the names of syntactic categories.

```
form ::= exp =_exp exp
         exp < exp
         all id:exp.form
         some id:exp.form
         ¬ form
         form & form                 R  means & is right associative
         form | form                 R
         form ⇒ form                 R
         (form)


 exp ::= void
         f(exp)                      f = any, hd, tl, lin, rin, FIX, DOM, Dom
         exp(exp)                    L, application is left associative
         atoms
         'character list'
         int
         ±n                          n a decimal numeral
         n
         exp op exp                  L  op = +,-,*,/
         ind exp do <n_1→id,id.exp,n_1→exp,...,>n_p→id,id.exp od
         exp list
         exp.exp                     R
         lind exp do 0→exp,...,n→id,...,id.exp od
      id:exp#exp                     R
         exp#exp                     R
         <exp,exp>
         $exp over id,id.exp$
      id:exp→exp                     R
         exp→exp                     R
         λid.exp
         exp|exp                     R
         if exp left id.exp right id.exp fi
         exp//exp                    L  means // is left associative
         {id:exp|form}
         type(id)
         prop(id)
         type(n)
         prop(n)
      least id >= exp where exp =_int exp
      well id:exp.exp
      sup(exp,exp)
      tind exp do id,id,id.exp od
```

The precedence of operators is as follows - the unary operators come first, then application, then

//	L
#	R
|	R
→	R
*, /	L
+, -	L
•	R

In {x:A|B|C} the precedence yields {x:A|(B|C)}.

In the following constructs id is a _binding occurrence_ of a variable and the operator name is a _binding operator_. The exp term or the form term is the _scope_ of the operator. An occurrence of id in the scope is _bound_. It is bound by the innermost binding operator in whose scope it lies.

construct	_operator_
id:T→exp	function space
id:T#exp	product
λid.exp	abstraction
{id:T|form}	set formation
least id>=t.exp $=_{int}$ s	least number operator
all id:T.form	all quantifier
some id:T.form	some quantifier
well id:T.exp	well-formed trees

In addition in the constructs ind_do_od, lind_do_od, tind_do_od if_left_right_fi, $_over_$ the subexpressions of the form id,...,id.exp indicate binding of the id's in exp. These are also binding occurrences and exp is the scope.

An occurrence of an identifier, say x, in a formula is considered _free_ iff it is not bound. A term t is said to be _free for a free identifier x in a term or formula F_ iff no free identifier of t becomes bound in F(t/x).

2. The Form of Sequents

The rules will require a mechanism for indicating the types of free variables. Just as in [4] we use <u>declarations</u> of the form x:A to indicate that x is a variable of type A. In [4] the list of such declarations is called the <u>environment</u>. We write these declarations in the hypothesis list of a sequent. We also agree to label the assumption formulas in a hypothesis list so that we can refer to them by label (another more critical reason will appear in section IV.3). If F is a formula we write x:F as a labeled formula.

We consider sequents of the form

$$x_1{:}A_1,\ x_2{:}A_2,\ \cdot\ \cdot\ \cdot\ , x_n{:}A_n >> A$$

where the A_i are readable formulas or expressions. If A_i has a free occurrence of variable x, then if $x{:}A_j$ occurs to the left of A_i we say that A_j is the type of x.

In the rules below we assume that H is a list of readable declarations and labeled formulas and that A,B,T are readable. We use x,y,z as variables (identifiers).

These rules will make sense only when the assumptions are sensible. Assumptions are <u>sensible</u> when they are readable and in addition, when all free variables of an expression A_i are declared before (to the left of) the expression, say $x{:}A_j$, $j<i$ and there is a deduction of $A_j \in$ type(k) for some k from the hypotheses to the left of $x{:}A_j$. Any complete deduction starting from no assumptions is guaranteed to generate only sensible assumptions.

3. λPRL Logic (First Order Arithmetic)

PREDICATE CALCULUS

OBJECT FORMATION (TERMS)

INTEGERS

TYPE

T1. H >> int ∈ type(i)

CONSTANTS

T2. H >> n ∈ int
where n is a signed decimal constant

INTEGER DECLARATION (VARIABLES)

T3. H, x:int, H' >> x ∈ int

T4-T7. H >> a op b ∈ int op = +,*,-,/
H >> a ∈ int
H >> b ∈ int

H >> ¬b=0 if op=/

LISTS

TYPE

T8. H >> A list ∈ type(i)
H >> A ∈ type(i)

CONSTANTS

T9. H >> $[a_1,\ldots,a_n]$ ∈ A list
H >> a_i ∈ A or n=0

LIST DECLARATIONS

T10. H, x:A list, H' >> x ∈ A list

OPERATIONS

T11. H >> hd(a) ∈ A
H >> a ∈ A list

T12. H >> tl(a) ∈ A list
H >> a ∈ A list

T13. H >> a.b ∈/A list
H >> a ∈ A
H >> b ∈ A list

PROPOSITION FORMATION (FORMULAS)

ATOMIC

F1. H >> false ∈ prop(i)

F2. H >> a $=_A$ b ∈ prop(i)
H >> A ∈ type(i)
H >> a ∈ A
H >> b ∈ A

```
F3.  H >> a<b ∈ prop(i)
       H >> a ∈ int
       H >> b ∈ int
```

COMPOUND

```
F4-F5.  H >> A op B ∈ prop(i)        op = &, |
          H >> A ∈ prop(i)
          H >> B ∈ prop(i)

F6.     H >> A ⇒ B ∈ prop(i)
          H >> A ∈ prop(i)
          H, A >> B ∈ prop(i)

                some
F7-F8.  H >> { all } x:A.B ∈ prop(i)

          H >> A ∈ type(i)
          H, x:A >> B ∈ prop(i)
```

LOGICAL RULES

INTRODUCTION / ELIMINATION

AND

```
L1. H >> A&B i by intro
      H >> A i
      H >> B i

L2. H, x:(A&B), H' >> T  i by elim x
      H, x1:A, x2:B, H' >> T i
```

OR

```
L3. H >> (A|B) i by intro 1
      H >> A i
    * H >> B ∈ prop(i)

    H >> (A|B) i by intro 2
      H >> B i
    * H >> A ∈ prop(i)

L4. H, x:(A|B), H' >> T i by elim x
      H, x:A, H' >> T i
      H, y:B, H' >> T i
```

IMPLIES

```
L5. H >> (A⇒B) i by intro
      H, x:A >> B
    * H >> A ∈ prop(i)
      (x:A of level i)

L6. H, f:(A⇒B), H' >> T i by elim f
      H, f:(A⇒B), H' >> A k
      H, f:(A⇒B), H', y:B >> T i
      where f:(A⇒B) and y:B
                  are of level k.
```

ALL

L7. $H >> all\ x{:}A.B\ i$ by intro
$H, x{:}A >> B\ i$
* $H >> A \in type(i)$

L8. $H, f{:}(all\ x{:}A.B), H' >> T\ i$ by elim a
$H, f{:}(all\ x{:}A.B), H' >> a \in A\ k$
$H, f{:}(all\ x{:}A.B), H', y{:}B(a/x) >> T\ i$
where $f{:}(all\ x{:}A.B)$ and $y{:}B(a/x)$ are of level k.

SOME

L9. $H >> some\ x{:}A.B\ i$ by intro a
$H >> a \in A\ i$
$H >> B(a/x)\ i$
* $H, x{:}A >> B \in prop$
($x{:}A$ of level i)

L10. $H, p{:}(some\ x{:}A.B), H' >> T\ i$ by elim p
$H, x{:}A, y{:}B, H' >> T\ i$

HYPOTHESIS

L11. $H, x{:}A, H' >> A\ i$ by hyp x
provided $x{:}A$ is of level j, $j \geq i$

FALSE (ELIMINATION)

L12. $H, z{:}false, H' >> T\ i$ by elim z

CONSEQUENCE

L13. $H >> T\ i$ by seq A
$H >> A\ j$ new
$H, A >> T\ i$

In all elimination rules, if the formula on which elimination is performed, say $A|B$, is of level k, then the subterms in the subgoals, say A or B, are also of level k. In rules such as L5, L9, when subexpressions of the goal, say $A \Rightarrow B$, are added to the hypothesis list, say $x{:}A$, they are assigned the level of the goal, say i.

Equality

E1. $H >> a =_A b\ i$ by sym
$H >> b =_A a\ i$

$a =_A b\ \ i$ by sym
$H >> b =_A a\ \ i$

E2. $H >> a =_A b\ \ i$ by trans on c
$H >> a =_A c\ \ i$
$H >> c =_A b\ \ i$

E3. $H >> a \in A\ \ i$ by equal types B
$H >> A =_{type(i)} B$
$H >> b \in B\ \ i$

E4. $H >> a =_A b \quad i$ by equal types B
$H >> A =_{type(i)} B$
$H >> a =_B b \quad i$

E5. $H >> t_1(a/x) =_{T(a/x)} t_2(b/x) \quad i$ by subst a,b,A in $x.t_1$, $x.t_2$, x.T
$H >> a =_A b \quad j$ new
$H, x{:}A >> t_1 =_T t_2$

E6. $H >> B(a/x) \in prop(i)$ by family A over x.B
$H >> a \in A \quad j$ new
$H, x{:}A >> B \in prop(i) \quad i$
(x:A of level j)

ARITHMETIC†

A1. $H >> a =_{int} b$ by arith [i op j]
* $H >> a \in int$
* $H >> b \in int$

when conclusion follows in the theory of
restricted arithmetic (see [13])

A2. $H >> (a =_{int} b) \mid \neg(a =_{int} b)$ by arith
* $H >> a \in int$
* $H >> b \in int$

A3. $H >> a<b$ by arith [i op j]
* $H >> a \in int$
* $H >> b \in int$
same proviso as above

A4. $H >> (a<b) \mid \neg(a<b)$ by arith
* $H >> a \in int$
* $H >> b \in int$

A5. $H >>$ false by arith
same proviso as above

†The first five rules are repeated at the place they would occur in the TYPES rules following the pattern established for the new types not occurring in λPRL. Likewise for the first LIST rule.

A6. H >> all x:int.T i by ind $-k_1(d,u)k_2$
* H >> (all x:int.T) $\in$ prop(i)
H, x:int, $z:x<d$, $v:T(x+k_1/x)$ >> T i
H >> T(d/x) i
.
.
.
H >> T(u/x) i
H, x:int, $z:u<x$, $v:T(x-k_1/x)$ >> T i

provided $d \le u$, $0<k_j \le (u-d)+1$ $j=1,2$
and provided the new assumptions are assigned level i.

LISTS

Li1. H >> a $=_{Alist}$ b i by equality
* H >> a $\in$ A list i
* H >> b $\in$ A list i

provided that a=b follows by the equality
decision procedure (see [28])

Li2. H >> all x:A list.T i by ind x_1. . . . $.x_n$.a
* H >> (all x:Alist.T) $\in$ prop(i)
H >> T([]/x) i
H, x_1:A, a:A list, u:T(a/x) >> T(x_1.a/x) i
.
.
.
H, x_1:A, . . . ,x_n:A, a:A list, u:T(a/x) >> T(x_1. . . . $.x_n$.a/x) i
provided the new assumptions are of level i

4. Higher Order Arithmetic

In this section the type structure is enriched to include the empty type, void, the type of atoms, and constructors for building dependent cartesian products, x:A#B, disjoint unions, A|B, and dependent function spaces, x:A→B. When B does not depend on x in x:A#B and in x:A→B, then the ordinary cartesian product, A#B, and function space A→B result as in int#int→int. The subtheory of these well-known operators is familiar under the name higher order arithmetic. An interesting fact about such a theory is that the methods of type checking used in λPRL can be extended to this theory. It is also interesting that the category of recursive function definitions needed in λPRL are replaced in form of the elimination rule on the types int and A list and the function space introduction rule.

