Formalization of Isabelle Meta Logic in NuPRL

Pavel Naumov*

Department of Computer Science, Cornell University, Ithaca, NY 14853
pavel@cs.cornell.edu

September 15, 1999

Abstract. NuPRL and Isabelle are two general purpose theorem provers. Both of them are based on a version of Constructive Higher Order Type Theory. In an earlier work the author has proposed an informal semantics of Isabelle Meta Logic in an extension of NuPRL Type Theory. An automated converter, based on this semantics, has been developed, that translates Isabelle theorem statements into NuPRL.

This work presents a formalization of the above semantics in NuPRL. It starts with a deep embedding of Isabelle type and term syntax into NuPRL Constructive Type Theory. Next, two internal NuPRL functions are defined. One of them maps Isabelle types into NuPRL types and the other maps Isabelle terms into elements of appropriate NuPRL types. These two functions provide an interpretation of Isabelle in NuPRL.

Finally, interpretations of all Isabelle Meta Logic rules are proven as theorems in some classical extension of NuPRL Type Theory. This formalization is aimed to provide a more secure foundation for the interaction between two systems.

1 Introduction

This work studies connection between two different proof development environments: Isabelle [12] and NuPRL [2]. Previous works in this area were directed towards creating effective semi-automated procedures for translating mathematical results from one system into another. D. Howe in [5] and [4] defined a shallow embedding of HOL [3] into a classical extension of NuPRL Type Theory, proved its soundness using set-theoretical semantics of NuPRL, and wrote a converter from HOL into NuPRL based on this embedding.

Following Howe's approach, in [10] we defined an embedding of Isabelle Meta Logic into NuPRL and wrote a converter that automates such translation. An important novelty of [10] was in the way the soundness of the embedding is justified. Instead of relying on semantical arguments, it is done in a purely syntactical proof that shows how the translation of any instance of an Isabelle inference rule can be decomposed into several NuPRL inference rule applications. A similar result for Howe's embedding of HOL into NuPRL was independently obtained by J. Meseguer and M.-O. Stehr [8].

* This work was supported by DARPA grant F30602-98-2-0198 “An Open Logical Programming Environment: a Practical Framework for Sharing Formal Models”.
The syntactical soundness proof opens the door to two new directions in the research. First, it becomes feasible to convert formal proofs, not just statements of the theorems, from one system into another. Such proof translation mechanism will actually eliminate the need for justification of conversion soundness, since all translated results will have proofs in the target system. The second opportunity is that now the justification argument can be formalized. Although such formalization does not completely eliminate the need for one system to trust the soundness of another, it provides a more secure foundation for interaction between theorem provers.

In this work we have explored the second opportunity. We have defined a deep embedding of Isabelle syntax into NuPRL Constructive Type Theory, formalized the translation as an internal NuPRL function and proved translations of Isabelle Meta Logic inference rules as theorems in a classical extension of NuPRL.

In addition to providing a secure foundation for the translator, this work also shows that NuPRL, as Type Theory and a proof development environment, is mature enough to reflect not just one particular mathematical theory, but an entire formal system.

The paper is structured as follows. Section 2 describes NuPRL formal notations, used throughout the manuscript. Section 3 deals with generic mathematical facts that were added to standard Nuprl 4.2 library in order to accomplish the formalization of Isabelle. Sections 4 and 5 formalize Isabelle type and term syntax correspondingly. Together they define a deep embedding of Isabelle Meta Logic into NuPRL Type Theory. Section 6 defines the interpretation of Isabelle Meta Logic in NuPRL as an internal NuPRL function. The final Section 7 shows that this function maps Isabelle meta inference rules into NuPRL propositions, derivable in a classical extension of NuPRL Type Theory.

The presentation of the material closely follows to corresponding formal NuPRL theories. In particular, all key definitions and theorems are reproduced the way they are formalized in NuPRL. For space considerations, proofs are omitted, but they are available from the author’s Web page\(^1\).

## 2 NuPRL Formal Notations

One of the features that makes NuPRL stand apart from the other formal proof development environments is its advance graphical user interface. Terms are entered into NuPRL not by typing in an ASCII text, but by filling fields in a structural term editor. The same term editor is normally used to display formal theories. Hyperlink mechanism, provided by the editor, allows the reader to inspect the abstraction definition just by clicking on any instance of this abstraction in any term. The editor also relies on an extensive use of abstraction display forms to achieve better readability. For example, the default display form for the addition operator \( \text{add}(x, y) \) is \( x + y \) and the one for universal quantifier \( \forall x : T.P \) is \( \forall x : T.P \). Display forms can be set to hide some of operator parameters if their values are assumed to be obvious. For instance, equality operator

\(^1\) [http://www.cs.cornell.edu/home/pavel](http://www.cs.cornell.edu/home/pavel)
equal(T, x, y) which states that elements x and y of type T are equal, is normally displayed as x = y. Parameters missing from the operator display form are known in among NuPRL users as hidden parameters.

Because of mentioned above features, conversion of NuPRL proofs into a paper-based form is not trivial. The standard NuPRL printing function, which converts NuPRL formal theories into $\LaTeX$ files, uses abstraction display forms, and, as a result, loses all hidden parameters. Therefore, NuPRL theory printouts do not provide enough information to reconstruct formal definitions and proofs in their original form. Instead, they should be considered as “semi-formal” descriptions of original theories.

In this work we will be incorporating fragments of these formal library printouts in the text. In most cases we will benefit from this approach since it allows to present formal results in a form close to the original. Nevertheless, in some cases these fragments are ambiguous and will need additional comments. In any case, it will be important to keep in mind that these printout fragments are not the real formalization. The other facts about NuPRL library structure, important for understanding of the formalization, are summarized below.

### 2.1 Library Objects

Each abstraction, theorem, or display form is normally stored in its own library object. Any object printout contains four fields (see Figure 1).

<table>
<thead>
<tr>
<th>status</th>
<th>kind</th>
<th>name</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>T</td>
<td>int_trichot</td>
<td>$\forall i,j: \mathbb{Z}. i &lt; j \lor i = j \lor i &gt; j$</td>
</tr>
</tbody>
</table>

Fig. 1. NuPRL Library Object Structure

**Status** This field shows current status of the object. Symbol * stands for “completely installed”. There are special symbols for partially installed, empty, and error objects.

**Kind** The object can have the following kinds: theorem (T), abstraction (A), display form (D), comment (C), or ML code (M).

**Name** This field contains a unique ID of the object in the entire NuPRL library.

**Content** The content is object term printout produced using appropriate display forms.
2.2 Definitions

In most cases there are three objects in the library that correspond to a definition: a display form, an abstraction, and a well-formness lemma. For example, boolean negation is defined in NuPRL by the following library objects

*D bnot df Parems :: Prec(preop):: \(~_b:b:bool:E\) == bnot\{\}(\langle b \rangle)

*A bnot \(~_b\) = if \(b\) then \(ff\) else \(tt\) fi

*T bnot_wf \(\forall b: \mathbb{B}. \; \neg_b \in \mathbb{B}\)

To save space, here we usually reproduce only abstraction object from each definition. In some rare cases well-formness lemmas also will be given. Majority of recursive functions are defined in NuPRL using \(Y\)-combinator. Since such definitions are hard to read, appropriate abstraction object is hidden in the library and a more intuitive recursive "definition" is displayed instead. For instance,

*M length_ml ||as|| == \(r\) case as
  of \(\langle \rangle\) => 0
  \(| a::as^' \rangle \Rightarrow ||as|| + 1\)
  esac

Technically, this recursive pseudo definition is an ML object.

3 Auxiliary Mathematical Facts

Before presenting Isabelle syntax and semantics formalization, we need to state some general mathematical facts and notations that will be used in this work, but which are not part of standard NuPRL 4.2 library. These facts can be divided into three categories.

- Library enhancement by new abstractions, theorems, and tactics, that are not using any new primitive notions or inference rules besides NuPRL standard library.
- NuPRL theory of parametrized recursive types that was developed in [6] by Mendler and formalized by the author in [9].
- A classical extension of NuPRL Constructive Type Theory. Consistency of such extension follows from Howe [5] set-theoretical semantics of NuPRL.

Below all three categories are discussed in details.

