An Executable Formalization of the HOL/Nuprl Connection in Twelf

Carsten Schürmann[1] Mark-Oliver Stehr[2]

[1] Yale University, carsten@cs.yale.edu
[2] University of Urbana-Champaign, stehr@uiuc.edu

Abstract. Howe's HOL/Nuprl connection is an interesting example of a translation between two fundamentally different logics, namely a typed higher-order logic and a polymorphic extensional type theory. In earlier work we have established a proof-theoretic correctness result of the translation in a way that complements Howe's semantics-based justification and furthermore goes beyond the original HOL/Nuprl connection by providing the foundation for a proof translator. Using the Twelf logical framework, the present paper goes one step further. It presents the first rigorous formalization of this treatment in a logical framework, and hence provides a safe alternative to the translation of proofs.

1 Introduction

Doug Howe's HOL/Nuprl connnection [6, 9] establishes a link between two very different logics, namely the classical logic of the HOL system [4] and a classical variant of the Nuprl type theory [3], so that formal developments in Nuprl can integrate theorems or entire libraries developed in HOL.

Based on a proof-theoretic understanding of the HOL/Nuprl connection obtained in earlier work [18], we present a rigorous formalization of the relevant parts of HOL and Nuprl as deductive systems, and a foundational transformation between the two in the logical framework Twelf [15]. Both encodings of the deductive systems are adequate, the transformation is executable and machine-verified, using Twelf's termination [16], coverage [17], and uniquness checker [2].

According to [18], the HOL/Nuprl connection as it was originally implemented in [10] proceeds in two stages:

1. The first stage is a *translation* of an *axiomatic HOL theory* into an *axiomatic Nuprl theory*. The use of the term "axiomatic" emphasizes the fact that the theories are not necessarily only definitional extensions of the base logic. The translation is, by its very nature, metalogical, in the sense that, by relating two different logics, it is beyond the scope of each of them. It is therefore a critical stage whose correctness cannot be reduced to that of the two theorem provers involved and requires careful analysis.
2. The second stage is the *interpretation* of an axiomatic Nuprl theory inside Nuprl. In this way we can often obtain a *computationally meaningful* theory, which is closer to the spirit of Nuprl, that favors definitional extensions. As

in Howe's extension of Nuprl, this interpretation stage can take place inside the Nuprl system in a formally rigorous way.

The key correctness property established in [18] is the soundness of the translation. If $\mathcal{L}$ and $\mathcal{L}'$ are the source and target logics, respectively, and α is the mapping of $\mathcal{L}$ into $\mathcal{L}'$ then *soundness* is the property that $\Gamma \vdash_{\mathcal{L}} P$ implies $\alpha(\Gamma) \vdash_{\mathcal{L}'} \alpha(P)$ for any set Γ of axioms and any formula P. Hence, a proof of soundness would demonstrate that for each proof of P from Γ in $\mathcal{L}$ there is a corresponding proof of $\alpha(P)$ from $\alpha(\Gamma)$ in $\mathcal{L}'$. Although soundness is a necessary requirement for the correctness of the translator, it is noteworthy that soundness can always be achieved by extending the target system by additional axioms and inference rules. Of course, such an extension could make the target system inconsistent, which is why the soundness proof is meaningful only in the presence of a consistency proof for the target system extension, which for the classical variant of Nuprl has been achieved by Howe's hybrid computational/set-theoretic semantics [7, 8].

It has been recognized in [18] that, since the soundness proof is conducted in a constructive way, it implicitly contains an algorithm for proof translation. In fact, a proof translator based on our earlier study has been developed by Pavel Naumov as an extension of the Nuprl system [12]. In spite of the high degree of safety achieved by translating proofs, we have found that, computationally, proof translation can be very expensive if the proofs are large and unstructured.

A viable alternative, which achieves a similar degree of assurance is formal verification. This paper can be seen as a first step in that direction by giving a fully formal account of the translation and its correctness proof in the logical framework of Twelf. Furthermore, the Twelf specification is executable, and hence constitutes a uniform certified translator that can translate theories as well as proofs if this is considered necessary.

In summary we present in this paper a lightweight formalization of [18] with a few simplifications: (1) We do not make explicit the categorical structure, e.g. the fact that the translation constitutes a natural transformation. (2) We use higher-order abstract syntax to represent terms in the object logics HOL and Nuprl. (3) We use signatures of the metalogical framework Twelf to uniformly represent signatures and theories of the object logics. (4) Without loss of generality we work with a simplified notion of sentences that are obtained from sequents by a universal closure. (5) We confine ourselves to a fixed HOL theory (namely the logical theory of HOL) and represent the composition of theory translation and the subsequent theory interpretation by a single function.

The paper is organized as follows. After a brief introduction to Twelf we give representations of the objects logics HOL and Nuprl in Sections 3 and 4, respectively. Then, in Section 5, we present the translation as formalized in Twelf, followed by Section 6, which establishes its correctness. Details about the formalization of Nuprl and selected Twelf proofs can be found in the appendix.

2 The Twelf Logical Framework

The Twelf logical framework is an implementation of LF [5] designed as a meta-language for the representation of deductive systems and used in this work for representation the relevant rules of HOL and Nuprl. Judgments are represented as types, and derivations as objects leading two the three standard syntactical categories of the Twelf system.

Kinds:	K	$::= \texttt{type} \mid \{\texttt{x}:A\}\ K \mid A\ \texttt{->}\ K$
Types:	A, B	$::= a \mid A\ M \mid \{\texttt{x}:A\}\ B \mid A\ \texttt{->}\ B$
Objects:	M	$::= c \mid \texttt{x} \mid \texttt{[x:}A\texttt{]}\ M \mid M_1\ M_2$

We write a for type level constants (also called type families), and c for object level constants. In Twelf, $\texttt{[x:}A\texttt{]}\ M$, $\{\texttt{x:}A\}\ K$ and $\{\texttt{x:}A\}\ B$ denote λ-terms, Π-kinds and Π-types, respectively. Type-level constants and object-level constants declarations form signatures in Twelf, and thus, the entire formal development of the HOL-Nuprl connection can be thought of as one signature that we explain piece by piece. Type constants are declared in form of declarations "$a\ \texttt{:}\ K\texttt{.}$", and object level constant are either declared "$c\ \texttt{:}\ A\texttt{.}$", or defined "$c\ \texttt{:}\ A\ \texttt{=}\ M\texttt{.}$" Constants c may be be used infix, below the declaration "$\texttt{\%infix}\ n\ m\ c\texttt{.}$" n defines the associativity of c, which can either be `left` or `right`, and m the binding tightness of c. We sometimes use "_", a variable that is instantiated by Twelf using unification.

