A Verified Theorem Prover Backend Supported by a Monotonic and Name-Invariant Library

Vincent Rahli[1], Liron Cohen[2], and Mark Bickford[2]

[1] Luxembourg University, Luxembourg
[2] Cornell University, Ithaca, NY, USA

Abstract. Building a verified proof assistant entails implementing and mechanizing the concept of a library. This includes adding support for standard manipulations on the library, such as adding new definitions and proofs. In this work we develop such mechanism for the Nuprl proof assistant, and integrate it into the formalization of Nuprl's theory in Coq. We *formally* verify that standard operations on the library preserve its validity. This is a key property for any interactive theorem prover, since it ensures consistency. Some unique features of Nuprl, such as the presence of undefined abstractions, make the proof of these property nontrivial. Thus, e.g., to achieve monotonicity the semantics of sequents had to be refined. We also show that the library is invariant with respect to the names of its objects. On a broader view, this work provides the backend for a verified version of Nuprl, which in turn can be used as a proof checker for Nuprl by importing Nuprl proofs into our Coq formalization.

1 Introduction

Interactive theorem provers are remarkable software systems with the aim of mechanizing formal reasoning at the highest level of rigor. Nowadays they are playing a major role in high value applications of formal methods such as certified programming, advanced programming language design, cyber security, cyber physical systems, etc. But what makes a prover trustworthy? There are many different components underlying a theorem prover that must be trusted in order for one to trust the proofs it generates, from the logic it implements down to the hardware on which the code runs. One such key component is the underlying library. In most mainstream theorem provers all the definitions, lemmas, and proofs are stored in a library which underlines the computation system. Such digital libraries allow users to organize, share, search, and reuse their code and knowledge, as well as manage the complexity of large implementations [4]. In this paper we internalize the concept of the underlying library and present a formally verified backend of Nuprl [17; 5]. Nuprl is a theorem prover that implements a dependent type theory, in the spirit of Martin-Löf's extensional theory [40], which is built on top of an untyped and lazy λ-calculus.

The first stage in building a formally verified theorem prover was taken in [8] where Nuprl's meta-theory was formalized in Coq [11; 20]. The implementation includes Nuprl's computation and type systems. Nuprl's derivation rules were proven to be valid w.r.t. Allen's PER semantics, and the consistency of Nuprl relative to that of Coq's was established. The full implementation is available at `https://github.com/vrahli/NuprlInCoq`, and additional information can be found at `http://www.nuprl.org/html/Nuprl2Coq/`.

In this paper we take the next step by adding support for the dependency of the computation and type systems on the library. This includes: (1) formalizing the concept of a proof as a tree of instances of derivation rules; (2) formalizing the concept of a library of interdependent definitions and proofs, where definitions can refer to proofs, and proofs can refer to definitions; (3) defining the concept of *validity* for such a library; (4) adding support for standard manipulations on a library such as creating new definitions and proofs; (5) and proving that these manipulations preserve the validity of libraries. These operations on the library can be invoked by users through commands that can be combined into a *script*. Furthermore, we build support for using this verified theorem prover backend as a proof checker for Nuprl in the form of a tool that takes as input a Nuprl proof and converts it into a script, which can then be checked by the Coq implementation. That is, it can check that applying the commands of the script yields a valid library.

The first step is to add support for the dependency of the system on a library. In Nuprl most computation steps amount to pattern matching or β-reductions. However, because proofs are built in the context of a library of definitions and lemmas, performing a δ-reduction, i.e., unfolding a definition from the library, counts as a computation step and therefore as a proof step too. In order to be able to verify the validity of proof steps that involve computations, we must have a formal account of what a single computation step is. To support this, we must first extend the computation system with user definitions, which are also called *abstractions* (not to be confused with λ-abstractions).