3.1 Library Enhancement

**List Equality** For any decidable type \(T\), type \(T\ List\) is also decidable. In other words, if there is a boolean equality relation on type \(T\), there is one on type \(T\ List\). Function
\[\begin{align*}
\text{eq_list.ml} & \quad l = l_m \\
& \quad \Rightarrow \text{if null}(l) \wedge_b \text{null}(m) \text{ then } tt \\
& \quad \text{if null}(l) \lor_b \text{null}(m) \text{ then } ff \\
& \quad \text{else } E[hd(l); \ \text{hd}(m)] \wedge_b \text{tl}(l) = l \ \text{tl}(m) \\
& \quad \text{fi}
\end{align*}\]

defines such boolean equality on \textit{T List} assuming that \(E\) is a boolean equality on type \(T\). List boolean equality is written as \(=\). This notation hides parameter \(E\). The fact that \(=\) is indeed a boolean equality on type \(T\text{ List}\) is expressed in the following lemma:

\[\begin{align*}
\text{eq_listAssert} & \quad \forall T : U. \ \forall E : T \rightarrow T \rightarrow \mathbb{B}.
\quad (\forall t, t' : T. \ \uparrow E[t; t'] \iff t = t') \iff \\
& \quad (\forall ts, ts' : T\text{ List}. \ \uparrow ts = l \ ts' \iff ts = ts')
\end{align*}\]

If relation \(E\) is a boolean equality only on a subset \(T'\) of type \(T\), \(=\) is a boolean equality on type \(T'\text{ List}\). In even more general form, the first and the second arguments of \(E\) can range over different subsets of \(T\). In these cases the following variation of the above theorem will be useful:

\[\begin{align*}
\text{eq_listAssert_2} & \quad \forall T : U. \ \forall Q, R : T \rightarrow P.
\quad \forall E : \{x : T | Q[x]\} \rightarrow \{x : T | R[x]\} \rightarrow \mathbb{B}.
\quad (\forall t : x : T | Q[x]. \ \forall t' : x : T | R[x]. \ \uparrow E[t; t'] \iff t = t') \\
& \quad \Rightarrow (\forall ts : x : T | Q[x] \text{ List}. \ \forall ts' : x : T | R[x] \text{ List}. \\
& \quad \uparrow ts = l \ ts' \iff ts = ts')
\end{align*}\]

\textbf{Records} While defining terms recursively, we need to deal with bindings of bounded variables in partially dis-assembled terms. Such binding is basically a mapping from variable names into type names. Since Isabelle bound variables are named by de Bruijn \([\!][\!][\!]\) indices, binding is a function from an initial segment of natural numbers into a type of Isabelle type names. Such functions are commonly called \textit{records}. In type theories which support dependent function types, record is a notion more general than list, because different fields can have different types. We will be using this feature later.

There are three basic operations on records that are used in this work: \textit{update}, \textit{shift}, and \textit{tail}. Function \textit{update}

\[\begin{align*}
\text{update} & \quad f[n \rightarrow a] = \lambda z. \text{if } (z = n) \text{ then } a \text{ else } f \ z \ fi
\end{align*}\]

extends a record \(f\) of length \(n\) by a new element \(a\), added in the end of the record. This function is similar to appending a single-element list to the end of a given list. Function \textit{shift}

\[\begin{align*}
\text{shift} & \quad [s \gg f] = \lambda z. \text{if } (z = 0) \text{ then } s \text{ else } f \ (z - 1) \ fi
\end{align*}\]
extends record \( f \) by a new element \( s \), added in the beginning of the record. New element gets number 0 and all other element numbers are incremented by 1. This function is similar to \textit{cons} on lists. Function \textit{tail}

\[ \text{tail} \quad (f|n) \to \lambda x. f \ (n \ + \ x) \]

is similar to the \textit{n-th tail} on lists. It removes first \( n \) elements of record \( f \) and re-enumerates the rest accordingly. The following properties of record operators will be used later:

\[ \text{T shift_update} \]
\[
\forall T : U. \ \forall n : N. \ \forall f : \mathbb{N}n \to T. \ \forall a, z : T.
[a \gg f[n \to z]] = [a \gg f][n + 1 \to z]
\]

\[ \text{T shift_zero} \] \hspace{1cm} \forall T : U. \ \forall t' : T. \ \forall n : N. \ \forall f : \text{Top}.
\[ t = t' \Rightarrow n = 0 \Rightarrow [t \gg f] \ n = t' \]

\[ \text{T shift_pos} \] \hspace{1cm} \forall n : N. \ \forall T' : U. \ \forall t', T' : U. \ \forall n : N. \ \forall f : \text{Top}.
\[ \forall i : [i \in (n + 1)^{-}] \ \forall t' : T'.
\]
\[ T' = T' \Rightarrow T[i - 1] = T' \Rightarrow f \ (i - 1) = t' \]
\[ \Rightarrow [t \gg f] \ i = t' \]

\[ \text{T shift_nil_update} \]
\[
\forall T : U. \ \forall f : \text{Nil} \to T. \ \forall s : T. \ [s \gg f] = f[0 \to s]
\]

\[ \text{T zero_tail} \]
\[
\forall T : U. \ \forall n : N. \ \forall f : \mathbb{N}n \to T. \ \forall n', n' : N. \ \forall g : \text{Nil} \to T.
\]
\[ n = n' \Rightarrow n' = 0 \Rightarrow g = (f|n') \]

\textbf{Equality Re-writing} Lack of a good and fast automated re-writing tactic is one of known NuPRL weaknesses. While not intending to solve this problem in general, we introduce an automated re-write tactic EqRW to simplify some terms in our proofs. This tactic is not the most efficient one but it is sufficiently fast for our purposes.

EqRW recursively re-writes a hypothesis, using bool_to_propC conversion and lemmas from eq_rw_lemmas list. Initially this list includes stated above lemma eq_list_assert as well as the following theorems:

\[ \text{T inl_eq} \]
\[
\forall T, S : U. \ \forall x, y : T. \ (\text{inl} \ x) = (\text{inl} \ y) \iff x = y
\]

\[ \text{T inr_eq} \]
\[
\forall T, S : U. \ \forall x, y : T. \ (\text{inr} \ x) = (\text{inr} \ y) \iff x = y
\]

\[ \text{T inr_inl_eq} \]
\[
\forall T, S : U. \ \forall x : T. \ \forall y : S.
\]
\[ (\text{inr} \ y) = (\text{inl} \ x) \iff \text{False} \]

\[ \text{T inl_inr_eq} \]
\[
\forall T, S : U. \ \forall x : T. \ \forall y : S.
\]
\[ (\text{inl} \ x) = (\text{inr} \ y) \iff \text{False} \]
\*T \text{pair\_eq} \quad \forall T, S : \mathcal{U}. \forall t, t' : T. \forall s, s' : S. \\
\quad \langle t, s \rangle = \langle t', s' \rangle \iff t = t' \land s = s' \\

Some lemmas from the classical NuPRL extension and lemmas about Isabelle-specific abstractions are also on the \texttt{eq\_rw\_list}. We will discuss them later.

### 3.2 Parametrized Recursive Types

Recursive (inductive) type is a widely studied type constructor. It allows for any monotonic function $b : \mathcal{U} \rightarrow \mathcal{U}$ to define a new type $\text{rec}(X, B(X))$ that can be informally viewed as the minimal solution of the type equation

$$X = B(X).$$

This equation naturally leads us to a more general type constructor that gives the minimal solution to a system of type equations:

$$X_1 = b_1(X_1, \ldots, X_n)$$

$$\ldots$$

$$X_n = b_n(X_1, \ldots, X_n)$$

Another way to express the same system of equations is to think about variables $X_1, \ldots$ as about a function from index type $I$ into the type universe $\mathcal{U}$

$$X = \lambda i.b(i, X),$$

where $X : I \rightarrow \mathcal{U}$, $b : I \rightarrow (I \rightarrow \mathcal{U}) \rightarrow \mathcal{U}$. This approach allows for infinite systems of type equations if type $I$ is infinite. If $X_0$ is the minimal solution of the above equations, we will denote the application of $X_0$ to an element $i_0$ of type $I$ by $\text{parec}(X, i_0, b(X, i_0) \circ i_0)$.

Inference rules and semantics for such types have been introduced in \cite{7} and \cite{6}. These rules were added to NuPRL as an extension of Constructive Type Theory by the author in \cite{9}.

### 3.3 Classical Extension of NuPRL Type Theory

Isabelle syntax formalization will be done entirely in Constructive Type Theory. A classical extension is used only to provide a semantics of Isabelle Meta Logic in NuPRL. In fact, since Meta Logic is itself intuitionistic, non-constructivity probably can be avoided.