Among the many algorithms that Twelf offers, we comment only on the most important that are directly relevant to the formalization of the HOL-Nuprl connection. The type inference algorithm [14] permits inferable arguments to remain implicitly Π-quantified indicated by logic variables that start with an uppercase letter. The logic programming engine Elf [14] as part of Twelf defines an operational interpration of the Twelf signature. For example, a type $\texttt{a}\ M_1\ \texttt{X}$ defines a query, whose execution results in an object M and an instantiation M_2 of `X`, such that $M\ \texttt{:}\ \texttt{a}\ M_1\ M_2$ holds. Twelf's mode system [16] assigns input/output roles to arguments of type familes. For example a declaration $\texttt{\%mode}\ a\ \texttt{+X -Y}$ indicates that the first argument to a plays the role of an input, and the second the role of an output. Furthermore, once modes are declared, Twelf will check that ground (or closed) inputs entail ground outputs. Twelf's termination declaration [16] $\texttt{\%terminates X (}a\ \texttt{X Y)}$ states that any call to a reduces the size of the first argument, and the corresponding termination checker verifies this property. Correspondingly, the coverage declaration [17] $\texttt{\%covers (}a\ \texttt{+X -Y)}$ establishes that upon invocation with any input object M_1 as first argument, the evaluation of $a\ M_1\ \texttt{X}$ will always make progress and does not get stuck. $\texttt{\%total X (}a\ \texttt{X Y)}$ verifies that a is a total function, i.e. for that every input M_1 for `X`, there exists an M and M_2, such that $M\ \texttt{:}\ a\ M_1\ M_2$.

3 The Logic of HOL

HOL [4] is a proof development system based on higher-order logic. It uses a Hindley-Milner-style polymorphic λ-calculus together with an axiomatization of

the logic using polymorphic equality, implication, and Hilbert's choice operator as basic ingredients. As most higher-order logics it is a logic of total functions. The HOL system favors conservative theory extensions (to introduce new constants and/or new data types) but axiomatic extensions are also supported. The following higher-order abstract syntax representation of HOL in Twelf is close to the informal presentation of [4].

3.1 Syntax

We introduce a Twelf type `tp` to represent the set of *HOL types* with `o` representing the type of *HOL formulas* and `-->` the *HOL function type* constructor.

```
tp  : type.
--> : tp -> tp -> tp.                    %infix right 10 -->.
o   : tp.
```

A dependent Twelf type `tm` σ is used to represent the set of *HOL terms* (including *HOL formulas*) over a given HOL type σ with `=>` representing *logical implication*, `==` representing *polymorphic equality*, `@` representing *polymorhic function application*, and `\` representing *polymorphic* λ*-abstraction*. Since `=>` and `==` represent HOL constants, we also introduce convenience functions `==>` and `===` that can be directly applied to HOL formulas/terms.

```
tm  : tp -> type.
=>  : tm (o --> o --> o).
==  : tm (A --> A --> o).
@   : tm (A --> B) -> tm A -> tm B.      %infix left 15 @.
\   : (tm A -> tm B) -> tm (A --> B).
==> = [A:tm o] [B:tm o] => @ A @ B.      %infix right 13 ==>.
=== = [A:tm T] [B:tm T] == @ A @ B.      %infix left 14 ===.
```

HOL theories, more precisely their signatures, provide a way to extend the syntax of HOL by additional constants. We define an *HOL type constant declaration* as a Twelf declaration of the form c : `tp -> tp -> ... -> tp`. An *HOL constant declaration* is a Twelf declaration of the form c : {α_1 : `tp`} ... {α_n : `tp`} `tm` σ, where σ is an HOL type over α_1 : `tp` ... α_n : `tp` and where each type variable α_i occurs in σ. An *HOL signature* Σ is a Twelf signature consisting of *HOL type constant declarations* and *HOL constant declarations*. The categoy of HOL signatures equipped with the standard notion of signature morphism (see [18]) will be denoted by **HolSign**.

An *HOL sentence* over Σ has the form {α_1 : `tp`} ... {α_n : `tp`}A with an HOL formula A over α_1 : `tp` ... α_n : `tp`. The set of sentences over a given HOL signature Σ is denoted by $HolSen(\Sigma)$. This notion of an HOL sentence is less general than that of an HOL sequent used in [4], but it is sufficient for our purposes, because each HOL sequent can be converted into an equivalent HOL sentence of the form above by means of a universal closure.

3.2 Deduction in HOL

In this section we inductively define the *HOL derivability predicate* that characterizes all derivable HOL sentences. Using the propositions-as-types interpretation (sometimes called judgements-as-types interpretation in this setting) this predicate is formalized in Twelf as follows.

```
|- : tm o -> type.
```

Each HOL deduction rule is then represented as a function in Twelf that operates on proofs of derivability. The function allows us to construct a proof of the conclusion if we provide a proof for each premise.

```
mp    : |- H -> |- H ==> G -> |- G.
disch : (|- H -> |- G) -> |- H ==> G.
refl  : |- H === H.
beta  : |- (\ H) @ G === (H G).
sub   : {G:tm A -> tm o} |- H1 === H2 -> |- G H1 -> |- G H2.
abs   : |- \ H === \ G <- ({x} |- H x === G x).
```

We do not need an explicit representation of HOL's assumption and type instantiation rules, because they are inherited from the logical framework. The latter is a special case of Twelf's substitution rule.

Given a signature Σ, the *HOL entailment relation* $(\vdash^{Hol}_{\Sigma}) \subseteq \mathcal{P}_{\text{fin}}(HolSen(\Sigma)) \times HolSen(\Sigma)$ is defined as follows: $\{\phi_1, ..., \phi_n\} \vdash^{Hol}_{\Sigma} \phi$ holds iff a proof of $\texttt{|-}\phi$ can be constructed from proofs of $\texttt{|-}\phi_1 \ldots \texttt{|-}\phi_n$ in Twelf. Using the terminology of [11], the structure $(\mathbf{HolSign}, HolSen, \vdash^{Hol})$ constitutes an entailment system. We call it the *entailment system of HOL*.

3.3 Theories

An *(axiomatic) HOL theory* (Σ, Γ) consists of a signature Σ together with a set Γ of sentences over Σ called *axioms*. A signature morphism $H : \Sigma \to \Sigma'$ is said to be a *theory morphism* $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ iff $\Gamma' \vdash^{Hol}_{\Sigma'} HolSen(H)(\phi)$ for all $\phi \in \Gamma$. This gives a category of theories that will be denoted by **HolTh**.

