The Definition of $\mu p r l$

Joseph L. Bates  
Robert L. Constable

TR 82-492  
October 1981

Department of Computer Science  
Cornell University  
Ithaca, New York  14853

This research was supported by the National Science Foundation under grant NSF-80-03349
The Definition of \$\text{apr1}\$

Joseph L. Bates

Robert L. Constable

Department of Computer Science
Cornell University

TR 82-492

PRELIMINARY DOCUMENT -- October 7, 1981

This research was supported by the National Science Foundation under grant NSF-80-03349.
1. **Introduction**

μprl is a small type theory designed as a first attempt at a complete PRL logic. PRL is the latest project in a series of efforts to allow the formalization of computational reasoning (PL/CV2 [CO 78], PL/CV3 [CZ 80]). We have created this theory as a concise, understandable foundation from which to continue our work. This will include construction of an automated environment to support the development and maintenance of proofs.

μprl is based very heavily on the research of Martin-Lof [Ma 79] and earlier work by Constable and Zlatin [CZ 81]. It differs from the former, most significantly, in taking an intensional view of functions and types. It differs from the latter in the approach to intensionality and in the style of proofs, which follow the refinement paradigm advocated by Bates [Ba 79].

Though we are striving for the understandable foundation mentioned above, the logic remains in flux. We frequently discover "new" properties, some consistent with our intuitions, some surprising, and some disturbing. Thus, we present the formalism as a subtle construction, open to exploration. We welcome all comments.

1.1. **What is the Logic About?**

μprl is a formalism for describing computations and for reasoning about them. There are ways of writing

1. values -- these are called **canonical terms.**
2. expressions that produce values when executed -- these are called **non-canonical terms.**
3. relationships between the values of expressions -- these are called **formulas.**

The logic claims certain formulas as true and claims that a variety of inference rules are valid. The main use of the formalism is to allow the derivation of interesting properties of terms. Because of the logic's great expressive power, this may be thought of as developing correct programs, or as proving mathematical theorems.

μprl does not look like a predicate calculus. There are no quantifiers and no propositional connectives. The main concept of the logic is the classification of terms into types. The language for writing types is very rich, so they act as specifications, expressing properties of terms. Proving that a term is in a type is a way of saying that the term, perhaps a very complex expression, evaluates to a useful value.

An implementation of the logic should provide both help in proving formulas and an evaluator for expressions. Thus, once a proof is completed, perhaps showing that an expression yields an interesting object, the expression may be evaluated to obtain that object. Further, because of the nature of the logic, we believe that proofs can often be carried out in such a way that the particular term satisfying the desired type (specification) need not be shown. Thus, programming can proceed as a process of reasoning about specifications (in the form of types), as suggested by Bates [Ba 79].

Some proofs will be carried out purely for their mathematical content -- to develop knowledge about what can be computed (or what is constructively
valid). Other proofs will be built to establish correct programs for real-world problems. We believe these activities are closely related in theory and in practice. The PRL system is intended to combine them into one.

1.2. About Understanding the Logic

In order to explain the logic, we will present ways of building terms and rules for proving formulas. The meanings of terms and formulas are implicit in the rules. They express precisely our beliefs about terms, so to understand the logic one must study the rules. However, to aid the reader in reaching this understanding, we present informal accounts of our insights. These informal accounts are meta-theory, i.e., discussions about the logic. Some meta-theory will be precise, it will look like mathematics; other meta-theory will be vague, our attempts to communicate intuition. We intend these discussions to be helpful, but finally, alone, the reader must come to understand the logic.

One might wonder whether it would be possible to provide a model for μprl as an aid to understanding. We believe that, essentially, the answer is no. Any model that captures the ideas presented here must be as complex as the theory. We admit that the rules are hard to grasp at first, but that is because they have content. The issue is how to present the content in as clear and elegant a form as possible. We have addressed ourselves to precisely this task; herein is the result.

2. The Structure of the Logic

μprl allows one to prove and accumulate formulas. Since formulas are built from terms, we begin with a discussion of the latter.

2.1. Terms

The simplest terms are variables and constants. Variables are written as strings of letters, e.g., alpha, cost, j. Constants take a variety of forms, some rather complicated, e.g., 5, red, <13,λx.blue>. All other terms are built from simpler terms in prescribed ways, e.g., <x,1>, suc(alpha). Much of this presentation is an explanation of ways for forming terms.

Terms with variables occurring free are called open. All other terms are closed. We intend that closed terms may always be evaluated (i.e., executed) to produce canonical terms (i.e., values). This intent is captured largely in the form of reduction rules, a particular sub-class of our inference rules. Open terms cannot, in general, be evaluated until values are supplied for their free variables. Whenever we introduce a variety of term, we will explain which variables free in sub-terms remain free in the complete term and which variables become bound.

μprl is based on the idea that we can conceive of several fundamentally distinct ways of building values. Each method of building is named by a type (or type schema). Thus, the natural numbers are associated with the type N; this is the type resulting from the concept of indefinite repetition of a "successor" operation. Another notion, that of packaging several known values together into a new value, results in the class of tuple types. These types
are written \([A, B]\) with elements of the form \(<a, b>\).

One source of the power of this theory is that types are treated as objects. There are canonical terms that denote types and non-canonical terms that describe type yielding computations. As with other terms, type terms may have free variables. An example of this in PASCAL notation might be

\[
\text{array lower..100 of } T
\]

where lower and \(T\) are somewhere established to be variables of type \(N\) and \(TYPE\), respectively.

2.2. Formulas

Having conveyed a general understanding of terms, we can now provide a similar understanding of formulas. A formula is either

\[
\begin{align*}
t_1 = t_2 = \ldots = t_n & \quad \text{called a typing,} \\
t_1 \approx t_2 \approx \ldots \approx t_n & \quad \text{called a similarity,}
\end{align*}
\]

where \(t_1, \ldots, t_n, T_1, \ldots, T_n\) are terms with \(t_1, \ldots, t_n\) denoting arbitrary values (possibly types), and \(T_1, T_2, \ldots, T_n\) denoting types. The terms in typings and similarities are unordered, we will permute them freely.

A proved typing, \(t_1 = \ldots = t_n \in T\), is intended to say that some value is in the type obtained by evaluating \(T\), and that each \(t_i\), when evaluated, will produce that value. We say that two values are equal just when they are written as the same sequence of marks (subject to \(\alpha\)-conversion as described below).

A proved similarity, \(T_1 \approx \ldots \approx T_n\), is intended to say that, when evaluated, every \(T_i\) has the same set of values as its members. Thus, \(x \in T_i \iff x \in T_j\). As stated, equality on types is intensional -- two types are equal when they look the same. We believe this is an appropriate view when dealing with types as computational objects. In particular, equality on types is decidable. Similarity is an extensional equality on types. It is useful when viewing types as classifiers.