The main reason for using a classical extension of NuPRL is that we want results, brought over from Isabelle theories, to be useful in NuPRL proofs. Thus, ideally, Isabelle type of meta-propositions $o$ should be mapped into NuPRL type of propositions $\mathcal{P}$. Unfortunately, this mapping is hard to implement since NuPRL type $\mathcal{P}$ has infinitely many elements and Isabelle type $o$ is assumed to
have only two elements: true and false. It means that, for example, Isabelle meta
rule
\[
\begin{array}{c}
[\phi] [\psi] \\
\psi \quad \phi
\end{array}
\]
\[
\frac{\phi \equiv \text{prop} \, \psi}
\]
would not be valid when Isabelle propositions are interpreted as NuPRL prop-
sitions and Isabelle equality \( \equiv \) is translated as NuPRL equality. A possible way
around this problem would be to interpret Isabelle type \( o \) as NuPRL quotient


type \( Q = \mathbb{P} / (x, y. x \Leftrightarrow y) \). Type \( Q \) is a factorization of type \( \mathbb{P} \) by the equiva-
lence relation “if and only if”. From author’s experience, dealing with quotient
propositions in NuPRL is not an easy task and conversion of theorems stated
as quotient propositions into standard NuPRL propositions is not trivial either.
But the most importantly, using type \( Q \) does not solve our problems. Unlike
Isabelle Meta Logic, NuPRL propositions do not belong all to the same type.
They are split into a chain of propositional types of different levels:

\[ \mathbb{P}_1 \subseteq \mathbb{P}_2 \subseteq \cdots \subseteq \mathbb{P}_n \subseteq \cdots \]

such that equality of two propositions of level \( n \) is actually an element of type
\( \mathbb{P}_{n+1} \). It means that type \( Q \) in NuPRL is also a sequence of types
\( Q_n = \mathbb{P}_{n} / (x, y. x \Leftrightarrow y) \). If we choose some \( Q_{n_0} \) to be the interpretation of Isabelle

type \( o \), then equality on elements of \( o \) would need to be translated as equality
of \( Q_{n_0} \) elements which in NuPRL belong to a higher-level type \( Q_{n_0+1} \).

Therefore, there probably is no reasonably simple adjustment to NuPRL type
\( \mathbb{P}_i \) that can be used as an interpretation of Isabelle type \( o \).

All mentioned above problems can be avoided by interpreting Isabelle type \( o \)
as NuPRL boolean type \( \mathbb{B} \). Type \( \mathbb{B} \) has only two elements and there is a boolean
equality \( =_b \) on type \( \mathbb{B} \) such that for any elements \( x \) and \( y \) of type \( \mathbb{B} \), \( x =_b y \) is
also an element of type \( \mathbb{B} \).

On the other hand, type \( \mathbb{B} \) brings problems of its own. Not every NuPRL type
has a boolean equality relation. In fact, only decidable types can have boolean
equality in NuPRL because boolean equality, just as any other NuPRL func-
tion, would be assumed to be computable. We will extend NuPRL Constructive
Type Theory by a boolean equality predicate for any type. Such an extension is
non-constructive, or, in other words, classical. Boolean equality is a three-place
predicate that takes any type \( T \) and two elements \( x \) and \( y \) of type \( T \) as
arguments. It returns boolean true if and only if \( x \) and \( y \) are equal as elements of the


type \( T \). To make formulas more readable, NuPRL display mechanism normally
hides type argument of boolean equality and show the boolean equality just as
\( x =_b y \).

There are two properties of boolean equality that we assume. They normally
would be stated as inference rules in NuPRL, but we state them as proof-less
theorems in case if somebody would want to use some other primitive abstraction
instead of boolean equality. These two theorems are

\[ \text{bequal_wf} \quad \forall T : U. \forall x, y : T. \ (x =_b y) \in \mathbb{B} \]
*T assert_bequal \( \forall T:U. \forall x,y:T. \uparrow(x =_b y) \iff x = y \)

In the last formula, \( \uparrow \) stands for NuPRL “assert” operator that converts booleans to propositions. It is important to remember that both boolean equality \( =_b \) and propositional equality \( = \) have hidden type parameter \( T \).

We also need to add boolean universal quantifier to our theory. It is very similar to standard NuPRL propositional universal quantifier with the exception that it works on boolean terms. Boolean universal quantifier has two arguments: a type term \( T \) and a boolean term \( B \) with one bound variable \( x \). It will be displayed as \( \forall_b x : T.B \). Two assumptions about boolean universal quantifier are also stated as theorems:

\[
\begin{align*}
\text{*T ball_wf} & \quad \forall T:U. \forall b:T \rightarrow \mathbb{B}. \ \forall_b x:T. \ b[x] \in \mathbb{B} \\
\text{*T assert_ball} & \quad \forall T:U. \forall b:T \rightarrow \mathbb{B}. \ \uparrow\forall_b x:T. \ b[x] \iff (\forall x:T. \ \uparrow b[x])
\end{align*}
\]

\( \text{Note that the proposed classical NuPRL extension is not minimal. Boolean universal quantifier can be expressed via propositional universal quantifier and boolean equality as} \)

\( \forall_b x : T.b[x] \equiv ((\forall x : T. \ \uparrow b[x]) =_b \text{true}) \).

Alternatively, more compact single-argument operator \( \downarrow \) that converts propositions to booleans can be used as a primitive notion. Boolean equality can be obviously defined via this operator and propositional equality.

Consistency of a NuPRL classical extension can be shown using D. Howe [5] set-theoretical semantics for NuPRL.

Finally, we put lemmas \texttt{bequal_assert} and \texttt{ball_assert} on the list \texttt{eq} \texttt{rw} \texttt{lemmas} to be used by \texttt{EqRw} tactic.

4 Isabelle Type Term Syntax

Isabelle types are elements of Isabelle classes. Each type can belong to a finite number of classes. The list of classes, to which any given type belongs is called a \textit{sort} of this type. Our formalization of Isabelle syntax includes classes and sorts, but they will be ignored later, during interpretation definition. Informally, all classes will be mapped into a NuPRL universal type \( U \) of an arbitrary level \( i \).

4.1 Classes and Sorts

Isabelle defines SML type \texttt{class} as type \texttt{string}, SML type \texttt{sort} as type \texttt{class list}, and SML type \texttt{index_name} as Cartesian product of \texttt{string} and \texttt{int}. We formalize these abstractions in NuPRL in almost identical form.
*A class    Class == Atom

*A sort    Sort == Class List

*A indexname    Indexname == Atom × Z

where Atom is NuPRL type of tokens. All three types defined above are decidable. Boolean equality on classes is inherited from type Atom

*A eq_class    c =e d == c = a d

boolean equality on sorts can be defined using boolean list equality, discussed in Section 3.1, and boolean equality on classes

*A eq_sort    s1 =s s2 == s1 = l s2

boolean equality on type indexname is defined via boolean equalities on tokens and integers

*A eq_indexname    i1 =ixn i2 ==
    let <a1,z1> = i1 in let <a2,z2> = i2 in
    a1 =a a2 ∧ (z1 =z z2)

The following lemmas are put on the eq_rw_lemmas list to enhance EqRW tactic

*T eq_class_assert    ∀c,d:Class. ↑c =e d ⟷ c = d

*T eq_sort_assert    ∀s,t:Sort. ↑s =s t ⟷ s = t

*T eq_indexname_assert    ∀i1,i2:Indexname. ↑i1 =ixn i2 ⟷ i1 = i2

4.2 Type Term

Isabelle defines SML type of Isabelle type terms, called typ, as

datatype typ = Type of string * typ list
    | TFree of string * sort
    | TVar of indexname * sort

There are two adjustments that we make to this definition in order to simplify reasoning about this type in NuPRL.

1. Two different kinds of free variables (TFree and TVar) will be combined into one kind

| TVar of TypVarName

where TypVarName is a disjoint union of types string * sort and indexname * sort.

2. Among different Isabelle type constructors, functional type constructor
S --> T == Type(‘‘fun’’, [S,T])
plays a special role in the definition of Isabelle type “term”. It is convenient
to separate this type constructor into a kind of its own.

Therefore, our type term formalization is based on the following, slightly
modified, version of SML typ datatype

datatype typ = Type of string * typ list
     | TFun of typ * typ
     | TVar of TypVarName

Obvious homomorphism maps original typ type into its modified version.
Using inductive types, the above definition can be written in NuPRL as

*A typ_var_name TypVarName == Atom × Sort + Indexname × Sort

*A typ       Typ == rec(T.Atom × T List + T × T + TypVarName)

*A type      Type(a;ts) == inl <a, ts>

*A t_fun       (t ⇒ s) == inr (inl <t, s> )

*A t_var      TVar(q) == inr inr q

Many functions on type Typ\(^2\) will be defined using the following case split operator

*A typ_cases  case t
     of Type(a,l) -> typ_case[a; l]
     | (x ⇒ y) -> fun_case[x; y]
     | Var(q) -> var_case[q]
     ==
     case t
     of inl(z0) ⇒ let <a,l> = z0 in typ_case[a; l]
     | inr(z1) ⇒ case z1
     of inl(z2) ⇒ let <x,y> = z2 in
     fun_case[x; y]
     | inr(q) ⇒ var_case[q]

In particular, boolean equality on type Typ is defined by recursion as

*A eq_typ_ml (t1 =tp t2)
     ==r case t1
     of Type(a1,ts1) ->
        case t2

