Towards a Formally Verified Proof Assistant

Abhishek Anand and Vincent Rahli

Cornell University

Abstract. This paper presents a formalization of Nuprl's metatheory in Coq. It includes a nominal-style definition of the Nuprl language, its reduction rules, a coinductive computational equivalence, and a Curry-style type system where a type is defined as a *Partial Equivalence Relation* (PER) à la Allen. This type system includes Martin-Löf dependent types, a hierarchy of universes, inductive types and partial types. We then prove that the typehood rules of Nuprl are valid w.r.t. this PER semantics and hence reduce Nuprl's consistency to Coq's consistency.

1 Introduction

Trustworthiness of Proof Assistants. In order to trust a proof checked by a proof assistant, we have to trust many aspects of both its theory and implementation. Typically, the core of a proof assistant consists of a proof checking machinery which ensures that proof terms are indeed proofs of the corresponding statements. In constructive type theories such as the ones implemented in Agda [10], Coq [9], and Nuprl [4], this is accomplished with typechecking rules, which are derived from a semantic model, e.g., a computational model based on an applied λ-calculus. Parts of these theories have been formally described in various documents [37,7,3,15,24,36].

This is not a completely satisfactory state of affairs because: (1) It is possible to overlook inconsistencies between the different parts formalized on paper; and (2) Mistakes are possible in these large proofs (often spanning hundreds of pages) which are never carried out in full details. For example, we at least once added an inconsistent rule to Nuprl even after extensive discussions regarding its validity. A Bug that lead to a proof of False was found in Agda's typechecker[1]. Recently, the Propositional Extensionality axiom was found to be inconsistent with Coq[2]. However, consistency of Propositional Extensionality is a straightforward consequence of Werner's proof-irrelevant semantics [37] of the Prop universe. Werner allows only structurally recursive definitions while Coq's termination analyzer seems to be more permissive. A similar bug was discovered in Agda[3].

Fortunately, proof assistants have matured enough [26,19] that we can consider formalizing proof assistants in themselves [8], or in other proof assistants. By Tarski's undefinability theorem, these rich logics cannot formalize their own

[1] https://lists.chalmers.se/pipermail/agda/2012/003700.html

[2] See https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html for more details. We do not use this axiom in our development.

[3] https://lists.chalmers.se/pipermail/agda/2014/006252.html

semantics. However, because Martin-Löf's type theories are stratified into cumulative universes, it is plausible that a universe can be modeled in a higher universe of the same theory. Even better, universes of two equally powerful theories can be interleaved so that universes up to a level i of one of the theories could be modeled using at most universes up to level $i + 1$ of the other theory, and vice-versa. The latter approach seems likely to catch more mistakes if the two theories and their implementations are not too closely correlated.

Also, some type theories are proof-theoretically stronger than others. For example, Agda supports inductive-recursive definitions and can likely prove Nuprl and the predicative fragment of Coq consistent. Among other things, this paper also illustrates how one can define Nuprl's entire hierarchy of universes in Agda.

Advantages of a Mechanized Metatheory. A mechanized metatheory can guide and accelerate innovation. Currently, developers of proof assistants are reluctant to make minor improvements to their proof assistants. Many questions have to be answered. Is the change going to make the theory inconsistent? Is it going to break existing developments? If so, is there a way to automatically transform developments of the old theory to the new one while preserving the semantics? A mechanized metatheory can be used to confidently answer these questions. Moreover, we would no longer have to sacrifice efficiency for simplicity.

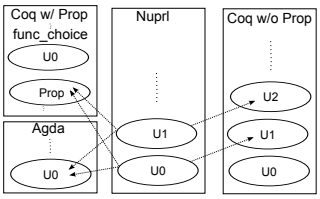
Mechanized Formalization of Nuprl. Therefore, this paper tackles the task of formalizing Nuprl in Coq. We chose Coq over other constructive proof assistants [10] because of its powerful tactic mechanism, and over other non-constructive proof assistants [32] because of the convenience of extracting programs from proofs for free. The two theories differ in several ways. While Coq (like Agda) is based on an intensional type theory with intrinsic typing, Nuprl is based on an extensional one with extrinsic typing. Also, over the years Nuprl has been extended with several expressive types not found in Coq, e.g., quotient types [4]; refinement types [4]; intersection and union types [24]; partial types [15]; and recursive types [28]. Partial types enable principled reasoning about partial functions (functions that may not converge on all inputs). This is better than the approach of Agda[4] and Idris [11] where one can disable termination checking and allow oneself to accidentally prove a contradiction.

Following Allen [3] we implement W types (which can be used to encode inductive types [2] in an extensional type theory like Nuprl) instead of Mendler's recursive types, therefore making the entire Nuprl metatheory predicatively justifiable as evidenced by our illustrative Agda model in Sec. 5. Mendler's recursive types can construct the least fixpoint of an arbitrary monotone endofunction (not necessarily strictly positive) in a Nuprl universe. Formalizing this notion seems to require an impredicative metatheory. Our current system includes a hierarchy of universes, function types, partial types, and we also extend Allen's work to include parametrized W types (similar to parametrized inductive types [33]).

Our formalization of Nuprl's metatheory in Coq proceeds as follows in three steps: (1) We define an inductive type of Nuprl terms. This definition is parametrized by a collection of operators and makes it possible to add new

[4] http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.AgdaVsCoq

constructs without changing the core definitions. We then define substitution and α-equality and prove several of their properties. (2) We define Nuprl's lazy computation system and a coinductive computational approximation relation that was defined and proved to be a congruence by Howe [21]. We formalize this proof as well as the domain theoretic properties that were used by Crary to justify some typehood rules about partial types [15]. We then define a computational equivalence relation [21] which plays a key role in the definition of our type system. (3) Following Allen's approach, we define types as PERs. This definition determines which closed terms denote equal types and which closed terms are equal members of types. Finally, we define Nuprl's sequents and prove the validity of many inference rules. We also show that using induction-recursion in Agda results in a more intuitive and simple definition of Nuprl's type system.



We describe below the key details of each of these steps. More details can be found in our technical report or in our publicly available code [5]. A key aspect of this formalization is that it gives us a *verified trusted core* of Nuprl. Although one can use Nuprl's tactics to prove typehood properties[5], these tactics can only use the above mentioned rules in the end. Moreover, this work tackles the task of formally describing in a unified framework all the extensions and changes that were made to Nuprl's type theory since CTT86 [14] and since Allen's PER semantics [3]. The core of this work is a formalization of [21,3,15,24]. Unlike previous works, we pin down three precise metatheories that can model (parts of) Nuprl. This is best illustrated by the figure above. An arrow from a Nuprl universe A to some universe B means that the PERs of types in A can be defined as relations in the universe B. As expected of large mechanized formalizations like ours, we found at least one minor mistake in one of these extensions. With the help of the original author, we were able to fix this proof in our formalization.