EQUALITY (continued)

E7. $H \gg b(a/x) \in B(a/x)$ i by spec of x.b in A over x.B
$H \gg a \in A$ j new
$H, x{:}A \gg b \in B$ i
(x:A of level j)

E8. $H \gg T_1(a/x) =_{type(i)} T_2(b/x)$ by subst a,b,A in $x.T_1$, $x.T_2$
$H \gg a =_A b$ j new
$H, x{:}A \gg T_1 =_{type(i)} T_2$

VOID

1. Formation

$H \gg void \in type(i)$

2. Introduction

none

3. Elimination

$H, z{:}void, H' \gg any(z) \in A$

4. Equality

$H \gg any(e) =_A any(e')$ i new
* $H \gg A \in type(i)$
$H \gg e =_{void} e'$

ATOMS

1. Formation

$H \gg atoms \in type(i)$

2. Introduction

$H \gg$ 'a...' $\in$ atoms

3. Elimination

none

4. Equality

$H \gg (a =_{atoms} b) \mid \neg(a =_{atoms} b)$ by equality
* $H \gg a \in atoms$
* $H \gg b \in atoms$

$H \gg$ 'a...' $=_{atoms}$ 'b...' by equality
* $H \gg$ 'a...' $\in$ atoms (decision procedure)
* $H \gg$ 'b...' $\in$ atoms

5. Computation

none

6. Disequality

$H \gg \neg($'a...' $=_{atoms}$ 'b...'$)$ by equality
* $H \gg$ 'a...' $\in$ atoms
* $H \gg$ 'b...' $\in$ atoms

INTEGERS

1. Formation

$H \gg int \in type(i)$ (repeated)

2. Introduction

$H \gg n \in int$ (repeated)
where n is a signed decimal constant

$H \gg a\ op\ b \in int$ op = +,*,-,/
$H \gg a \in int$
$H \gg b \in int$

$H \gg \neg\, b=0$ if op = /

3. Elimination

$H, x:int \gg ind\ x\ do <d \to u,v.t_1,\ d \to t_d, \ldots, h \to t_h, >h \to u,v.t_p\ od$
$\in T_i$ new by elim

$H, x:int, x<d, v:T(x+k/x) \gg t_1(x+k/u) \in T_i$
$H, x:int, \gg t_d \in T(d/x)_i$
.
.
.
$H, x:int, \gg t_h \in T(h/x)_i$
$H, x:int, h<x, v:T(x-k/x) \gg t_p(x-k/u) \in T_i$
where $1 \le k \le (h-d)+1$ and
new hypotheses are of level i

4. Equality

H >> a $=_{int}$ b by arith [i op j] (repeated)
* H >> a ∈ int
* H >> b ∈ int

H >> (a $=_{int}$ b) | ¬(a $=_{int}$ b) by arith (repeated)
* H >> a ∈ int
* H >> b ∈ int

H >> ind e do <d→u,v.t_1,...,>h→u,v.t_h od $=_{T(e/x)}$ i new by indeq on x.T
    ind e' do <d→u,v.t'_1,...,>h→u,v.t'_h od
  H >> e $=_{int}$ e'
  H, u:int, u<d, v:T(u/x) >> t_1 $=_{T(u/x)}$ t'_1 i
  .
  .
  .
  H, u:int, h<u, v:T(u/x) >> t_p $=_{T(u/x)}$ t'_p i
  new hypotheses are of level i

5. Computation

H, w:e<d,H' >> ind e do <d→u,v.t_1,...,>h→u,v.t_h od $=_{T(e/x)}$
    t_1(e+k_1/u,ind(e+k_1)do_od/v) i new by equality over x.T

* H, w:e<d, H' >> T(e+k_1/x) ∈ type(i)
    * H, H', u:int, z:u<d, v:T(u/x) >> t_1 ∈ T(u-k_1/x) i
      .
      . call these hypotheses IH
      .
    * H, H', u:int, z:h<u, v:T(u/x) >> t_h ∈ T(u+k_2/x) i

H >> ind d do <d→u,v.t_1, d→t_d, ... od $=_{T(d/x)}$ t(d/u) i new
    by equality over x.T

* H >> T(c/x) ∈ type(i)
  IH
  .
  .
  .

H, w:h<e,H' >> ind e do <d→u,v.t_1,...,>h→u,v.t_h od $=_{T(e/x)}$
    t_h(e-k_2/u,ind(e-k_2)do_od/v) i new by equality over x.T

  * H, w:h<e, H' >> T(e/x) ∈ type(i)
    IH

6. Order

H >> a<b ∈ prop(i)
  H >> a ∈ int
  H >> b ∈ int

$H \gg a<b$ by arith [i op j]
$\quad * H \gg a \in int$
$\quad * H \gg b \in int$

$H \gg (a<b) \mid \neg(a<b)$ by arith
$\quad * H \gg a \in int$
$\quad * H \gg b \in int$

LISTS

1. Formation

$H \gg A$ list $\in$ type(i) (repeated)
$\quad H \gg A \in$ type(i)

2. Introduction

$H \gg [a_1,\ldots,a_n] \in A$ list (repeated)
$\quad H \gg a_i \in A$ or $n=0$

$H \gg hd(a) \in A$ (repeated)
$\quad H \gg a \in A$ list

$H \gg tl(a) \in A$ list (repeated)
$\quad H \gg a \in A$ list

$H \gg a.b \in.A$ list (repeated)
$\quad H \gg a \in A$
$\quad H \gg b \in A$ list

3. Elimination

$H, x{:}A$ list $\gg$ lind x do $0 \rightarrow t_0,\ldots,n \rightarrow x_1,\ldots,x_n,\ y.t_n$ od $\in T$ i new by elim

$H \gg t_0 \in T([]/x)$ i
$H, x_1{:}A, y{:}A$ list $\gg t_1 \in T(x_1.y/x)$ i
.
.
.
$H, x_1{:}A,\ldots,x_n{:}A, y{:}A$ list $\gg t_n \in T(x_1 \cdot \cdot \cdot \cdot \cdot x_n.y/x)$ i
new hypotheses are of level i

4. Equality (see Lil)

type $H \gg A$ list $=_{type(i)} B$ list by equality
$\quad H \gg A =_{type(i)} B$

elim

H >> lind e do $0 \to t_0, \ldots n \to x_1, \ldots, x_n, y.t_n$ od $=_T$
lind e' do $0 \to t'_0, \ldots n \to x_1, \ldots, x_n, y.t'_n$ od
by A list eq over x.T i new
H >> e $=_{A\ list}$ e' i
H >> $t_0 =_{T([]/x)} t_0'$ i
H, x_1:A, y:A list >> $t_1 =_{T(x_1.y/x)} t_1'$ i
.
.
.
H, x_1:A,...,x_n:A, y:A list >> $t_n =_{T(x_1 \cdot \ldots \cdot x_n.y/x)} t_n'$ i
new hypotheses are of level i

5. Computation

H >> lind [] do $0 \to t_0, \ldots, n \to x_1, \ldots, x_n, y.t_n$ od $=_{T([]/x)} t_0$
i new by equality on x.T
* H >> T([]/x) ∈ type(i)
* H >> t_0 ∈ T([]/x) i
.
.
.
H, x_1:A,...,x_n:A, y:A list >> $t_n \in T(x_1 \cdot \ldots \cdot x_n.y/x)$ i
new hypotheses are of level i

UNION

1. Formation

H >> (A|B) ∈ type(i) formation
H >> A ∈ type(i)
H >> B ∈ type(i)

2. Introduction

H >> rin(b) ∈ (A|B) new i intro
* H >> A ∈ type(i)
H >> b ∈ B

H >> lin(a) ∈ (A|B) new i intro
* H >> B ∈ type(i)
H >> a ∈ A i

3. Elimination

H, d:(A|B), H' >> if d left u.t_1 right u.t_2 fi
∈ T new i elim d
H, u:A, H'(lin(u)/d) >> t_1 ∈ T(lin(u)/d) i
H, v:B, H'(rin(v)/d) >> t_2 ∈ T(rin(v)/d) i

where A,B have the same level as (A|B), say k.

Notice that $d:(A|B)$ is <u>replaced</u> causing replacements in H'. Also note the use of u,v in hyp and in bindings $u.t_1$, $v.t_2$.

4. Equality

$H >> t =_{A|B} t' \in type(i)$ by union type
* $H >> t \in (A|B)$ i
* $H >> t' \in (A|B)$ i

$H >> lin(a) =_{(A|B)} lin(a')$ new i by equality
* $H >> B \in type(i)$
  $H >> a =_A a'$ i

likewise for rin

$H >> if\ d\ left\ u.t_1\ right\ v.t_2\ fi =_{T(d/z)} if\ d'\ left\ u.t'_1\ right\ v.t'_2\ fi$
new i by unioneq over z.T
using (A|B)

$H >> d =_{(A|B)} d'$ new k

$H, u:A >> t_1 =_{T(lin(u)/z)} t'_1$ i
$H, v:B >> t_2 =_{T(rin(v)/z)} t'_2$ i

5. Computation

$H >> if\ lin(a)\ left\ u.t_1\ right\ v.t_2\ fi =_{T(lin(a)/z)} t_1(a/u)$ new i

by equality over z.T using A|B new i

* $H >> (A|B) \in U(k)$ new k
  $H >> a \in A$ k
  $H, u:A >> t_1 \in T(lin(u)/z)$ i
  $H, v:B >> t_2 \in T(rin(v)/z)$ i

Note global convention on the level of hypotheses.

PRODUCT

1. Formation

$H >> x:A \# B \in type(i)$ by formation
  $H >> A \in type(i)$
  $H, x:A >> B \in type(i)$

2. Introduction

$H >> <a,b> \in (x:A\#B)$ i new by intro
* $H, x:A >> B \in type(i)$
  $H >> a \in A$ i
  $H >> b \in B(a/x)$ i

3. Elimination

H, p:(x:A#B), H' >> \$p over x,y.t\$ ∈ T i new by elim
  H, x:A, y:B, H'(<x,y>/p) >> t ∈ T (<x,y>/p) i
  where A,B have same level as x:A#B, say k.