\(^2\) We start the world Typ with the capital letter when it refers to NuPRL formalization
of SML datatype typ.
of Type(a2,ts2) ->
  a1 =a a2 ∧ b ts1 =1 ts2
| (p2 ⇒ q2) -> ff
| Var(n2) -> ff
| (p1 ⇒ q1) ->
  case t2
  of Type(a2,ts2) -> ff
  | (p2 ⇒ q2) ->
    (p1 =tp p2) ∧ b (q1 =tp q2)
  | Var(n1) ->
  case t2
  of Type(a2,ts2) -> ff
  | (p2 ⇒ q2) -> ff
  | Var(n2) -> (n1 =n n2) ,

where =n is a boolean equality on type TypVarName:

*A eq_typ_var_name
(n1 =n n2) ==
  case n1
  of inl(as1) =>
    let <a1,s1> = as1 in
    case n2
    of inl(as2) => let <a2,s2> = as2 in
      a1 =a a2 ∧ b s1 =s s2
    | inr(is2) => ff
  | inr(is1) =>
    let <i1,s1> = is1 in
    case n2
    of inl(as2) => ff
    | inr(is2) => let <i2,s2> = is2 in
      i1 =ixn i2 ∧ b s1 =s s2

The following lemmas are added to EqRW package:

*T eq_typ_var_name_assert
∀n1,n2:TypVarName. ↑(n1 =n n2) ⟷ n1 = n2

*T eq_typ_assert ∀t1,t2:Typ. ↑(t1 =tp t2) ⟷ t1 = t2

Note that although informal proofs of both of these lemmas are rather trivial, the formal proof by induction of the second lemma involves consideration of 9 different cases. Fortunately, EqRW tactic can reduce most of these cases to tautologies that can be handled by NuPRL Auto tactic.

5 Term Syntax

Isabelle defines SML type term as
datatype \text{term} = \text{Const of string} \times \text{typ} \\
| \text{Free of string} \times \text{typ} \\
| \text{Var of indexname} \times \text{typ} \\
| \text{Bound of int} \\
| \text{Abs of string} \times \text{typ} \times \text{term} \\
| \text{op $ of term} \times \text{term}

This type includes well-formed terms as well as non-well-formed terms. There are two conditions which an element of type \text{term} should satisfy in order to be well-formed:

- De Bruijn index \text{i} in any subterm \text{Bound}(i) should be non-negative and less than the number of abstractions above this subterm in the term tree.
- For each occurrence of application operator \text{t}_1 \text{t}_2, the type of term \text{t}_1 should have the form \text{t}_1 \Rightarrow \text{t}_2 where \text{t}_1 is the type of term \text{t}_2. Function “type of” is a recursively defined partial function.

Only well-formed terms are used in Isabelle proofs. Before any term is used in Isabelle, its well-formness is checked via certification process. Well-formed terms are naturally split into groups of terms of the same type.

It is possible to encode this term definition into NuPRL directly. The main disadvantage of this approach is its complexity. All terms will have the same NuPRL type, but different Isabelle types. Hence, Isabelle term types will be defined without using already existing NuPRL type mechanism. More attractive seems to be the idea that a group of Isabelle terms that have the same Isabelle type, should constitute a separate NuPRL type. In this case already existing in NuPRL tactics and theorems can be used to deal with Isabelle types. This approach can be implemented using NuPRL parametrized recursive type constructor.

The key idea is to recursively define a parametrized family of NuPRL types \text{Term}(\text{tp}), where parameter \text{tp} ranges over NuPRL type \text{Typ}. Type \text{Term}(\text{tp}) represents well-formed terms of type \text{tp}. It is defined as a disjoint union of several types, corresponding to different Isabelle term constructors.

\textbf{Constant Terms} Constant terms of Isabelle type \text{tp} are pairs, whose first elements are arbitrary tokens and the second element is \text{tp}:

\*A \text{const\_term} \quad \text{ConstTerm}(\text{tp}) == \text{Atom} \times \{x:\text{Typ} \mid x = \text{tp}\}

\textbf{Variable Terms} Following the definition of type \text{Typ}, constructors \text{Free} and \text{Var} will be combined into one entity

\*A \text{var\_term} \quad \text{VarTerm}(\text{tp}) == \text{VarName} \times \{x:\text{Typ} \mid x = \text{tp}\}

where

\*A \text{var\_name} \quad \text{VarName} == \text{Atom} \times \text{Indexname}
Note that \texttt{VarName} is a decidable type. Boolean equality on this type can be defined through boolean equality on types \texttt{Atom} and \texttt{Indexname}.

\begin{verbatim}
*A eq_var_name (v1 = v2) ==
  case v1
    of inl(a1) => case v2
      of inl(a2) => a1 = a2
      | inr(i2) => ff
    | inr(i1) => case v2
      of inl(a2) => ff
      | inr(i2) => i1 = ixn i2
\end{verbatim}

The following lemma is added to \texttt{eq_rw_lemmas} list.

\begin{verbatim}
*T eq_var_name_assert \forall v, w: VarName. \uparrow(v = v w) \iff v = w
\end{verbatim}

**Bound Variable Term** Since we want to define type \texttt{Term(tp)} recursively, any "partially dis-assembled" term should also be considered to be an element of type \texttt{Term(tp)}. In particular, \texttt{Bound(i)} needs to be an element of type \texttt{Term(tp)} for some element \texttt{tp} of type \texttt{Typ}. At the same time, Isabelle type of subterm \texttt{Bound(i)} can be determined only from the abstraction operator that binds index \texttt{i}. In a "partially dis-assembled" terms an appropriate abstraction operator may not exist. Hence, we can talk about Isabelle type of subterm \texttt{Bound(i)} only with respect to some kind of environment, that stores types from binding abstractions. Such environment will be called \textit{binding}. Binding is specified by its length \texttt{n} and a function \texttt{bd : \mathbb{N}_n \rightarrow Typ} that maps de Bruijn indices into Isabelle types.

Formally, binding is a parameter of type \texttt{Term(tp)}, which, therefore, should be written as \texttt{Term(tp,n,bd)}. A bound variable term of type \texttt{tp} is an integer \texttt{k} such that \texttt{k < n} and \texttt{bd(n - 1 - k) = tp}.

\begin{verbatim}
*A bound_term BoundTerm(tp;n;bd) == {k: \mathbb{N}_n | bd(n - 1 - k) = tp}
\end{verbatim}

**Abstraction Term** In SML any Isabelle abstraction term is composed of variable name \texttt{a} (used only for display purposes), type of this variable \texttt{s}, and a term \texttt{t}. Isabelle type of term \texttt{t} is not a part of the abstraction syntax, but it can be re-constructed using SML \texttt{typ_of} function. In NuPRL, it will be convenient to add the type of the term \texttt{t} to the abstraction syntax.

\begin{verbatim}
*A abs_term AbsTerm(tp;n;bd;tm) ==
  Atom \times s:Typ \times q:Typ\{q:Typ| tp = (s \Rightarrow q)} \times tm < q, n + 1, bd[n \rightarrow s]>
\end{verbatim}

The fourth argument of \texttt{AbsTerm} is a function that maps a triple \langle \texttt{tp,n,bd} \rangle into the type \texttt{Term(tp,n,bd)}. Later this argument will be bound by the parametrized recursive type constructor.

\footnote{We assume here that binding stores type information in the "reverse" order. This unusual assumption greatly simplifies the formalization below.}
**Application Term** Similarly to the abstraction term case, we add one extra type parameter to application term syntax, namely, the Isabelle type of the argument

\[
\text{op_term} \quad 0pTerm(tp;n;bd;tm) \equiv \\
s:Typ \times tm \to (s \Rightarrow tp), n, bd \times tm < s, n, bd>
\]

**Finalizing Term Definition** Any Isabelle term is either a constant, or a free variable, or a bound term, or an abstraction, or an application. Hence, we want to define type \(Term(tp,n,bd)\) to be the minimal solution of the following type equation:

\[
Term(tp,n,bd) = ConstTerm(tp) + VarTerm(tp) + BoundTerm(tp,n,bd) + \\
AbsTerm(tp,n,bd,\lambda tp, \lambda n, \lambda bd. Term(tp,n,bd)) + \\
OpTerm(tp,n,bd,\lambda tp, \lambda n, \lambda bd. Term(tp,n,bd))
\]

This recursive type has three parameters: \(tp, n\), and \(bd\). Since our parametrized recursive type theory (see Section 3.2) permits only one parameter, these three parameters should be combined into one

\[
\text{proto_term} \quad ProtoTerm(p) \equiv \\
\text{parec(tm,p}. \text{let } <tp,z> = p \text{ in let } <n,bd> = z \\
in \\
\text{ConstTerm}(tp) + \text{VarTerm}(tp) + \\
\text{BoundTerm}(tp;n;bd) + \\
\text{AbsTerm}(tp;n;bd;tm) + \\
\text{OpTerm}(tp;n;bd;tm) \to p)
\]

\[
\text{term} \quad Term(t;n;bd) \equiv \text{ProtoTerm(<t, n, bd>)}
\]

\[
\text{const} \quad \text{Const}(a;t) \equiv \text{inl} <a, t>
\]

\[
\text{vari} \quad \text{Vari}(a;t) \equiv \text{inr (inl <a, t> )}
\]

\[
\text{bound} \quad \text{Bound}(k) \equiv \text{inr inr (inl k )}
\]

\[
\text{abs} \quad \lambda a:s. \text{tm}:q \equiv \text{inr inr inr (inl <a, s, q, tm> )}
\]

\[
\text{op} \quad (tm1 \circ tm2) \equiv \text{inr inr inr <s, tm1, tm2>}
\]

Intermediate notion of \textit{proto term} will be extensively used later in proofs by induction on type \(Term(tp,n,bd)\). Since induction rule for parametrized recursive types assumes existence of only one parameter in recursive types, it is
hard to do inductive proofs directly over type $\text{Term}(tp, n, bd)$. Instead, we normally prove an auxiliary theorem carrying induction over type $\text{ProtoTerm}(p)$ and derive from it the main result, stated in terms of type $\text{Term}(tp, n, bd)$.