2 Uniform Term Model and Computation System

We use a nominal approach (bound variables have names) to define Nuprl terms. This definition closely matches the way terms are represented in Nuprl's current implementation. It is also very close to definitions used in paper descriptions of Nuprl [21]. Many alternative approaches have been discussed in the literature. See [6] for a survey. However, our choice avoided the overhead of translating the paper definitions about Nuprl to some other style of variable bindings.

We often show colored code[6]: blue is used for inductive types, dark red for constructors of inductive types, green for defined constants, functions and lemmas, red for keywords and some notations, and purple for variables.

Fig. 1 defines NTerm, the type of Nuprl terms. Variable bindings are made explicit by the simultaneously defined BTerm type. bterm takes a list of variables

[5] Nuprl has never had a typechecker. It relies on customizable tactics to do typechecking.

[6] Some of the colored items are hyperlinked to the place they are defined, either in this document, in the standard library, or in our publicly available code.

```
Inductive NVar : Set :=
| nvar : nat → NVar.

Inductive Opid : Set :=
| Can : CanonicalOp → Opid
| NCan : NonCanonicalOp → Opid.
```

```
Inductive NTerm : Set :=
| vterm: NVar → NTerm
| oterm: Opid → list BTerm → NTerm
with BTerm : Set :=
| bterm: (list NVar) → NTerm → BTerm.
```

Fig. 1 Uniform Term Model

lv and a term nt and constructs a bound term. Intuitively, a variable that is free in nt gets bound to its first occurrence in lv, if any. The rest of our term definition is parametrized by a collection of operators Opid. We divide our Opids into two groups, CanonicalOps and NonCanonicalOps (see [5, Sec. 2.1] for their definitions). This distinction is only relevant for defining the computation system and its properties. Intuitively, an operator constructs a term from a list BTerm. For example, oterm (Can NLambda) [bterm [nvar 0] (vterm (nvar 0))] represents a λ-term of the form $\lambda x.x$. Nuprl has a lazy computation system and any NTerm of the form (oterm (Can _) _) is already in canonical form and is called a value.

Not all the members of NTerm are well-formed: (nt_wf t) asserts that t is well-formed. For example, a well-formed λ-term must have exactly one bound term as an argument. Moreover, that bound term must have exactly one bound variable. Fig. 2 compactly describes the syntax of an illustrative subset of the language that we formalized. There, v ranges over values. There is a member of CanonicalOp for each clause of v. For example, the constructor Nint of type $\mathbb{Z} \to$ CanonicalOp corresponds to the first clause that denotes integers. In Fig. 2, vt ranges over the values that represent types. A term t is either a variable, a value, or a non-canonical term represented as (oterm (NCan _) _) in our term model. These have arguments (marked in boxes) that are said to be *principal*. As mentioned below, principal arguments are subterms that have to be evaluated to a canonical form before checking whether the term itself is a redex or not.

We next define our simultaneous substitution function (lsubst) and α-equality (alpha_eq and alpha_eq_bterm). As expected of a nominal approach to variable bindings, we spent several weeks proving their properties that were required to formalize Nuprl. One advantage is that these and many other definitions and proofs [5, Sec. 2.2] apply to any instantiation of Opid in which equality is decidable. This considerably simplifies the process of extending the language.

We formalize Nuprl's computation system by defining a one step computation function [5, Sec. 3.1] on NTerm. When evaluating a non-canonical term, it first checks whether one of the principal arguments is non-canonical. If so, it recursively evaluates it. The interesting cases are when all the principal arguments are canonical. Fig. 2 compactly describes these cases for an illustrative subset of the formalized system.

3 Computational Approximation and Equivalence

When we define the type system in Sec. 6, we want it to respect many computational relations. For example, most type systems satisfy the subject reduction property. In Agda, Coq and Nuprl, if t reduces to t', and t is in some type

$v ::= vt$	(type)	$\mid \texttt{inl}(t)$	(left injection)	$\mid \texttt{Ax}$	(axiom)
$\mid \underline{i}$	(integer)	$\mid \texttt{inr}(t)$	(right injection)	$\mid \langle t_1, t_2 \rangle$	(pair)
$\mid \lambda x.t$	(lambda)	$\mid \texttt{sup}(t_1, t_2)$	(supremum)		
$vt ::= \mathbb{Z}$	(integer type)	$\mid x{:}t_1 \to t_2$	(function type)		
$\mid x : t_1 \times t_2$	(product type)	$\mid t_2 = t \in t_1$	(equality type)		
$\mid \texttt{Base}$	(base type)	$\mid \overline{t}$	(partial type)		
$\mid \cup x : t_1.t_2$	(union type)	$\mid \cap x{:}t_1.t_2$	(intersection type)		
$\mid t_1//t_2$	(quotient type)	$\mid t_1 + t_2$	(disjoint union type)		
$\mid \{x : t_1 \mid t_2\}$	(set type)	$\mid \texttt{W}(x{:}t_1, t_2)$	(W type)		
$t ::= x$	(variable)	$\mid v$	(value)		
$\mid \boxed{t_1}\ t_2$	(application)	$\mid \texttt{fix}(\boxed{t})$	(fixpoint)		
$\mid \texttt{let}\ x := \boxed{t_1}\ \texttt{in}\ t_2$	(call-by-value)	$\mid \texttt{let}\ x, y = \boxed{t_1}\ \texttt{in}\ t_2$	(spread)		
$\mid \texttt{case}\ \boxed{t_1}\ \texttt{of inl}(x) \Rightarrow t_2 \mid \texttt{inr}(y) \Rightarrow t_3$	(decide)				
$\mid \texttt{if}\ \boxed{t_1} =_{\mathbb{Z}} \boxed{t_2}\ \texttt{then}\ t_3\ \texttt{else}\ t_4$	(integer equality)				

$(\lambda x.F)\ a$	$\to$	$F[x\backslash a]$	$\texttt{fix}(v)$	$\to$	$v\ \texttt{fix}(v)$
$\texttt{let}\ x, y = \langle t_1, t_2 \rangle\ \texttt{in}\ F$	$\to$	$F[x\backslash t_1; y\backslash t_2]$	$\texttt{let}\ x := v\ \texttt{in}\ t$	$\to$	$t[x\backslash v]$
$\texttt{case inl}(t)\ \texttt{of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G$	$\to$	$F[x\backslash t]$			
$\texttt{case inr}(t)\ \texttt{of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G$	$\to$	$G[y\backslash t]$			
$\texttt{if}\ \underline{i}_1 =_{\mathbb{Z}} \underline{i}_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2$	$\to$	t_1, if $\underline{i}_1 = \underline{i}_2$			
$\texttt{if}\ \underline{i}_1 =_{\mathbb{Z}} \underline{i}_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2$	$\to$	t_2, if $\underline{i}_1 \neq \underline{i}_2$			