4. Equality
type

H >> x:A#B $=_{type(i)}$ x:C#D by equality
  H >> A $=_{type(i)}$ C
  H, A $=_{type(i)}$ C, x:A >> B $=_{type(i)}$ D

intro

H >> <a,b> $=_{(x:A\#B)}$ <c,d> i new by equality
  * H, x:A >> B ∈ type(i)
    H >> a $=_{A}$ c i
    H >> b $=_{B(a/x)}$ d i

H >> \$p over u,v.t\$ $=_{T(p/z)}$ \$p' over u',v'.t'\$ i new
  by equality on x:A#B over z.T
  * H >> T ∈ type(i)
  * H >> p $=_{(x:A\#B)}$ p' k new
    H, u:A, v:B >> t $=_{T(\langle u,v\rangle/z)}$ t'(u/u',v/v') i
    where A,B are level k

5. Computation

H >> \$<a,b> over x,y.t\$ $=_{T(\langle a,b\rangle/z)}$ t(a/x,b/y) x:A#B
  prodcomp over z.T i new
  * H, B:(x:A#B) >> T ∈ type(i)
  * H >> x:A#B ∈ U(k)
  * H >> a ∈ A k k new
  * H >> b ∈ B(a/x) k k new

FUNCTIONS

1. Formation

H >> (x:A→B) ∈ type(i) by formation
  * H >> A ∈ type(i)
  * H, x:A >> B ∈ type(i)

2. Introduction

H >> λx.b ∈ x:A→B new i by intro
  * H >> A ∈ type(i)
    H, x:A >> b ∈ B i

3. Elimination

H, f:(x:A→B), H' >> t(f(a)/y) ∈ T(f(a)/y)
  by elim on f over y.T, y.t
  H, f:(x:A→B), H' >> a ∈ A k
  H, f:(x:A→B), H', y:B(a/x) >> t ∈ T i

Note: The generality of the elim rule is dictated by the extract form, otherwise a more appropriate rule would be

$H, f{:}(x{:}A \to B), H' \gg f(a) \in B(a/x)$ by elim f
  $H \gg a \in A$

4. Equality

$H \gg (f =_{x:A\to B} f') \in \mathrm{type}(i)$ by equality
  $* \ H \gg f \in (x{:}A \to B)$ i
  $* \ H \gg f' \in (x{:}A \to B)$ i

intro $H \gg \lambda x.b =_{x:A\to B} \lambda x.b'$ new i by equality

elim $* \ H \gg x{:}A \to B \in \mathrm{type}(i)$
  $H, x{:}A \gg b =_{B} b'$ i

$H \gg f(a) =_{B(a/x)} g(b)$ new i by equality over $x{:}A \to B$
  $H \gg f =_{(x:A\to B)} g$ i
  $H \gg a =_{A} b$ i

5. Computation

$H \gg (\lambda x.b)(a) =_{B(a/x)} b(a/x)$ i new by \ red on $x{:}A \to B$
  $H \gg \lambda x.b \in (x{:}A \to B)$ i
  $H \gg a \in A$ i

5. Sets and Quotients

The concept of a set is clearly important in computational mathematics. But the set concept can sometimes be considered a circumlocution in propositions, for example instead of saying all $x{:}\{i{:}int \mid P(i)\}.R(x)$ one can say $allx.int.P(x) \Rightarrow R(x)$. However the notion of a subset type has arisen in programming language notations. For instance we might want to express the notion that function f takes prime numbers as inputs. We can assign to the type $\{x{:}int \mid prime(x)\} \to A$. For this reason we introduce the set constructor (for a deeper analysis of this constructor see [9]).

It is also useful to have a mechanism for defining new equality operators on types. For example, one might wish to treat pairs of integers as rational numbers by introducing the equality $\langle a,b\rangle = \langle c,d\rangle$ iff $a*d = b*c$. This is accomplished using the quotient constructor; examples of its use appear in section III.4 and III.5.

QUOTIENTS

1. Formation

$H >> (A/E) \in type(i)$ by formation
  $H >> A \in type(i)$
  $H, x{:}A, y{:}A >> E(x)(y) \in prop(i)$ i
  $H, x{:}A >> E(x)(x)$ i
  $H, x{:}A, y{:}A, u{:}E(x)(y) >> E(y)(x)$ i
  $H, x{:}A, y{:}A, z{:}A, u{:}E(x)(y), w{:}E(y)(z) >> E(x)(z)$ i
  new hypotheses are of level i

2. Introduction

$H >> a \in A/E$ new i by intro
  * $H >> (A/E) \in type(i)$
    $H >> a \in A$

3. Elimination

$H, u{:}A/E, H' >> t \in T$ i new by /elim
  * $H, u{:}A/E, H' >> T \in type(i)$
    $H, u{:}A, H', x{:}A, H'(x/u), v{:}E(u)(x) >>$
      $t =_T t(x/u)$ i
    new hypotheses are of level i

4. Equality

$H >> A/E =_{type(i)} B/F$
  * $H >> A/E \in type(i)$
  * $H >> B/F \in type(i)$
    $H >> A =_{type(i)} B$
  $H, u{:}(A =_{type(i)} B), x{:}A, y{:}A >>$
      $E(x)(y) \Leftrightarrow F(x)(y)$ i

x:A, y:A are of level i, the other hypothesis is of level k>i

$H >> a =_{A/E} a' \in prop(i)$
* $H >> A/E \in type(i)$
* $H >> a \in (A/E)$ i
* $H >> a' \in (A/E)$ i

$H >> a =_{A/E} a'$ i new
* $H >> A/E \in type(i)$
  $H >> a \in A$ i
  $H >> a' \in A$ i
  $H >> E(a)(a')$

Note, when * is not used, the provisos on the thm $H >> A/E \in type(i)$ must be added to the list of provisions.

5. Computation

none

SETS

1. Formation

$H \gg \{x{:}A|B\} \in type(i)$ by formation
  $H \gg A \in type(i)$
  $H, x{:}A \gg B \in type(i)$ i

2. Introduction

$H \gg a \in \{x{:}A|B\}$ i new by intro
  * $H \gg \{x{:}A|B\} \in type(i)$
    $H \gg a \in A$ i
    $H \gg B(a/x)$ i

3. Elimination

$H, u{:}\{x{:}A|B\}, H' \gg t \in T$ i new by elim
  * $H, u{:}\{x{:}A|B\}, H' \gg T \in type(i)$
    $H, u{:}A, y{:}B, H' \gg t \in T$
    provided y new
    y does not occur in t

4. Equality

$H \gg a =_{\{x:A|B\}} a' \in type(i)$ by eq
  * $H \gg \{x{:}A|B\} \in type(i)$
  * $H \gg a \in \{x{:}A|B\}$ i
  * $H \gg a' \in \{x{:}A|B\}$ i
    $H \gg a =_A a'$ i

5. Computation

none

6. Universes

One way of achieving sufficient flexibility in a rigidly typed language is to permit so-called polymorphic operations (see [19]). This can be accomplished by allowing types to be objects; then they can be passed as arguments to functions, and such functions are thus parameterized with respect to type. If types are objects, then functions can be of the sort f:type→type, and one must ask whether type ∈ type or whether type itself belongs to a higher kind of classification, say type ∈ large type. One is either led to examine reflexive notions of type, as in [34], or a hierarchical notion as in [33]. We have adopted a hierarchical account based on [33].

UNIVERSES

H >> A ∈ type(j)
  H >> A ∈ type(i)   provided i<j

1. Formation

H >> type(i) ∈ type(j) provided i<j

H >> (a $=_A$ b) ∈ type(j)
  * H >> A ∈ type(i) provided i<j
  * H >> a ∈ A
  * H >> b ∈ A

2. Introduction

These rules are distributed among the specific types

3. Elimination

H, u:type(i), H' >> case(i)(u, t_1,t_2,t_3, x,y.t_4, x,f.t_5, x,f.t_6, x,f.t_7, x,y_1,y_2.t_8, k.t_9, x,e,z1,z2,z3, ax1,ax2,ax3.t_{10}, u.t_{11}) ∈ T new j by elim u

0. H, u:type(i), H' >> T_u ∈ type(i)

1. H, -, $H'_{int/u}$ >> $T_{(int/u)}$ t_1 ∈ T(int/u)

2. H, -, $H'_{atom/u}$ >> $T_{(atom/u)}$ t_2 ∈ T(int/u)

3. H, -, $H'_{false/u}$ >> $T_{(false/u)}$ t_3 ∈ T(int/u)

4. H, x:type(i), y:type(i), H'((x|y)/u) >> $T_{(x|y/u)}$ t_4 ∈ T(int/u)

5. H, x:type(i), f:x→type(i), $H'_{(z:x\to f(z)/u)}$ >> $T_{(z:x\to f(z)/u)}$ t_5 ∈ T(int/u)

6. H, x:type(i), f:x→type(i), $H'_{(z:x\#f(z)/u)}$ >> $T_{(z:x\#f(z)/u)}$ t_6 ∈ T(int/u)

7. H, x:type(i), f:x→type(i), $H'_{(\{z:x|f(z)\}/u)}$ >> $T_{(\{z:x|f(z)\}/u)}$ t_7 ∈ T(int/u)

8. H, x:type(i), y_1:x, y_2:x, $H'_{(y_1=_x y_2/u)}$ >> $T_{(y_1=_x y_2/u)}$ t_8 ∈ T(int/u)

9.j. H, - $H'_{(U(j)/u)}$ >> $T_{(U(j)/u)}$ t_9 ∈ T(int/u)
for j=1, ..., i-1.

10. H, x:type(i), e:x→x→type(i),
z1:x, z2:x, z3:x, ax1:e(z1)(z1), ax2:e(z1)(z2)→e(z2)(z1),
ax3:e(z1)(z2)#e(z2)(z3)→e(z1)(z3),
$H'_{((x/e)/u)}$ >> T_{10} ∈ ((x/e)/u)

11. H, u:type(i), H' >> t_{11} ∈ T

4. Equality

$$H \gg \mathrm{type}(i) =_{\mathrm{type}(j)} \mathrm{type}(i)$$
provided $i<j$

$$H \gg (a =_A b) =_{\mathrm{type}(i)} (a' =_B b')$$
$$H \gg A =_{\mathrm{type}(i)} B$$
$$H,\ A=B \gg a =_A a'$$
$$H,\ A=B \gg b =_A b'$$

5. Computation

(omitted)

7. Higher Order Logic

There is an important class of argument which we cannot express in the logic as it stands now, namely proofs of expressions which are abstracted with respect to propositions or to propositional functions. for instance, we cannot say "for all propositions p and q, $\neg p | q$ implies $p \Rightarrow q$."