Among all types $\text{Term}(tp, n, bd)$, special role play those with $n = 0$ since elements of such types correspond to actual Isabelle terms. Display form for type $\text{term}(tp, 0, bd)$ is just $\text{Term}(tp)$.

### 5.1 Operations on Terms

Such operations and predicates on Isabelle terms as “substitution” and “occurs free” will be used later to state Isabelle Meta Logic inference rules. Definitions of these operators are based on term case split constructor

* A term_cases

  case t
  of Const(a, tp) -> const_case[a; tp]
  | Var(a1, tp1) -> var_case[a1; tp1]
  | Bound(k) -> bound_case[k]
  | $\lambda$a2:s. tm:q -> abs_case[a2; s; q; tm]
  | (tm1 o tm2 | s2) -> op_case[s2; tm1; tm2]

  case t
  of inl(z) => let <a, tp> = z in const_case[a; tp]
  | inr(z3) =>
    case z3 of inl(z4) => let <i, tp2> = z4 in var_case[i; tp2]
    | inr(z5) =>
      case z5 of inl(j) => bound_case[j]
      | inr(z6) =>
        case z6 of inl(z7) =>
          let <a2, z6> = z7 in
          let <tp3, z8> = z8 in
          let <tp33, tm> = z88 in
          abs_case[a2; tp3; tp33; tm]
          | inr(z9) =>
            let <tp4, z10> = z9 in
            let <tm1, tm2> = z10 in
            op_case[tp4; tm1; tm2]

**Binding** If $tm$ is a term with a free variable $vn$ of type $tp$, then, before this variable can be bound by an abstraction, it should be converted into a bound
variable. If $tm \in Term(tp', n, bd)$, then such conversion can be done by the function

*\[\begin{align*}
\text{bind} & (tm; vn; tp; n; bd) \\
\text{case } tm & \\
\text{of } \text{Const}(a, t) & \to tm \\
\text{Var}(v, t) & \to \text{if } (v = v \text{ vn}) \land b \text{ (} t = \text{tp tp) then Bound(n) else tm fi} \\
\text{Bound}(j) & \to tm \\
\lambda a : s. \; m : q & \to \lambda a : s. \; \text{bind}(m; vn; tp; n + 1; bd[n \to s]):q \\
(f \circ m \mid s) & \to (\text{bind}(f; vn; tp; n; bd) \circ \text{bind}(m; vn; tp; n; bd))
\end{align*}\]

This function substitutes an appropriate de Bruijn index $Bound(i)$ for every occurrence of variable $Var(vn, tp)$ in term $tm$. The resulting term has one extra element in the binding

*\[\begin{align*}
\text{bind_wf} & (tm) \\
\forall t : \text{Typ}. \; \forall n : \text{N}. \; \forall bd : \text{N}n & \to \text{Typ}. \; \forall tm : \text{Term}(t; n; bd). \\
\forall vn : \text{VarName}. \; \forall tp : \text{Typ}. \; \forall k : \text{N}. \; k = n + 1 & \Rightarrow \\
\text{bind}(tm; vn; tp; n; bd) & \in \text{Term}(t; k; [tp >> bd])
\end{align*}\]

where $[tp >> bd]$ is the operator shift on records that was discussed in Section 3.1.

**Substitution** The operator

*\[\begin{align*}
\text{subs} & (tm[vn, tp \to t]) \\
\text{case } tm & \\
\text{of } \text{Const}(a, s) & \to tm \\
\text{Var}(w, s) & \to \text{if } (w = v \text{ vn}) \land b \text{ (} s = \text{tp tp) then } t \text{ else } tm \text{ fi} \\
\text{Bound}(j) & \to tm \\
\lambda a : s. \; m : q & \to \lambda a : s. \; m[vn, tp \to t]:q \\
(f \circ m \mid s) & \to (f[vn, tp \to t] \circ m[vn, tp \to t])
\end{align*}\]

substitutes term $t$ for every occurrence of variable $Var(vn, tp)$ in the term $tm$. In order to avoid de Bruijn indices collision, term $t$ in the above definition should have nil binding

*\[\begin{align*}
\text{subs} & (tm[vn, tp \to m]) \\
\forall t : \text{Typ}. \; \forall n : \text{N}. \; \forall bd : \text{N}n & \to \text{Typ}. \; \forall tm : \text{Term}(t; n; bd). \\
\forall vn : \text{VarName}. \; \forall tp : \text{Typ}. \; \forall b : \text{N} & \to \text{Typ}. \; \forall m : \text{Term}(tp). \\
\text{tm}[vn, tp \to m] & \in \text{Term}(t; n; bd)
\end{align*}\]

**Free Occurrence** Some logical rules require a variable not to occur free in a term. We prefer to define boolean predicate “variable $Var(vn, tp)$ does occur in term $tm$”:
6 Interpretation

6.1 Type Interpretation

In order to define Isabelle type term interpretation in NuPRL, one needs to select evaluation of type variables and type constructors. After that an interpretation can be extrapolated on all type terms. The basic evaluation of constructors and variables will be called type evaluation

*A t_eval TEval == (Atom → U List → U) × (TypVarName → U)

For any given type evaluation ev, an interpretation of a type term t is defined by recursion:

*M t_interp_ml ρ(t) ==r
case t
    of Type(a,ts) → ev.1 a map(λx.ρ(x); ts)
        | (p ⇒ q) → ρ(p) → ρ(q)
        | Var(n) → ev.2 n

Type interpretation ρ(t) has two arguments: type term t and type evaluation ev. The last one is not normally displayed.

6.2 Term Evaluation

Interpretation of an Isabelle term will be defined with respect to a type evaluation tev and term evaluation ev, where term evaluation assigns values to atomic terms – constants and variables. Term evaluation is called just evaluation

*A eval Eval(tev) == (Atom → t:Typ → ρ(t)) ×

         | (VarName → t:Typ → ρ(t))

Application of an evaluation ev to constants and variables is called constant evaluation and variable evaluation correspondingly:

*A const_eval ConstEval(a;t|e) == e.1 a t
*A var_eval VarEval(i;t|e) == e.2 i t
Later, verifying Isabelle Meta Logic rules, we will be using operator

\*A eval_update ev[vn,tp \to val] ==
<ev.1, \lambda w,s.if (w =v vn) \land_b (s =tp tp) then val else ev.2 w s fi >

that changes evaluation function \( ev \) at the point \( Var(vn,tp) \). Although similar update operator can be defined to change evaluation on constants, it is never used.

6.3 Binding Evaluation

If an Isabelle term \( tm \) belongs to a NuPRL type \( Term(tp,n,bd) \) and \( n > 0 \), then \( tm \) can have occurrences of de Bruijn indices that are not bound by any abstraction. Interpretation of such dis-assembled terms can be defined only if we assign first some specific values to unbound indices. We call this assignment a binding value. Binding value of elements of type \( Term(tp,n,bd) \) has to map each integer index \( i \), such that \( i < n \), into an element of NuPRL type \( \rho(bd(i)) \):

\*A bind_value BndVal(n;bd) == i:Nn \to \rho(bd i)

Binding values are essentially records that we have discussed in Section 3.1. Introduced there operators update, shift, and tail can be applied to binding values as well. Although it is more convenient to define duplicates of these operators to deal specifically with binding values. For example,

\*A bdv_update (bdv:bd)[n \to v:t] == bdv[n \to v]

Note that extra parameters \( bd \) and \( t \) do not appear on the right hand side of the definition. These are “dummy” parameters incorporated into bdv_update to assist NuPRL type guessing procedure. They can be ignored from the logical point of view. Normally we would make such parameters hidden, but in this particular case they are sometimes helpful for proof understanding.