All mathematical developments in HOL take place in *standard theories* extending the *logical theory* `bool`. Therefore, for the remainder of this paper we define `bool` as `o`, and we use `bool` to emphasize that we are working with classical extensional logic. The logical theory `bool` has a signature Σ which contains the standard type constant `bool` (i.e. `o`) and the standard constants `==` and `=>`. The remaining constants of Σ together with their definitional axioms in Γ are:

```
true  : tm bool = (\ [x : tm bool] x) === (\ [x: tm bool] x).
false : tm bool = all (\ [P] P).
neg   : tm (bool --> bool) = \ [P] P ==> false.
/|\   : tm (bool --> bool --> bool)
      = \ [P] \ [Q] all (\ [R] (P ==> Q ==> R) ==> R).
/\    = [A] [B] /|\ @ A @ B.            %infix right 12 /\.
```

```
\|/   : tm (bool --> bool --> bool)
      = \ [P] \ [Q] all (\ [R] (P ==> R) ==> (Q ==> R) ==> R).
\/    = [A] [B] \|/ @ A @ B.            %infix right 11 \/.
all|  : tm ((A --> bool) --> bool)
      = \ [P:tm (A --> o)] P === \ [x] true.
all   = [A] all| @ A.
the|  : tm ((A --> o) --> A).
the   = [A] the| @ A.
ex|   : tm ((A --> bool) --> bool)
      = \ [P:tm (A --> o)] P @ (the (\ [x] P @ x)).
ex    = [A] ex| @ A.
```

Moreover, there are some nondefinitional axioms in Γ, namely

```
bool-cases-ax  : |- all (\ [x:tm bool] x === true \/ x === false).
imp-antisym-ax : |- all (\ [x:tm bool]
                           all (\ [y:tm bool]
                                  (x ==> y) ==> (y ==> x) ==> x === y)).
eta-ax         : |- (\ [x] F @ x) === F.
select-ax      : |- all (\ [P]  all (\ [x] P @ x ==>  P @ (the P))).
```

Some HOL contants and axioms have been omitted from our current formalization, because they are unnecessary for the core fragment of the translation. They, however, can be easily added.

Our encoding of HOL in Twelf is adequate. This means that every derivation of a sentence ϕ in HOL corresponds bijectively to an object in β-normal η-long of type `|-` ϕ, where ϕ stands for the representation of the sentence ϕ in Twelf. The reverse direction also holds. Furthermore, the encoding is compositional, in the sense, that the substitution property of HOL is captured by Twelf's built-in β-rule.

4 The Type Theory of Nuprl

Nuprl's type theory [3] is a variant of Martin-Löf's 1982 polymorphic, extensional type theory (the version contained in [13] with extensional equality). Although Nuprl has very advanced features (e.g. subset types, subtyping, quotient types, recursive types, intersection types, partial functions, and direct computation, which make these type theories rather different), semantically Nuprl can be viewed as an extension of Martin-Löf's type theory, in the sense that it has a richer variety of types and more flexible rules which give rise to a richer collection of well-typed terms.[1]

In constrast to HOL, terms in Nuprl are neither explicitly nor implicitly equipped with types. Instead types are ordinary terms, and the judgement that a type can be *assigned* to a term is a sentence in the logical language which is not decidable in general. Indeed, since Nuprl is polymorphic, a term may be associated with different types.

[1] For some subtle differences between Martin-Löf's type theory and Nuprl see [1].

Even though the advanced features of Nuprl provide an important motivation for the HOL/Nuprl connection, the connection itself does not rely on features that go beyond Martin-Löf's type theory as presented in [13]. Hence, we have selected a set of rules as the basis of our formalization that can be derived in both Martin-Löf's type as well as in Nuprl. We do not attempt to give a complete presentation of these type theories, but we rather show that the given rules are sufficient to establish the connection to HOL. In the following we give a simplified presentation of Nuprl based on [3]. As for HOL we use a Twelf representaton based on higher-order abstract syntax.

4.1 Syntax

We introduce a Twelf type `n-tm` to represent the set of *Nuprl terms* which as discussed above also includes all potential *Nuprl types*. The subsets of well-typed Nuprl terms and types are determined by the deduction rules of Nuprl given in the next subsection.

Nuprl has the following term or type constructors. We begin with the term `uni i` representing the predicative Nuprl universe at level `i`.

```
n-tm : type.
uni : integer -> n-tm.
```

We use `eq x y t` to represent Nuprl's typed equality, that is `x` and `y` are equal in type `t`. Membership, written `x # t`, is a derived notion in Nuprl and stands for `eq x x t`. The constant `axiom` is an element that denotes an anonymous proof in Nuprl, e.g. a proof by means of a computation or decision procedure.

```
eq  : n-tm -> n-tm -> n-tm -> n-tm.
#  = [n1][n2] eq n1 n1 n2.              %infix left 18 #.
axiom : n-tm.
```

In the following, `pi` represents the Nuprl dependent function type constructor with `->>` representing the special case of ordinary function types. The constants `app` and `lam` represent function application and the untyped λ-abstraction of Nuprl. For instance, we represent Nuprl's dependent function type $x : S \to T$ as $[\![x : S \to T]\!] = \texttt{pi}\ [\![S]\!]\ [x][\![T]\!]$, where $[\![M]\!]$ denotes the Twelf representation of M.

```
pi  : n-tm -> (n-tm -> n-tm) -> n-tm.
->> = [A:n-tm] [B:n-tm] pi A [x] B.     %infix right 20 ->>.
app : n-tm -> n-tm -> n-tm.
lam : (n-tm -> n-tm) -> n-tm.
```

Nuprl has strong existential types (also called strong Σ-types) represented by the function `sigma` with an element constructor `pair` and projections `fst` and `snd`.

```
sigma  : n-tm -> (n-tm -> n-tm) -> n-tm.
pair : n-tm -> n-tm -> n-tm.
fst  : n-tm -> n-tm.
snd  : n-tm -> n-tm.
```

The function `+` represents the disjoint sum type constructor. It comes with left and right injections `inl` and `inr`, and a function `decide` to perform case analysis.

```
+   : n-tm -> n-tm -> n-tm.              %infix right 19 +.
inl : n-tm -> n-tm.
inr : n-tm -> n-tm.
decide : n-tm -> (n-tm -> n-tm) -> (n-tm -> n-tm) -> n-tm.
```

Finally, we have Nuprl's singleton type `unit` with `bullet` as its only element, and the empty type `void` with a function `any` for the elimination principle.

```
unit : n-tm.   bullet : n-tm.
void : n-tm.   any : n-tm -> n-tm.
```

Finally, the HOL/Nuprl connection makes use of Nuprl's subset types, here represented by the type constructor `set`. A set $\{x : T \mid P\}$ in Nuprl is then represented as $[\![\{x : T \mid P\}]\!] = \texttt{set}\ [\![T]\!]\ [x][\![P]\!]$.

```
set : n-tm -> (n-tm -> n-tm) -> n-tm.
```

Similar to HOL, the Nuprl syntax can be extended by additional (untyped) constants. A *Nuprl signature* Σ is a Twelf signature consisting of *Nuprl constant declaration* of the form $c : \texttt{n} - \texttt{tm}$. The category of Nuprl signatures equipped with the standard notion of signature morphism (see [18]) will be denoted by **NuprlSign**. Given a signature Σ, we define *Nuprl sentences* simply as Nuprl terms over Σ (in practice these will be Nuprl types interpreted as propositions). Although this is more restrictive that the Nuprl sequents of [3], there is no loss of generality, because by universal closure each sequent can be converted to a Nuprl sentence of this form. The set of sentences over Σ is denoted by $\mathit{NuprlSen}(\Sigma)$.