2.3. Bindings

During proofs, we will often want to demonstrate relations between open formulas (e.g., an implication). Thus, we must explain what it means to have such a relation on open formulas. The particular relations allowed are explained in the section on "Proofs", below. For now we consider them abstractly.

Whenever we wish to express a relation in which free variables occur, we will state the type that each free variable ranges over. These statements are called bindings. They are written \(x : T\), where \(x\) is a variable and \(T\) is a type. The meaning of a relation \(R\) with bindings \(x_1 : T_1, \ldots, x_n : T_n\) is that \(R\) holds when \(x_1\) is replaced by any element of \(T_1\), \ldots, \(x_n\) is replaced by any element of \(T_n\), the substitutions occurring simultaneously.

We present more detail on bindings, together with an example, after discussing several notational conventions.
2.4. Notation

By \( t(x) \) we mean a term, \( t \), that may have free occurrences of the variable \( x \) (perhaps with other free variables as well). Given such a term, we write \( t(\exp/x) \) to mean the same term with \( \exp \) substituted for every occurrence of \( x \). Sometimes we write \( t(\exp) \) as an abbreviation, when it is clear which free variable of \( t \) is being instantiated.

We adopt the convention that \( x, y, z \) stand for variables; \( a, b, c, d, t, t', t_1, \ldots \) stand for terms; \( A, B, C, D, T, T', T_1, \ldots \) stand for types; \( F, G, H, H_1, \ldots \) stand for formulas; \( E \) stands for a binding. We use the notation \( \overline{q} \) to stand for a list of \( q' \)'s separated by commas, for any sort of form \( q \). However, the notation \( \equiv \overline{q} \) stands for a list of \( q' \)'s separated by equal signs and \( \equiv \overline{q} \) stands for a list of \( q' \)'s separated by similarity signs. Thus,

- \( \overline{H} \) stands for a list of formulas separated by commas,
- \( \equiv \overline{E} \) stands for a typing formula,
- \( \equiv T \) stands for a similarity formula.

We will be writing a variety of forms involving free variables and bindings. These include \( \lambda \)-terms, environment bindings, and (dependent) types. We take the position that the particular variables named in bindings are unimportant. Thus, we allow any uniform change of variables to occur (\( \alpha \)-conversion), subject to the constraint that no capture results.

We have defined a canonical form as one having itself as value. This means, e.g., that \( \lambda x.x \) evaluates to \( \lambda x.x \) and similarly for \( \lambda y.y \). But since these forms are to be indistinguishable in the logic, it certainly must be that their values are equal. Thus, our "blindness" to choice of variables extends to informal reasoning -- we agree that two values are the same, even when they are not identical sequences of marks, if they are \( \alpha \)-convertible.

2.5. Proofs

A proof is a formal demonstration that a single formula follows from a set of formulas. Formulas are proved by carrying out a process similar to top down programming. To show that \( F \) follows from hypotheses \( \overline{H} \) (possibly empty), one starts with the goal

\[ \overline{H} \vdash F. \]

This forms a (degenerate) tree of goals, called a refinement tree. A complete tree is finite and every node follows from its children by some refinement rule. In a partial tree, the leaves are unproved subgoals. A proof proceeds by repeatedly expanding leaves into subgoals by the application of rules. Since some rules generate no subgoals, this process can terminate.

The root of a proof is always of the form shown above. However, inside a proof we may want to show a goal in which certain variables occur free. Such a goal must be presented with bindings for the free variables. Thus, the general form of a goal is

\[ \overline{E}, \overline{H} \vdash F \]

where \( \overline{E} \) is a list of bindings, called the environment.
All variables that occur free in $\overline{H}$ or $F$ must be present on the left of some binding in $\overline{E}$. In addition, $\overline{E}$ is ordered and the type part of a binding can have free occurrences of variables that are bound earlier in the environment. Thus, for example, one may build the following sorts of goals:

$$x:T, xeT \vdash xeT$$
$$x:A, y:B(x), z:C(x,y), F(x,y,z) \vdash G(x,y,z)$$

Finally, no variable should occur as the left side of more than one binding. If this should seem to arise during application of a rule, an appropriate renaming of bound variables must be performed.

An example of a complete refinement (in full detail) is:

$$a=\text{beN} \vdash \text{suc}(a)=\text{suc}(b)\text{eN} \quad \text{by substitution}$$
$$a=\text{beN} \vdash a=\text{beN} \quad \text{by hypothesis}$$
$$x:\text{N}, a=\text{beN} \vdash \text{suc}(x)=\text{suc}(x)\text{eN} \quad \text{by equality}$$
$$x:\text{N}, a=\text{beN} \vdash \text{suc}(x)\text{eN} \quad \text{by N intro}$$
$$x:\text{N}, a=\text{beN} \vdash xeN \quad \text{by binding}$$

This tree has the structure:

```
    /
   /
  /
/
```

```
substitution
```

```
  /
  /
  /
```

```
hypothesis
```

```
  /
  /
```

```
equality
```

```
  /
  /
```

```
N intro
```

```
  /
  /
```

```
binding
```

We note several properties of goals. The hypotheses of goals are not ordered, we assume without further explanation that they may be re-ordered at will. The bindings of goals are ordered, but we allow re-orderings that maintain the property specified above (free variables in bindings are bound earlier in the environment). It should be clear that bindings are not formulas. In particular, they never appear to the right of $\vdash$.

We note several properties of refinement proofs. Every subgoal of a goal is independent, e.g., there is no notion of sequencing. Goals are restricted sequents, only one conclusion can appear. There is no notion of "accessibility of earlier formulas" (Constable and O'Donnell [CO 78]), the only formulas available in the proof of a goal's conclusion are those present in the hypothesis list.

The refinement style of proof is not well suited to manual manipulation with pencil and paper. However, we believe it may be an appropriate style in an automated, interactive environment ([Ba 79], [Kr 81]). One of the goals of the PRL project is to discover effective methods for interacting with machine supported logics.

2.6. Collecting Results -- The Library

There is a single structure of proved results. We call it the library. The library starts off empty and formulas are added to it as they are proved. At any time the library is a sequence of formulas (each justified by a proof
recorded elsewhere). Results are not named in any way, nor is there any notion of sub-library (e.g., book).

If the user can prove a refinement whose root is $\Delta \vdash F$ and if all the assumptions of $\Delta$ are present in the library, then the conclusion $F$ may be added to the library. Thus, each formula in the library comes from a refinement. Refinements are not actually present in the library (they have no semantics there), but they are used to justify the introduction of formulas into the library.

2.2. The Organization of the Rules

We are now nearly ready to present the refinement rules. As mentioned above, the logic is organized around a variety of types, i.e., methods for building objects. Each type gives rise to rules for reasoning about it.

