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 $add(x, y)$ is $x + y$ and the one for universal quantifier $all(T, x.P)$ 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

$equal(T, x, y)$ 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).

status	kind	name	content
*	T	int_trichot	∀i,j:ℤ. i < j ∨ i = j ∨ i > j

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   Parens ::Prec(preop)::  ¬b<b:bool:E>== bnot{}(<b>)

*A bnot      ¬bb ==  if b then ff else tt fi

*T bnot_wf   ∀b:𝔹. ¬bb ∈ 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 [] => 0
                           | a::as' => ||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

```
*M eq_list_ml  l =l m
               ==r if null(l) ∧b null(m) then tt
                   if null(l) ∨b null(m) then ff
                   else E[hd(l); hd(m)] ∧b tl(l) =l tl(m)
                   fi
```

defines such boolean equality on T *List* assuming that E is a boolean equality on type T. List boolean equality is written as $=_l$. This notation hides parameter E. The fact that $=_l$ is indeed a boolean equality on type T *List* is expressed in the following lemma:

```
*T eq_list_assert
               ∀T:𝕌. ∀E:T → T → 𝔹.
                 (∀t,t':T.  ↑E[t;t'] ⟺ t = t') ⇒
                    (∀ts,ts':T List.  ↑ts =l ts' ⟺ ts = ts')
```

If relation E is a boolean equality only on a subset T' of type T, $=_l$ is a boolean equality on type T' *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:

```
*T eq_list_assert_2
     ∀T:𝕌. ∀Q,R:T → ℙ.
       ∀E:{x:T| Q[x]}  → {x:T| R[x]}  → 𝔹.
        (∀t:{x:T| Q[x]} . ∀t':{x:T| R[x]} .  ↑E[t;t'] ⟺ t = t')
           ⇒ (∀ts:{x:T| Q[x]}  List. ∀ts':{x:T| R[x]}  List.
                 ↑ts =l ts' ⟺ ts = ts')
```

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 [1] indices, binding is a function from an initial segment of natural numbers into a type of Isabelle type names. Such functions are commonly called *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: *update*, *shift*, and *tail*. Function *update*

```
*A update     f[n → a] ==  λz.if (z =z n) then a else f z fi
```

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 *shift*

```
*A shift      [s >> f] ==  λz.if (z =z 0) then s else f (z - 1) fi
```

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 *cons* on lists. Function *tail*

```
*A tail        (f|n) ==  λx.f (n + x)
```

is similar to the 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:

```
*T shift_update
               ∀T:𝕌. ∀n:ℕ. ∀f:ℕn → T. ∀a,z:T.
                  [a >> f[n → z]] = [a >> f][n + 1 → z]

*T shift_zero ∀T:𝕌. ∀t',t:T. ∀n:ℕ. ∀f:Top.
                  t = t' ⇒ n = 0 ⇒ [t >> f] n = t'

*T shift_pos  ∀n:ℕ. ∀T:ℕn → 𝕌. ∀T',T'':𝕌. ∀t:Top.
                 ∀f:j:ℕn → T[j]. ∀i:{1..(n + 1)⁻}. ∀t':T'.
                    T' = T'' ⇒ T[i - 1] = T' ⇒ f (i - 1) = t'
                      ⇒ [t >> f] i = t'

*T shift_nil_update
              ∀T:𝕌. ∀f:ℕ0 → T. ∀s:T.  [s >> f] = f[0 → s]

*T zero_tail  ∀T:𝕌. ∀n:ℕ. ∀f:ℕn → T. ∀n',n'':ℕ. ∀g:ℕ0 → T.
                  n = n' ⇒ n'' = 0 ⇒ g = (f|n')
```

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:

```
*T inl_eq       ∀T,S:𝕌. ∀x,y:T.  (inl x ) = (inl y ) ⟺ x = y

*T inr_eq       ∀T,S:𝕌. ∀x,y:T.  (inr x ) = (inr y ) ⟺ x = y

*T inr_inl_eq   ∀T,S:𝕌. ∀x:T. ∀y:S.
                  (inr y ) = (inl x ) ⟺ False

*T inl_inr_eq   ∀T,S:𝕌. ∀x:T. ∀y:S.
                  (inl x ) = (inr y ) ⟺ False
```

```
*T pair_eq      ∀T,S:𝕌. ∀t,t':T. ∀s,s':S.
                  <t, s> = <t', s'> ⟺ t = t' ∧ s = s'
```