The syntax of the judgement part of the logic suggests that there is a type of propositions, e.g. when we write $(x =_{int} x) \in \mathrm{prop}(1)$, the expression prop(1) plays the role of a type. If we allowed declarations of the form p:prop, q:prop then we could express the propositional concept mentioned as

$$p{:}\mathrm{prop},\ q{:}\mathrm{prop} \gg (\neg p | q) \Rightarrow (p \Rightarrow q).$$

If we treat prop(i) as a type, then we obtain a version of higher order logic because we can quantify over propositions and propositional functions. But in order to understand such an idea we must be clear about the mathematical status of propositions as objects. Unlike in the systems of Frege [17] and Russell [33] a proposition does not denote a truth value, either <u>true</u> or <u>false</u>. Thus before we can completely understand higher order logic we must examine the semantics of propositions. As a start we can take propositions as objects by taking prop(i) as a type. To do this we need to specify equality on propositions. Here are the rules.

propositions are objects of type prop(i)

$H \gg \text{prop}(i) \in \text{type}(j)$
provided $i<j$

equalities are propositions

$H \gg (a =_A a') \in \text{prop}(i)$ (repeated)
$H \gg a \in A$
$H \gg a' \in A$
$H \gg A \in \text{type}(i)$

equality of atomic propositions

$H \gg (a =_A a') =_{\text{prop}(i)} (b =_B b')$
$H \gg A =_{U(i)} B$
$H, A =_{\text{prop}(i)} B \gg a =_A b$
$H, A =_{\text{prop}(i)} B \gg a' =_A b'$

inequalities are propositions (repeated)

$H \gg a<b \in \text{prop}(i)$
$H \gg a \in \text{int}$
$H \gg b \in \text{int}$

equality of inequalities

$H \gg (a<b) =_{\text{prop}(i)} (a'<b')$
$H \gg a =_{\text{int}} a'$
$H \gg b =_{\text{int}} b'$

compound propositions are objects

(see FORMATION RULES)

equality of compound propositions

$H \gg A \text{ op } B =_{\text{prop}(i)} A' \text{ op } B'$ op = $\&, |, \Rightarrow$
$H \gg A =_{\text{prop}(i)} A'$
$H \gg B =_{\text{prop}(i)} B'$

$H \gg \text{quant } x{:}A.B =_{\text{prop}(i)} \text{quant } y{:}A'.B'$ for quant = all, some
$H \gg A =_{\text{type}(i)} A'$
$H, x{:}A, A =_{\text{type}(i)} A' \gg B =_{\text{prop}(i)} B'(x/y)$

8. Summary

This concludes the major stages of evolution of the logic. In the next section we consider a formal semantics of this logic and discuss the propositions-as-types principle which will allow dramatic simplification of the logic and which will add a powerful new rule.

III. Examples

This section consists of five examples which illustrate various aspects of the theory. The first two examples are very elementary and serve to illustrate the proof structure. The third example illustrates the value of a rich theory even for problems which can be solved well in a first order theory. The last two examples illustrate the development of mathematics.

1. A Propositional Calculus Proof

We can express the usual propositional arguments in an abstract form. For example

```
     >> all p,q,r:prop(k).(p⇒q⇒r) ⇒ (p⇒q) ⇒ (p⇒r) i by intro

(1)  1. p,q,r:prop(k)  >> (p⇒q⇒r) ⇒ (p⇒q) ⇒ (p⇒r) i by intro

(1)  1.   , 2. p⇒q⇒r  >> (p⇒q) ⇒ (p⇒r)           i by intro

(1)  1.   , 2.   , 3. p⇒q >> p⇒r                  i by intro

(1)  1.   , 2.   , 3.   , 4. p >> r               i by elim 3

     (1)  1,2,3,4  >>  p        i hyp 4
          [any k<i]
```

(2) 1,2,3,4, 5. q >> r   i by elim 2

(1)   >> p   i hyp

[any k<i]

(2) 1,2,3,4,5, 6. q⇒r >> r   i by elim 6

(1) 1,2,3,4,5,6 >> q   i hyp

[any k<i]

(2) 1,2,3,4,5,6, 7. r >> r   i hyp

[any k<i]

We have numbered the subgoals as (1), (2). With these numbers we can give the tree-address of any rule application. For example, the first application of hyp is at node 1.1.1.1.1 and elim 2 is at 1.1.1.1.2. Also as is obvious, we abbreviate the hypotheses by number after their first occurrence. This allows us to list all hypotheses at each node. A similar kind of abbreviation underlies the implementation.

Notice that level numbers play a trivial role here. This is a proof scheme valid at every level i>k. It can be instantiated at any level prop(k).

2. <u>A Precidate Calculus Proof</u>

1. A:type , 2. B:A#A→prop >> some y:A all x:A.B(<x,y>) ⇒
all x:A.some y:A.B(<x,y>)   by intro

(1) 1.   , 2.   , 3. some y:A.all x:A.B(<x,y>) >> all x:A.some y:A.B(<x,y>)
by intro

```
(1) 1.   , 2.   , 3.   , 4. x:A              >> some y:A.B(<x,y>)
                                                 by elim 3

(1) 1.   , 2.   , 3. y:A , 4. all x:A.B(<x,y>) , 5. x:A >> some y:A.B(<x,y>)
                                                 elim 4, x

    (1) 1,2,3,4,5               >> x ∈ A            hyp 5

    (2) 1,2,3,4,5, 6. B(<x,y>) >> some y:A.B(<x,y>)  by intro x

        (1) 1,2,3,4,5,6          >> B(<x,y>)    hyp 6
```

Here we have suppressed level numbers and let prop abbreviate prop(k).

3. Maximum Segment Sums

The vPRL language not only allows us to define concepts that are ineffable in λPRL, but permits a more succinct and general treatment of ideas that can be expressed in λPRL. For example, in the paper "Proofs as Programs" [4], we solve the following problem:

given an integer sequence of length n, say [a(1),...,a(n)], write a program to find the sum, $\sum_{i=p}^{q} a(i)$, that is maximum among all such sums.

We called a consecutive subsequence, [a(p), a(p+1),...,a(p+q)], a segment and we called this the maximum segment sum problem. We solved this problem in λPRL by proving the following assertion

all a:list.some(M,L):int.
some(a,b,s):int.all(p,q):int.

$$(1 \le p \le q \le len(a) \Rightarrow M = \sum_{i=a}^{b} a(i) \ \& \ M \ge \sum_{j=p}^{q} a(j) \ \&$$

$$L = \sum_{i=s}^{len(a)} a(i) \ \& \ L \ge \sum_{j=p}^{len(a)} a(j))$$

The inductive proof of this assertion given in [4] provides a linear time algorithm for finding M; the other values L, a, b, s are auxiliary, used to define M and compute it efficiently.

In νPRL we can analyze the problem in a more general way, achieving at the same time a more compact and algebraic proof. We start with the concept of the maximum of a "two dimensional" (i.e. two argument) function.

(1) Define pos = $\{n:int \mid n \geq 1\}$

(2) Define max1:pos # (pos→int) → int by extracting from

$\forall f:(pos \to int).\forall n:pos.\exists m:int.\exists j:pos.$

$(\forall i:pos.(1 \leq i \leq n \Rightarrow f(i) \leq m) \;\&\; 1 \leq j \leq n \;\&\; m = f(j)))$

(3) Define max2:pos # pos # (pos # pos → int) → int by the rule

max2(n,m,f) = max1(n,λx.max1(m,λy.f(x,y)))

(or we could define max2 as well by extraction).

Now we notice that the two dimensional maximum can be computed in a different order.

(4) Lemma: max2(n+1,m+1,f) =
max(max1(m+1,λy.f(n+1,y)),
max2(n,λx.max1(m+1,λy.f(x,y)))).

We can then try to express max2(n+1, m+1, f) in terms of max(n,m,f) in order to understand how we might compute max2(n+1, m+1, f) iteratively. To this end we prove

(5) Lemma: max2(n,λx.max1(m+1,λy.f(x,y))) =
max(max1(n,λx.max1(m,λy.f(x,y))),
max1(n,λx.f(x,m+1))).

We can now view max2(n+1, m+1, f) as max(S_1, max(S_2,S_3)), where S_2 is max2(n, m, f).

When $f(x,y) = \sum_{i=x}^{y} a(i)$, then these laws about max1 and max2 take a simpler form, and when we consider the form of the maximum segment sum problem we notice that we are computing the special form $\mathrm{max2}(n,n,\lambda x,y.\ \sum_{i=x}^{y} a(i))$. In this case, S_2, that is $\mathrm{max1}(n+1,\lambda y.f(n\_1,y))$, is just $\sum_{i=n+1}^{n+1} a(i) = a(n+1)$. So the problem reduces to

```
(6)   max2(n+1,n+1,f) =
      max(a(n+1),
          max(max2(n,n,f),max1(n,λx.f(x,n+1))))
```

When we analyze the term S_3, i.e. $\mathrm{max1}(n,\lambda x.f(x,n+1))$, we see that it is the maximum sum of segments that include $a(n+1)$. Notice that the maximum is unchanged if we replace $\mathrm{max1}(n,\lambda x.f(x,n+1))$ by $\mathrm{max1}(n+1,\lambda x.f(x,n+1))$. We simply cover the term $f(n+1,n+1)$ twice. It will be convenient to use this term as S_3 because we can describe its computation iteratively. Following the notation in [4], let $L_{n+1} = \mathrm{max1}(n+1,\lambda x.f(x,n+1))$, and let $M_{n+1} = \mathrm{max2}(n+1,n+1,f)$.

(7) $M_{n+1} = \max(a(n+1), \max(M_n, L_{n+1}))$.

If we can compute L_{n+1} in terms of L_n, then we will have an iterative solution to the problem. But this is an easy computation from the definition, namely

$$(8)\quad \mathrm{max1}(n+1,\lambda x.\ \sum_{i=x}^{n+1} a(i)) = \max(a(n+1), \mathrm{max1}(n,\lambda x.\ \sum_{i=x}^{n} a(i)) + a(n+1))$$

$$= \max(a(n+1), L_n + a(n+1))$$

Using this form we get

(9) $M_{n+1} = \max(a(n+1), M_n, L_n + a(n+1))$.

From the above equation we can write a very simple inductive proof of the following characterization of the maximum segment sum problem.

Theorem: all a:int list.some m:int.

$$(m = \text{max2}(\text{len}(a), \text{len}(a), \lambda x,y. \sum_{i=x}^{y} a(i))).$$

Proof: (by induction on a).
Base case: a=[], take m as any integer.
Induction case:
assume a = k.b and $M = \text{max2}(\text{len}(b), \text{len}(b), \lambda x,y. \sum_{i=x}^{y} b(i))$
and $L = \text{max1}(\text{len}(b), \lambda x. \sum_{i=1}^{\text{len}(b)} b(i))$, then
take m = max(k,M,L+k) and use the above algebraic analysis.
QED

This proof is impossible in λPRL because we cannot define max1, max2 since they take functions as arguments. We could mimic this proof by treating equations (2)-(7) as specific facts about a function of the form max2(n,m,a,x,y), but such a course is discouraging because the facts actually being proved cannot be expressed. One is then led to seek a more direct proof which avoids some of the lines that encode useful general lemmas.

We see from this example that the richness of the language determines the way we analyze a problem at the level of formal detail.

4. Rational Numbers

Let us consider how the rational numbers, rat, can be defined as a type in this theory. First we define the legitimate fractions, e.g. those without zero denominators.

$$Fr = int \# \{z:int \mid \neg z=0\}$$

Now on Fr we define equality of fractions, namely <a,b> = <c,d> iff a*d = b*c. For a precise definition we would like simple notation for the selection functions from pairs. To this end define:

lof(x) = $x over u,v.u$

2of(x) = $x over u,v.v$

Then we know lof(<a,b>) = $<a,b> over u,v.u$ = a.

Next define equality on fractions as

eq(x,y) == (lof(x) * 2of(y) = 2of(x) * lof(y))

We must then prove these simple propositions

1. all x:Fr.(eq(x,x))

2. all x:Fr.all y:Fr.(eq(x,y) ⇒ eq(y,x))

3. all x:Fr.all y:Fr.all z:Fr.(eq(x,y) & eq(y,z) ⇒ eq(x,z))

It is interesting to note that we can trivially prove these properties informally by appealing to the symmetry properties of the <u>expression</u> lof(x) * 2of(y) = lof(y) * 2of(x). This is, of course, metamathematical reasoning. It is also easy to give purely arithmetic proofs.