In explaining a type, we must

1. introduce a name for the type (or type schema),
2. show how to form values (canonical terms) in the type,
3. show how to form expressions (non-canonical terms) for values in the type,
4. show how to form other expressions associated with the type,
5. give rules for reducing expressions to values.

The types are relatively independent, so each type is placed in a section with an informal intuitive account and associated formal rules. As in [Ma 79], for each rule, "the reader is asked to try to make the conclusion evident to himself on the presupposition that he knows the premises". However, we will provide some hints on how to accomplish this.

The rules are roughly divided in four classes: formation, introduction, elimination, and reduction rules. They are classified to aid our understanding. It is useful to do so because each sort of rule has certain properties. However, the classification does not follow definite rules, it is a matter of judgement.

Introduction rules show how to build the values of a type. All canonical terms are formed here, sometimes along with non-canonical terms. Introduction rules have the general form:

$E, \Delta \vdash \text{new-term}\in T$ by intro
: (subgoals to build parts of new-term)

Elimination rules show how to use an arbitrary value of a type. Expressions are formed that contain as a part a term of the type being discussed. These expressions are never canonical, they map "out of" the type. Elimination rules have the general form:

$E, \Delta, t\in T \vdash \text{new-term}\in \text{Some-Type}$ by elimination
: (subgoals to show new-term is well-formed)

Reduction rules tell how to evaluate an "elimination" term. When the parts of the elimination term are canonical, the reduction rule must apply. It may be stated so that it is applicable other times as well. Reduction rules have the general form:

\[\text{pprl definition}\]

October 7, 1981
\[ E, H \vdash \text{non-canonical-term=reduced-term}\in T \text{ by reduction} \]

(subgoals to show terms are well-formed)

Finally, formation rules are the introduction rules for TYPE (really, universes) -- they explain how to build types. They are spread throughout this document because we have chosen to present them with their associated intro, elim, and reduction rules. Formation rules have the general form:

\[ E, H \vdash \text{New-Type}\in \text{TYPE by formation} \]

(subgoals to build parts of New-Type)

In addition to presenting the rules, we must explain what terms are canonical and when variables become bound in terms. These issues are discussed immediately before each set of rules using a semi-formal notation. Each discussion is divided into sections on Canonicality and Binding.

The notation "VAL t" means that t is a canonical term of type T, given that t\in T is known. In sections labeled Canonicality, one finds unqualified assertions, such as VAL 0, and conditional assertions, such as VAL a \& VAL b \implies VAL <a,b>. The assertions can be considered a very simple system for proving VAL statements. A term t is canonical in T exactly when \vdash t\in T is provable and "VAL t" is demonstrable.

Whenever a term is constructed from sub-terms, all variables free in the sub-terms generally remain free in the entire term. However, the term is admitted into our world only by the grace of a refinement rule. In the Binding discussion preceding the refinement rule, it may be stated that certain variables free in parts become bound in the entire term. Thus, the understanding is that variables are to be considered free, unless a Binding discussion declares otherwise.

2. Universes: V1

We begin the process of explaining \$\text{prl}\$ types with a look at the type hierarchy. This order of presentation is intended to aid in understanding the logic, but the concept of a hierarchy is not necessary for most of the rules to be meaningful. The philosophically minded may want to read about the other types before finishing this section.

\$\text{prl}\$ is a system with types as objects. This means that types such as N and [N,N], i.e., N\times N, must be members of some "larger" type, say TYPE. Further, if types are to be objects, we must be able to form pairs of types and functions from TYPE to TYPE. This implies that [TYPE,TYPE] and TYPE\times TYPE must be types.

Since we will be explaining how to form structured types from simpler ones, it seems desirable to let TYPE be treated the same as other "lower" types. Then we will be able to build such types as TYPE\times TYPE using existing mechanisms. However, it is known that simply postulating TYPE\in TYPE can lead to severe difficulties, hence some other approach is necessary.

We adopt the approach of Martin-Lof [Ma 79], initially advocated by Russell and Whitehead in Principia Mathematica -- a hierarchy of types. TYPE is treated as a synonym for V1 and we postulate that V1\in V2. V1 and V2 are the first two universes. This process could proceed indefinitely, but because we
wish only to build structured types based on V1, we stop here.

Our attempt to keep the logic simple by stopping at V2 creates a small anomaly. Since we wish to say that all types are objects, and V2 cannot be an object (else if would need a type), we establish henceforth the following terminology. Whenever a term is in V1 or V2, i.e., t\in V1 or t\in V2, we say that the term denotes a type. Being a type is exactly being an element of some Vi, and thus, V2 is not a type. However, we allow V2 to appear explicitly on the right side of a typing, and whenever we speak of a type in that position, we admit the possibility that V2 might appear. This last, somewhat ugly, explanation would not be necessary if the hierarchy extended indefinitely. We expect this to be the case in full PRL.

In summary, the type theory starts with a form denoted V2. V2 is not a type, because it is not an element of any Vi. V2 is not even an object, because it is not a member of any type. Nonetheless, it is a useful form and we shall call elements of V2 large types. The first rule says that V2 is not empty.

\[ E, \bar{H} \vdash V1 \in V2 \quad \text{by V1 formation} \]

Since V1 \in V2, we know that V1 is a type, it is our first large type. We sometimes write V1 as TYPE and call elements of V1 small types. V1 is a canonical term, thus: VAL V1.

As with other types, V1 is both a classifier and an object. However, unlike many of them (the small types), it turns out that V1 (and the other large types) are nearly useless as objects because no functions can be defined on them (i.e., V2 \rightarrow V2 cannot be formed). As described above, the main reason for considering elements of V2 to be objects is that the description of the formalism is simplified.

Universes are not a necessary concept for every type theory. In fact, a theory without them is sufficiently powerful for most programming. However, by including at least small types as objects, users are able to build "program writing programs" -- a very important sort of extension to the logic.

At the end of this document is a diagram of "all" the objects and types in the system showing how they relate by \varepsilon and indicating what purposes they serve.

4. General Rules

There are refinement rules that are not associated with any type. They are divided into several classes.