Some lemmas from the classical NuPRL extension and lemmas about Isabelle-specific abstractions are also on the `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 $b : \mathbb{U} \to \mathbb{U}$ to define a new type $rec(X.B(X))$ 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:

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

$$\dots$$

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

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

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

where $X : I \to \mathbb{U}$, $b : I \to (I \to \mathbb{U}) \to \mathbb{U}$. 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 $parec(X, i.b(X, i)@i_0)$.

Inference rules and semantics for such types have been introduced in [7] and [6]. These rules were added to NuPRL as an extension of Constructive Type Theory by the author in [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 $\mathbb{P}$. Unfortunately, this mapping is hard to implement since NuPRL type $\mathbb{P}$ 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

$$\frac{\begin{matrix}[\phi] & [\psi] \\ \psi & \phi\end{matrix}}{\phi \equiv_{prop} \psi}$$

would not be valid when Isabelle propositions are interpreted as NuPRL propositions 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 equivalence 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:

$$\mathbb{P}_1 \subseteq \mathbb{P}_2 \subseteq \ldots \subseteq \mathbb{P}_n \subseteq \ldots$$

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 function, 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

```
*T bequal_wf     ∀T:𝕌. ∀x,y:T.  (x =_b y) ∈ 𝔹
```

```
*T assert_bequal ∀T:𝕌. ∀x,y:T.  ↑(x =_b y) ⟺ 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:

```
*T ball_wf       ∀T:𝕌. ∀b:T → 𝔹.  ∀_b x:T. b[x] ∈ 𝔹

*T assert_ball   ∀T:𝕌. ∀b:T → 𝔹.  ↑∀_b x:T. b[x] ⟺ (∀x:T. ↑b[x])
```

Note that the proposed classical NuPRL extension is not minimal. Boolean universal quantifier can be expressed via propositional universal quantifier and boolean equality as

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

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 `bequal_assert` and `ball_assert` on the list `eq_rw_lemmas` to be used by `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 *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 $\mathbb{U}_i$ of an arbitrary level i.

4.1 Classes and Sorts

Isabelle defines SML type `class` as type `string`, SML type `sort` as type `class list`, and SML type `indexname` as Cartesian product of `string` and `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 =_c 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 ∧_b (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 =_c 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

```
*M 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 =l 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(n2) -> ff
                       | 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 term = Const of string * typ
              | Free  of string * typ
              | Var   of indexname * typ
              | Bound of int
              | Abs   of string * typ * term
              | op $  of term * term
```

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

- De Bruijn index i in any subterm $Bound(i)$ should be non-negative and less than the number of abstractions above this subterm in the term tree.
- For each occurrence of application operator $t_1\$t_2$, the type of term t_1 should have the form $T_1 \Rightarrow T_2$ where T_1 is the type of term 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 $Term(tp)$, where parameter tp ranges over NuPRL type Typ. Type $Term(tp)$ represents well-formed terms of type tp. It is defined as a disjoint union of several types, corresponding to different Isabelle term constructors.

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

```
*A const_term   ConstTerm(tp) ==  Atom × {x:Typ| x = tp}
```

Variable Terms Following the definition of type Typ, constructors `Free` and `Var` will be combined into one entity

```
*A var_term     VarTerm(tp) ==  VarName × {x:Typ| x = tp}
```

where

```
*A var_name     VarName ==  Atom + Indexname
```

Note that VarName is a decidable type. Boolean equality on this type can be defined through boolean equality on types Atom and Indexname

```
*A eq_var_name (v1 =v v2) ==
                case v1
                of inl(a1) => case v2
                              of inl(a2) => a1 =a a2
                               | inr(i2) => ff
                 | inr(i1) => case v2
                              of inl(a2) => ff
                               | inr(i2) => i1 =ixn i2
```

The following lemma is added to eq_rw_lemmas list.