Finally we can take the rationals, rat, to be

rat = Fr/λx.λy.eq(x,y)

Notice that $=_{rat}$ denotes the equality relation on rationals. It is interesting to note that although <2,4> $=_{rat}$ <1,2>, when we print the rational <2,4>, we will see <2,4>. So just as school children must be careful to reduce fractions if they want compact answers, so must we. To do this we can build a function

reduce:Fr → Fr

which divides out the gcd of numeration and denomination. We can prove that reduce maps rat to rat and indeed reduce(x) $=_{rat}$ x. If we want to see reduced output, then we must print reduce(x); even though this equals x in rat, it does not equal x in Fr, and it is Fr that we "see".

Using reduce one can define "the numerator and denominator" of a rational as follows

$$\begin{aligned} \mathrm{num}(x) &= \mathrm{1of}(\mathrm{reduce}(x)) \\ \mathrm{denom}(x) &= \mathrm{2of}(\mathrm{reduce}(x)) \end{aligned}$$

These are maps rat → int

5. Real Numbers

Let us define a very interesting type, the <u>real</u> numbers. Following Bishop [6] we take them to be the cauchy convergent sequences of rationals. First define the cauchy condition on sequences of rationals. Recall that nat = {z:int|¬z<0}.

$$\begin{aligned} &\mathrm{cauchy}(f) = \mathrm{all}\ n{:}nat.\mathrm{all}\ m{:}nat. \\ &\qquad (|f(n)-f(m)| \le \langle 1,n\rangle+\langle 1,m\rangle) \end{aligned}$$

where |x| is the absolute value of a rational and <1,n>+<1,m> is the rational sum of the fractions 1/n and 1/m.

Now define the cauchy convergent sequences as

$$\mathrm{cseq} = \{f{:}nat \to rat \mid \mathrm{cauchy}(f)\}.$$

To define the real numbers we must agree on the definition of equality. We choose this, again following Bishop:

$$\mathrm{realeq}(x,y) = \mathrm{all}\ n{:}nat.(|x(n)-y(n)| \le \langle 2,n\rangle).$$

We must verify that this is an equivalence relation. This requires real work which is not shown here. Then the reals are defined as

$$\mathrm{real} = \mathrm{cseq}/\lambda x.\lambda y.\mathrm{realeq}(x,y).$$

Notice that $=_{real}$ is the equality relation on reals.

It is interesting to define the division operation on integers into reals and the embedding operations of nat → rat and rat → real. Suppose these are named as

$$/\text{:int} \# \{z\text{:int} \mid \neg z=0\} \rightarrow \text{real}$$
$$\text{rl:rat} \rightarrow \text{real}$$

Then we can prove:

$$\text{all a:int.all b:}\{z\text{:int} \mid \neg z=0\}.(a/b =_{\text{real}} \text{rl}(\langle a,b \rangle))$$

This establishes the relationship between the <a,b> and a/b forms of rational numbers. We can think of a/b as the real number form of "the fraction a/b."

Operations on real are defined first on the cauchy sequences and then extended to real by the rules for quotient types. For example here is how + is defined.

$$\text{for x,y:cseq} \qquad (x+y)(n) = x(2*n)+y(2*n).$$

We must then show that

$$\text{if } x =_{\text{real}} x' \text{ and } y =_{\text{real}} y'$$
$$\text{then } x+y =_{\text{real}} x'+y'$$

This proposition allows us to treat + as a map real#real → real.

The reader interested in developing calculus and analysis along these lines should consult the books of Bishop [6] and Bridges [7].

IV. SEMANTICS

1. Goals

So far the meaning of formulas in this logic has been given only informally in terms of intuitive operations on proofs. We can recognize that the rules of inference preserve the constructive meaning of the logical operations as described in I.3. The semantic explanations of I.3 rely on our intuitive grasp of the concept of a rule or method and on our intuitive grasp of the notion of proof or evidence. We understand these ideas well enough to confirm that the rules of II are indeed rules of evidence or rules of constructive truth.

We are now interested in giving a more rigorous and mathematical account of meaning. We would like to have a mathematical semantics for this theory similar in its precision to Tarski's account of classical semantics [37]. Such an account would not only play an analogous role to Tarski's and give rise to a model theory of constructive logic, but it would also provide the theoretical blueprint for implementing the logic. To implement it is to build a computer program which will carry out the operations which define the logic. This is the critical element missing from classical semantics which leaves it inert.

2. Analysis of Informal Constructive Semantics

According to the constructive meaning of $A \Rightarrow B$, a proof of this statement implicitly exhibits a rule which will take proofs of A into proofs of B. In this theory we take rules to be given by functions, denoted by λ expressions. We might then expect to associate to this proof a λ expression which codifies the rule.

If for each inference rule we can find a mathematical object which codifies its meaning, then perhaps we can give a precise mathematical meaning to the logical operations. This can in fact be done. To explain how, we add semantic information to the rule. The new information shows explicitly what mathematical object is associated with the rule of inference.

Here is the augmented form of the implication introduction rule. It is called the <u>extract-form</u>.

```
H >> A⇒B i EXT λx.b
  H, x:A >> B i    EXT b
* H >> A ∈ prop(i)
```

The EXT clauses are to be read bottom-up. Thus the rule says that if the term b is extracted from the proof of the subgoal B from hypotheses H, x:A; then the term λx.b is extracted from the implication introduction step.

This isolated example suggests a pattern of analysis. We might be able to explain for each inference rule how to put together the semantic information from the subgoals into semantic information for the goals. We could then inductively assign meaning to the entire proof. However, this example does not suggest what to do at the basis of induction, i.e. what semantic information do we assign to axioms such as $0 =_{int} 0$. Let us leave this question aside for the moment and press our investigation where it is yielding information. <u>We will try to assign meaning to inference rules under the assumption that we can assign it to the subgoals</u>. So we assume that we know what it means to build a term t which denotes a proof of proposition T.

3. Meaning of the Inference Rules

Here are all of the predicate calculus rules in their extract forms (with level and well-formedness information suppressed)

AND

```
    H >> A&B              by intro EXT <a,b>
      H >> A   EXT a
      H >> B   EXT b

    H, x:(A&B),H' >> T    by elim  EXT  $x over x1,x2.t$
      H, x1:A, x2:B, H' >> T   EXT  t
```

```
OR
    H >> (A|B)    by intro 1   EXT  lin(a)
      H >> A   EXT a

    H >> (A|B)    by intro 2   EXT  rin(b)
      H >> B   EXT b

    H, x:(A|B), H' >> T   by elim  EXT  if x left u.t1 right v.t2 fi
      H, u:A, H'(lin(u)/x) >> T  EXT  t1
      H, v:B, H'(rin(v)/x) >> T  EXT  t2


IMPLIES
    H >> A⇒B   by intro  EXT  λx.b
      H, x:A >> B   EXT  b

    H, f:(A⇒B), H' >> T   by elim   EXT  t(f(a)/y)
      H, f:(A⇒B), H' >> A   EXT  a
      H, f:(x:A⇒B), H', y:B >> T   EXT  t


FALSE
    H, z:false, H' >> T   by elim  EXT  any(z)


ALL
    H >> all x:A.B   by intro   EXT   λx.b
      H, x:A >> B   EXT   b

    H, f:(all x:A.B), H' >> T   by elim a   EXT   t(f(a)/y)
      H, f:(all x:A.B), H' >> a ϵ A
      H, f:(all x:A.B), H', y:B(a/x) >> T   EXT   t


SOME
    H >> some x:A.B   by intro a   EXT   <a,b>
      H >> B(a/x)   EXT   b

    H, p:(some x:A.B), H' >> T  by elim   EXT  $p over x,y.t$
      H, x:A, y:B, H' >> T   EXT   t


CONSEQUENCE
    H >> T by seq A   EXT   t(a/x)
      H >> A   EXT  a
      H, x:A >> T   EXT  t
```

4. Analogy Between Propositions and Types

The truly remarkable feature of these extract forms is that the extracted terms in all cases are terms from the type theory. We can see intuitively that these are the correct terms to extract; that is, the computational meaning of the term as given by the type theory rules is congruent to the intuitive operational meaning of the inference rule. For example, if we know that b is a term which denotes a proof of B from H, x:A, then λx.b denotes a proof of A⇒B because λx.b is a rule for computing proofs of B from proofs of A.

The above observation suggests that the logical operation of implication corresponds to the type constructor →. The correspondence extends as the reader can easily verify to the following:

Logical Operation	Type Constructor
&	#
|	|
⇒	→
false	void
some x:A.B	x:A#B
all x:A.B	x:A→B

Table 1

This correspondence suggests that compound propositions are very much like types in their computational meaning. If there were a way to extend this correspondence to all atomic propositions as well, then we could in fact identify propositions with types, which in some real sense correspond to mathematical objects which codify their proofs. We would then have a method of attacking the open issue of how to assign mathematical meaning to the atomic propositions thereby discharging the open assumption in this semantic analysis.

5. Meaning of Atomic Propositions

If we try to extend the interpretation of propositions-as-types to the atomic propositions, then we must in some way construe false, $a =_A b$ and $a<b$ as types. We want the correspondence to be that the proposition denotes the type of its proofs. This works well for false which can be identified with the void type as we observed above. But what happens to $a =_A b$?

Consider the atomic proposition $1 =_{int} 1$. How are we to prove this? One proof is to take it as an axiom. So we might create an element called ax and assign it to the type $1 =_{int} 1$. What about the type $0 =_{int} 1$? This should not have any canonical proofs. In fact, $0 =_{int} 1$ should be an empty type. Should it be the void type? Do we think of $0 =_{int} 1$ and $1 =_{int} 2$ as the same propositions? Certainly they are not denoted by the same formulas, but they might express the same idea. As types they have the same extension, thus in a set theoretic setting they would be equal. But consider the extensionally equal proposition $\lambda x.x =_{int \to int} \lambda x.1$. This is another proposition with no canonical proofs. But our understanding of this requires different knowledge than our understanding of $0 =_{int} 1$.

We can see the kind of knowledge required for equality if we consider how we might define $n =_{int} m$ from the single atomic proposition $0 =_{int} 0$ and an inductive definition of the positive and negative numbers. Such a definition for positive numbers might be that $(n-1) =_{int} (m-1)$. This is quite a different idea than that behind the equality of two functions. Therefore if we take seriously the notion that propositions are understood by grasping how to prove them, then these must be distinct propositions. This suggests that $0 =_{int} 1$ and $1 =_{int} 2$ are also different. Consequently, equality of propositions and types is not extensional. This in fact accords with our account of types.

If we accept $a =_A b$ as a type, then we must decide what its canonical elements will be. At first sight it seems that we might need to enumerate all possible ways of proving equalities and consider the normal forms of these proofs. At this point one might argue that the meaning of an equality proposition can be simplified considerably without affecting the meaning of compound propositions. We might argue that the essential information about $a =_A b$ in terms of subsequent use is simply that it holds. This is indeed the position that Martin-Löf [26,27] arrived at and it is a position that we independently confirmed to be useful in our earlier work on constructive logic [2,11,13]. It is the position taken in this theory. We introduce a single new element ax which represents all of the useful information of any true equality statement.