4.1. Structural Rules

\[ E, H_1, \ldots, H_n \vdash H_i \quad \text{by hypothesis i (sometimes called projection)} \]

\[ E, \bar{H} \vdash F \quad \text{by sequence (sometimes called composition)} \]

\[ E, \bar{H} \vdash G \]

\[ E, H, G \vdash F \]
\[ E, x : T, H \vdash x \varepsilon T \] by binding

4.2. **Equality Rules**

\[ E, H \vdash \bar{a} \varepsilon T \] by equality

\[ E, H \vdash =5 i \varepsilon T \]

\[ E, H \vdash =5 n \varepsilon T \]

given that each term in \( =\bar{a} \) occurs in some \( =5 i \)
and that for each \( i > 1 \), some term in \( =5 i \) occurs in
an earlier \( =5 j \) (i.e., where \( j < i \))

\[ E_1, E_2(\bar{a}), H(\bar{a}) \vdash F(\bar{a}) \] by \( = \) substitution

\[ E_1, E_2(\bar{a}), H(\bar{a}) \vdash =5 e \varepsilon T \]

\[ E_1, x : T, E_2(x^*), H(x^*) \vdash F(x^*) \]

where \( x^* \) is \( x \) repeated to match the length of \( a \),
given no capture and that each term in \( a \)
occurs in \( =5 \)

4.3. **Similarity Rules**

These rules express the idea that types as classifications are treated extensionally.

\[ E, H \vdash \approx A \] by similarity

\[ E, H \vdash =5 v_i \] where \( i \) is 1 or 2

given that each term in \( \approx A \) occurs in \( =5 \)

\[ E, H \vdash \approx A \] by similarity

\[ E, H \vdash \approx B_i \]

\[ E, H \vdash \approx B_n \]

given that each term in \( \approx A \) occurs in some \( \approx B_i \)
and that for each \( i > 1 \), some term in \( \approx B_i \) occurs in
an earlier \( \approx B_j \) (i.e., where \( j < i \))

\[ E, H \vdash \approx e A \] by \( \approx \) substitution

\[ E, H \vdash \approx T \]

\[ E, H \vdash =5 e B \]

given that \( A \) and \( B \) occur in \( \approx T \)

5. **Finite Enumerations:** \{\}

Consider the types constructed from a finite number of distinct, determined objects. We call these enumeration types and write them with curly brackets, e.g., \{red,blue,green\}. The values of such a type are the names appearing inside the brackets, e.g., \{red,blue,green\}.

How could one use an arbitrary element of such a type? That is, given that \( t \varepsilon \{q_1, \ldots, q_n\} \), what fundamental operation(s) could make use of \( t \)? We
believe that the essential activity on these types is reasoning by cases. If we know how to solve a problem for each qi, we know how to solve it for arbitrary t{q1,...,qn}. Thus, we introduce the form
\[
\text{case}(t; q1:t1, q2:t2, ..., qn:tn)
\]
which is executed by evaluating t, and given that the result is qi, then evaluating ti and yielding the result.

Canonicality:
\[
\text{VAL} \{q1,...,qn\} \quad \text{VAL} q1 \\
\text{VAL} \quad : \\
\text{VAL} qn
\]

Binding:
\[
\text{no} \{\} \text{forms bind any variables}
\]
\[
\mathcal{E}, \lambda \vdash \{q1,...,qn\}\epsilon \text{VAL} \quad \text{by} \{\} \text{formation}
\]
\[
\mathcal{E}, \lambda \vdash qi\epsilon\{q1,...,qn\} \quad \text{by} \{\} \text{intro}
\]
\[
\mathcal{E}, \lambda, t\epsilon\{q1,...,qn\} \vdash \text{case}(t; q1:t1,...,qn:tn)\epsilon \text{T}(t) \quad \text{by} \{\} \text{elim}
\]
\[
\mathcal{E}, \lambda \vdash t1\epsilon \text{T}(q1) \\
\vdash \\
\mathcal{E}, \lambda \vdash t\epsilon \text{T}(qn)
\]
\[
\mathcal{E}, \lambda \vdash \text{case}(q1; q1:t1,...,qn:tn) = ti\epsilon \text{T}(qi) \quad \text{by} \{\} \text{red}
\]
\[
\mathcal{E}, \lambda \vdash t1\epsilon \text{T}(q1) \\
\vdash \\
\mathcal{E}, \lambda \vdash t\epsilon \text{T}(qn)
\]

We could define the finite type \text{Bool} as \{true,false\} and \text{Void} as \{\}.

6. \textbf{Natural Numbers: N}

Consider the process of forming objects starting with a particular object and repeatedly building a "successor". This process forms the natural numbers, denoted \(\mathbb{N}\), containing the values 0, suc(0), suc(suc(0)), .... Thus, we have \(0\in\mathbb{N}\) and for any number \(n\), \(\text{suc}(n)\in\mathbb{N}\).

How can a number be decomposed? The general method is that of inductive definition: explain how to use 0 and explain how to use \text{suc}(n) in terms of the use of \(n\). For example, produce \text{twice} \(n\) by
\[
\text{twice } 0 = 0 \\
\text{twice } \text{suc}(n) = \text{suc}(\text{suc}(\text{twice } n))
\]

This method gives rise to the non-canonical form
\[
\text{ind}(a;n;x,y,b)
\]
To execute such a term, first evaluate \(n\), if the result is 0 then evaluate \(a\) and yield its value, otherwise the result is \(\text{suc}(m)\) so evaluate \(\text{ind}(a;m;x,y,b)\) then evaluate and return \(b(x,y)\) with \(x\) replaced by \(m\) and \(y\) replaced by the
value just obtained for ind(a;m;x,y;b).

Canonicality:
\[ \text{VAL N} \quad \text{VAL 0} \quad \text{VAL a} \implies \text{VAL suc(a)} \]

Binding:
\[ x \text{ and } y \text{ free in } b \text{ become bound in } \text{ind}(a;n;x,y;b) \]

\[ \mathcal{E}, \mathcal{H} \vdash N \in \text{V1} \quad \text{by N formation} \]

\[ \mathcal{E}, \mathcal{H} \vdash 0 \in N \quad \text{by N intro} \]

\[ \mathcal{E}, \mathcal{H} \vdash \text{suc}(a) \in N \quad \text{by N intro} \]

\[ \mathcal{E}, \mathcal{H} \vdash a \in \text{N} \]

\[ \mathcal{E}, \mathcal{H}, n \in \text{N} \vdash \text{ind}(a;n;x,y;b) \in \text{T}(n) \quad \text{by N elim (sometimes called induction)} \]

\[ \mathcal{E}, \mathcal{H} \vdash a \in \text{T}(0) \]

\[ \mathcal{E}, x : N, y : \text{T}(x), \mathcal{H} \vdash b \in \text{T}(\text{suc}(x)) \]

\[ \mathcal{E}, \mathcal{H} \vdash \text{ind}(a;0;x,y;b) \in \text{T}(0) \quad \text{by N red} \]

\[ \mathcal{E}, \mathcal{H} \vdash a \in \text{T}(0) \]

\[ \mathcal{E}, x : N, y : \text{T}(x), \mathcal{H} \vdash b \in \text{T}(\text{suc}(x)) \]

\[ \mathcal{E}, \mathcal{H}, n \in \text{N} \vdash \text{ind}(a;\text{suc}(n);x,y;b) \in b(n/x, \text{ind}(a;n;x,y;b)/y) \in \text{T}(\text{suc}(n)) \quad \text{by N red} \]

\[ \mathcal{E}, \mathcal{H} \vdash a \in \text{T}(0) \]

\[ \mathcal{E}, x : N, y : \text{T}(x), \mathcal{H} \vdash b \in \text{T}(\text{suc}(x)) \]