```
*T eq_var_name_assert   ∀v,w:VarName. ↑(v =v w) ⟺ v = w
```

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

Formally, binding is a parameter of type $Term(tp)$, which, therefore, should be written as $Term(tp, n, bd)$. A bound variable term of type tp is an integer k such that $k < n$ and $bd(n - 1 - k) = tp$[3]

```
*A bound_term   BoundTerm(tp;n;bd) ==  {k:ℕn| bd (n - 1 - k) = tp}
```

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

```
*A abs_term    AbsTerm(tp;n;bd;tm) ==
                Atom × s:Typ × q:{q:Typ| tp = (s ⇒ q)}  ×
                      tm <q, n + 1, bd[n → s]>
```

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

[3] 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

```
*A op_term     OpTerm(tp;n;bd;tm) ==
                   s:Typ × tm <(s ⇒ tp), n, bd> × 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 would 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

```
*A proto_term  ProtoTerm(p) ==
                 parec(tm,p. let <tp,z> = p in let <n,bd> = z
                             in
                             ConstTerm(tp) + VarTerm(tp) +
                                BoundTerm(tp;n;bd) +
                                AbsTerm(tp;n;bd;tm) +
                                OpTerm(tp;n;bd;tm) @ p)

*A term         Term(t;n;bd) ==  ProtoTerm(<t, n, bd>)

*A const        Const(a;t)   ==  inl <a, t>

*A vari         Vari(a;t)    ==  inr (inl <a, t> )

*A bound        Bound(k)     ==  inr inr (inl k )

*A abs          λa:s. tm:q    ==  inr inr inr (inl <a, s, q, tm> )

*A op           (tm1 o tm2)  ==  inr inr inr inr <s, tm1, tm2>
```

Intermediate notion of *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 $Term(tp, n, bd)$. Instead, we normally prove an auxiliary theorem carrying induction over type $ProtoTerm(p)$ and derive from it the main result, stated in terms of type $Term(tp, n, bd)$.

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

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]
   | λa2: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,z8> = z7
                                              in
                                              let <tp3,z88> = 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

```
*M bind_ml
 bind(tm;vn;tp;n;bd)
==r case tm
 of Const(a,t) -> tm
  | Var(v,t) -> if (v =v vn) ∧b (t =tp tp) then Bound(n) else tm fi
  | Bound(j) -> tm
  | λa:s. m:q -> λa:s. bind(m;vn;tp;n + 1;bd[n → s]):q
  | (f o m | s) -> (bind(f;vn;tp;n;bd) o bind(m;vn;tp;n;bd))
```

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

```
*T bind_wf_tm  ∀t:Typ. ∀n:ℕ. ∀bd:ℕn → Typ. ∀tm:Term(t;n;bd).
               ∀vn:VarName. ∀tp:Typ. ∀k:ℕ. k = n + 1 ⇒
               bind(tm;vn;tp;n;bd) ∈ Term(t;k;[tp >> bd])
```

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

Substitution The operator

```
*M subs_ml
  tm[vn,tp→t]
 ==r
  case tm
   of Const(a,s) -> tm
    | Var(w,s) -> if (w =v vn) ∧b (s =tp tp) then t else tm fi
    | Bound(j) -> tm
    | λa:s. m:q -> λa:s. m[vn,tp→t]:q
    | (f o m | s) -> (f[vn,tp→t] o m[vn,tp→t])
```

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

```
*T subs_wf     ∀t:Typ. ∀n:ℕ. ∀bd:ℕn → Typ. ∀tm:Term(t;n;bd).
               ∀vn:VarName. ∀tp:Typ. ∀b:ℕ0 → Typ. ∀m:Term(tp).
                  tm[vn,tp→m] ∈ Term(t;n;bd)
```

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”:

```
*M free_ml
   Free(vn;tp;tm)
==r
   case tm
    of Const(a,tp') -> ff
     | Var(vn',tp') -> (vn' =v vn) ∧_b (tp' =tp tp)
     | Bound(j) -> ff
     | λa:s. tm':q -> Free(vn;tp;tm')
     | (f o m | s) -> Free(vn;tp;f) ∨_b Free(vn;tp;m)
```