Operators bdv_shift and bdv_tail also have dummy parameters for type guessing, but in our formalization they are hidden:

\*A bdv_shift [v >>> bdv] == [v >> bdv]

\*A bdv_tail (bdv|k) == (bdv|k)

We also define bdv_apply operator as a duplicate of the standard NuPRL application. This definition is also needed only in order to assist type guessing procedure.

\*A bdv_apply [bdv](i) == bdv i

Basic properties of operators update and shift can be re-formulated in terms of newly-introduced abstractions:
\*T \texttt{bdv\_shift\_zero} \quad \forall T: U. \quad \forall t', t: T. \quad \forall i: N. \quad \forall tp, bdv, n, bd: \text{Top}.
\quad t = t' \Rightarrow i = 0 \Rightarrow [[[t >>> bdv]](i) = t'

\*T \texttt{bdv\_shift\_pos} \quad \forall tv: \text{TEval}. \quad \forall n: N. \quad \forall bd: \text{Nn} \rightarrow \text{Typ}.
\quad \forall t: \text{Top}. \quad \forall bdv: \text{BindVal}(n; bd). \quad \forall i: \{1..(n + 1)\}.
\quad \forall j: Nn. \quad \forall tp: \text{Typ}. \quad \text{bd } j = \text{tp} \Rightarrow j = i - 1
\quad \Rightarrow [[[v >>> bdv]](i) = [bdv](j)

\*T \texttt{bdv\_shift\_update} \quad \forall tv: \text{TEval}. \quad \forall n: N. \quad \forall bd: \text{Nn} \rightarrow \text{Typ}. \quad \forall t, s: \text{Typ}.
\quad \forall v: \rho(t). \quad \forall w: \rho(s). \quad \forall bdv: \text{BindVal}(n; bd).
\quad \forall v >>> (bdv: bd)[n \Rightarrow w: s] =
\quad (\forall v >>> (bdv: bd)[t \Rightarrow bd])[n + 1 \Rightarrow w: s]

\*T \texttt{bdv\_shift\_nil\_update} \quad \forall tv: \text{TEval}. \quad \forall n: N. \quad \forall bd: \text{Nn} \rightarrow \text{Typ}. \quad \forall t: \text{Typ}.
\quad \forall bdv: \text{BindVal}(0; bd). \quad \forall v: \rho(t).
\quad \forall v >>> (bdv: bd)[0 \Rightarrow v: t]

6.4 Finalizing Interpretation Definition

For any binding value \( bdv \), and an evaluation \( ev \), we define an interpretation of a term \( t \) recursively as

\*M \texttt{interp\_ml} \quad \beta(t \mid bdv, ev)
\quad \Rightarrow \text{case } t \text{ of }
\quad \quad \text{Const}(a, tp) \rightarrow \text{ConstEval}(a; tp \mid ev)
\quad \quad \text{Var}(i, tp) \rightarrow \text{VarEval}(i; tp \mid ev)
\quad \quad \text{Bound}(j) \rightarrow [bdv](n - 1 - j)
\quad \quad \lambda a: s. \quad \text{tm}: q \rightarrow \lambda z, \beta(tm \mid (bdv: bd)[n \Rightarrow z: s], ev)
\quad \quad (f \circ \text{tm} \mid s) \rightarrow \beta(f \mid bdv, ev) \beta(tm \mid bdv, ev)

If term \( t \) has Isabelle type \( tp \), then its interpretation belongs to NuPRL type \( \rho(tp) \)

\*T \texttt{interp\_wf} \quad \forall tv: \text{TEval}. \quad \forall ev: \text{Eval}(tev). \quad \forall tp: \text{Typ}. \quad \forall n: N.
\quad \forall bd: \text{Nn} \rightarrow \text{Typ}. \quad \forall t: \text{Term}(tp; n; bd). \quad \forall bdv: \text{BindVal}(n; bd).
\quad \beta(t \mid bdv, ev) \in \rho(tp)

The following lemmas show the relation between term operators and the defined above interpretation

\*T \texttt{interp\_bind\_tm}
\quad \forall tv: \text{TEval}. \quad \forall ev: \text{Eval}(tev). \quad \forall vn: \text{VarName}. \quad \forall tp, t: \text{Typ}. \quad \forall n: N.
\quad \forall bd: \text{Nn} \rightarrow \text{Typ}. \quad \forall tm: \text{Term}(t; n; bd). \quad \forall \rho(tp). \quad \forall bdv: \text{BindVal}(n; bd).
\quad \forall k: N. \quad k = n + 1 \Rightarrow \beta(\text{bind}(tm; vn; tp; n; bd) \mid [w >>> bdv], ev) =
\quad \beta(tm \mid bdv, ev[vn, tp \rightarrow w])

\*T \texttt{interp\_subs\_tm}
\quad \forall t: \text{Typ}. \quad \forall n: N. \quad \forall bd: \text{Nn} \rightarrow \text{Typ}. \quad \forall tm: \text{Term}(t; n; bd). \quad \forall tv: \text{TEval}.
\[ \forall \text{ev:Eval}(\text{tev}). \forall \text{bdv:BindVal}(n;bd). \forall \text{vn:VarName}. \forall \text{tp:Typ}. \]
\[ \forall b: \text{N} \rightarrow \text{Typ}. \forall \text{bv:BindVal}(0;b). \forall \text{m:Term}(\text{tp}). \]
\[ \beta(\text{tm}[\text{vn, tp} \mapsto \text{m}])|_{\text{bdv}, \text{ev}} = \beta(\text{tm}|_{\text{bdv}, \text{ev}[, \text{vn, tp} \mapsto \beta(\text{m}|_{\text{bdv}, \text{ev})]}}) \]

* \text{T interp_free_tm}

\[ \forall \text{vn:VarName}. \forall \text{tp:Typ}. \forall \text{tev:TEval}. \forall \text{ev:Eval}(\text{tev}). \forall t: \text{Typ}. \forall n: \text{N}. \]
\[ \forall \text{bdv:N} \rightarrow \text{Typ}. \forall \text{tm:Term}(t;n;bd). \forall \text{bdv:BindVal}(n;bd). \forall w: \rho(\text{tp}). \]
\[ \rightarrow \text{Free}(\text{vn};\text{tp};\text{tm}) \Rightarrow \beta(\text{tm}|_{\text{bdv, ev}[, \text{vn, tp} \mapsto w]) = \beta(\text{tm}|_{\text{bdv, ev})} \]

They will be used in the next section to verify Isabelle meta rules.

7 Isabelle Meta Logic

In the previous sections we have formalized Isabelle syntax and defined its interpretation in NuPRL. In this section we state Isabelle meta rules and show that they are translated into valid NuPRL statements.

7.1 Meta Logic Syntax

**Propositions** Isabelle Meta Logic declares a type constant o, which stands for Isabelle type of all propositions. In NuPRL we represent this declaration by the following definition:

\[ * \text{A prop_typ 0} = \text{Type("prop"; \text{\[\]})} \]

We restrict the class of possible type evaluations to such evaluations that map constant o into NuPRL boolean type \( \mathbb{B} \). Technically, this restriction is put by adding condition

\[ (\text{pr}_1(\text{tev}))("\text{prop";} []) = \mathbb{B} \]

as a hypothesis to all theorems that we are proving about type o. We call the above condition *propositional signature*

\[ * \text{A prop_sign PropSign} = \text{tev.1 "prop" \[\] = B} \]

Propositional signature has type evaluation tev and universe level i as hidden parameters.

**Implication** Isabelle declares \( \Rightarrow \) as a constant of type o \( \Rightarrow o \Rightarrow o \). Accordingly, in NuPRL we define Isabelle implication as

\[ * \text{A imp_const \[\Rightarrow\] = Const("\Rightarrow";} (0 \Rightarrow (0 \Rightarrow 0))) \]

We will assume that Isabelle implication is interpreted as NuPRL boolean implication:

\[ * \text{A imp_sign ImpSign = ConstEval("\Rightarrow";} (0 \Rightarrow (0 \Rightarrow 0))|_{\text{ev}) = (\lambda x. y. x \Rightarrow b \ y) \]

Each time when constant \( \Rightarrow \) is used in Isabelle meta rules, it is applied to a pair of arguments. As a result, the following notation is handy:

\[ * \text{A imp \[\Rightarrow \} o q) = ((\Rightarrow o p) o q) \]
Universal Quantifier  Isabelle declares universal quantifier $\forall$ as a constant of the type $\forall (\alpha \Rightarrow o) \Rightarrow \alpha$. Although at the first glance it seems that the same constant $\forall$ belongs to the type $\forall (\alpha \Rightarrow o) \Rightarrow o$ for any type $\alpha$ of class $\text{term}$, this is not true. Any instance of constant $\forall$ in any Isabelle term has form $\text{Const("all"}, (\alpha \Rightarrow o) \Rightarrow o)$ for a particular type term $\alpha$. Hence, $\forall$ is actually a family of constants, parametrized by $\alpha$.