It is worthwhile mentioning that there is another reason why the notion of Nuprl sentence is a proper specialization of the judgements admitted in [3], which (disregarding the left-hand side) take the form $\vdash\ T\,[\mathrm{ext}\,P]$, the pragmatic intention being that the extraction term P is usually hidden from the user, but it can be extracted from a completed proof. In this paper we are not interested in the extraction term P. Therefore we will only use *abstract judgements* of the form $\vdash\ T$. We define such an abstract judgement to be derivable iff $\vdash\ T\,[\mathrm{ext}\,P]$ is derivable for some P.

4.2 Deduction in Nuprl

In this section we inductively define the *Nuprl derivability predicate*. We consider derivability in the fragment of Classical Nuprl given by the inference rules below, which are either basic inference rules or trivially derivable in Nuprl. Similar to derivability in HOL we formalize the derivability predicate of Nuprl as follows:

```
!- : n-tm  -> type.                  %prefix 10 !-.
```

There is no need to formalize the basic Nuprl assumption, weakening, and cut rules, because they are inherited from Twelf. We begin with the representation of the rules for Nuprl's hierarchy of universes. There is a formation rule for each universe and a rule stating that the hierarchy is cummulative.

```
uni-form  : !- uni I # uni J <- J - 1 >= I.
uni-culm  : !- T # uni J <- J - 1 >= I <- !- T # uni I.
```

For Nuprl's equality we have a formation rule, and rules for symmetry, transitivity and substitution. Reflexivity is a trivial consequence of fact that membership `M # T` is defined as a special case of equality `eq`, namely as `eq M M T`.

```
equality-form  : !- (eq M N T) # (uni K)
  <- !- T # (uni K) <- !- M # T <- !- N # T.

equality-sym : !- eq M N T -> !- eq M M T.
equality-trans : !- eq M N T -> !- eq N K T -> !- eq M K T.

subst : !- eq M M' (P N) <- !- eq M M' (P N') <- !- eq N N' T
  <- ({x:n-tm} !- x # T -> !- P x # uni K).
```

The following rule `ax-intro` implies that `axiom` serves as an anonymous proof of every membership. The next rule `ax-elim` allows us to abstract from a proof.

```
ax-intro : !- axiom # (M # T) <- !- M # T.
ax-elim  : !- T <- !- _ # T.
```

Dependent function types are at the core of Nuprl's type theory. We follow the standard scheme to first give a formation rule, which introduces the type, and then introduction and elimination rules for the elements of this type, followed by equational/computation rules.[2]

```
fun-form : !- (pi S T) # (uni K)
  <- ({x:n-tm} !- x # S -> !- (T x) # (uni K)) <- !- S # (uni K).
fun-intro: !- (lam M) # (pi S T)
  <- ({x:n-tm} !- x # S -> !- (M x) # (T x)) <- !- S # (uni K).
fun-elim : !- (app M N) # (T N)  <- !- N # S  <- !- M # (pi S T).
fun-xi1   : !- eq (lam M) (lam N) (pi S T) <- !- S # (uni K)
  <- ({x:n-tm} !- x # S -> !- eq (M x) (N x) (T x)).
fun-beta : !- eq (app (lam M) N) (M N) (T N) <- !- N # S
  <- ({x:n-tm} !- x # S  -> !- (M x) # (T x)).
fun-ext  : !- eq M N (pi S T) <- !- M # (pi S T) <- !- N # (pi S T)
  <- ({x:n-tm} !- x # S -> !- eq (app M x) (app N x) (T x)).
```

For sake of brevity we have omitted the rules concerned with Σ-types and subset types, but the interested reader can find them in Appendix A.

Nuprl has a singleton type `unit` with one element `bullet`.

[2] Instead of Nuprl's untyped computation rules, we use the weaker typed computation rules to cover Martin-Löf's type theory as well.

```
unit-form : !- unit # (uni 1).
unit-intro: !- bullet # unit.
unit-eq  : !- eq M N unit <- !- M # unit <- !- N # unit.
```

Finally, we have the rules for the empty type `void` that does not have any introduction rules, but an elimination rule that allows us to prove anything from the existence of an element in `void`.

```
void-form : !- void # (uni 1).
void-elim : !- (any N) # T <- !- N # void <- !- T # uni K
```

The heavily used Nuprl type `boolean` is defined as a disjoint union in Nuprl:

```
boolean = unit + unit. tt = inl bullet. ff = inr bullet.
if = [x][l][r] decide x ([z] l) ([z] r).
```

The propositions-as-types interpretation is made explicit using the following logical abbreviations. We also introduce the abbreviation `nP k` for `uni k` to emphasize that we are interpreting types in this universe in a logical way.

```
nP = [k] uni k.
ntrue = unit. nfalse = void.
n/\ = [A] [B] sigma A [x] B.              %infix right 27 n/\.
n\/ = [A] [B] A + B.                      %infix right 26 n\/.
nall = [A] [B] pi A B.
nex  = [A] [B] sigma A B.
=n=>  = [A] [B] pi A [x] B.               %infix right 26 =n=>.
n~ = [A] A =n=> nfalse.                   %prefix 29 n~.
n<=> = [A] [B] (A =n=> B) n/\ (B =n=> A). %infix right 26 n<=>.
```

4.3 Classical Extension

The translation described in the next section makes use of Nuprl's operator

```
^ = [b:n-tm] if b ntrue nfalse.           %prefix 27 ^.
^-form : {b:n-tm} !- b # boolean -> !- if b ntrue nfalse # uni 1 .
```

which converts an element of `boolean` into a (propositional) type. The following properties have been proved using Twelf:

```
fact5 : !- _ # nall boolean [b] ^ b # nP 1
fact4 : !- _ # nall boolean [b] (eq b tt boolean) =n=> ^ b
fact6 : !- _ # nall boolean [b] (^ b) =n=> (eq b tt boolean)
```

For the translation of HOL's equality we wish to define a boolean polymorphic equality using Nuprl's propositional equality, but so far we do not have any means for converting a proposition into a boolean, which amounts to deciding whether a propositional type is inhabited. So we add a standard constant `inhabited` and we assume the following family of axioms stating that `inhabited t` decides if its argument, a type `t` in `uni k`, is inhabited, and that it returns an element of `t` if this is the case. Under the logical reading this assumption is known as the *axiom of the excluded middle*.

```
inhabited : n-tm.
inh-intro : !- inhabited # (pi (uni K) [x] x + (x ->> void)).
```

Equipped with this axiom we can easily define an operator `v` casting a propositional type into a boolean value deciding the proposition:

```
v = [P:n-tm] decide (app inhabited P) ([x] tt)  ([y] ff).
```

Recall that `decide` performs case analysis for elements of a disjoint union type. The following *casting lemmas* have been verified using Twelf:

```
v-form : !- N # uni K -> !- v N # boolean .
law4 : !- _ # nall (nP K) [P]  (v P) # boolean
law5 : !- _ # nall (nP K) [P]  (^ (v P)) =n=> P
law6 : !- _ # nall (nP K) [P]  P =n=> (^ (v P))
```

In complete analogy to the entailment relation of HOL we now define the *Nuprl entailment relation* $(\vdash_{\Sigma}^{Nuprl}) \subseteq \mathcal{P}_{\text{fin}}(\mathit{NuprlSen}(\Sigma)) \times \mathit{NuprlSen}(\Sigma)$ where $\{\phi_1, ..., \phi_n\} \vdash_{\Sigma}^{Nuprl} \phi$ holds iff a proof of `!-`ϕ can be constructed from proofs of `!-`$\phi_1 \ldots$ `!-`ϕ_n in Twelf. The structure $(\mathbf{NuprlSign}, \mathit{NuprlSen}, \vdash^{Nuprl})$ constitutes an entailment system. We call it the *entailment system of Nuprl*.