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 $\rho(t)$ 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 → val] ==
<ev.1, λw,s.if (w =v vn) ∧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    BindVal(n;bd) ==  i:ℕn → ρ(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 => v:t] ==  bdv[n → 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 bdv_shift_zero  ∀T:𝕌. ∀t',t:T. ∀i:ℕ. ∀tp,bdv,n,bd:Top.
                       t = t' ⇒ i = 0 ⇒ [[t >>> bdv]](i) = t'

*T bdv_shift_pos  ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ.
                  ∀t,v:Top. ∀bdv:BindVal(n;bd). ∀i:{1..(n + 1)⁻}.
                   ∀j:ℕn. ∀tp:Typ. bd j = tp ⇒ j = i - 1
                     ⇒ [[v >>> bdv]](i) = [bdv](j)

*T bdv_shift_update ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀t,s:Typ.
                    ∀v:ρ(t). ∀w:ρ(s). ∀bdv:BindVal(n;bd).
                     [v >>> (bdv:bd)[n => w:s]] =
                     ([v >>> bdv]:[t >> bd])[n + 1 => w:s]

*T bdv_shift_nil_update ∀tev:TEval. ∀bd:ℕ0 → Typ. ∀t:Typ.
                         ∀bdv:BindVal(0;bd). ∀v:ρ(t).
                          [v >>> bdv] = (bdv:bd)[0 => 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 interp_ml   β(t|bdv,ev)
   ==r case t
        of Const(a,tp) -> ConstEval(a;tp|ev)
         | Var(i,tp) -> VarEval(i;tp|ev)
         | Bound(j) -> [bdv](n - 1 - j)
         | λa:s. tm:q -> λz.β(tm|(bdv:bd)[n => z:s],ev)
         | (f o tm | s) -> β(f|bdv,ev) β(tm|bdv,ev)
```

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

```
*T interp_wf ∀tev:TEval. ∀ev:Eval(tev). ∀tp:Typ. ∀n:ℕ.
              ∀bd:ℕn → Typ. ∀t:Term(tp;n;bd). ∀bdv:BindVal(n;bd).
                 β(t|bdv,ev) ∈ ρ(tp)
```

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

```
*T interp_bind_tm
   ∀tev:TEval. ∀ev:Eval(tev). ∀vn:VarName. ∀tp,t:Typ. ∀n:ℕ.
    ∀bd:ℕn → Typ. ∀tm:Term(t;n;bd). ∀w:ρ(tp). ∀bdv:BindVal(n;bd).
     ∀k:ℕ. k = n + 1 ⇒ β(bind(tm;vn;tp;n;bd)|[w >>> bdv],ev) =
                                  β(tm|bdv,ev[vn,tp → w])

*T interp_subs_tm
   ∀t:Typ. ∀n:ℕ. ∀bd:ℕn → Typ. ∀tm:Term(t;n;bd). ∀tev:TEval.
```

```
  ∀ev:Eval(tev). ∀bdv:BindVal(n;bd). ∀vn:VarName. ∀tp:Typ.
   ∀b:ℕ0 → Typ. ∀bv:BindVal(0;b). ∀m:Term(tp).
    β(tm[vn,tp→m]|bdv,ev) = β(tm|bdv,ev[vn,tp → β(m|bv,ev)])
```

```
*T interp_free_tm
   ∀vn:VarName. ∀tp:Typ. ∀tev:TEval. ∀ev:Eval(tev). ∀t:Typ. ∀n:ℕ.
    ∀bd:ℕn → Typ. ∀tm:Term(t;n;bd). ∀bdv:BindVal(n;bd). ∀w:ρ(tp).
     ↑¬bFree(vn;tp;tm) ⇒ β(tm|bdv,ev[vn,tp → w]) = β(tm|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:

```
*A prop_typ   O ==  Type("prop";[])
```

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

$$(pr_1(tev))(\text{"}prop\text{"}, []) = \mathbb{B}$$

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

```
*A prop_sign  PropSign ==  tev.1 "prop" [] = 𝔹
```

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

Implication Isabelle declares $\Longrightarrow$ as a constant of type $o \Rightarrow o \Rightarrow o$. Accordingly, in NuPRL we define Isabelle implication as

```
*A imp_const  ==> ==  Const("==>";(O ⇒ (O ⇒ O)))
```