Z. Tuples: []

Consider the process of taking two existing values and packaging them up into one value. This process leads to the product types, written [A,B], with elements \(<a,b>\). In general, we can imagine building products in which the type of the second component depends on the value of the first component. For example, \(<\text{number},1>\) and \(<\text{type},N>\) could both be elements of the dependent product

\[ [x : \{\text{number, type}\}, \text{case}(x; \text{number:N, type:V1})]. \]

The general form of a product type is \([x:A,B(x)]\) where A is a type and B is a type valued expression. A pair \(<a,b>\) is in the type if \(a \in A\) and \(b \in \text{B}(a)\). We require that there be no free variables in a product type, thus any term of the form [\(\ldots\)] is canonical. However, there is a mechanism, the "with" term, for constructing product types as the result of computation.

Using [\(\ldots\)] formation, we can get easily the following products:

\[ [N,N] \in \text{V1} \quad \text{meaning } [x:N,N] \text{ where T2 has no x in it} \]

\[ [V1,N] \in \text{V2} \quad \text{whose elements are } <\text{small type, natural number}> \]

\[ [x:V1,x] \in \text{V2} \quad \text{whose elements are } <\text{small type, element of that type}> \]

The last arises because we can prove \(x:V1 \vdash x:V1\), so the second sub-goal of [\(\ldots\)] formation can be proved if T1 is V1 and T2(x) is x.
Given a value of a product type, what sort of use might we make of it? The operation of tearing open a tuple and using the parts is called spread. We introduce the form

\[
\text{spread}(t; x, y, d)
\]

in which \(x\) and \(y\) free in \(d\) become bound. To evaluate this term, first evaluate \(t\) yielding \(\langle a, b \rangle\), then evaluate and yield the result of \(d(a/x, b/y)\). For example, if \(d\) is simply \(x\) then the spread term is \(\text{first}(t)\), for \(y\) it is \(\text{second}(t)\).

We say that two product types \([(x:A, B(x))]\) and \([(x:C, D(x))]\) are equal when the expressions that describe the types of their components denote the "same" computation. It is not sufficient that the expressions are extensionally equal, they must produce their values in the same way. Thus as suggested earlier, we require the products to be identical up to α-conversion.

Similarity of two product types means that the same pairs are in both types. This is the case when \(A\) is similar to \(C\) and \(B(x)\) and \(D(x)\) produce similar types on the same values of \(x\). This is captured in a rule that is much like Martin-Löf's rule for \(Σ\)-type equality.

Canonicality:

\[
\text{VAL } [x:A, B(x)] \quad \text{VAL } a \& \text{VAL } b \Rightarrow \text{VAL } \langle a, b \rangle
\]

Binding:

\(x\) free in \(B\) becomes bound in \([x:A, B(x)]\)

\(x\) and \(y\) free in \(d\) become bound in \(\text{spread}(t; x, y, d)\)

\(E, H \vdash [x:A, B(x)] \epsilon \text{Vi} \) by \([\ ]\) formation

\(E, H \vdash A \epsilon \text{Vj} \)
\(E, x:A, H \vdash B(x) \epsilon \text{Vk} \)

where \(i, j, k\) are 1 or 2

and \(i = \text{max}(j, k)\)

given that \(A\) has no free variables and \(B\) has no free variables other than \(x\)

\(E, H \vdash \langle a, b \rangle \epsilon [x:A, B(x)]\) by \([\ ]\) intro

\(E, H \vdash a \epsilon A\)

\(E, H, a \epsilon A \vdash b \epsilon B(a)\)

\(E, H, t \epsilon [x:A, B(x)] \vdash \text{spread}(t; x, y, d) \epsilon \text{T}(t)\) by \([\ ]\) elim

\(E, x:A, y:B(x), H \vdash d \epsilon \text{T}(\langle x, y \rangle)\)

\(E, H, a \epsilon A, b \epsilon B(x) \vdash \text{spread}(\langle a, b \rangle; x, y, d) = d(a/x, b/y) \epsilon \text{T}(\langle a, b \rangle)\) by \([\ ]\) reduction

\(E, x:A, y:B(x), H \vdash d \epsilon \text{T}(\langle x, y \rangle)\)

\(E, H \vdash [x:A, B(x)] \epsilon [x:C, D(x)]\) by \([\ ]\) similarity

\(E, H \vdash A \epsilon C\)

\(E, x:A, H \vdash B(x) \epsilon D(x)\)

7.1. With Terms

So far we have discussed a very limited variety of terms for product types. The terms have all been canonical, they are the product types

\(\text{pprl definition}\) October 7, 1981
themselves. As the reader might expect, we need a method for building products from the results of subsidiary expressions. In particular, we need product valued terms which contain free variables.

The "most powerful" non-canonical term will be seen to be the application of a product valued function to some arguments. This might seem to be a way to compute products. However, since there is no way, as yet, to build a product other than by constructing a constant product, there is no way to build a function that does more than yielding one of a finite collection of products. We need some basic means for building a product from computed values. The method provided is the with form.

The term \([x:A(\bar{y}), B(x, \bar{y})]\) with \(\bar{y}+\bar{t}\) denotes the product obtained by replacing \(\bar{y}\) in \([x:A(\bar{y}), B(x, \bar{y})]\) with the values denoted by \(\bar{t}\). Thus, the [...part serves as a template with holes in it. The holes are filled with the values of the terms \(\bar{t}\), when those values become known. The \(\bar{t}\) part of the construct may have arbitrary free variables and may be substituted into without restriction. The template part, however, is considered quoted -- by definition it contains no free variables and may not be disturbed by substitution, \(\beta\)-reduction, or any other mechanism. The rules for dealing with these terms formalize our conception.

Using "with" terms, we can build product valued terms with parts that depend on some computations, e.g. \([y,y]\) with \(y\)-type-valued-term. The conventional approach is to allow templates as terms. In this example \([y,y]\) would be considered a product valued term with free variables. However, this approach leads to problems because of the interplay between intensionality and substitution.

Consider any non-canonical closed term \(T\) that evaluates to \(N\). That is, \(T=N\) is provable. We should be able to interchange \(T\) and \(N\) as permitted by the substitution rules. If we had the form \([y,y]\) where \(y\) is a type variable, we would be able to prove \([T,T]=[N,N]\). Yet this is false according to our notion of equality on types (\(T\) and \(N\) are not the same sequence of marks). The difficulty is that we are treating descriptions interchangably in a place where they are not interchangeable. The whole point of intensionality is to discriminate between different descriptions of the same value. Thus, one must be careful in mixing intensionality with substitution.