This device of forcing $a =_A a'$ to be a type is somewhat artificial at first glance. We manufacture new elements called ax whose only purpose is to inhabit types which were themselves manufactured to correspond exactly to propositions. However unnatural this correspondence may at first seem at the atomic level, it does permit a very powerful unification of the logic and type theory, and it leads to a complete solution of the problem of assigning mathematical meaning to propositions. We can express the propositions-as-types correspondence with a few simple additional rules. First we replace the prop(i) notion by the universe type(i). We also identify false with void. We replace the judgement $A \in$ prop(i) by the assertion $A =_{type(i)} A$ (e.g. by $A \in$ type(i)).

6. Propositions-As-Types Rules

inhabitation yields truth

$$H \gg a =_A a'$$
$$\quad H \gg ax \in (a =_A a')$$

truth yields inhabitation

$$H \gg ax \in (a =_A a')$$
$$\quad H \gg (a =_A a')$$

In addition to these rules we use the logical operations as abbreviations for the type constructors according to the above table. We can also take the well-formedness judgement $A \in prop(i)$ to correspond to the proposition $A =_{U(i)} A$.

7. Reduction of Judgements

The form of the logic can be made more conventional if judgements of the form $a \in A$ could be construed as assertions. This is quite possible if we take $a \in A$ to mean $a =_A a$. By reading all of the above rules of the form $a \in A$ as $a =_A a$ we can see that only one form of judgement is necessary, namely the judgement of truth. While this reduction is possible, it does not greatly simplify the logic. In fact, it has the undesirable affect of allowing absurd assumptions to spread into the deduction of well-formedness. It becomes entirely possible to conclude $z{:}false \gg 2 \in prop(i)$.

V. FORMAL TYPE THEORETIC SEMANTICS

1. Forms of Judgement

Once we have recognized that propositions can be treated as types, it is possible to imagine presenting a type theory which in a sense subsumes logic. Another way to say this is in terms of the categories of judgement used in section II. The rules are there stated in terms of the judgement that an assertion is true, say H >> A and in terms of the judgement that a term is of a certain type, either H >> $A \in$ prop(i) or H >> $a \in A$. We know how to reduce both forms of judgement to assertion, namely treat $a \in A$ as $a =_A a$ and reduce prop(i) to universe type(i) so that $A \in$ prop(i) becomes $A =_{U(i)} A$.

The propositions-as-types principle also allows us to reduce in the other way, namely assertion can be construed as a type judgement. To assert A with proof a is to judge that $a \in A$. Per Martin-Löf [26,27] has created a type theory based on judgements of these four forms:

A is a type
A and B are equal types
a is of type A
a and b are equal elements of A

He symbolizes these as A type, A=B, $a \in A$, $a=b \in A$ respectively.

The theory presented in section II can be understood as a type theory based on fewer judgements, namely on $a \in A$ with the "A type" being reduced to $A \in$ type(i), with A=B reduced to $A =_{type(i)} B$ and $a=b \in A$ reduced to $ax \in (a =_A b)$ where $a =_A b$ is a type.

The logic presented to the user can be seen as a presentation of this type theory in which the term a in the judgement $a \in A$ can be suppressed. A careful derivation of the logic from the type theory can be found in [21]. We present the complete logic in its type theoretic form in Appendix A.

VI. PROOF TACTICS

1. General

As in the λPRL system, a great deal of the expressive power of ∨PRL comes from a formalization of the metatheory in the programming language ML [19]. The mechanism of interaction is the same in both theories, but the details of the syntactic types are different.

The rich type structure of ∨PRL suggests that the metamathematics can be modeled naturally in a programming language which is essentially a fragment of PRL itself. A method of doing this is discussed in [12].

2. Type Checking Tactics and Conservative Mode

The standard way of proving theorems in mathematics is to state only well-formed formulas as candidates for theoremhood. The well-formedness argument takes place in the metatheory. In order to mimic this mode of operation in ∨PRL, we build proof tactics which take readable formulas as input and attempt to establish that they are well-formed. In most cases, such proofs are straightforward. The result of such a tactic is a proof of the judgement $F \in type(i)$ (or what is the same, $F =_{type(i)} F$). This judgement is recorded in the library by storing the entry as

	t PROP(i)	or as	t TYPE(i)
rather than as			
	t FORM		

Once it is known that a goal is a well-formed formula, then all of the subgoals marked by * in the rules can be ignored.

Acknowledgements

We would like to thank Stuart Allen and Bob Harper for numerous suggestions which have shaped the logic and improved the presentation of the rules.

We also thank Donette Isenbarger for preparing the manuscript in its many versions.

References

[1] Aho, Alfred V., J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974, 470pp.

[2] Bates, J.L., A Logic for Correct Program Development, Ph.D. Thesis, Department of Computer Science, Cornell University, 1979.

[3] Bates J. and R.L. Constable, "Definition of Micro-PRL", Technical Report TR 82-492, Computer Science Department, Cornell University, October 1981.

[4] Bates, J.L. and R.L. Constable, "Proofs as Programs", Dept. of Computer Science Technical Report, TR 82-530, Cornell University, Ithaca, NY, 1982.

[5] Bibel, W., "Syntax-Directed, Semantics-Supported Program Synthesis", Artificial Intelligence, 14, 1980, pp. 243-261.

[6] Bishop, E., Foundations of Constructive Analysis, McGraw Hill, NY, 1967, 370 pp.

[7] Bridges, D.S., Constructive Functional Analysis, Pitman, London, 1979.

[8] Brouwer, L.E.J., "On the Significance of the Principle of Excluded Middle in Mathematics, Especially in Function Theory", J. für die reine und angewandte Math, 154, 1923, pp. 1-7 in From Frege to Gödel, ed. J. van Heijenoort, Harvard University Press, Cambridge, 1967, pp. 334-345.

[9] Constable, Robert L., "Constructive Mathematics as a Programming Logic I: Some Principles of Theory", Cornell University Technical Report TR83-554, May 1983 (to appear in Proc. of FCT Conf., Springer-Verlag, 1983).

[10] Constable, Robert L., "Partial Functions in Constructive Formal Theories", Proc. of 6th G.I. Conference, Lecture Notes in Computer Science, Vol. 45, Springer-Verlag, 1983.

[11] Constable, Robert L. and D.R. Zlatin, "The Type Theory of PL/CV3", IBM Logic of Programs Conference, Lecture Notes in Computer Science, Vol. 131, Springer-Verlag, NY, 1982, 72-93.

[12] Constable, R., T. Knoblock, and J. Bates, "On Writing Programs to Find Proofs" report, Computer Science Department, Cornell University, 1983.

[13] Constable, Robert L., S.D. Johnson and C.D. Eichenlaub, Introduction to the PL/CV2 Programming Logic, Lecture Notes in Computer Science, Vol. 135, Springer-Verlag, NY, 1982.

[14] Conway, R.W., and D. Gries, An Introduction to Programming: A Structured Approach, Little, Brown & Co., 1979.

[15] deBruijn, N.G., "A Survey of the Project AUTOMATH", Essays on Combinatory Logic, Lambda Calculus and Formalism, (eds. J.P. Seldin and J.R. Hindley), Academic Press, NY, 1980, 589-606.

[16] Fitting, M., Proof Methods for Model and Intuitionistic Logics, D. Reidel, Dordrecht, 1983.

[17] Frege, G., "Begriffsschrift, A Formula Language, Modeled Upon that for Arithmetic, for Pure Thought", Halle, 1879 (reprinted in From Frege Gödel, ed. J. van Heijenoort, Harvard University Press, Cambridge, 1967, pp. 1-82).

[18] Gentzen, G., "Investigations Into Logical Deduction", pp. 68-131, (original 1935) reprinted in The Collected Papers of Gerhard Gentzen, North-Holland, Amsterdam, 1969 (ed. M.E. Szabo).

[19] Gordon, M., R. Milner and C. Wadsworth, Edinburgh LCF: A Mechanized Logic of Computation, Lecture Notes in Computer Science, Vol. 78, Springer-Verlag, 1979.

[20] Gries, David, The Science of Programming, Springer-Verlag, 1982.

[21] Harper, R., "Formalization of Type Theory", report, Computer Science Department, Cornell University, 1983.

[22] Heyting, A., Die formalen Regeln der intuitionistischen Logik, Sitzungsberichte der Preussischen Akademie der Wissenschaften, Phipikalisch - mathematische Klass, 1930, pp. 42-56.

[23] Howard, W.A., "The Formulas-As-Types Notion of Construction" in Essays on Combinatory Logic, Lambda Calculus and Formalism, (eds., J.P. Seldin and J.R. Hindley), Academic Press, NY, 1980.

[24] Kleene, S.C., Introduction to Metamathematics, D. Van Nostrand, Princeton, 1952, 550 pp.

[25] Manna, Z. and R. Waldinger, "A Deductive Approach to Program Synthesis", Department of Computer Science Technical Report, STAN-CS-78-690, Stanford University, Stanford, CA, 1978.

[26] Martin-Löf, Per, "An Intuitionistic Theory of Types: Predicative Part", Logic Colloquium, 1973, (eds. H.E. Rose and J.C. Shepherdson), North-Holland, Amsterdam, 1975, 73-118.

[27] Martin-Löf, Per, "Constructive Mathematics and Computer Programming", 6th International Congress for Logic, Method and Phil. of Science, North-Holland, Amsterdam, 1982.

[28] Nelson, Greg, "Techniques for Program Verification", Ph.D. Thesis, Stanford University, Stanford, CA, 1980. (Also Technical Report CSL-81-10, Xerox, June 1981.)

[29] Nordstrom, B., "Programming in Constructive Set Theory: Some Examples", Proc. 1981 Conf. on Functional Prog. Lang. and Computer Archi, Portsmouth, 1981, 141-153.

[30] Petersson, K., "A Programming System for Type Theory", LPM Memo 21, Dept. of Computer Science, University of Goteborg/Chalmers University of

Technology, Goteborg, Sweden, 1982.

[31] Proofrock, J.A., "PRL: Proof Refinement Logic Programmer's Manual (Lambda PRL VAX Version), Computer Science Department, Cornell University, 1983.

[32] Reynolds, John C, "Towards a Theory of Type Structure", Proc. Colloque su, la Programmation, Lecture Notes in Computer Science, 19, Springer-Verlag, pp. 408-425, 1974.

[33] Russell, B., Mathematical Logic as Based on a Theory of Types, Am. J. of Math., 30, 1908, pp. 222-262.

[34] Scherlis, W.L. and D.S. Scott, "First Steps Toward Inferential Programming", Proc. IFIP Congress, Paris, 1983.

[35] Scott, Dana, "Data Types as Lattices", SIAM Journal on Computing, 5:3, September, 1976.

[36] Scott, Dana, "Constructive Validity", Symposium on Automatic Demonstration, Lecture Notes in Mathematics, 125, Springer-Verlag, 1970, 237-275.

[37] Tarski, A., "The Semantic Conception of Truth and the Foundations of Semantics", Philos and Phenom. Res., 4, 1944, pp. 341-376.

[38] van Wijngaarden, A.B.J. et al, Revised Report on the Algorithmic Language ALGO 68, Supplement to ALGO BULLETIN, University of Alberta, 1974.

[39] Weyhrauch, R., "Prolegomena to a Theory of Formal Reasoning", Artificial Intelligence, 13, 1980, pp. 133-170.

APPENDIX A

The following table is a translation between the notation used in the ν-PRL implementation and the logic of the appendix.