4.4 Theories

An *(axiomatic) Nuprl theory* (Σ, Γ) consists of a signature Σ together with a set Γ of sentences over Σ called *axioms*. A signature morphism $H : \Sigma \to \Sigma'$ is said to be a *theory morphism* $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ iff $\Gamma' \vdash_{\Sigma'}^{Nuprl} \mathit{NuprlSen}(H)(\phi)$ for all $\phi \in \Gamma$. This gives a category of theories that will be denoted by **NuprlTh**.

Our encoding of Nuprl and its classical extension in Twelf are adequate. As in the case of HOL, this means that every derivation of a sentence ϕ in Nuprl corresponds bijectively to an object in β-normal η-long of type `!-` ϕ, where ϕ stands for the representation of the sentence ϕ in Twelf. The reverse direction also holds. Again, the encoding is compositional, in the sense, that the substitution property of Nurpl is captured by Twelf's built-in β-rule.

5 Theory Translation

In [12] the translation from HOL theories to Nuprl theories is given by a functor Φ : **HolSign** $\to$ **NuprlTh** which translates HOL signatures into Nuprl theories together with a natural transformation $\alpha : \mathit{HolSen} \to \mathit{NuprlSen} \circ \Phi$ which translates HOL sentences into Nuprl sentences.

Since signatures of the object logics do not have a formal status in our Twelf formalization beyond being represented as Twelf signatures, we cannot express a function like Φ in our current formalization. Instead, we will show in Section 5.1 how to translate a concrete signature using the logical theory of HOL as an example. In the following, we focus on the formalization of the core translation function α, which has three main components. In the logic-programming-style

of Twelf functions are represented as predicates, and uniqueness and totality are established independently.

The first component is the translation of HOL types into Nuprl types. Notice that the above HOL type `o` of propositions is translated classically as the Nuprl data type `boolean`.

```
transtp    : tp -> n-tm -> type.          %mode transtp +A -N.
transtp--> : transtp (A --> B) (pi T [x] S) <- transtp B S <- transtp A T.
transtpo   : transtp o boolean.
```

The second component of the translation function α is the translation of HOL terms into Nuprl terms:

```
transtm : tm A -> n-tm -> type.           %mode transtm +A -N.
trans=> : transtm => =p=>.
trans== : transtm (== : tm (A --> A --> o)) (=p= N) <- transtp A N.
trans@  : transtm ((H :tm (B --> A)) @ (G:tm B)) (app T S)
           <- transtm G S <- transtm H T.
trans\  : transtm (\ H : tm (A --> B)) (lam M)
           <- ({x:tm _}{y:n-tm} transtm x y -> transtm (H x) (M y))
           <- transtp A _.
```

where we have employed the abbreviations

```
=p=> = lam [x] lam [y] if x y tt.
=b=> = [x][y] app (app =p=> x) y.          %infix right 26 =b=>.
=p= = [T:n-tm] lam [x] lam [y] v (eq x y T).
=b= = [T][x][y] app (app (=p= T) x) y.
```

The final component of α is the translation of HOL sentences into Nuprl sentences. Here, a Nuprl term is obtained from the translation of an HOL formula, and hence we only need to cast it into propositional type to obtain a meaningful Nuprl sentence.

```
transsen : tm o -> n-tm -> type.          %mode transsen +A -N.
t-base : transsen A (^ N) <- transtm A N.
```

5.1 Interpreting the Logical Theory

The logical HOL theory `bool` is a theory like every other theory, and does not need any special consideration. It is however important to notice that `bool` is not a pure definitional theory in HOL. We follow [10] where the proof obligations have been verified inside Nuprl using the interpretation given below, but first we recall the general concept of a theory interpretation.

Given Nuprl theories (Σ, Γ) and (Σ', Γ'), we say that (Σ, Γ) is *interpreted* in (Σ', Γ') by I iff $I : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ is a theory morphism in the category **NuprlTh**. Notice that these morphisms are not necessarily axiom-preserving, since it is typically the point of such an interpretation to get rid of axioms. Instead, we have to verify $\Gamma' \vdash^{Nuprl}_{\Sigma'} \mathit{NuprlSen}(I)(\phi)$ for all $\phi \in \Gamma$, the sentences

NuprlSen$(I)(\phi)$ are called *proof obligations*. As explained in the introduction of this paper, the activity of setting up a theory morphism and verifying the proof obligations characterizes the second stage of the HOL/Nuprl connection which requires user interaction in general.

In [18] each HOL constant is translated into a Nuprl constant with the same name. If the HOL constant is polymorphic the resulting Nuprl constant is a function representing a family of constants indexed by types. In the interpretation stage this Nuprl constant is then interpreted. In most cases the interpretation is the same constant but with an associated definitional axiom which equates the constant to a Nuprl term.

To accomplish this in Twelf for the concrete logical HOL theory `bool` we simply extend `transtm` by the composition of: (1) the translation of HOL constants into Nuprl contants and (2) the interpretation of Nuprl constants by their associated Nuprl terms. As a result, `transtm` represents the composition of the translation and the interpretation function in our formalization.

```
tc-true : transtm true tt.  tc-false: transtm false ff.
tc-neg  : transtm neg (lam [x] if x ff tt).
tc-/|\  : transtm /|\ (lam [x] lam [y] if x y ff).
tc-\|/  : transtm \|/ (lam [x] lam [y] if x tt y).
tc-all| : transtm (all| : tm ((A --> o) --> o))
                  (lam [P] v (pi T [x] ^ (app P x))) <- transtp A T.
tc-ex|  : transtm (ex| : tm ((A --> o) --> o))
                  (lam [P] v (sigma T [x] ^ (app P x))) <- transtp A T.
```

As abbreviations we introduce `inh T` to express that a type `T` is nonempty and `arb T`, which picks an arbitrary element inhabiting a nonempty type `T`.

```
inh = [T:n-tm] nex T [y] ntrue.
arb = [T:n-tm] decide (app inhabited T) ([x] x) ([x] bullet).
arb-intro : {t:n-tm} {u : !- t # (S # uni 1) n/\ inh S} !- arb S # S
```

Hilbert's choice operator `the| P`, where `P` is a boolean predicate on some type `A`, picks an element of the subset of `A` specified by `P` if this subset is nonempty, or yields an arbitrary element of `A` otherwise.

```
tc-the| : transtm (the| : tm ((A --> o) --> A))
  (lam [p] decide (app inhabited
     (set T ([x] ^ app p x))) ([x] x) ([x] arb T)) <- transtp A T.
```

Using this interpretation, all the proof obligations, i.e. the translated axioms of the HOL theory `bool`, can be derived in Classial Nuprl. The fact that HOL types are nonempty is critical to verify the proof obligation corresponding to the declaration of Hilbert's ϵ-operator `the`.