Canonicality:
no "with" forms are canonical

Binding:
no "with" forms bind any variables (the template of a "with" is not considered a sub-term, so there are no variables free in it)

\(E, \overline{H} \vdash ([x:A(\bar{y}), B(x, \bar{y})]\) with \(\bar{y}+\bar{t}\)\) by \([\] with formation
\(E, \overline{H} \vdash A(\bar{t})\)\) by \(V_j\)
\(E, x:A(\bar{t}), \overline{H} \vdash B(x, \bar{t})\)\) by \(V_k\)

where \(i, j, k\) are 1 or 2
and \(i=\max(j, k)\)
given that only \(\bar{y}\) are free in \(A\)
and only \(x\) and \(\bar{y}\) are free in \(B\)
\[ E, \bar{H} \vdash ([x:A(\bar{y}), B(x, \bar{y})] \with \bar{y} +\bar{c}) = [x:A(\bar{c}) \in B(x, \bar{c})] \in \forall \bar{y} \] by \[ \] with reduction \\
\[ E, \bar{H} \vdash [x:A(\bar{y}), B(x, \bar{y})] \with \bar{y} +\bar{c} \in \forall \bar{y} \] \\
given that all terms in \[ \bar{y} \] are canonical

8. Functions: \[ \to \]

The arrow type constructor builds dependent function types of the form \((x:A \to B(x))\). A value of such a type is a \(\lambda\)-term \(\lambda x. b\), with the property that for any \(a \in A\), \(b(a/x) \in B(a)\). Thus, the type of the result depends on the value of the argument. If \(B\) does not depend on \(x\) then the type is simply \(A \to B\) and a member function always returns an element of \(B\).

Given a value of a function type, what sort of use might we make of it? The obvious operation is application, written here as 

\[ \text{ap}(f, t) \]

where \(f\) is the function and \(t\) an argument. To evaluate this term, first evaluate both \(f\) and \(t\). The former should yield a value \(\lambda x. b\) while the latter yields a value \(v\), so evaluate and yield the result of \(b(v/x)\).

The reduction process just discussed is commonly known as beta-reduction. We also provide eta-reduction, \(\lambda x. \text{ap}(f, x) \Rightarrow f\), as a valid method of reasoning about functions.

Canonicality: 
\[ \text{VAL} \ (x:A \to B(x)) \]
\[ \text{VAL} \ \lambda x. b \]

Binding:
\[ x \text{ free in } B \text{ becomes bound in } (x:A \to B(x)) \]
\[ x \text{ free in } b \text{ becomes bound in } \lambda x. b \]

\[ E, \bar{H} \vdash (x:A \to B(x)) \in \forall \bar{v} \] by \(\to\) formation \\
\[ E, \bar{H} \vdash \text{Ac} \in V_j \]
where \(i, j, k\) are 1 or 2 \\
\[ E, x:A, \bar{H} \vdash B(x) \in \forall \bar{v} \]
and \(i = \max(j, k)\) \\
given that \(A\) has no free variables and \(B\) has no free variables other than \(x\)

\[ E, \bar{H} \vdash \lambda x. b \in (x:A \to B(x)) \] by \(\to\) intro \\
\[ E, \bar{H} \vdash \text{Ac} \in V_i \]
where \(i\) is 1 or 2 \\
\[ E, x:A, \bar{H} \vdash b \in B(x) \]
given that \(b\) has no free variables other than \(x\)

\[ E, \bar{H}, t \in (x:A \to B(x)) \vdash \text{ap}(t, a) \in B(a) \] by \(\to\) elim \\
\[ E, \bar{H} \vdash a \in A \]

\[ E, \bar{H}, \lambda x. b \in (x:A \to B(x)), a \in A \vdash \text{ap}(\lambda x. b \in (x:A \to B(x)), a) = b(a) \in B(a) \] by \(\to\) beta reduction \\
\[ E, \bar{H}, t \in (x:A \to B(x)) \vdash \lambda x. \text{ap}(t, x) = t \in (x:A \to B(x)) \] by \(\to\) eta reduction
\[ E, \overline{H} \vdash (x : A \rightarrow B(x)) \approx (x : C \rightarrow D(x)) \text{ by } \approx \text{ similarity} \]
\[ E, \overline{H} \vdash A \equiv C \]
\[ E, x : A, \overline{H} \vdash B(x) \approx D(x) \]

2.1. With Terms

As with products, we need "with" forms for arrow types. These rules are variants of those for tuples.

Canonicality:
no "with" forms are canonical

Binding:
no "with" forms bind any variables (the template of a "with" is not considered a sub-term, so there are no variables free in it)

\[ E, \overline{H} \vdash ((x : A(x)) \rightarrow B(x, \overline{y})) \text{ with } \overline{y} + \overline{e} \in \overline{V} \text{ by } + \text{ with formation} \]
\[ E, \overline{H} \vdash A(x) \in \overline{V} \]
\[ E, x : A, \overline{H} \vdash B(x, \overline{y}) \in \overline{V} \text{ where } i, j, k \text{ are } 1 \text{ or } 2 \]
\[ E, x : A, \overline{H} \vdash \overline{y} + \overline{e} \in \overline{V} \text{ and } i = \max(j, k) \]
given that only \( \overline{y} \) are free in \( A \)
and only \( x \) and \( \overline{y} \) are free in \( B \)

\[ E, \overline{H} \vdash ((x : A(x)) \rightarrow B(x, \overline{y})) \text{ with } \overline{y} + \overline{e} = (x : A(x) \rightarrow B(x, \overline{e})) \in \overline{V} \text{ by } + \text{ with reduction} \]
\[ E, \overline{H} \vdash ((x : A(x)) \rightarrow B(x, \overline{y})) \text{ with } \overline{y} + \overline{e} \in \overline{V} \]
given that all terms in \( \overline{e} \) are canonical

\[ E, \overline{H} \vdash (\lambda x : b(x, \overline{y}) \in (x : A \rightarrow B(x)) \text{ by } \lambda \text{ with intro} \]
\[ E, \overline{H} \vdash A \in \overline{V} \text{ where } i \text{ is } 1 \text{ or } 2 \]
\[ E, x : A, \overline{H} \vdash b(x, \overline{e}) \in B(x) \]
given (as everywhere) that no capture happens in \( b(x, \overline{e}) \)
and only \( x \) and \( \overline{y} \) are free in \( b \)

\[ E, \overline{H} \vdash (\lambda x : b(x, \overline{y}) \in (x : A \rightarrow B(x)) \text{ by } \lambda \text{ with reduction} \]
\[ E, \overline{H} \vdash (\lambda x : b(x, \overline{y}) \text{ with } \overline{y} + \overline{e} = \lambda x : b(x, \overline{e}) \in (x : A \rightarrow B(x)) \text{ by } \lambda \text{ with reduction} \]
\[ E, \overline{H} \vdash (\lambda x : b(x, \overline{y}) \text{ with } \overline{y} + \overline{e} \in (x : A \rightarrow B(x)) \text{ given that all terms in } \overline{e} \text{ are canonical} \]

2. Identity: I

The I type constructor builds a notion of equality into the mathematical side of the logic, i.e., into the "real world" of values.