For...	*Read...*
U_i	$ty(i)$
$decide(e; x.t_1; y.t_2)$	*if* e *left* $x.t_1$ *right* $y.t_2$ *fi*
$spread(e; x, y.t_2)$	$\$\, e\ \mathit{over}\ x, y.t\, \$$

VOID

Formation

$H \gg void \in U_k \langle l \rangle$ by void form $[k<l]$

Introduction

(None)

Elimination

$H \gg libet(e) \in T(e)_z \langle k \rangle$ by void elim over T_z
$H \gg e \in void \langle i \rangle$
$H, z: void \langle i \rangle \gg T \in U_k \langle l \rangle$

Equality

$H \gg libet(e) = libet(e') \in T(e)_z \langle k \rangle$ by libet eq over T_z
$H \gg e = e' \in void \langle i \rangle$
$H, z: void \langle i \rangle \gg T \in U_k \langle l \rangle$

Computation

(None)

NATURAL NUMBERS

Formation

$H \gg N \in U_k \langle l \rangle$ by N form $[k<l]$

Introduction

$H \gg 0 \in N \langle k \rangle$ by N intro

$H \gg \sigma n \in N \langle k \rangle$ by N intro
$\quad H \gg n \in N \langle k \rangle$

Elimination

$H \gg ind(n; t_1; x, y.t_2) \in T(n)_x \langle k \rangle$ by induction over T_x
$\quad H \gg n \in N \langle i \rangle$
$\quad H \gg t_1 \in T(0)_x \langle k \rangle$
$\quad H, x{:}\, N \langle i \rangle, y{:}\, T \langle k \rangle \gg t_2 \in T(\sigma x)_x \langle k \rangle$
$\quad \ddagger H, x{:}\, N \langle i \rangle \gg T \in U_k \langle l \rangle$

Equality

$H \gg \sigma n = \sigma n' \in N \langle k \rangle$ by σ eq
$\quad H \gg n = n' \in N \langle k \rangle$

$H \gg ind(n; t_1; x, y.t_2) = ind(n'; t_1'; x, y.t_2') \in T(n)_x \langle k \rangle$ by ind eq over T_x
$\quad H \gg n = n' \in N \langle i \rangle$
$\quad H \gg t_1 = t_1' \in T(0)_x \langle k \rangle$
$\quad H, x{:}\, N \langle i \rangle, y{:}\, T \langle k \rangle \gg t_2 = t_2' \in T(\sigma x)_x \langle k \rangle$
$\quad \ddagger H, x{:}\, N \langle i \rangle \gg T \in U_k \langle l \rangle$

Computation

$H \gg ind(0; t_1; x, y.t_2) = t_1 \in T(0)_x \langle k \rangle$ by ind red over T_x
$\quad H, x{:}\, N \langle i \rangle \gg ind(x; t_1; x, y.t_2) \in T \langle k \rangle$

$H \gg ind(\sigma n; t_1; x, y.t_2) = t_2(n, ind(n; t_1; x, y.t_2))_{x,y} \in T(\sigma n)_x \langle k \rangle$ by ind red over T_x
$\quad H \gg \sigma n \in N \langle i \rangle$
$\quad H, x{:}\, N \langle i \rangle \gg ind(x; t_1; x, y.t_2) \in T \langle k \rangle$

Formation

$H \gg A \mid B \in U_k \langle l \rangle$ by union form $[k<l]$
$H \gg A \in U_k \langle l \rangle$
$H \gg B \in U_k \langle l \rangle$

Introduction

$H \gg inl(a) \in A \mid B \langle k \rangle$ by union intro
$H \gg a \in A \langle k \rangle$
$H \gg B \in U_k \langle l \rangle$

$H \gg inr(b) \in A \mid B \langle k \rangle$ by union intro
$H \gg b \in B \langle k \rangle$
$H \gg A \in U_k \langle l \rangle$

Elimination

$H \gg decide(e; x.t_1; y.t_2) \in T(e)_z \langle k \rangle$ by union elim in $A \mid B$ over T_z
$H \gg e \in A \mid B \langle i \rangle$
$H, x{:}\, A \langle i \rangle \gg t_1 \in T(inl(x))_z \langle k \rangle$
$H, y{:}\, B \langle i \rangle \gg t_2 \in T(inr(y))_z \langle k \rangle$

Equality

$H \gg A \mid B = A' \mid B' \in U_k \langle l \rangle$ by union eq $[k<l]$
$H \gg A = A' \in U_k \langle l \rangle$
$H \gg B = B' \in U_k \langle l \rangle$

$H \gg inl(a) = inl(a') \in A \mid B \langle k \rangle$ by inl eq
$H \gg a = a' \in A \langle k \rangle$
$H \gg B \in U_k \langle l \rangle$

$H \gg inl(b) = inl(b') \in A \mid B \langle k \rangle$ by inl eq
$H \gg b = b' \in B \langle k \rangle$
$H \gg A \in U_k \langle l \rangle$

$H \gg decide(e; x.t_1; y.t_2) = decide(e'; x.t_1'; y.t_2') \in T(e)_z \langle k \rangle$ by decide eq in $A \mid B$ over T_z
$H \gg e = e' \in A \mid B \langle i \rangle$
$H, x{:}\, A \langle i \rangle \gg t_1 = t_1' \in T(inl(x))_z \langle k \rangle$
$H, y{:}\, B \langle i \rangle \gg t_2 = t_2' \in T(inr(y))_z \langle k \rangle$

Computation

$H \gg decide(inl(a); x.t_1; y.t_2) = t_1(a)_x \in T(inl(a))_z \langle k \rangle$ by decide red in $A \mid B$ over T_z
$\quad H \gg inl(a) \in A \mid B \langle i \rangle$
$\quad H, z{:}A \mid B \langle i \rangle \gg decide(z; x.t_1; y.t_2) \in T \langle k \rangle$

$H \gg decide(inr(b); x.t_1; y.t_2) = t_2(b)_y \in T(inr(b))_z \langle k \rangle$ by decide red in $A \mid B$ over T_z
$\quad H \gg inr(b) \in A \mid B \langle i \rangle$
$\quad H, z{:}A \mid B \langle i \rangle \gg decide(z; x.t_1; y.t_2) \in T \langle k \rangle$

PRODUCT

Formation

$H \gg x{:}A \# B \in U_k \langle l \rangle$ **by product form** $[k<l]$

$\quad H \gg A \in U_k \langle l \rangle$

$\quad H, x{:}\, A \langle k \rangle \gg B \in U_k \langle l \rangle$

Introduction

$H \gg \langle a\ b \rangle \in x{:}A \# B \langle k \rangle$ **by product intro**

$\quad H \gg a \in A \langle k \rangle$

$\quad H \gg b \in B(a)_x \langle k \rangle$

$\quad H, x{:}\, A \langle k \rangle \gg B \in U_k \langle l \rangle$

Elimination

$H \gg spread(p; x, y.t) \in T(p)_z \langle k \rangle$ **by product elim in** $x{:}A \# B$ **over** T_z

$\quad H \gg p \in x{:}A \# B \langle i \rangle$

$\quad H, x{:}\, A \langle i \rangle, y{:}\, B \langle i \rangle \gg t \in T(\langle x\ y \rangle)_z \langle k \rangle$

Equality

$H \gg x{:}A \# B = x{:}A' \# B' \in U_k \langle l \rangle$ **by product eq** $[k<l]$

$\quad H \gg A = A' \in U_k \langle l \rangle$

$\quad H, x{:}\, A \langle k \rangle \gg B = B' \in U_k \langle l \rangle$

$H \gg \langle a\ b \rangle = \langle a'\ b' \rangle \in x{:}A \# B \langle k \rangle$ **by pair eq**

$\quad H \gg a = a' \in A \langle k \rangle$

$\quad H \gg b = b' \in B(a)_x \langle k \rangle$

$\quad H, x{:}\, A \langle k \rangle \gg B \in U_k \langle l \rangle$

$H \gg spread(e; x, y.t) = spread(e'; x, y.t') \in T(e)_z \langle k \rangle$ **by spread eq in** $x{:}A \# B$ **over** T_z

$\quad H \gg e = e' \in x{:}A \# B \langle i \rangle$

$\quad H, x{:}\, A \langle i \rangle, y{:}\, B \langle i \rangle \gg t = t' \in T(\langle x\ y \rangle)_z \langle k \rangle$

Computation

$H \gg spread(\langle a\ b \rangle; x, y.t) = t(a, b)_{x,y} \in T(\langle a\ b \rangle)_z \langle k \rangle$ **by spread red in** $x{:}A \# B$ **over** T_z

$\quad H \gg \langle a\ b \rangle \in x{:}A \# B \langle i \rangle$

$\quad H, z{:}\, (x{:}A \# B) \langle i \rangle \gg spread(z; x, y.t) \in T \langle k \rangle$

FUNCTION

Formation

$H \gg x{:}A \rightarrow B \in U_k \langle l \rangle$ **by arrow form** $[k<l]$
  $H \gg A \in U_k \langle l \rangle$
  $H, x{:}\,A \langle k \rangle \gg B \in U_k \langle l \rangle$

Introduction

$H \gg \lambda x.b \in x{:}A \rightarrow B \langle k \rangle$ **by arrow intro**
  $H, x{:}\,A \langle k \rangle \gg b \in B \langle k \rangle$
  $H \gg A \in U_k \langle l \rangle$

Elimination

$H \gg f(a) \in B(a)_x \langle k \rangle$ **by arrow elim**
  $H \gg f \in x{:}A \rightarrow B \langle k \rangle$
  $H \gg a \in A \langle k \rangle$

Equality

$H \gg x{:}A \rightarrow B = x{:}A' \rightarrow B' \in U_k \langle l \rangle$ **by arrow eq** $[k<l]$
  $H \gg A = A' \in U_k \langle l \rangle$
  $H, x{:}\,A \langle k \rangle \gg B = B' \in U_k \langle l \rangle$

$H \gg \lambda x.b = \lambda x.b' \in x{:}A \rightarrow B \langle k \rangle$ **by lambda eq**
  $H, x{:}\,A \langle k \rangle \gg b = b' \in B \langle k \rangle$
  $H \gg A \in U_k \langle l \rangle$

$H \gg f(a) = f'(a') \in B(a)_x \langle k \rangle$ **by application eq in** $x{:}A \rightarrow B$
  $H \gg f = f' \in x{:}A \rightarrow B \langle k \rangle$
  $H \gg a = a' \in A \langle k \rangle$

Computation

$H \gg (\lambda x.b)(a) = b(a)_x \in B(a)_x \langle k \rangle$ **by lambda red in** $x{:}A \rightarrow B$
  $H \gg \lambda x.b \in x{:}A \rightarrow B \langle k \rangle$
  $H \gg a \in A \langle k \rangle$

QUOTIENT

Formation

$H \gg A/E \in U_k \langle l \rangle$ by quotient form $[k<l]$

$H \gg A \in U_k \langle l \rangle$

$H \gg E \in A \to A \to U_k \langle l \rangle$

$H, x{:}\, A \langle k \rangle \gg e_1 \in E(x)(x) \langle k \rangle$

$H, x{:}\, A \langle k \rangle, y{:}\, A \langle k \rangle, u{:}\, E(x)(y) \langle k \rangle \gg e_2 \in E(y)(x) \langle k \rangle$