* $A \text{ all} \_\text{const}$  $\bigcap (\forall a) = \text{Const("all"}; (\forall a \Rightarrow (\forall a \Rightarrow 0))$

* $A \text{ all} \_\text{sign}$  $\text{AllSign} = \forall \alpha : \text{Typ. ConstEval("all"}; (\forall a \Rightarrow (\forall a \Rightarrow 0)) | \text{ev} = (\lambda b. \forall x : \rho (\forall a). b x)$

* $A \text{ iall}$  $\bigcap (\forall a : s. p) = (\bigcap (s) \circ \alpha a : s. p : 0)$

We added letter $i$ in the name “iall” in order to avoid name collision with standard NuPRL universal quantifier.

Equality Isabelle declares equality predicate as a constant of the type $\forall (\alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow o)$. Just like the universal quantifier, this constant actually is a family of constants, parametrized by $\alpha$:

* $A \text{ eq} \_\text{const}$  $\text{eq\_const}(\forall a) = \text{Const("=="}; (\forall a \Rightarrow (\forall a \Rightarrow 0))$

* $A \text{ eq} \_\text{sign}$  $\text{EqSign} = \forall \alpha : \text{Typ. ConstEval("=="}; (\forall a \Rightarrow (\forall a \Rightarrow 0)) | \text{ev} = (\lambda x, y : (x \equiv y))$

* $A \text{ eq}$  $(x \equiv y) = ((\text{eq\_const}(t p) \circ x) \circ y)$

Operator $eq$ hides parameter $t p$ to make formulas more readable.

Re-writing enhancements The following three lemmas are added to the lemma list used by EqRW tactic:

* $T \text{ imp} \_\text{interp}$  $\forall a : \text{N. Vbd: Nn} \Rightarrow \text{Typ. Vp, q : Term(0; n; bd). Vtev:\text{TEval. Vev : Eval(tev). Vbdv: BindVal(n; bd). PropSign \Rightarrow ImpSign \Rightarrow \beta (p \Rightarrow q) \mid \text{bdv, ev} = \beta (p \mid \text{bdv, ev}) \Rightarrow \beta (q \mid \text{bdv, ev})}$

* $T \text{ all} \_\text{interp}$  $\forall a : \text{Atom. Vbdv: Nn} \Rightarrow \text{Typ. Vbdv: BindVal(n; bd). PropSign \Rightarrow AllSign \Rightarrow \beta (\forall (a : s. p) \mid \text{bdv, ev}) = \forall x : \rho (s). \beta (p \mid (\text{bdv; bd}) [n \Rightarrow x : s], ev)$

* $T \text{ eq} \_\text{interp}$  $\forall a : \text{N. Vbdv: Nn} \Rightarrow \text{Typ. Vtp: Typ. Vx, y : Term(tp; n; bd). Vtev:TEval. Vev : Eval(tev). Vbdv: BindVal(n; bd). PropSign \Rightarrow EqSign \Rightarrow \beta (x \equiv y) \mid \text{bdv, ev} = (\beta (x \mid \text{bdv, ev}) \equiv \beta (y \mid \text{bdv, ev}))$
7.2 Meta Logic Rules

Isabelle Meta Logic rules stated in [11] are reproduced on Figure 2. The following

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>2</td>
<td>$\phi \vdash \phi$</td>
</tr>
<tr>
<td>3</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>4</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>5</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>6</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>7</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>8</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>9</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>10</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>11</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>12</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>13</td>
<td>$\phi \vdash \psi$</td>
</tr>
<tr>
<td>14</td>
<td>$\phi \vdash \psi$</td>
</tr>
</tbody>
</table>

---

**Fig. 2.** Isabelle Meta Logic rules

Theorem verify these rules in the classical extension of NuPRL. Accompanying them proof sketches are aimed to give a general idea about the proof. Complete formal proofs are available from the author's Web page.

* * rule_1

$\forall \text{tev:TEval}. \forall n: N. \forall bd: \mathbb{N} \rightarrow \text{Typ}. \forall \text{bdv:BindVal}(n; bd).$

$\forall p, q: \text{Term}(0 ; n ; bd). \forall \text{ev: Eval}(\text{tev}). \text{PropSign} \Rightarrow \text{ImpSign} \Rightarrow$

$(\uparrow \beta(p \mid \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta(q \mid \text{bdv}, \text{ev})) \Rightarrow$

$(\uparrow \beta(p \Rightarrow q) \mid \text{bdv}, \text{ev})$

**Proof Sketch** Tactic Eq\v{r}W reduces this theorem to a propositional tautology of the form $a \Rightarrow a$. □

* * rule_2

$\forall \text{tev:TEval}. \forall n: N. \forall bd: \mathbb{N} \rightarrow \text{Typ}. \forall \text{bdv:BindVal}(n; bd).$

$\forall p, q: \text{Term}(0 ; n ; bd). \forall \text{ev: Eval}(\text{tev}). \text{PropSign} \Rightarrow \text{ImpSign} \Rightarrow$

$(\uparrow \beta(p \Rightarrow q) \mid \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta(p \mid \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta(q \mid \text{bdv}, \text{ev})$

**Proof Sketch** Similar to Rule 1. □

* * rule_3

$\forall \text{tev:TEval}. \forall bd: \mathbb{N} \rightarrow \text{Typ}. \forall \text{bdv:BindVal}(0; bd). \forall p: \text{Term}(0).$

$\forall a: \text{Atom}. \forall s: \text{Typ}. \forall v: \text{VarName}. \text{PropSign} \Rightarrow$

$(\forall \text{ev: Eval}(\text{tev}). \text{AllSign} \Rightarrow \uparrow \beta(p \mid \text{bdv}, \text{ev})) \Rightarrow$

$(\forall \text{ev: Eval}(\text{tev}). \text{AllSign} \Rightarrow \uparrow \beta(\cap(a: a. \text{bind}(p; v; s)) \mid \text{bdv}, \text{ev}))$
Proof Sketch. After the reduction by EqRW, the statement of this theorem follows from lemma interp.bind tm.

* T rule.4
\[ \forall \text{tev}:T\text{Eval}. \forall \text{ev}:\text{Eval(tev)}. \forall \text{bd}:\mathbb{N} \rightarrow \text{Typ}. \forall \text{bdv}:\text{BindVal}(0; \text{bd}). \]  
\[ \forall \text{p}:\text{Term}(0). \forall \text{a}:\text{Atom}. \forall \text{s}:\text{Typ}. \forall \text{v}\text{n}:\text{VarName}. \forall \text{v}\text{m}:\text{Term}(\text{s}). \]  
\[ \text{PropSign} \Rightarrow \text{AllSign} \Rightarrow \uparrow \beta(\langle \text{a} : \text{s}. \text{bind}(\text{p}; \text{v}\text{n}; \text{s}) \rangle | \text{bdv}, \text{ev}) \]  
\[ \Rightarrow \uparrow \beta(\langle \text{v}\text{n}, \text{s} \rightarrow \text{m} \rangle | \text{bdv}, \text{ev}) \]

Proof Sketch. After the reduction by EqRW, the statement of this theorem follows from lemmas interp.bind tm and interp.subs tm.

* T rule.5
\[ \forall \text{tev}:T\text{Eval}. \forall \text{ev}:\text{Eval(tev)}. \forall \text{n}:\mathbb{N}. \forall \text{bd}:\mathbb{N} \rightarrow \text{Typ}. \]  
\[ \forall \text{bdv}:\text{BindVal}(\text{n}; \text{bd}). \forall \text{tp}:\text{Typ}. \forall \text{va}:\text{Term}(\text{tp}; \text{bd}). \text{PropSign} \Rightarrow \]  
\[ \text{EqSign} \Rightarrow \uparrow \beta(\langle \text{a} \equiv \text{a} \rangle | \text{bdv}, \text{ev}) \]

Proof Sketch. Tactic EqRW reduces this theorem to a tautology of the form 
\[ x = x. \]

* T rule.6
\[ \forall \text{tev}:T\text{Eval}. \forall \text{ev}:\text{Eval(tev)}. \forall \text{n}:\mathbb{N}. \forall \text{bd}:\mathbb{N} \rightarrow \text{Typ}. \]  
\[ \forall \text{bdv}:\text{BindVal}(\text{n}; \text{bd}). \forall \text{tp}:\text{Typ}. \forall \text{va}, \text{vb}:\text{Term}(\text{tp}; \text{bd}). \text{PropSign} \Rightarrow \]  
\[ \text{EqSign} \Rightarrow \uparrow \beta(\langle \text{a} \equiv \text{b} \rangle | \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta(\langle \text{b} \equiv \text{a} \rangle | \text{bdv}, \text{ev}) \]