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

```
*A imp_sign   ImpSign ==
     ConstEval("==>";(O ⇒ (O ⇒ O))|ev) = (λx,y.x ⇒b y)
```

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

```
*A imp         (p ==> q) ==  ((==> o p) o q)
```

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

```
*A all_const   ∩('a) ==  Const("all";(('a ⇒ O) ⇒ O))

*A all_sign    AllSign ==
                ∀'a:Typ. ConstEval("all";(('a ⇒ O) ⇒ O)|ev) =
                           (λb.∀_b x:ρ('a). b x)

*A iall         ∩(a:s. p) ==  (∩(s) o λa:s. p:O)
```

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 $\alpha \Rightarrow \alpha \Rightarrow o$. Just like the universal quantifier, this constant actually is a family of constants, parametrized by α:

```
*A eq_const    eq_const(‘a) ==  Const("==";(‘a ⇒ (‘a ⇒ O)))
*A eq_sign     EqSign ==
   ∀‘a:Typ. ConstEval("==";(‘a ⇒ (‘a ⇒ O))|ev) = (λx,y.(x =_b y))
*A eq           (x ≡ y) ==  ((eq_const(tp) o x) o y)
```

Operator *eq* hides parameter *tp* to make formulas more readable.

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

```
*T imp_interp
   ∀n:ℕ. ∀bd:ℕn → Typ. ∀p,q:Term(O;n;bd). ∀tev:TEval.
    ∀ev:Eval(tev). ∀bdv:BindVal(n;bd). PropSign ⇒ ImpSign ⇒
     β((p ==> q)|bdv,ev) = β(p|bdv,ev) ⇒_b β(q|bdv,ev)

*T all_interp
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀a:Atom. ∀s:Typ. ∀p:Term(O;n + 1;bd[n → s]). ∀ev:Eval(tev).
     PropSign ⇒ AllSign ⇒
      β(∩(a:s. p)|bdv,ev) = ∀_b x:ρ(s). β(p|(bdv:bd)[n => x:s],ev)

*T eq_interp
   ∀n:ℕ. ∀bd:ℕn → Typ. ∀tp:Typ. ∀x,y:Term(tp;n;bd). ∀tev:TEval.
    ∀ev:Eval(tev). ∀bdv:BindVal(n;bd). PropSign ⇒ EqSign ⇒
     β((x ≡ y)|bdv,ev) = (β(x|bdv,ev) =_b β(y|bdv,ev))
```

7.2 Meta Logic Rules

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

1. $\dfrac{\begin{matrix}[\phi]\\ \psi\end{matrix}}{\phi \Rightarrow \psi}$ 2. $\dfrac{\phi \Rightarrow \psi \quad \phi}{\psi}$

3. $\dfrac{\phi}{\bigwedge_\sigma x.\phi}$ 4. $\dfrac{\bigwedge_\sigma x.\phi}{\phi[b/x]}$

5. $\dfrac{}{a \equiv_\sigma a}$ 6. $\dfrac{a \equiv_\sigma b}{b \equiv_\sigma a}$ 7. $\dfrac{a \equiv_\sigma b \quad b \equiv_\sigma c}{a \equiv_\sigma c}$

8. $\dfrac{}{(\lambda x.a) \equiv_\sigma (\lambda y.a[y/x])}$ 9. $\dfrac{}{((\lambda x.a)b) \equiv_\sigma a[b/x]}$ 10. $\dfrac{f(x) \equiv_\sigma g(x)}{f \equiv_{\rho \to \sigma} g}$

11. $\dfrac{a \equiv_\sigma b}{(\lambda x.a) \equiv_{\rho \to \sigma} (\lambda x.b)}$ 12. $\dfrac{f \equiv_{\rho \to \sigma} g \quad a \equiv_\rho b}{f(a) \equiv_\sigma g(b)}$

13. $\dfrac{\begin{matrix}[\phi] & [\psi]\\ \psi & \phi\end{matrix}}{\phi \equiv_{prop} \psi}$ 14. $\dfrac{\phi \equiv_{prop} \psi \quad \phi}{\psi}$