We have omitted in our Twelf formalization the straightforward interpretation of the HOL data type `ind` by the Nuprl data type of natural numbers. Since we only need to prove the translation of the HOL infinity axiom, any other infinite type would do.

6 Correctness of the Translation

The key property of a map of entailment systems [11] is soundness, i.e. the preservation of entailment. In our lightweight formalization in Twelf this boils down to `lemma5`, which is given at the end of this section. Its proof closely follows the informal proof given in [18], but instead of using the Nuprl system to prove some intermediate lemmas, all parts of the proof have been uniformly conducted in Twelf. We begin with a number of simple Nuprl lemmas:

```
disch_lemma : !- _ # nall boolean [p] nall boolean [q]
                     (^ p =n=> ^ q) =n=> ^ (p =b=> q)
mp_lemma : !- _ # nall boolean [p] nall boolean [q]
                  (^ (p =b=> q)) =n=> (^ p =n=> ^ q)
beta_lemma:  !- _ # nall (uni 1) [T] nall T [a] nall T [b]
                    eq a b T =n=>  ^ =b= T a b
beta_inverse : !- _ # nall (uni 1) [T] nall T [a] nall T [b]
                      ^ =b= T a b =n=> eq a b T
```

To prove soundness as expressed by `lemma5`, it remains to show that the translation of each HOL rule can be derived in Nuprl. Most of the translated inference rules have surprisingly short proofs (see Appendix B) in Nuprl if we use the lemmas above together with the following well-formedness lemmas for translated HOL types and HOL terms, which have been proved in Twelf by induction over HOL types and HOL terms, respectively.

```
lemma1: {A:tp} transtp A T -> type.
%mode lemma1 +TTM -TTP.
lemma2:  {H:tm A} {M:n-tm} transtm H M -> type.
%mode lemma2 +TM -TTM -TTP.
lemma3: transtp A T -> !- _ # ((T # (uni 1)) n/\ (inh T)) -> type.
%mode lemma3 +TT -NP.
lemma4: transtm (H:tm A) N -> transtp A T -> !- N # T -> type.
%mode lemma4 +TM -TT -NP.

lemma5 : |- H -> transsen H T -> !- M # T -> type.
%mode lemma5 +HOL -TTS -NUPRL.
```

Informally, `lemma5` states that the translation `T` of each derivable HOL sentence `H` is inhabited in Nuprl.

7 Final Remarks

We have presented a lightweight formalization of earlier work [18] complementing Howe's semantics-based justification of the HOL/Nuprl connection with a proof-theoretic counterpart. Our correctness result does not only provide a formal proof-theoretic justification for *translating theories*, but it simulataneously provides a formalization of *proof translation* that was beyond the scope of the original HOL/Nuprl connection. A noteworthy point is that the translation does

not rely on the more advanced features of Nuprl that go beyond Martin-Löf's extensional polymorphic type theory as presented in [13]. Therefore, the translation can also be regarded as a translation between HOL and a classical variant of Martin-Löf's type theory. This paper makes use of a few simplifications, but on the other hand it goes beyond [18] in the sense that is precisely spells out the rules that are suficient to establish the logical connection. Furthermore, the entire development including some verifications that were delegated to Nuprl in [18] has been uniformly verified in Twelf. In addition to the translation, there is the logical theory interpretation stage, which seems less critical, because the associated proof obligations have been verified by Howe inside the Nuprl system. Still we plan to extend our formalization to include a detailed verification of the interpretation stage in Twelf.

The feasibility of proof translation has been demonstrated by the proof translator presented in [12], but a remaining practical problem is that proof translation can be computationally very expensive, especially in view of the large size of HOL proofs generated by some HOL tactics. The approach taken in this paper is a rigorously formal certification of the translator by formalizing not only the translation function but also the deductive system of the logics involved and the soundness proof in a metalogical framework like Twelf. So instead of verifying the correctness of each single translated HOL proof in Nuprl, so to say at runtime, we have formalized our general soundness result, which enhances our confidence in the correctness of our earlier informal mathematical treatment, and hence can be regarded as a resonable safe alternative to proof translation.

One the other hand, if the high assurance of proof translation is needed, the Twelf specification can serve as a certified proof translator. However, the practical feasibility of translating actual HOL proofs in this way has not been investigated yet and is left as a possible direction for future work. Other items for future work include the explicit representation of theories as objects in Twelf as well as a more modular development that separates the two stages of theory translation and theory interpretation.

The complete formal development of the HOL/Nuprl connection in Twelf can be found at `www.logosphere.org`.

References

1. S. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* PhD thesis, Cornell University, September 1987.
2. Penny Anderson and Frank Pfenning. Verifying uniqueness in a logical framework. In *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'04)*, Park City, Utah, September 2004. Springer Verlag.
3. R. L. Constable, S. Allen, H. Bromely, W. Cleveland, et al. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, 1986.
4. M. J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.
5. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

6. 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, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–282. Springer Verlag, 1996.
7. D. J. Howe. Semantical foundations for embedding HOL in Nuprl. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101, Berlin, 1996. Springer-Verlag.
8. D. J. Howe. A classical set-theoretic model of polymorphic extensional type theory. Manuscript (submitted for publication), 1997.
9. D. J. Howe. Toward sharing libraries of mathematics between theorem provers. In *Frontiers of Combining Systems, FroCoS'98, ILLC, University of Amsterdam, October 2–4, 1998, Proceedings*. Kluwer Academic Publishers, 1998.
10. D. J. Howe. Source Code of the HOL-Nuprl Translator (including Extensions to Nuprl), January 1999.
11. J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Logic Colloquium'87, Granada, Spain, July 1987, Proceedings*, pages 275–329. North-Holland, 1989.
12. P. Naumov, M.-O. Stehr, and J. Meseguer. The HOL/NuPRL proof translator — A practical approach to formal interoperability. In *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs'2001, Edinburgh, Scotland, UK, September 3–6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 329 – 345. Springer-Verlag, 2001.
13. K. Petersson, J. Smith, and B. Nordstroem. *Programming in Martin-Löf's Type Theory. An Introduction*. International Series of Monographs on Computer Science. Oxford: Clarendon Press, 1990.
14. Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.
15. Frank Pfenning and Carsten Schürmann. System description: Twelf — a metalogical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
16. Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.
17. Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In David Basin and Burkhart Wolff, editors, *Proccedings of Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume LNCS-2758, Rome, Italy, 2003. Springer Verlag.
18. M.-O. Stehr, P. Naumov, and J. Meseguer. A proof-theoretic approach to HOL-Nuprl connection with applications to proof translation (extended abstract). In *WADT/CoFI'01, 15th International Workshop on Algebraic Development Techniques and General Workshop of the CoFI WG, Genova, Italy, April 1-3, 2001*, 2001. Full version available at http://formal.cs.uiuc.edu/stehr/biblio_stehr.html.