Fig. 2 Syntax (top) and operational semantics (bottom) of Nuprl

```
Definition olift (R : NTerm → NTerm → [univ]) (x y :NTerm) : [univ] :=
  nt_wf x × nt_wf y ×
  ∀ sub: Substitution, wf_sub sub → programs [lsubst x sub, lsubst y sub]
                                   → R (lsubst x sub) (lsubst y sub).

Definition blift (R: NTerm → NTerm → [univ]) (bt1 bt2: BTerm): [univ] :=
  {lv: (list NVar) × {nt1 ,nt2 : NTerm × R nt1 nt2
    × alpha_eq_bterm bt1 (bterm lv nt1) × alpha_eq_bterm bt2 (bterm lv nt2) }}.

Definition lblift (R: NTerm → NTerm → [univ]) (l r: list BTerm): [univ] :=
  length l = length r × ∀ n : nat, n < length l → blift R (l[n]) (r[n]).
```

Fig. 3 Lifting operations. The notation {_:_× _} denotes sigma types (sigT)

T, then t and t' are equal in T. In addition, it is useful to have our types respect a *congruence* that contains the computation relation. For example, Coq has a notion of definitional equality which is a congruence. In Nuprl, we have a computation equivalence $\sim$ [21], which is a coinductively defined congruence that permits more powerful equational reasoning. For example, all diverging programs are equivalent under $\sim$. The same holds for all programs that generate an infinite stream of zeroes. Howe first defines a preorder approx on closed terms, proves that it is a congruence and finally defines $t1 \sim t2$ as approx $t1$ $t2$ $\times$ approx $t2$ $t1$. We first define olift, blift, lblift in Fig. 3. These will be used to lift a binary relation on closed NTerms to one on terms that are not necessarily closed, to BTerms, and to lists of BTerms, respectively. Note that programs l asserts that every member of l is both well-formed and closed; wf_sub sub as-

```
CoInductive approx : (tl tr :NTerm) : [univ] :=
| approx_fold: close_compute approx tl tr → approx tl tr.

CoInductive approx_aux (R : NTerm → NTerm → [univ]) (tl tr: NTerm): [univ] :=
| approx_fold: close_compute (approx_aux R \2/ R) tl tr → approx_aux R tl tr.

Definition approx := approx_aux (fun _ _ : NTerm ⇒ False).
```

Fig. 4 Computational approximation.

serts that the range of the substitution *sub* consists of well-formed terms; $t \Downarrow tv$ asserts that t converges to the value tv [5, Sec. 3.1]. $t \Downarrow$ asserts that t converges to some value. Although the notation `[univ]` currently stands for `Type`, most of our development works unchanged if we change it to `Prop`.

One can think of approx as the greatest fixpoint of the following operator on binary relations:

```
Definition close_compute (R: NTerm → NTerm → [univ]) (tl tr : NTerm): [univ]:=
programs [ tl, tr ] × ∀ (c : CanonicalOp) (tls : list BTerm),
  (tl ⇓ oterm (Can c) tls)
  → {trs : list BTerm × (tr ⇓ oterm (Can c) trs)× lblift (olift R) tls trs }.
```

One could now directly define approx as at the top of Fig. 4, where `\2/` denotes disjunction of binary relations. However, this approach would require using the `cofix` tactic of Coq for proving the properties of approx. Unfortunately, `cofix` does very conservative productivity checking [23] and often rejects our legitimate proofs. So we use parametrized coinduction [23] to define it in a slightly indirect way (the next two items in Fig. 4). With this technique, we only need to use `cofix` once to prove a "coinduction-principle" [5, Sec. 3.2] for approx and use that principle for the coinductive proofs about approx. Howe then proves that (olift approx) (abbreviated as approx_open) is a congruence [21]:

```
Theorem approx_open_congruence : ∀ (o : Opid) (lbt1 lbt2 : list BTerm),
  lblift approx_open lbt1 lbt2
  → nt_wf (oterm o lbt2) → approx_open (oterm o lbt1) (oterm o lbt2).
```

The proof is not easy. He first defines another relation approx_star, which contains approx_open and which is a congruence by definition. Then he proves that approx_star implies approx_open. This proof assumes that all Opids satisfy a condition called *extensionality*. We formalize his proof and prove that all the Opids of the current Nuprl system are extensional. Hence, we obtain that approx_open, approx and $\sim$ are congruences [5, Sec. 3.2.1].

Domain Theoretic Properties. The preorder approx has interesting domain theoretic properties [15] such as compactness and the least upper bound principle. Let $\bot$ be $\mathtt{fix}(\lambda x.x)$. It is the least element (up to $\sim$) w.r.t. approx, i.e., for any closed term t, approx $\bot$ t. The least upper bound principle says that for any terms G and f, $G(\mathtt{fix}(f))$ is the least upper bound of the (approx) chain $G(f^n(\bot))$ for $n \in \mathbb{N}$. Compactness says that if $G(\mathtt{fix}(f))$ converges, then there exists a natural number n such that $G(f^n(\bot))$ converges. We formalized proofs of both these properties [5, Sec. 3.3]. Crary used compactness to justify his fixpoint induction principle. It provides an important way for proving that a term of the form $\mathtt{fix}(f)$ is in a partial type [5, Sec. 5.2]. We have used the least upper

T `type`	iff	$T{\equiv}T$	$T_1{\equiv}T_2$ iff $T_1 \Downarrow T_1' \wedge T_2 \Downarrow T_2' \wedge T_1'{\equiv}T_2'$
$t{\in}T$	iff	$t{\equiv}t{\in}T$	$t_1{\equiv}t_2{\in}T$ iff $t_1 \Downarrow t_1' \wedge t_2 \Downarrow t_2' \wedge T \Downarrow T' \wedge t_1'{\equiv}t_2'{\in}T'$