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.

```
*T rule_1
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀p,q:Term(0;n;bd). ∀ev:Eval(tev). PropSign ⇒ ImpSign ⇒
     (↑β(p|bdv,ev) ⇒ ↑β(q|bdv,ev)) ⇒
      ↑β((p ==> q)|bdv,ev)
```

Proof Sketch Tactic `EqRW` reduces this theorem to a propositional tautology of the form $a \Rightarrow a$. □

```
*T rule_2
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀p,q:Term(0;n;bd). ∀ev:Eval(tev). PropSign ⇒ ImpSign ⇒
     ↑β((p ==> q)|bdv,ev) ⇒ ↑β(p|bdv,ev) ⇒ ↑β(q|bdv,ev)
```

Proof Sketch Similar to Rule 1. □

```
*T rule_3
   ∀tev:TEval. ∀bd:ℕ0 → Typ. ∀bdv:BindVal(0;bd). ∀p:Term(0).
    ∀a:Atom. ∀s:Typ. ∀vn:VarName. PropSign ⇒
     (∀ev:Eval(tev). AllSign ⇒ ↑β(p|bdv,ev)) ⇒
      (∀ev:Eval(tev). AllSign ⇒ ↑β(∩(a:s. bind(p;vn;s))|bdv,ev))
```

Proof Sketch After the reduction by `EqRW`, the statement of this theorem follows from lemma `interp_bind_tm`. □

```
*T rule_4
   ∀tev:TEval. ∀ev:Eval(tev). ∀bd:ℕ0 → Typ. ∀bdv:BindVal(0;bd).
    ∀p:Term(0). ∀a:Atom. ∀s:Typ. ∀vn:VarName. ∀m:Term(s).
     PropSign ⇒ AllSign ⇒ ↑β(∩(a:s. bind(p;vn;s))|bdv,ev)
      ⇒ ↑β(p[vn,s→m]|bdv,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
   ∀tev:TEval. ∀ev:Eval(tev). ∀n:ℕ. ∀bd:ℕn → Typ.
    ∀bdv:BindVal(n;bd). ∀tp:Typ. ∀a:Term(tp;n;bd). PropSign ⇒
     EqSign ⇒ ↑β((a ≡ a)|bdv,ev)
```

Proof Sketch Tactic `EqRW` reduces this theorem to a tautology of the form $x = x$. □

```
*T rule_6
   ∀tev:TEval. ∀ev:Eval(tev). ∀n:ℕ. ∀bd:ℕn → Typ.
    ∀bdv:BindVal(n;bd). ∀tp:Typ. ∀a,b:Term(tp;n;bd). PropSign ⇒
     EqSign ⇒ ↑β((a ≡ b)|bdv,ev) ⇒ ↑β((b ≡ a)|bdv,ev)
```

Proof Sketch Tactic `EqRW` reduces this theorem to a tautology of the form $x = y \Rightarrow y = x$. □

```
*T rule_7
   ∀tev:TEval. ∀ev:Eval(tev). ∀n:ℕ. ∀bd:ℕn → Typ.
    ∀bdv:BindVal(n;bd). ∀tp:Typ. ∀a,b,c:Term(tp;n;bd). PropSign ⇒
     EqSign ⇒ ↑β((a ≡ b)|bdv,ev) ⇒
      ↑β((b ≡ c)|bdv,ev) ⇒ ↑β((a ≡ c)|bdv,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
   ∀tev:TEval. ∀ev:Eval(tev). ∀n:ℕ. ∀bd:ℕn → Typ.
    ∀a,b:Atom. ∀s,q:Typ. ∀tm:Term(q;n + 1;bd[n → s]).
     ∀bdv:BindVal(n;bd). PropSign ⇒ EqSign ⇒
      ↑β((λa:s. tm:q ≡ λb:s. tm:q)|bdv,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
   ∀tev:TEval. ∀ev:Eval(tev). ∀bd:ℕ0 → Typ. ∀bdv:BindVal(0;bd).
    ∀a:Atom. ∀s,q:Typ. ∀vn:VarName. ∀t:Term(s). ∀m:Term(q).
     PropSign ⇒ EqSign ⇒
      ↑β(((λa:q. bind(t;vn;q):s o m) ≡ t[vn,q→m])|bdv,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
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀s,q:Typ. ∀f,g:Term((s ⇒ q);n;bd). ∀x:VarName. PropSign ⇒
     ↑¬bFree(x;s;f) ⇒ ↑¬bFree(x;s;g) ⇒
      (∀ev:Eval(tev). EqSign ⇒
          ↑β(((f o Vari(x;s)) ≡ (g o Vari(x;s)))|bdv,ev))
       ⇒ (∀ev:Eval(tev). EqSign ⇒ ↑β((f ≡ g)|bdv,ev))
```