Proof Sketch. Tactic EqRW reduces this theorem to a tautology of the form 
\[ x = y \Rightarrow y = x. \]

* T rule.7
\[ \forall \text{tev}:T\text{Eval}. \forall \text{ev}:\text{Eval(tev)}. \forall \text{n}:\mathbb{N}. \forall \text{bd}:\mathbb{N} \rightarrow \text{Typ}. \]  
\[ \forall \text{bdv}:\text{BindVal}(\text{n}; \text{bd}). \forall \text{tp}:\text{Typ}. \forall \text{va}, \text{vb}, \text{vc}:\text{Term}(\text{tp}; \text{bd}). \text{PropSign} \Rightarrow \]  
\[ \text{EqSign} \Rightarrow \uparrow \beta(\langle \text{a} \equiv \text{b} \rangle | \text{bdv}, \text{ev}) \Rightarrow \]  
\[ \uparrow \beta(\langle \text{b} \equiv \text{c} \rangle | \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta(\langle \text{a} \equiv \text{c} \rangle | \text{bdv}, \text{ev}) \]

Proof Sketch. Tactic EqRW reduces this theorem to a tautology of the form 
\[ x = y \Rightarrow y = z \Rightarrow x = z. \]

* T rule.8
\[ \forall \text{tev}:T\text{Eval}. \forall \text{ev}:\text{Eval(tev)}. \forall \text{n}:\mathbb{N}. \forall \text{bd}:\mathbb{N} \rightarrow \text{Typ}. \]  
\[ \forall \text{va}, \text{vb}:\text{Atom}. \forall \text{vs}, \text{q}:\text{Typ}. \forall \text{vtm}:\text{Term}(\text{q}; \text{n} + 1; \text{bd}[\text{n} \rightarrow \text{s}]). \]  
\[ \forall \text{bdv}:\text{BindVal}(\text{n}; \text{bd}). \text{PropSign} \Rightarrow \text{EqSign} \Rightarrow \]  
\[ \uparrow \beta(\langle \lambda \text{s}:\text{tm}:\text{q} \equiv \lambda \text{b}:\text{s}. \text{tm}:\text{q} \rangle | \text{bdv}, \text{ev}) \]

Proof Sketch. Tactic EqRW reduces this theorem to a tautology of the form 
\[ x = x, \]  
because an interpretation of an abstraction does not depend on the name of the variable used in the abstraction syntax.
\*T rule_9
\( \forall \text{tev}: \text{TEval}. \ \forall \text{ev}: \text{Eval} (\text{tev}). \ \forall \text{bd} : \text{No} \rightarrow \text{Typ}. \ \forall \text{bdv}: \text{BindVal}(0; \text{bd}). \ \forall \text{a}: \text{Atom}. \ \forall s, q : \text{Typ}. \ \forall \text{vn} : \text{VarName}. \ \forall t : \text{Term}(s). \ \forall m : \text{Term}(q). \ PropSign \Rightarrow \ EqSign \Rightarrow \\
\uparrow \beta ((\lambda a. q. \text{bind}(t; \text{vn}; q) : s \circ m) \equiv t[\text{vn}, q \rightarrow m]) \mid \text{bdv}, \text{ev}) \)

**Proof Sketch** After the reduction by EqRW, the statement of this theorem follows from lemmas interp\_bind\_tm and interp\_subs\_tm.

\*T rule_10
\( \forall \text{tev}: \text{TEval}. \ \forall n : \text{N}. \ \forall \text{bd} : \text{No} \rightarrow \text{Typ}. \ \forall \text{bdv}: \text{BindVal}(n; \text{bd}). \ \forall s, q : \text{Typ}. \ \forall f, g : \text{Term}(s \Rightarrow q); n; \text{bd}). \ \forall \text{vn} : \text{VarName}. \ PropSign \Rightarrow \\
\uparrow \gamma \text{Free}(x; s) \Rightarrow \uparrow \gamma \text{Free}(x; s; g) \Rightarrow \\
(\forall \text{ev}: \text{Eval}(\text{tev}). \ EqSign \Rightarrow \uparrow \beta ((f \circ \text{Var}(x; s)) \equiv (g \circ \text{Var}(x; s))) \mid \text{bdv}, \text{ev})) \\
\Rightarrow (\forall \text{ev}: \text{Eval}(\text{tev}). \ EqSign \Rightarrow \uparrow \beta ((f \equiv g) \mid \text{bdv}, \text{ev})) \)

**Proof Sketch** After the reduction by EqRW, the statement of this theorem follows from lemma interp\_free\_tm.

\*T rule_11
\( \forall \text{tev}: \text{TEval}. \ \forall \text{bd} : \text{No} \rightarrow \text{Typ}. \ \forall \text{bdv}: \text{BindVal}(0; \text{bd}). \ \forall t : \text{Typ}. \ \forall s, q : \text{Typ}. \ \forall \text{vn} : \text{VarName}. \ \forall p, q : \text{Term}(t). \ PropSign \Rightarrow \\
(\forall \text{ev}: \text{Eval}(\text{tev}). \ EqSign \Rightarrow \uparrow \beta ((p \equiv q) \mid \text{bdv}, \text{ev})) \Rightarrow \\
(\forall \text{ev}: \text{Eval}(\text{tev}) \ EqSign \Rightarrow \\
\uparrow \beta ((\lambda a : s. \text{bind}(p; \text{vn}; s) : \text{tp} \equiv \lambda a : s. \text{bind}(q; \text{vn}; s) : \text{tp}) \mid \text{bdv}, \text{ev})) \)

**Proof Sketch** After the reduction by EqRW, the statement of this theorem follows from lemmas interp\_bind\_tm.

\*T rule_12
\( \forall \text{tev}: \text{TEval}. \ \forall \text{ev}: \text{Eval}(\text{tev}). \ \forall t : \text{Typ}. \ \forall n : \text{N}. \ \forall \text{bd} : \text{No} \rightarrow \text{Typ}. \ \forall \text{bdv}: \text{BindVal}(n; \text{bd}). \ \forall s : \text{Typ}. \ \forall f, g : \text{Term}(s \Rightarrow \text{tp}); n; \text{bd}). \ \forall a, b: \text{Term}(s; n; \text{bd}). \ PropSign \Rightarrow \\
EqSign \Rightarrow \uparrow \beta ((f \equiv g) \mid \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta ((a \equiv b) \mid \text{bdv}, \text{ev}) \Rightarrow \\
\uparrow \beta (((f \circ a) \equiv (g \circ b)) \mid \text{bdv}, \text{ev}) \)

**Proof Sketch** Tactic EqRW reduces this theorem to a tautology of the form 
\( u = v \Rightarrow x = y \Rightarrow u(x) = v(y) \).

\*T rule_13
\( \forall \text{tev}: \text{TEval}. \ \forall n : \text{N}. \ \forall \text{bd} : \text{No} \rightarrow \text{Typ}. \ \forall \text{bdv}: \text{BindVal}(n; \text{bd}). \ \forall p, q : \text{Term}(0; n; \text{bd}). \ \forall \text{ev}: \text{Eval}(\text{tev}). \ PropSign \Rightarrow EqSign \Rightarrow \\
(\uparrow \beta (p \mid \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta (q \mid \text{bdv}, \text{ev})) \Rightarrow \\
(\uparrow \beta (q \mid \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta (p \mid \text{bdv}, \text{ev})) \Rightarrow \uparrow \beta ((p \equiv q) \mid \text{bdv}, \text{ev}) \)

**Proof Sketch** The statement of this theorem, after the reduction by tactic EqRW, follows from the standard NuPRL lemma \texttt{iff_imp_equal_bool}.
\*T \text{rule}_{14} \\
\forall \text{tev}: \text{Eval}. \ \forall n: \mathbb{N}. \ \forall \text{bd}: \mathbb{N} \rightarrow \text{Typ}. \ \forall \text{bdv}: \text{BindVal}(n; \text{bd}). \\
\forall p,q: \text{Term}(0; n; \text{bd}). \ \forall \text{tev}: \text{Eval}(\text{tev}). \ \text{PropSign} \Rightarrow \text{EqSign} \Rightarrow \\
\uparrow \beta((p \equiv q) | \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta(p | \text{bdv}, \text{ev}) \Rightarrow \uparrow \beta(q | \text{bdv}, \text{ev})

\textbf{Proof Sketch} Tactic \text{EqRW} reduces this theorem to a tautology of the form \((x = y) \Rightarrow x \Rightarrow y\). \hfill \square

8 Acknowledgements

Author would like to thank Robert L. Constable for numerous discussions and support of this project and Doug J. Howe for the time and effort of explaining technical details of his work.

References