$t_1{\equiv}t_2{\in}$`Base`	iff	$t_1 \sim t_2$
`Ax`$\equiv$`Ax`$\in(a = b \in A)$	iff	$(a = b \in A)$ `type` $\wedge\ a{\equiv}b{\in}A$
$t_1{\equiv}t_2{\in}\overline{A}$	iff	$(\overline{A})$ `type` $\wedge\ (t_1\Downarrow \iff t_2\Downarrow) \wedge (t_1\Downarrow \Rightarrow\ t_1{\equiv}t_2{\in}A)$
$f_1{\equiv}f_2{\in}x{:}A \to B$	iff	$(x{:}A \to B)$ `type` $\wedge\ \forall a_1, a_2.\ a_1{\equiv}a_2{\in}A \Rightarrow f_1(a_1){\equiv}f_2(a_2){\in}B[x\backslash a_1]$
$\mathtt{sup}(a_1, f_1){\equiv}\mathtt{sup}(a_2, f_2){\in}\mathtt{W}(x{:}A, B)$	iff	$(\mathtt{W}(x{:}A, B))$ `type` $\wedge\ a_1{\equiv}a_2{\in}A$ $\wedge\ \forall b_1, b_2.\ b_1{\equiv}b_2{\in}B[x\backslash a_1]\ \Rightarrow\ f_1(b_1){\equiv}f_2(b_2){\in}\mathtt{W}(A{:}x, B)$

`Base`$\equiv$`Base`		
$(a_1 = a_2 \in A){\equiv}(b_1 = b_2 \in B)$	iff	$A{\equiv}B \wedge (a_1{\equiv}b_1{\in}A \vee a_1 \sim b_1) \wedge (a_2{\equiv}b_2{\in}A \vee a_2 \sim b_2)$
$\overline{A}{\equiv}\overline{B}$	iff	$A{\equiv}B \wedge (\forall a.\ a{\in}A \Rightarrow a\Downarrow)$
$x_1{:}A_1 \to B_1{\equiv}x_2{:}A_2 \to B_2$	iff	$A_1{\equiv}A_2 \wedge \forall a_1, a_2.\ a_1{\equiv}a_2{\in}A_1 \Rightarrow B_1[x_1\backslash a_1]{\equiv}B_2[x_2\backslash a_2]$
$\mathtt{W}(x_1{:}A_1, B_1){\equiv}\mathtt{W}(x_2{:}A_2, B_2)$	iff	$A_1{\equiv}A_2 \wedge \forall a_1, a_2.\ a_1{\equiv}a_2{\in}A_1 \Rightarrow B_1[x_1\backslash a_1]{\equiv}B_2[x_2\backslash a_2]$

Fig. 5 Informal definition of a core part of a Nuprl universe

bound principle to justify some untyped computational equivalences that are useful for automatic optimization of Nuprl extracts [34].

4 PER Semantics

We now define Nuprl's type system. Several semantics have been proposed for Nuprl over the years, such as: Allen's PER semantics [3,15]; Mendler's adaptation of Allen's PER semantics [28]; and Howe's set-theoretic semantics [22]. In this paper, we use Allen's PER semantics because it can be defined predicatively. Also, the various additions made to Nuprl over the years have been validated using this semantics. Allen's method determines which closed terms are types and which closed terms are equal members of types, therefore, defining types as PERs. A PER is symmetric and transitive, but not necessarily reflexive. Partiality is required because the domain of these relations is the entire collection of closed terms, and not just the members of the type. We say that t is a member of type T when the PER definition of T relates t and t.

Fig. 5 informally presents a core part of Nuprl's type system à la Crary, which can be made formal using induction-recursion (where equality in W types has to be defined inductively). We write $T_1{\equiv}T_2$ to mean that T_1 and T_2 are equal types, and $a{\equiv}b{\in}T$ to mean that a and b are equal in type T. Allen designed his semantics to work with systems that have an extensional type equality (meaning that two types T and S are equal if for all t and s, $t{\equiv}s{\in}T$ iff $t{\equiv}s{\in}S$) and suggests a way to deal with type systems that have a more intensional type equality, such as Nuprl. For example, two true equality types such as $0 = 0 \in \mathbb{N}$ and $1 = 1 \in \mathbb{N}$ are not equal types in Nuprl even though they have the same PER. Also, note that type equality in Nuprl is coarser than computational equivalence ($\sim$). For example $(\lambda x.((x + 1) - 1) = \lambda x.x \in \mathbb{N} \to \mathbb{N})$ and $(\lambda x.x = \lambda x.x \in \mathbb{N} \to \mathbb{N})$

are equal equality types, but the two terms are not computationally equivalent because $\lambda x.((x+1)-1)$ gets stuck when applied to a term that is not a number while $\lambda x.x$ does not. Crary [15] provides a formal account of the adaptation of Allen's semantics to deal with systems that have non-fully extensional type equality (he adds a few type constructors and leaves off the W types). Following Crary, our type system defines which terms denote equal types instead of simply defining which terms denote types as in Allen's semantics.

As mentioned by Allen, simple induction mechanisms such as the one of Coq are not enough to provide a straightforward definition of Nuprl's semantics, where one would define typehood and member equality by mutual induction [3, Sec. 4.2]. The problem is that in the inductive clause that defines the dependent function types, the equality predicate occurs at a non-strictly-positive position. Allen suggests that the definition should however be valid because it is "half-positive". This is what induction-recursion, as implemented in Agda, achieves [17,18]. Instead of making that induction-recursion formal, the approach taken by Allen was to define ternary relations between types and equalities.

This trick of translating a mutually inductive-recursive definition to a single inductive definition has been formally studied by Capretta [13]. He observes that this translation is problematic when the return type of the function mentions a predicative universe. Indeed, we experienced the same problem while formalizing Allen's definition in a precise metatheory like Coq. In particular, we can only predicatively formalize a finite number of universes of Nuprl in Coq. This is not surprising given the results of Setzer that intensional and extensional versions of various dependent type theories have the same proof theoretic strength [35].

We will first explain how induction-recursion can be used to define the type system in an intuitive way. Although the inductive-recursive definition is easier to understand and lets us predicatively prove Nuprl's consistency, Agda lacks a tactic language, which is critical to automate many otherwise tedious proofs. Therefore, we chose Coq over Agda and used Allen's trick to define Nuprl's type system. At first, this purely inductive definition in Sec. 6 might seem overly complicated. However, it can be understood as applying Capretta's general recipe [13] to the inductive-recursive definition.