$H, x{:}\, A \langle k \rangle, y{:}\, A \langle k \rangle, u{:}\, E(x)(y) \langle k \rangle, z{:}\, A \langle k \rangle, v{:}\, E(y)(z) \langle k \rangle \gg e_3 \in E(x)(z) \langle k \rangle$

Introduction

$H \gg a \in A/E \langle k \rangle$ by quotient intro

$H \gg a \in A \langle k \rangle$

$H \gg A/E \in U_k \langle l \rangle$

Elimination

$H \gg t(e)_x \in T(e)_x \langle k \rangle$ by quotient elim in A/E over t_x in T_x

$H \gg e \in A/E \langle i \rangle$

$H, x{:}\, A \langle i \rangle \gg t \in T \langle k \rangle$

$H, x{:}\, A \langle i \rangle, y{:}\, A \langle i \rangle, z{:}\, E(x)(y) \langle i \rangle \gg t = t(y)_x \in T \langle k \rangle$

$H, x{:}\, A/E \langle i \rangle \gg T \in U_k \langle l \rangle$

Equality

$H \gg A/E = A'/E' \in U_k \langle l \rangle$ by quotient eq $[k<l]$

$H \gg A = A' \in U_k \langle l \rangle$

$H \gg A/E \in U_k \langle l \rangle$

$H \gg A'/E' \in U_k \langle l \rangle$

$H, z{:}\, A = A' \in U_k \langle l \rangle, x{:}\, A \langle k \rangle, y{:}\, A \langle k \rangle \gg e \in E(x)(y) \to E'(x)(y) \langle k \rangle$

$H, z{:}\, A = A' \in U_k \langle l \rangle, x{:}\, A \langle k \rangle, y{:}\, A \langle k \rangle \gg e' \in E'(x)(y) \to E(x)(y) \langle k \rangle$

$H \gg a = a' \in A/E \langle k \rangle$ by quotient member eq

$H \gg a \in A/E \langle k \rangle$

$H \gg a' \in A/E \langle k \rangle$

$H \gg e \in E(a)(a') \langle k \rangle$

Computation

(None)

SET

Formation

$H \gg \{x{:}\,A \mid B\} \in U_k \langle l \rangle$ **by set form** $[k<l]$

$\quad H \gg A \in U_k \langle l \rangle$

$\quad H, x{:}\,A \langle k \rangle \gg B \in U_k \langle l \rangle$

Introduction

$H \gg a \in \{x{:}\,A \mid B\} \langle k \rangle$ **by set intro**

$\quad H \gg a \in A \langle k \rangle$

$\quad H \gg b \in B(a)_x \langle k \rangle$

$\quad H, x{:}\,A \langle k \rangle \gg B \in U_k \langle l \rangle$

Elimination

$H \gg t(a)_x \in T(a)_x \langle k \rangle$ **by set elim in** $\{x{:}\,A \mid B\}$ **over** t_x **in** T_x

$\quad H \gg a \in \{x{:}\,A \mid B\} \langle i \rangle$

$\quad H, x{:}\,A \langle i \rangle, y{:}\,B \langle i \rangle \gg t \in T \langle k \rangle$ $[y$ new; no free y in $t]$

Equality

$H \gg \{x{:}\,A \mid B\} = \{x{:}\,A' \mid B'\} \in U_k \langle l \rangle$ **by set eq**

$\quad H \gg A = A' \in U_k \langle l \rangle$

$\quad H, x{:}\,A \langle k \rangle \gg B = B' \in U_k \langle l \rangle$

$H \gg a = a' \in \{x{:}\,A \mid B\} \langle k \rangle$ **by set element eq**

$\quad H \gg a \in \{x{:}\,A \mid B\} \langle k \rangle$

$\quad H \gg a' \in \{x{:}\,A \mid B\} \langle k \rangle$

$\quad H \gg a = a' \in A \langle k \rangle$

Computation

(None)

UNIVERSES

Formation

$H \gg U_i \in U_j \langle k \rangle$ **by universe form** $[i<j<k]$

Introduction

(All formation rules are universe introduction rules.)

$H \gg A \in U_i \langle k \rangle$ **by cumulativity** $[j < i]$ $[i<k]$
$\quad H \gg A \in U_j \langle l \rangle$

Elimination

$H \gg case_i(u; t_1; \ldots; u.t_{10}) \in T(u)_z \langle k \rangle$ **by universe elim on** u
$\quad H \gg u \in U_i \langle j \rangle$
$\quad H \gg t_1 \in T(void)_z \langle k \rangle$
$\quad H \gg t_2 \in T(N)_z \langle k \rangle$
$\quad H, v: U_i \langle j \rangle, w: U_i \langle j \rangle \gg t_3 \in T(v \mid w)_z \langle k \rangle$
$\quad H, v: U_i \langle j \rangle, f: v \to U_i \langle j \rangle \gg t_4 \in T(x{:}v \# f(x))_z \langle k \rangle$
$\quad H, v: U_i \langle j \rangle, f: v \to U_i \langle j \rangle \gg t_5 \in T(x{:}v \to f(x))_z \langle k \rangle$
$\quad H, v: U_i \langle j \rangle, f: v \to U_i \langle j \rangle \gg t_6 \in T(\{x: v \mid f(x)\})_z \langle k \rangle$
$\quad H, v: U_i \langle j \rangle, e: v \to v \to U_i \langle j \rangle, * \gg t_7 \in T(v/e)_z \langle k \rangle$
$\quad H, v: U_i \langle j \rangle, x_1: v, x_2: v \gg t_8 \in T(x_1 = x_2 \in v)_z \langle k \rangle$
$\quad H \gg t_9 \in T(U_j)_z \langle k \rangle$ [for each $1 \le j < i$]
$\quad H, z: U_i \langle j \rangle \gg t_{10} \in T \langle k \rangle$

insert at $*$:
$\quad z_1: v \langle i \rangle, z_2: v \langle i \rangle, z_3: v \langle i \rangle,$
$\quad a_1: e(z_1)(z_1) \langle i \rangle,$
$\quad a_2: e(z_1)(z_2) \to e(z_2)(z_1) \langle i \rangle,$
$\quad a_3: e(z_1)(z_2) \# e(z_2)(z_3) \to e(z_1)(z_3) \langle i \rangle$

Equality

$H \gg case_i(u; \ldots) = case_i(u'; \ldots) \in T(u)_z \langle k \rangle$ **by case eq over** T_z
$\quad H \gg u = u' \in U_i \langle j \rangle$
$\quad \ldots$

Computation

$H \gg case_i(void; t_1; \ldots) = t_1 \in T(void)_z \langle k \rangle$ **by case red in** U_i **over** T_z
$\quad H, z: U_i \langle j \rangle \gg case_i(z; \ldots) \in T \langle k \rangle$
$\quad H \gg void \in U_i \langle j \rangle$

$\ldots$

EQUALITY TYPE

Formation

$H \gg (a = a' \in A) \in U_i \langle k \rangle$ **by equality form** $[i<k]$

$\quad H \gg a \in A \langle i \rangle$

$\quad H \gg a' \in A \langle i \rangle$

$\quad \dagger H \gg A \in U_i \langle k \rangle$

Introduction

$H \gg ax \in (a = a' \in A) \langle k \rangle$ **by equality intro**

$\quad H \gg a = a' \in A \langle k \rangle$

Elimination

$H \gg a = a' \in A \langle k \rangle$ **by equality elim**

$\quad H \gg e \in (a = a' \in A) \langle k \rangle$

Equality

$H \gg (a = b \in A) = (a' = b' \in A') \in U_k \langle l \rangle$ **by equality type eq** $[k<l]$

$\quad H \gg a = a' \in A \langle k \rangle$

$\quad H \gg b = b' \in A \langle k \rangle$

$\quad H \gg A = A' \in U_k \langle l \rangle$

Computation

(None)

GENERAL RULES

Equality

$H \gg b(a)_x = b'(a')_x \in B(a)_x \langle k \rangle$ **by substitution b_x for b'_x in B_x over A**
$\quad H \gg a = a' \in A \langle i \rangle$
$\quad H, x{:}\, A \langle i \rangle \gg b = b' \in B \langle k \rangle$

$H \gg a = a' \in A \langle k \rangle$ **by symmetry** [Open derivative]
$\quad H \gg a' = a \in A \langle k \rangle$

$H \gg a = a'' \in A \langle k \rangle$ **by transitivity via a'** [Open derivative]
$\quad H \gg a = a' \in A \langle k \rangle$
$\quad H \gg a' = a'' \in A \langle k \rangle$

$H \gg a = b \in A \langle i \rangle$ **by equaltypes B**
$\quad H \gg A = B \in U_i \langle k \rangle$
$\quad H \gg a = b \in B \langle k \rangle$

Families

$H \gg b(a)_x \in B(a)_x \langle k \rangle$ **by specialization of b_x in A over B_x**
$\quad H \gg a \in A \langle i \rangle$
$\quad H, x{:}\, A \langle i \rangle \gg b \in B \langle k \rangle$

Miscellaneous

$H \gg a \in A \langle k \rangle$ **by equality restriction** [Open derivative]
$\quad H \gg a = a' \in A \langle k \rangle$

$H \gg A \in U_i \langle k \rangle$ **by inhabitation** $[i<k]$
$\quad H \gg a = a' \in A \langle i \rangle$

$H \gg a \in A \langle k \rangle$ **by universe cumulativity** $[k<l]$
$\quad H \gg a \in A \langle l \rangle$

$H_1, x{:}\, A \langle i \rangle, H_2 \gg x \in A \langle i \rangle$ **by hypothesis**

SUBTYPE PRINCIPLES

Equality type

$H \gg a \in A \langle k \rangle$ **by eq type elim using** a'
 $H \gg (a = a' \in A) \in U_k \langle l \rangle$

$H \gg a' \in A \langle k \rangle$ **by eq type elim using** a
 $H \gg (a = a' \in A) \in U_k \langle l \rangle$

$H \gg A \in U_k$ **by eq type elim using** a, a' [Open derivative]
 $H \gg (a = a' \in A) \in U_k \langle l \rangle$

Union, product, function, and set

$H \gg A \in U_k \langle l \rangle$ **by subtype**
 $H \gg A \mid B \in U_k \langle l \rangle$ *or,*
 $H \gg B \mid A \in U_k \langle l \rangle$ *or,*
 $H \gg x{:}A \# B \in U_k \langle l \rangle$ *or,*
 $H \gg A \rightarrow B \in U_k \langle l \rangle$ *or,*
 $H \gg \{x{:}A \mid B\} \in U_k \langle l \rangle$

$H, x{:}A \gg B \in U_k \langle l \rangle$ **by subtype**
 $H \gg x{:}A \# B \in U_k \langle l \rangle$ *or,*
 $H \gg A \rightarrow B \in U_k \langle l \rangle$ *or,*
 $H \gg \{x{:}A \mid B\} \in U_k \langle l \rangle$

Quotient

$H \gg A \in U_k \langle l \rangle$ **by subtype**
 $H \gg A/E \in U_k \langle l \rangle$

$H \gg E \in A \rightarrow A \rightarrow U_k \langle l \rangle$ **by subtype**
 $H \gg A/E \in U_k \langle l \rangle$