A Deduction in Nuprl

For sake of brevity we have omitted several rules from our representation of Nuprl. To given a complete pricture of the fragment of Nuprl that is sufficent to conduct all proofs in this paper we present all remaining rules in this appendix.

We begin with the missing rules for strong existential types:

```
sig-form : !- (sigma S T) # (uni K)
           <- ({x:n-tm} !- x # S -> !- (T x) # (uni K))
           <- !- S # (uni K).

sig-intro: !- (pair M N) # (sigma S T)
             <- !- N # (T M)
             <- !- M # S.

sig-fst : !- (fst M) # S
           <- !- M # (sigma S T).

sig-snd : !- (snd M) # (T (fst M))
           <- !- M # (sigma S T).

sig-xi   : !- eq (pair M N) (pair M' N') (sigma S T)
           <- !- eq M M' S
           <- !- eq N N' (T M).

sig-redfst : !- eq (fst (pair M N)) M S
               <- !- M # S
               <- ({x:n-tm} !- x # S  -> !- N # (T x)).

sig-redsnd : !- eq (snd (pair M N)) N (T M)
               <- !- M # S
               <- ({x:n-tm} !- x # S  -> !- N # (T x)).
```

Then there are the rules for disjoint union types:

```
sum-form : !- (S + T) # (uni K)
             <- !- S # (uni K)
             <- !- T # (uni K).

sum-inl : !- (inl M) # (S + T)
            <- !- T # (uni K)
            <- !- M # S.

sum-inr : !- (inr M) # (S + T)
            <- !- M # T
            <- !- S # (uni K).

sum-decide :  !- (decide M N1 N2) # (T M)
            <- ({x:n-tm} !- x # S2 -> !- (N2 x) # T (inr x))
```

```
            <- ({x:n-tm} !- x # S1 -> !- (N1 x) # T (inl x))
            <- !- M # S1 + S2
            <- ({x:n-tm} !- x # (S1 + S2) -> !- T x # uni K).

sum-ell : !- eq (decide (inl N) N1 N2) M (T (inl N))
            <- !- eq (N1 N) M (T (inl N))
            <- ({x:n-tm} !- x # S2 -> !- (N2 x) # T (inr x))
            <- ({x:n-tm} !- x # S1 -> !- (N1 x) # T (inl x))
            <- ({x:n-tm} !- x # (S1 + S2) -> !- T x # uni K)
            <- !- S2 # uni K <- !- N # S1.

sum-elr : !- eq (decide (inr N) N1 N2) M (T (inr N))
            <- !- eq (N2 N) M (T (inr N))
            <- ({x:n-tm} !- x # S2 -> !- (N2 x) # T (inr x))
            <- ({x:n-tm} !- x # S1 -> !- (N1 x) # T (inl x))
            <- ({x:n-tm} !- x # (S1 + S2) -> !- T x # uni K)
            <- !- N # S2 <- !- S1 # uni K.
```

Finally, we use the following rules for subset types:

```
set-form : !- (set T P) # (uni K)
           <- ({x:n-tm} !- x # T -> !- P x # uni K)
           <- !- T # (uni K).

set-intro : !- M # (set T P)
            <- !- M # T
            <- !- P M
            <- ({x:n-tm} !- x # T -> !- P x # uni K).

set-elem : !- M # T
           <- !- M # (set T P)
           <- ({x:n-tm} !- x # T -> !- P x # uni K).

set-prop : !- (P M)
           <- !- M # (set T P)
           <- ({x:n-tm} !- x # T -> !- P x # uni K).

set-equality: !- eq (set T P) (set T' P') (uni K)
              <- !- eq T T' (uni K)
              <- ({x:n-tm} !- x # T -> !- P x -> !- P' x)
              <- ({x:n-tm} !- x # T -> !- P' x -> !- P x).
```

Note that in contrast to Σ-types, subset types do not carry an element of the second argument type, which in the case of subset types would be a proof of the characteristic property.