5 An Inductive-Recursive Approach

Crary first presents an intuitive definition of Nuprl's type system in the style of Fig. 5 and asserts that it is not a valid inductive definition. Then he uses Allen's trick to convert it to a purely inductive definition [15, page 51]. Using induction-recursion [18], this section shows that the definitions in Fig. 5 are meaningful. Moreover, we show how to define the entire predicative hierarchy of universes of Nuprl in the first universe of Agda's predicative hierarchy. This is not surprising, given that induction-recursion is known to increase the proof theoretic strength of type theories [18]. As mentioned above, because Agda does not have a tactic machinery necessary for our proof automation, this definition is only for illustrative purposes. We define universes having only integers and dependent function types ([5, Sec. 4.1] has more types, e.g., W types).

```
mutual
  data equalType (n : ℕ) (iUnivs : Vec PER n)
    (T1 T2 : NTerm) : Set where

  PINT : { _ : T1 ⇓ mk_Int}
    { _ : T2 ⇓ mk_Int}
    → (equalType n iUnivs T1 T2)
  PFUN : {A1 B1 A2 B2 : NTerm} {v1 v2 : NVar}
    { _ : T1 ⇓ (mk_Fun A1 v1 B1)}
    { _ : T2 ⇓ (mk_Fun A2 v2 B2)}
    (pA : equalType n iUnivs A1 A2)
    (pB : (a1 a2 : NTerm)
      (pa : equalInType iUnivs pA a1 a2)
        → equalType n iUnivs
          (subst B1 v1 a1)
          (subst B2 v2 a2))
    →   (equalType n iUnivs T1 T2)
  PUNIV : (m :  Fin n)
    { _ : T1 ⇓ (mk_Univ (toℕ m))}
    { _ : T2 ⇓ (mk_Univ (toℕ m))}
      → (equalType n iUnivs T1 T2)

equalInType : {T1 T2 : NTerm} {n : ℕ}
      (iUnivs : Vec PER n)
      (teq : equalType n iUnivs T1 T2)
        → PER

equalInType iUnivs PINT    t t' =
  ∑ ℤ (λ n
  → (t ⇓ (mk_int n))
    × t' ⇓ (mk_int n))

equalInType iUnivs (PFUN pA pB) t t' =
  (a1 a2 : NTerm)
  (pa : equalInType iUnivs pA a1 a2)
    → equalInType   iUnivs
                    (pB a1 a2 pa)
                    (mk_apply t a1)
                    (mk_apply t' a2)

equalInType iUnivs (PUNIV m) T1 T2 =
  lookup m iUnivs T1 T2
```

Fig. 6 Agda Inductive-Recursive definition

In Fig. 6, where , where PER is NTerm→NTerm→Set, equalType inductively defines which types are equal and equalInType recursively defines which terms are equal in a type[7]. These definitions refer to each other and are simultaneously defined using the mutual keyword. Both definitions are parametrized by a number n and *iUnivs*, a vector of PERs of size n. The idea is that we have already defined the first n universes by now and the PER defined by (equalType n *iUnivs*) will serve as the equality of types in the next universe. The m^{th} member (where $m < n$) of *iUnivs* is the already constructed PER that determines which two terms denote equal types in the m^{th} universe. Given an evidence that *T1* and *T2* are equal types in this universe, equalInType returns the PER of this type. Note that equalInType is structurally recursive on its argument *teq*. Note also that equalInType occurs at a negative position in the PFUN clause, but this is allowed since equalInType is defined by structural recursion and not induction.

6 An Inductive Approach Based on Allen's Semantics

6.1 Metatheory. Now, we return to Coq, where induction-recursion is not yet implemented. As mentioned above, all the Coq definitions presented so far would typecheck either in `Prop` or `Type`. This is not true about the definition of the type system. As mentioned above, we define two metatheories of Nuprl in Coq. One uses its predicative `Type` hierarchy. This metatheory uses $n + 1$ Coq universes to model n Nuprl universes. Because universe polymorphism is still under development in Coq, this currently requires that we duplicate the code at each level, which is impractical. Hence, we have only verified this translation for $n = 3$, and we will not discuss that metatheory further [5, Sec. 4.3].

[7] For brevity, we ignore the issue of closedness of terms here.

The other metatheory uses Coq's Prop impredicative universe with the FunctionalChoice_on axiom[8] ([5, Sec. 4.2.3] explains why this axiom is needed). In this metatheory, we can model all of Nuprl's universes. Also, it allows us to justify some principles that a classical mathematician might wish to have. For example, in the Prop model, using the law of excluded middle for members of Prop (known to be consistent with Coq[8]), following Crary [15] we have proved [5, Sec. 5.2] that the following weak version of the law of excluded middle is consistent with Nuprl: $\forall P : \mathbb{U}_i.\ \downarrow(P + (P \to \texttt{Void}))$. Because the computational content of the disjoint union is erased (using the squashing operator $\downarrow$), one cannot use this to construct a magical decider of all propositions.

6.2 Type Definitions. This section illustrates our method by defining base, equality, partial, function, and W types. As mentioned above, types are defined as PERs on closed terms. A CTerm is a pair of an NTerm and a proof that it is closed. Let per stand for CTerm → CTerm → Prop. A *type system* is defined below as a *candidate type system* that satisfies some properties such as symmetry and transitivity of the PERs, where a candidate type system is an element of the type cts, which we define as CTerm → CTerm → per→ Prop. Given a cts *c*, *c T1 T2 eq* asserts that *T1* and *T2* are equal types in the type system *c* and *eq* is the PER that determines which terms are equal in these types.

We now define the PER constructors (of the form per_TyCon) for each type constructor TyCon of Nuprl's type theory. Intuitively, each per_TyCon is a monotonic operator that takes a candidate type system *ts* and returns a new candidate type system where all the types compute to terms of the form (TyCon *T1* ... *Tn*) where *T1*, ..., *Tn* are types of *ts*.

In the definitions below we use {_:_,_} for propositional existential types (i.e., ex from the standard library). Also, we use Nuprl term constructors of the form mkc_TyCon. We omit their definitions as they should be obvious. Finally, for readability we sometimes mix Coq and informal mathematical notations.

Base. The values of type Base (suggested by Howe [21]) are closed terms and its equality is computational equivalence. In the following definition, *ts* is not used because Base does not have any arguments:

```
Definition per_base (ts : cts) T1 T2 (eq : per) : Prop :=
  T1 ⇓ Base × T2 ⇓ Base × ∀ t t', eq t t' ⇔ t ∼ t'.
```

Equality. Unlike Coq, Nuprl has primitive equality types which reflect the metatheoretical PERs as propositions that users can reason about. Note that Uniqueness of Identity Proofs, aka UIP, holds for Nuprl, i.e., Ax is the unique canonical inhabitant of equality types.

```
Definition per_eq (ts : cts) T1 T2 (eq : per) : Prop :=
  {A, B, a1, a2, b1, b2 : CTerm , {eqa : per
      , T1 ⇓ (mkc_equality a1 a2 A) × T2 ⇓ (mkc_equality b1 b2 B)
      × ts A B eqa × (eqa \2/ ∼) a1 b1 × (eqa \2/ ∼) a2 b2
      × (∀ t t', eq t t' ⇔ (t ⇓ Ax × t' ⇓ Ax × eqa a1 a2)) }}.
```

[8] http://coq.inria.fr/cocorico/CoqAndAxioms

This definition differs from the one present in earlier Nuprl versions, where `(eqa \2/ ∼)` was simply *eqa*. This means that $t \in T$ is now a type when T is a type and t is in `Base`. In earlier versions of Nuprl [15] (as well as in other type theories [36]) membership was a non-negatable proposition, i.e., $t \in T$ was not a proposition unless it was true. This change allows us to reason in the theory about a wider range of properties that could previously only be stated in the metatheory. For example, we can now define the *subtype* type $A \sqsubseteq B$ as $\lambda x.x \in A \to B$ because $\lambda x.x$ is in `Base`. Sec. 6.3 shows how we had to change the definition of a type system in order to cope with this modification.