We introduce the form \( I(A, a, b) \) where \( A \) is a type and \( a, b \) are elements of \( A \). The values in \( I(A, a, b) \) depend on whether \( a = b \in A \). If so then there is a trivial element, \( i \), in the I type. If not then the I type is empty.

This type allows us to begin building a logic of propositions inside the type theory. By combining this form with the other type constructors, we can get types that represent quantified propositions. These can be manipulated as values by functions in the theory.
Canonically:
VAL I(A,a,b)  

Val i

Binding:
no I forms bind any variables

\[ E, \overline{H} \vdash I(A,a,b) \varepsilon Vi \quad \text{by I formation} \]
\[ E, \overline{H} \vdash A \varepsilon Vi \]  
\[ E, \overline{H} \vdash a \varepsilon T \]
\[ E, \overline{H} \vdash b \varepsilon T \]  
\[ E, \overline{H} \vdash i \varepsilon I(A,a,b) \quad \text{by I intro} \]
\[ E, \overline{H} \vdash a = b \varepsilon A \]
\[ E, \overline{H}, t \varepsilon I(A,a,b) \vdash F \quad \text{by I elim} \]
\[ E, \overline{H}, a = b \varepsilon A \vdash F \]

2.1. With Terms

Canonically:
no "with" forms are canonical

Binding:
no "with" forms bind any variables (the template of a "with" is not considered a sub-term, so there are no variables free in it)

\[ E, \overline{H} \vdash (I(A(\overline{y}), a(\overline{y}), b(\overline{y}))) \varepsilon Vi \quad \text{by I with formation} \]
\[ E, \overline{H} \vdash A(\overline{E}) \varepsilon Vi \]  
\[ E, \overline{H} \vdash a(\overline{E}) \varepsilon A(\overline{E}) \]
\[ E, \overline{H} \vdash b(\overline{E}) \varepsilon A(\overline{E}) \]  
given that only \( \overline{y} \) are free in \( A,a,b \)
\[ E, \overline{H} \vdash (I(A(\overline{y}), a(\overline{y}), b(\overline{y}))) \varepsilon Vi \quad \text{by I with reduction} \]
\[ E, \overline{H} \vdash (I(A(\overline{y}), a(\overline{y}), b(\overline{y}))) \varepsilon Vi \]  
given that all terms in \( \overline{E} \) are canonical

10. Intensionality

We intend that functions and types be fully usable as objects. One fundamental requirement for computation on objects is that their identities be determinable, i.e., that equality on objects be decidable. This is our main reason for choosing to treat functions and types intensionally, as was done in the above rules. Our notion of intensional identity is that two objects are equal if they "look the same" under alpha reduction.

We believe a meta-theorem can be shown that provability of \( = \) in \( \mu \text{prl} \) is decidable. Given such a theorem, we could define a new form, \( \text{eq} \), so that for any type \( T \), and terms \( a,b \) in \( T \):

\[ \mu \text{prl definition} \quad \text{October 7, 1981} \]
\text{eq}(T, a, b) \equiv \text{Bool}

\text{and } \quad \text{eq}(T, a, b) = \text{true} \iff a = b \in T.

The eq form would be connected to the algorithm implied by decidability proof.

The difficulty with this approach is that it makes the sensibility of \mu prl directly dependent on a non-trivial meta-theoretic construction (though perhaps this is already the case since thinking about the logic is a non-trivial meta-theoretic process). An alternative is to introduce the notion of inequality into the theory as a new relation, \not\equiv, with new formulas \text{t} \not\equiv \text{t}' \in T. We would try to construct proof rules so that \text{t} \not\equiv \text{t}' \in T exactly when \text{t} and \text{t}' are of type \text{T}, but \text{t} = \text{t}' \in T is not provable. The eq form could be defined:

\begin{align*}
a = b \in T & \iff \text{eq}(T, a, b) = \text{true} \in \text{Bool} \\
a \not\equiv b \in T & \iff \text{eq}(T, a, b) = \text{false} \in \text{Bool}
\end{align*}

This approach does not eliminate the need to demonstrate sensibility. The eq form must be well defined, so we must show the consistency and completeness of the theory. This will require roughly the same reasoning as showing decidability of =. However, it has the (possible) advantage that inequality is explained in a positive fashion, provability of \not\equiv, rather than as lack of provability of =. Also, we already have a variety of concepts that need justification, e.g., the induction builder: \text{ind}, and eq is treated here in the same way. The main disadvantage is that another relation and many proof rules must be added to the theory.

Our choice is to present the proof rules for \not\equiv, thereby giving the reader a more precise intuition about equality, and to define eq in terms of = and \not\equiv. Later we will prove the decidability of =, couple eq directly to the resulting algorithm, and eliminate the notion of \not\equiv from the logic. Hence, the rules below can be viewed as temporary, intended only to allow the introduction of eq before the decidability of eq is shown.

\[\begin{align*}
\bar{E}, \bar{H} & \vdash \text{t} \not\equiv \text{t}' \in T & \text{by } \not\equiv \text{ symmetry} \\
\bar{E}, \bar{H} & \vdash \text{t}' \not\equiv \text{t} \in T
\end{align*}\]

\[\begin{align*}
\bar{E}, \bar{H} & \vdash a \not\equiv c \in T & \text{by } \not\equiv \text{ equality} \\
\bar{E}, \bar{H} & \vdash a = b \in T \\
\bar{E}, \bar{H} & \vdash b = c \in T
\end{align*}\]