B Soundness Proof

The soundness result formulated in `lemma5` requires us to prove that the translation of each HOL rule can be derived in Nuprl. The following proofs correspond to the HOL rules `mp`, `disch`, `refl`, `beta`, and `abs`, respectively.

```
case-5-1 : lemma5 (mp (D1:|- H) (D2 : |- H ==> G)) (t-base TTM2')
    (nall-elim (nall-elim (nall-elim (nall-elim mp_lemma ND4) ND3) ND2) ND1)
    <- lemma2 H _ TTM
    <- lemma2 (H ==> G) _ (trans@ (trans@ trans=> TTM2) TTM2')
    <- lemma4 TTM transtpo ND4
    <- lemma4 TTM2' transtpo ND3
    <- lemma5 D1 (t-base TTM) ND1
    <- lemma5 D2 (t-base (trans@ (trans@ trans=> TTM2) TTM2')) ND2.

case-5-2 : lemma5 (disch (D : |- H -> |- G))
                  (t-base (trans@ (trans@ trans=> TTM1) TTM2))
    (=n=>-elim (nall-elim (nall-elim disch_lemma ND1) ND2)
       (=n=>-intro (boolean-if (uni-form (+>= 1 0>=0)) ND1
                                  (ntrue-form)  nfalse-form) ND))
    <- lemma2 H _ (TTM1 : transtm H T1)
    <- lemma2 G _ (TTM2 : transtm G T2)
    <- lemma4 TTM1 transtpo ND1
    <- lemma4 TTM2 transtpo ND2
    <- ({u:|- H}{y:n-tm}{v:!- y # ^ T1}
          lemma5 u (t-base TTM1) v
          -> lemma5 (D u) (t-base TTM2) (ND y v)).

case-5-3 : lemma5 (refl : |- H === H)
                  (t-base (trans@ (trans@ (trans== TTP) TTM) TTM))
       (nall-elim (nall-elim refl_lemma boolean-form) ND)
    <- lemma2 H _ TTM
    <- lemma4 TTM TTP ND.

case-5-4 : lemma5 (beta : |- (\ H) @ G === (H G))
       (t-base (trans@ (trans@ (trans== TTP1)
                          (trans@ (trans\ TTP2 TTM1) TTM2)) (TTM1 _ _ TTM2)))
       (nall-elim
          (nall-elim
             (nall-elim
                 (=n=>-elim beta_lemma (ax-elim (n/\-fst ND1)))
                (fun-elim (fun-intro (ax-elim (n/\-fst ND2)) ND3) ND4))
             (ND3 _ ND4)) (ax-intro (fun-beta ND3 ND4)))
    <- ({x:tm _}{y:n-tm} {ttm:transtm x y}
          lemma2 x _ ttm
          -> lemma2 (H x) _ (TTM1 x y ttm))
    <- lemma2 G _ TTM2
    <- lemma4 TTM2 TTP2 ND4
    <- ({x:tm _} {y:n-tm} {ttm:transtm x y}
          lemma2 x y ttm ->
```

```
      {u: !- y # _} lemma4 ttm TTP2 u
      -> lemma4 (TTM1 x y ttm) TTP1 (ND3 y u))
  <- lemma3 TTP1 ND1
  <- lemma3 TTP2 ND2.

case-5-5 : lemma5 (abs D1 : |- \ ([x:tm A] H x) === \ G)
  (t-base (trans@ (trans@ (trans== (transtp--> TTP1 TTP2))
                  (trans\ TTP1 TTM1)) (trans\ TTP1 TTM2)))
  (=n=>-elim (nall-elim
               (nall-elim (nall-elim beta_lemma
                            (fun-form (ax-elim (n/\-fst ND1))
                               ([_][_] ax-elim (n/\-fst ND2))))
                 (fun-intro (ax-elim (n/\-fst ND1)) ND3))
              (fun-intro (ax-elim (n/\-fst ND1)) ND4))
     (ax-intro (fun-xi1
                 ([x][u] ax-elim
                    (=n=>-elim
                       (nall-elim
                          (nall-elim (nall-elim
                                       beta_inverse
                                       (ax-elim (n/\-fst ND2)))
                             (ND3 x u))
                          (ND4 x u))
                       (ND5 x u)))
                 (ax-elim (n/\-fst ND1)))))
  <- lemma1 A TTP1
  <- ({x:tm _}{y:n-tm} {ttm:transtm x y}
      lemma2 x _ ttm
      -> lemma2 (H x) _ (TTM1 x y ttm))
  <- ({x:tm _}{y:n-tm} {ttm:transtm x y}
      lemma2 x _ ttm
      -> lemma2 (G x) _ (TTM2 x y ttm))
  <- ({x:tm _} {y:n-tm} {ttm:transtm x y}
      lemma2 x y ttm ->
      {u: !- y # _} lemma4 ttm TTP1 u
      -> lemma4 (TTM1 x y ttm) TTP2 (ND3 y u))
  <- ({x:tm _} {y:n-tm} {ttm:transtm x y}
      lemma2 x _ ttm -> {u: !- y # _}
      lemma4 ttm TTP1 u
      -> lemma4 (TTM2 x y ttm) TTP2 (ND4 y u))
  <- ({x:tm _} {y:n-tm} {ttm:transtm x y}
      lemma2 x _ ttm
      -> {u: !- y # _} lemma4 ttm TTP1 u
      -> lemma5 (D1 x)
         (t-base (trans@ (trans@ (trans== TTP2)
                                 (TTM1 x y ttm))
                         (TTM2 x y ttm))) (ND5 y u))
  <- lemma3 TTP1 ND1
  <- lemma3 TTP2 ND2.
```

```
case-5-6 : lemma5 (sub ([x:tm A] G x)
                  (D1 : |- H1 === H2)
                  (D2 : |- G H1))
     (t-base (TTM3 _ _ TTM2))
     (subst' ([x][u] u)
        (fun-beta ([x] [u] ^-form (M x) (ND6 x u)) ND4)
        (=n=>-elim
           (=n=>-elim
              (=n=>-elim
                 (nall-elim
                     (nall-elim
                        (nall-elim
                           subst_lemma
                          (ax-elim (n/\-fst ND2)))
                       ND3)
                    ND4)
                 (fun-intro (ax-elim (n/\-fst ND2))
                       [x] [u] ^-form (M x) (ND6 x u)))
              ND1)
           (subst ([x] [u] u)
               (fun-beta ([x] [u] ^-form (M x) (ND6 x u)) ND3)
               ND5))
        )
    <- ({x:tm _}{y:n-tm} {ttm:transtm x y}
           lemma2 x y ttm
           -> lemma2 (G x) (M y) (TTM3 x y ttm))
    <- lemma5 D1 (t-base (trans@ (trans@ (trans== TTP1) TTM1) TTM2)) ND1
    <- lemma3 TTP1 ND2
    <- lemma4 TTM1 TTP1 ND3
    <- lemma4 TTM2 TTP1 ND4
    <- lemma5 D2 (t-base (TTM3 _ _ TTM1)) ND5
    <- ({x:tm A} {y:n-tm} {ttm:transtm x y}
            lemma2 x y ttm
           -> {u: !- y # _} lemma4 ttm TTP1 u
           -> lemma4 (TTM3 x y ttm) transtpo (ND6 y u)).


%block b0 : some {T:tp}
    block {x:tm T} {y:n-tm} {u:transtm x y}  {l2:lemma2 x y u}.

%block b1 : some {T:tp}{N:n-tm}{TP:transtp T N}
    block {x:tm T} {y:n-tm} {u:transtm x y}  {l2:lemma2 x y u}
          {v : !- y # N} {l4:lemma4 u TP v}.

%block b2 : some {A:tm o} {T1:n-tm} {TTMA : transtm A T1}
    block {u:|- A}{n:n-tm}{h:!- n # ^ T1} {k:lemma5 u (t-base TTMA) h}.

% Lemma 1
%worlds (b0 | b1 | b2) (lemma1 _ _).
```

```
%unique lemma1 +A -1TTP.
%terminates TT (lemma1 TT _).
%covers (lemma1 +TT -NP).
%total TT (lemma1 TT _).

% Lemma 2
%worlds ( b0 | b1 | b2) (lemma2 _ _ _).
%terminates TT (lemma2 TT _ _).
%covers (lemma2 +TT -K -NP).
%total TT (lemma2 TT K NP).

% Lemma 3
%worlds (b1 | b2) (lemma3 _ _).
%unique lemma3 +A -1D.
%terminates TT (lemma3 TT _).
%covers (lemma3 +TT -NP).
%total TT (lemma3 TT NP).

% Lemma 4
%worlds (b1 | b2) (lemma4 _ _ _).
%terminates TT (lemma4 TT _ _).
%covers (lemma4 +TT -TT' -NP).
% total TT (lemma4 TT _ NP).

% Lemma 5
%worlds (b1 | b2) (lemma5 _ _ _).
%terminates TT (lemma5 TT _ _).
%covers (lemma5 +TT -TT' -NP).
% total TT (lemma5 TT _ _).
```

Although termination, input coverage (all cases of input arguments are covered) and uniqueness (where applicable) have been machine checked, the output coverage property (all cases of output arguments of the recursive calls are covered) could not be mechanically verified for Lemma 4 and 5 due to an incompleteness in the implementation of the output coverage checker. The proof of the output coverage relies on the fact that the translation yields unique results, a fact inaccessable to the current implementation. An appropriate extension of the implementation of the coverage checker is work in progress.