Partial Type. Given a type T that has only converging terms, we form the partial type $\overline{T}$ (see Fig. 5). Equal members of $\overline{T}$ have the same convergence behaviour, and if either one converges, they are equal in T.

```
Definition per_partial (ts : cts) T1 T2 (eq : per) : Prop :=
  {A1 , A2 : CTerm , {eqa : per
       , T1 ⇓ (mkc_partial A1) × T2 ⇓ (mkc_partial A2)
       × ts A1 A2 eqa × (∀ a, eqa a a → a⇓)
       × (∀ t t', eq t t' ⇔ ((t⇓ ⇔ t'⇓) × (t⇓ → eqa t t'))) }}.
```

Type Family. Allen [3] introduces the concept of type families to define dependent types such as function types and W types. A type family *TyCon* is defined as a family of types B parametrized by a domain A. In the following definition `per-fam(eqa)` stands for `(∀ (a a' : CTerm) (p : eqa a a'), per)`, which is the type of PERs of type families over a domain with PER *eqa*; and `CVTerm l` is the type of terms with free variables contained in *l*.

```
Definition type_family TyCon (ts : cts) T1 T2 eqa (eqb : per-fam(eqa)) : Prop:=
  {A, A' : CTerm , {v, v' : NVar , {B : CVTerm [v] , {B' : CVTerm [v'] ,
     T1 ⇓ (TyCon A v B) × T2 ⇓ (TyCon A' v' B')
     × ts A A' eqa
     × (∀ a a', ∀ e : eqa a a', ts (B[v\a]) (B'[v'\a']) (eqb a a' e))}}}}.
```

Equalities of type families in our formalization (such as *eqb* above) are five place relations, while they are simply three place relations in Allen's and Crary's formalizations. This is due to the fact that conceptually `∀ (a' : CTerm)` and `∀ (p : eqa a a')` could be turned into intersection types because *eqb* only depends on the fact that the types are inhabited and does not make use of the inhabitants.

Dependent Function. Dependent function types are defined so that functional extensionality is a trivial consequence.

```
Definition per_func (ts : cts) T1 T2 (eq : per) : Prop :=
  {eqa : per, {eqb : per-fam(eqa)
       , type_family mkc_function ts T1 T2 eqa eqb
       × (∀ t t', eq t t' ⇔
            (∀ a a' (e : eqa a a'), eqb a a' e (mkc_apply t a) (mkc_apply t' a')))}}.
```

W. We define W types, by first inductively defining their PERs called weq [3]:

```
Inductive weq (eqa : per) (eqb : per-fam(eqa)) (t t' : CTerm) : Prop :=
| weq_cons :
    ∀ (a f a' f' : CTerm) (e : eqa a a'),
      t ⇓ (mkc_sup a f)
      → t' ⇓ (mkc_sup a' f')
      → (∀ b b', eqb a a' e b b' → weq eqa eqb (mkc_apply f b) (mkc_apply f' b'))
      → weq eqa eqb t t'.

Definition per_w (ts : cts) T1 T2 (eq : per) : Prop :=
  {eqa : per, {eqb : per-fam(eqa) ,
   type_family mkc_w ts T1 T2 eqa eqb × (∀ t t', eq t t' ⇔ weq eqa eqb t t')}}.
```

Our technical report [5, Sec. 4.2] extends W types to parametrized W types and uses them to define parametrized inductive types (e.g., vectors).

6.3 Universes and Nuprl's Type System.

Universes. As Allen [3] and Crary [15] did, we now define Nuprl's universes of types, and finally Nuprl's type system. First, we inductively define a close operator on candidate type systems. Given a candidate type system *cts*, this operator builds another candidate type system from *ts* that is closed w.r.t. the type constructors defined above:

```
Inductive close (ts : cts) (T T' : CTerm) (eq : per) : Prop :=
  | CL_init : ts T T' eq → close ts T T' eq
  | CL_base : per_base (close ts) T T' eq → close ts T T' eq
  | CL_eq : per_eq (close ts) T T' eq → close ts T T' eq
  | CL_partial : per_partial (close ts) T T' eq → close ts T T' eq
  | CL_func : per_func (close ts) T T' eq → close ts T T' eq
  | CL_w : per_w (close ts) T T' eq → close ts T T' eq.
```

We define $\mathbb{U}(i)$, the Nuprl universe type at level i, by recursion on $i \in \mathbb{N}$:

```
Fixpoint univi (i : nat) (T T' : CTerm) (eq : per) : Prop :=
  match i with
  | 0 ⇒ False
  | S n ⇒ (T ⇓ (𝕌(n)) × T' ⇓ (𝕌(n))
           × ∀ A A', eq A A' ⇔ {eqa : per, close (univi n) A A' eqa})
         {+} univi n T T' eq    end.
```

Finally, we define univ, the collection of all universes, and the Nuprl type system as follows:

```
Definition univ (T T' : CTerm) (eq : per) := {i : nat , univi i T T' eq}.
Definition nuprl := close univ.
```

We can now define $t_1 \equiv t_2 \in T$ as {eq : per , nuprl T T eq × eq t_1 t_2} and $T \equiv T'$ as {eq : per , nuprl T T' eq}.

Type System. Let us now prove that nuprl is a *type system*, i.e., that it is a candidate type system *ts* that satisfies the following properties [3,15]:

1. *uniquely valued*: ∀ T T' eq eq', ts T T' eq → ts T T' eq' → eq ⇐₂⇒ eq'.
2. *equality respecting*: ∀ T T' eq eq', ts T T' eq → eq ⇐₂⇒ eq' → ts T T' eq'.
3. *type symmetric*: ∀ eq, symmetric (fun T T' ⇒ ts T T' eq).
4. *type transitive*: ∀ eq, transitive (fun T T' ⇒ ts T T' eq).
5. *type value respecting*: ∀ T T' eq, ts T T eq → T ∼ T' → ts T T' eq.