Proof Sketch After the reduction by `EqRW`, the statement of this theorem follows from lemma `interp_free_tm`. □

```
*T rule_11
   ∀tev:TEval. ∀bd:ℕ0 → Typ. ∀bdv:BindVal(0;bd). ∀tp:Typ.
    ∀a:Atom. ∀s:Typ. ∀vn:VarName. ∀p,q:Term(tp). PropSign ⇒
     (∀ev:Eval(tev). EqSign ⇒ ↑β((p ≡ q)|bdv,ev)) ⇒
      (∀ev:Eval(tev) EqSign ⇒
      ↑β((λa:s. bind(p;vn;s):tp ≡ λa:s. bind(q;vn;s):tp)|bdv,ev))
```

Proof Sketch After the reduction by `EqRW`, the statement of this theorem follows from lemmas `interp_bind_tm`. □

```
*T rule_12
   ∀tev:TEval. ∀ev:Eval(tev). ∀tp:Typ. ∀n:ℕ.
    ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd). ∀s:Typ.
     ∀f,g:Term((s ⇒ tp);n;bd). ∀a,b:Term(s;n;bd). PropSign ⇒
      EqSign ⇒ ↑β((f ≡ g)|bdv,ev) ⇒ ↑β((a ≡ b)|bdv,ev) ⇒
       ↑β(((f o a) ≡ (g o b))|bdv,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
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀p,q:Term(0;n;bd). ∀ev:Eval(tev). PropSign ⇒ EqSign ⇒
     (↑β(p|bdv,ev) ⇒ ↑β(q|bdv,ev)) ⇒
      (↑β(q|bdv,ev) ⇒ ↑β(p|bdv,ev)) ⇒ ↑β((p ≡ q)|bdv,ev)
```

Proof Sketch The statement of this theorem, after the reduction by tactic `EqRW`, follows from the standard NuPRL lemma `iff_imp_equal_bool`. □

```
*T rule_14
   ∀tev:TEval. ∀n:ℕ. ∀bd:ℕn → Typ. ∀bdv:BindVal(n;bd).
    ∀p,q:Term(0;n;bd). ∀ev:Eval(tev). PropSign ⇒ EqSign ⇒
     ↑β((p ≡ q)|bdv,ev) ⇒ ↑β(p|bdv,ev) ⇒ ↑β(q|bdv,ev)
```

Proof Sketch Tactic `EqRW` reduces this theorem to a tautology of the form $(x = y) \Rightarrow x \Rightarrow y$. □

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

1. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.
2. R.L. Constable et al. *Implementing Mathematics with Nuprl Proof Development System.* Prentice Hall, 1986.
3. M.J.C. Gordon and T.F. Melham. *Introduction to HOL – A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
4. D.J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–282, Berlin, 1996. Springer-Verlag.
5. D.J Howe. Semantics foundation for embedding HOL in Nuprl. In M. Wirsing and A. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101, Berlin, 1996. Springer-Verlag.
6. N.P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–173, 1991.
7. P.F. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, 1988.
8. J. Meseguer and M.-O. Stehr. The HOL-NuPRL Connection from the Viewpoint of General Logic. Working paper, June 1999.
9. P. Naumov. *Formalizing Reference Types in NuPRL.* PhD thesis, Cornell University, August 1998.
10. P. Naumov. Importing Isabelle Formal Mathematics into NuPRL. Technical Report TR99-1734, Cornell University, Computer Science Department, Ithaca, NY, February 1999.
11. L.C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
12. L.C. Paulson. *Isabelle – A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.