<this rule is derivable via:

\[a = b \ & \& \ b \not\equiv c \implies \neg I(a, c) \quad \text{(by assume } I(a, c); a = c; b = c; \text{eq}(b, c) = 1 \quad \text{but } b \not\equiv c \text{ so eq}(b, c) = 0; 0 = 1 \#\#)\]

\[\neg I(a, c) \implies a \not\equiv c \quad \text{(by cases on eq}(a, c): 0 \implies a \not\equiv c \quad 1 \implies a = c; I(a, c); \#\#)\]

\[>\]

\[\begin{align*}
\bar{E}, \bar{H} & \vdash T \not\equiv T' \in V \quad \text{by type } \not\equiv \\
\bar{E}, \bar{H} & \vdash T \in V \\
\bar{E}, \bar{H} & \vdash T' \in V \\
\text{given that the forms of } T \text{ and } T' \text{ differ when classified into:} \\
\{\}, \{\}, \{\}, \text{ Vj, I}
\end{align*}\]

\[\begin{align*}
\bar{E}, \bar{H} & \vdash \{ai\} \not\equiv \{bj\} \in V \quad \text{by enum formation } \not\equiv \\
\text{given that the sequence } ai \text{ differs from the sequence } bj
\end{align*}\]
\[ \bar{E}, \bar{H} \vdash [x:A_B(x)] = [x:C_D(x)] \varepsilon \text{Vi} \] by [] formation ≠
\[ \bar{E}, \bar{H} \vdash [x:A_B(x)] \varepsilon \text{Vi} \]
\[ \bar{E}, \bar{H} \vdash [x:C_D(x)] \varepsilon \text{Vi} \] given that the [] forms are not α equivalent

\[ \bar{E}, \bar{H} \vdash (x:A \rightarrow B(x)) = (x:C \rightarrow D(x)) \varepsilon \text{Vi} \] by () formation ≠
\[ \bar{E}, \bar{H} \vdash (x:A \rightarrow B(x)) \varepsilon \text{Vi} \]
\[ \bar{E}, \bar{H} \vdash (x:C \rightarrow D(x)) \varepsilon \text{Vi} \] given that the \( \rightarrow \) forms are not α equivalent

\[ \bar{E}, \bar{H} \vdash a \neq b \varepsilon \{a, b\} \] by \{\}\) ≠
\[ \bar{E}, \bar{H} \vdash a \varepsilon \{a\} \]
\[ \bar{E}, \bar{H} \vdash b \varepsilon \{a\} \] given that \( a \) and \( b \) are not identical symbols

\[ \bar{E}, \bar{H} \vdash a \neq b \varepsilon \mathbb{N} \] by \( \mathbb{N} \) ≠
\[ \bar{E}, \bar{H} \vdash a \varepsilon \mathbb{N} \]
\[ \bar{E}, \bar{H} \vdash b \varepsilon \mathbb{N} \] given that \( a \) and \( b \) are both canonical, yet are not identical forms

\[ \bar{E}, \bar{H} \vdash <a, b> = <c, d> \varepsilon [x:A_B(x)] \] by [] ≠
\[ \bar{E}, \bar{H} \vdash <a, b> \varepsilon [x:A_B(x)] \]
\[ \bar{E}, \bar{H} \vdash <c, d> \varepsilon [x:A_B(x)] \]
\[ \bar{E}, \bar{H} \vdash a \neq c \varepsilon A \]

\[ \bar{E}, \bar{H} \vdash <a, b> = <c, d> \varepsilon [x:A_B(x)] \] by [] ≠
\[ \bar{E}, \bar{H} \vdash <a, b> \varepsilon [x:A_B(x)] \]
\[ \bar{E}, \bar{H} \vdash <c, d> \varepsilon [x:A_B(x)] \]
\[ \bar{E}, \bar{H} \vdash a = c \varepsilon A \]
\[ \bar{E}, \bar{H} \vdash a \neq c \varepsilon \mathbb{E} [a] \]
\[ \bar{E}, \bar{H} \vdash a \neq b \varepsilon \mathbb{E} (a) \]

\[ \bar{E}, \bar{H} \vdash \lambda x.b \neq \lambda x.d \varepsilon (x:A \rightarrow B(x)) \] by \( \rightarrow \) ≠
\[ \bar{E}, \bar{H} \vdash \lambda x.b \varepsilon (x:A \rightarrow B(x)) \]
\[ \bar{E}, \bar{H} \vdash \lambda x.d \varepsilon (x:A \rightarrow B(x)) \] given that the \( \lambda \) forms are not α equivalent

\[ \bar{E}, \bar{H} \vdash a \neq b \varepsilon T1 \] by similarity
\[ \bar{E}, \bar{H} \vdash T1 \varepsilon T2 \]
\[ \bar{E}, \bar{H} \vdash a \neq b \varepsilon T2 \]

We are now in a position to define the eq form. This is done as suggested earlier. (Note: eq is not a type, even though we have named the rules as if it were.)

\[ \bar{E}, \bar{H} \vdash \text{eq(T, a, b)} \varepsilon \text{Bool} \] by eq formation
\[ \bar{E}, \bar{H} \vdash T \varepsilon \text{Vi} \]
\[ \bar{E}, \bar{H} \vdash a \varepsilon T \]
\[ \bar{E}, \bar{H} \vdash b \varepsilon T \]

\[ \bar{E}, \bar{H}, a \varepsilon \text{eq(T, a, b)} = \text{true} \varepsilon \text{Bool} \] by eq true intro

\[ \bar{E}, \bar{H}, \text{eq(T, a, b)} = \text{true} \varepsilon \text{Bool} \vdash a \varepsilon \text{eq(T, a, b)} \] by eq true elim
11. Diagram of Classes and Objects

<table>
<thead>
<tr>
<th>individuals</th>
<th>small types</th>
<th>large types(V1) (V2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>( \lambda x: N \cdot x )</td>
<td>( N \to N )</td>
<td>( \varepsilon )</td>
</tr>
<tr>
<td>( &lt;1,1&gt; )</td>
<td>([1,1] )</td>
<td>( \varepsilon )</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>V1</td>
</tr>
<tr>
<td><strong>objects</strong></td>
<td><strong>types / objects</strong></td>
<td><strong>. . . .</strong></td>
</tr>
<tr>
<td>( &lt;N,N&gt; )</td>
<td>([N,N] )</td>
<td></td>
</tr>
<tr>
<td>( \lambda x: V1 \cdot x )</td>
<td>( V1 \to V1 )</td>
<td></td>
</tr>
<tr>
<td>( &lt;1,N&gt; )</td>
<td>([N,V1] )</td>
<td></td>
</tr>
<tr>
<td>( &lt;N,1&gt; )</td>
<td>([x; V1,N] )</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>not a type nor object</td>
</tr>
<tr>
<td>objects, but not types</td>
<td>types / objects, but not useful objects</td>
<td></td>
</tr>
</tbody>
</table>

12. Acknowledgements

Many people have contributed to the development of \( \mu p r l \). We wish to thank all who have participated. Special thanks are due to Stuart Allen, Fran Corrado, Alan Demers, and Bob Harper, who with the authors, constitute Cornell's Project on Refinement Logic.

13. References


[Ma 79]
Martin-Lof, P.; "Constructive Mathematics and Computer Programming"; Sixth International Congress for Logic, Methodology and Philosophy of Science, Hanover, Germany (August 1979).