6. *term symmetric*: $\forall\ T\ eq,\ ts\ T\ T\ eq \to \text{symmetric}\ eq$.
7. *term transitive*: $\forall\ T\ eq,\ ts\ T\ T\ eq \to \text{transitive}\ eq$.
8. *term value respecting*: $\forall\ T\ eq,\ ts\ T\ T\ eq \to \forall\ t\ t',\ eq\ t\ t \to t \sim t' \to eq\ t\ t'$.

A type system uniquely defines the PERs of its types. The last six properties state that $\_\equiv\_\in\_$ and $\_\equiv\_$ are PERs that respect computational equivalence. This definition differs from Crary's [15] as follows: (1) We added condition 2 because Allen and Crary consider equivalent PERs to be equal and we decided not to add the propositional and function extensionality axioms; (2) We strengthened conditions 5 and 8 by replacing $\Downarrow$ with $\sim$. This seemed necessary to obtain a strong enough induction hypothesis when proving that our new definition of per_eq preserves the type system properties [5, Sec. 4.2]. These properties and the congruence of $\sim$ allow us to do computation in any context in sequents.

Finally, the following lemma corresponds to Crary's Lemma 4.13 [15]: For all $i \in \mathbb{N}$, univi i is a type system; and the following theorem corresponds to Crary's Lemma 4.14 [15]: univ and nuprl are type systems.

6.4 Sequents and Rules. In Nuprl, one reasons about the nuprl type system types using a sequent calculus, which is a collection of rules that captures many properties of the nuprl type system and its types. For example, for each type we have introduction and elimination rules. This calculus can be extended as required by adding more types and/or rules. This section presents the syntax and semantics of Nuprl's sequents and rules. We then prove that these rules are valid w.r.t. the above semantics, and therefore that Nuprl is consistent. This paper provides a safe way to add new rules by allowing one to formally prove their validity, which is a difficult task without the help of a proof assistant[9].

Syntax of Sequents and Rules. Sequents are of the form $h_1, \ldots, h_n \vdash T\ \lfloor \textbf{ext}\ t \rfloor$, where t is the *extract/evidence* of T, and where an hypothesis h is either of the form $x : A$ (non-hidden) or of the form $[x : A]$ (hidden). Such a sequent states that T is a type and t is a member of T. A rule is a pair of a sequent and a list of sequents, which we write as $(S_1 \wedge \cdots \wedge S_n) \Rightarrow S$. To understand the necessity of hidden hypotheses, let us consider the following intersection introduction rule:

$$H, [x : A] \vdash B[x]\ \lfloor \textbf{ext}\ e \rfloor\ \wedge\ H \vdash A = A \in \mathbb{U}_i\ \Rightarrow\ H \vdash \cap a{:}A.B[a]\ \lfloor \textbf{ext}\ e \rfloor$$

This rule says that to prove that $\cap a{:}A.B[a]$ is true with extract e, one has to prove that $B[x]$ is true with extract e, assuming that x is of type A. The meaning of intersection types requires that the extract e be the same for all values of A, and is therefore called the *uniform evidence* of $\cap a{:}A.B[a]$. The fact that x is hidden means that it cannot occur free in e (but can occur free in B). The same mechanism is required to state the rules for, e.g., subset types or quotient types.

Semantics of Sequents and Rules. Several definitions for the truth of sequents occur in the Nuprl literature [14,15,24]. Among these, Kopylov [24]'s definition was the simplest. We provide here an even simpler definition and we have proved in Coq that all these definitions are equivalent [5, Sec. 5.1]. The semantics we

[9] Howe [22] writes: "Because of this complexity, many of the Nuprl rules have not been completely verified, and there is a strong barrier to extending and modifying the theory."

present uses a notion of *pointwise functionality* [15, Sec. 4.2.1], which says that each type in a true sequent must respect the equalities of the types on which it depends. This is captured by formula 1 below for the hypotheses of a sequent, and by formula 2 for its conclusion. For the purpose of this discussion, let us ignore the possibility that some hypotheses can be hidden.

Let H be a list of hypotheses of the form $x_1 : T_1, \ldots, x_n : T_n$, let s_1 be a substitution of the form $(x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)$, and let s_2 be a substitution of the form $(x_1 \mapsto u_1, \ldots, x_n \mapsto u_n)$.

Similarity. Similarity lifts the notion of equality in a type (i.e., $\_\equiv\_\in\_$) to lists of hypotheses. We say that s_1 and s_2 are similar in H, and write $s_1 \equiv s_2 \in H$, if for all $i \in \{1, \ldots, n\}$, $t_i \equiv u_i \in T_i[x_1 \backslash t_1; \cdots; x_{i-1} \backslash t_{i-1}]$. Let $s \in H$ be $s \equiv s \in H$.

Equal Hypotheses. The following notion of equality lifts the notion of equality between types (i.e., $\_\equiv\_$) to lists of hypotheses. We say that the hypotheses H are equal w.r.t. s_1 and s_2, and write $s_1(H) \equiv s_2(H)$, if for all $i \in \{1, \ldots, n\}$, $T_i[x_1 \backslash t_1; \cdots; x_{i-1} \backslash t_{i-1}] \equiv T_i[x_1 \backslash u_1; \cdots; x_{i-1} \backslash u_{i-1}]$.

Hypotheses Functionality. We say that the hypotheses H are pointwise functional w.r.t. the substitution s, and write $H@s$ if

$$\forall s'.\ s \equiv s' \in H \Rightarrow s(H) \equiv s'(H) \tag{1}$$

Truth of Sequents. We say that a sequent of the form $H \vdash T\ \lfloor \mathbf{ext}\ t \rfloor$ is true if

$$\forall s_1, s_2.\ s_1 \equiv s_2 \in H \wedge H@s_1\ \Rightarrow\ T[s_1] \equiv T[s_2] \wedge t[s_1] \equiv t[s_2] \in T[s_1] \tag{2}$$

In addition the free variables of t have to be non-hidden in H.

Validity of Rules. A rule of the form $(S_1 \wedge \cdots \wedge S_n) \Rightarrow S$ is valid if assuming that the sequents S_1, ..., S_n are true then the sequent S is also true.

Consistency. Using the framework described in this paper we have currently verified over 70 rules, including the usual introduction and elimination rules to reason about the core type system presented above in Sec. 6.2 [5, Sec. 5.2].

A Nuprl proof is a tree of sequents where each node corresponds to the application of a rule. Because we have proved that the above mentioned rules are correct, using the definition of the validity of a rule, and by induction on the size of the tree, this means that the sequent at the root of the tree is true w.r.t. the above PER semantics. Hence, a proof of False, for any meaningful definition of False, i.e., a type with an empty PER such as $(0 = 1 \in \mathbb{Z})$, would mean that the PER is in fact non-empty, which leads to a contradiction.

Building a Trusted Core of Nuprl. Using our proofs that the Nuprl rules are correct, and the definition of the validity of rules, we can then build a verified proof refiner (rule interpreter) for Nuprl. Our technical report [5, Sec. 5.3] illustrates this by presenting a Ltac based refiner in Coq that allows one to prove Nuprl lemmas. These proofs are straightforward translations of the corresponding Nuprl proofs and we leave for future work the automation of this translation. An appealing use of such a tool is that it can then be used as Nuprl's trusted core which checks that proofs are correct, i.e., if the translation typechecks in Coq, this means that the Nuprl proof is correct.

7 Related Work

Perhaps the closest work to ours is that of Barras [7]. He formalizes Werner's [37] set theoretic semantics for a fragment of Coq in Coq by first axiomatizing the required set theory in Coq. While this fragment has the Peano numbers, inductive types are missing. Werner's semantics assumes the existence of inaccessible cardinals to give denotations to the predicative universes of Coq as sets. In earlier work, Barras and Werner [8] provide a deep embedding of a fragment of Coq that excludes universes and inductive types. They thereby obtain a certified typechecker for this fragment.

Similarly, Harrison [20] verified: (1) a weaker version of HOL Light's kernel (with no axiom of infinity) in HOL Light (a proof assistant based on classical logic); and (2) HOL Light in a stronger variant of HOL light (by adding an axiom that provides a larger universe of sets). Myreen et al. are extending this work to build a verified implementation of HOL Light [31] using their verified ML-like programming language called CakeML [25]. They fully verified CakeML in HOL4 down to machine code. Similarly, Myreen and Davis formally proved the soundness of the Milawa theorem prover [30] (an ACL2-like theorem prover) which runs on top of their verified Lisp runtime called Jitawa [29]. Both these projects go further by verifying the implementations of the provers down to machine code. Also, Nuprl's logic is different from those of HOL and Milawa, e.g., HOL does not support dependent types and Milawa's logic is a first-order logic of total, recursive functions with induction.

Also, Buisse and Dybjer [12] partially formalize a categorical model of an intensional Martin-Löf type theory in Agda.

Uses of induction-recursion to define shallow embeddings of hierarchies of Martin-Löf universes have often been described in the literature [17,27]. However, because we have a deep embedding, our inductive-recursive definition is parametrized over the already defined terms. This deep-embedding approach is required for our goal of extracting an independent correct by construction proof assistant. Also, the extensionality of Nuprl complicates our definitions a bit. For example, we have to define equality of types instead of just typehood. Danielsson [16] uses induction-recursion to define a deep embedding of a dependently typed language that does not have universes and inductive types.

8 Future Work and Acknowledgments

As future work, we want to formalize a tactic language and build a user (Emacs/NetBeans) interface for our verified Nuprl version, based on the code extracted from our Coq development. Also, we plan to add a way to directly write inductive definitions (possibly parametrized and/or mutual) and have a formally verified and transparent translation to our formalized parametrized W types. Finally, we plan to formalize a typechecker for a large part of Nuprl.

We thank the Coq and Agda mailing lists' members for helping us with various issues while using these tools. We thank Mark Bickford, Robert L. Constable, David Guaspari, and Evan Moran for their useful comments as well as Jason Gross from whom we learned that Agda allows inductive-recursive definitions.

References

1. *Interactive Theorem Proving - 4th Int'l Conf.*, volume 7998 of *LNCS*. Springer, 2013.
2. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using W-types. In *ICALP 2004*, volume 3142 of *LNCS*, pages 59–71. Springer, 2004.
3. Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.
4. Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469, 2006. http://www.nuprl.org/.
5. Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. Technical report, Cornell University, 2014. http://www.nuprl.org/html/Nuprl2Coq/.
6. Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL 2008*, pages 3–15. ACM, 2008.
7. Bruno Barras. Sets in Coq, Coq in sets. *Journal of Formalized Reasoning*, 3(1):29–48, 2010.
8. Bruno Barras and Benjamin Werner. Coq in Coq. Technical report, INRIA Rocquencourt, 1997.
9. Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004. http://coq.inria.fr/.
10. Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd Int'l Conf.*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009. http://wiki.portal.chalmers.se/agda/pmwiki.php.
11. Edwin Brady. Idris —: systems programming meets full dependent types. In *5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011*, pages 43–54. ACM, 2011.
12. Alexandre Buisse and Peter Dybjer. Towards formalizing categorical models of type theory in type theory. *Electr. Notes Theor. Comput. Sci.*, 196:137–151, 2008.
13. Venanzio Capretta. A polymorphic representation of induction-recursion. www.cs.ru.nl/~venanzio/publications/induction_recursion.ps, 2004.
14. R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
15. Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, August 1998.
16. Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Types for Proofs and Programs, Int'l Workshop*, volume 4502 of *LNCS*, pages 93–109. Springer, 2006.
17. Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000.
18. Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. *Ann. Pure Appl. Logic*, 124(1-3):1–47, 2003.
19. Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould

Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *ITP'13* [1], pages 163–179.

20. John Harrison. Towards self-verification of hol light. In *IJCAR 2006*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.
21. Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of Fourth IEEE Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, 1989.
22. Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 85–101. Springer-Verlag, Berlin, 1996.
23. Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *POPL'13*, pages 193–206. ACM, 2013.
24. Alexei Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects*. PhD thesis, Cornell University, Ithaca, NY, 2004.
25. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *POPL'14*, pages 179–192. ACM, 2014.
26. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL'06*, pages 42–54. ACM, 2006.
27. Conor McBride. Hier soir, an OTT hierarchy, 2011. http://sneezy.cs.nott.ac.uk/epilogue/?p=1098.
28. P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.
29. Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In *ITP 2011*, volume 6898 of *LNCS*, pages 265–280. Springer, 2011.
30. Magnus O. Myreen and Jared Davis. The reflective milawa theorem prover is sound (down to the machine code that runs it), 2014. Accepted to ITP 2014.
31. Magnus O. Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of hol light. In *ITP'13* [1], pages 490–495.
32. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
33. Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *TLCA'93*, volume 664 of *LNCS*, pages 328–345. Springer, 1993.
34. Vincent Rahli, Mark Bickford, and Abhishek Anand. Formal program optimization in Nuprl using computational equivalence and partial types. In *ITP'13* [1], pages 261–278.
35. Anton Setzer. *Proof theoretical strength of Martin-Löf Type Theory with W-type and one universe*. PhD thesis, Ludwig Maximilian University of Munich, 1993.
36. I.A.S. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Univalent Foundations, 2013.
37. Benjamin Werner. Sets in types, types in sets. In *TACS*, volume 1281 of *LNCS*, pages 530–546. Springer, 1997.