Given that computations, and therefore proofs, depend on the current state of the library, it is essential to prove some preservation properties of the library. The main desired property is that standard actions on a library (e.g. adding new definitions and new lemmas) all preserve its validity. Nuprl has several features that pose some unique challenges in formally verifying such properties of a digital library. In particular, the existence of proof terms (extracts) and undefined abstractions. The main difficulty here is that in any given state of the library most terms, namely the abstractions that have not been defined yet, do not have any operational semantics. However, in the spirit of the open-ended nature of Nuprl [22], we do not want to syntactically forbid the use of *undefined abstractions*. Moreover, they offer some advantages such as providing a natural way to define (mutually-)recursive functions as illustrated in Sec. 3. Therefore, because undefined abstractions are part of the computation system and may appear in other syntactic objects, we must control their behavior semantically. Objects that mention undefined abstractions are unstable in the sense that they can change their value once these abstractions are defined. Therefore, the meaning of lemmas that mention undefined abstractions does not rely solely on the current library, and hence simply parameterizing the semantics of the system by a library is insufficient. To internalize the notion of undefined abstractions, the semantics must be extended to accommodate the evolving behavior of undefined objects. We here address this issue by first parameterizing the semantics of types and sequents by a library parameter, and then constraining the library parameter so that a sequent is true only if it is true in *all possible extensions* of the current library (in the spirit of Kripke's semantics [**Kripke1963**]).

We proved that this new semantics is a conservative generalization of the old one, in the sense that sound rules w.r.t. the old semantics remain sound w.r.t. the new semantics. Given this new semantics we prove the monotonicity of the system with respect to the library, i.e. that all standard operations on the library preserve the semantics. We also prove a name-invariance theorem, which states that renaming certain entries in the library results in the same theory modulo the same renaming. This is a critical feature of the library since in practice we often rename definitions and lemmas in our implementations for various reasons, e.g., to use a different convention, to avoid clashes, etc.

Contributions.

- Providing a formal treatment of the concept of computation in the context of a library.
- Building support for users to perform standard operations such as: starting a new proof; updating an incomplete proof; committing a complete proof to the library and building its extract[3]; and adding a new definition.
- Proving salient properties of the system, such as monotonicity and name-invariance. This involved incorporating libraries into the semantics of sequents in Nuprl, and establishing the conservativity of this new semantics.
- Constructing a fully automated tool converting Nuprl proofs into proofs in the Coq formalization of Nuprl, so as to be checked for validity.
- The entire development presented in this paper has been fully formalized in Coq, and can be accessed here: `https://github.com/vrahli/NuprlInCoq`.

Outline. The rest of the paper is organized as follows: Sec. 2 surveys some of Nuprl's key features which are essential to the work in this paper. Sec. 3 provides a detailed account of how user definitions are implemented and manipulated in Nuprl. This is done via second-order variables, and so their theory of substitution is developed, as well as their computation rules. In Sec. 4 we refine the semantics of sequents in Nuprl, which enables the monotonicity property of the library. We also prove the name-invariance property of the library. Sec. 5 describes the implementation of a library structure in Nuprl. Sec. 6 discusses a fully automated tool that we have implemented in order to use our verified proof assistant backend as a proof checker for the current version of Nuprl. Sec. 7 discusses related work, and Sec. 8 concludes.

[3] According to the proof-as-program paradigm, the extract of a proof of a proposition/type T is a term t that inhabits T. See section 4 for more details.

Fig. 1 Syntax (top) and operational semantics (bottom) of a subset of Nuprl

$$
\begin{array}{llll}
vt \in \texttt{Type} ::= \mathbb{Z} & \text{(integer type)} & \mid \texttt{Base} & \text{(base)} \\
\quad \mid t_1 = t_2 \in t & \text{(equality)} & \mid \mathbb{U}_i & \text{(universe)} \\
\quad \mid \boldsymbol{\Sigma} x{:}t_1.t_2 & \text{(sum)} & \mid \boldsymbol{\Pi} x{:}t_1.t_2 & \text{(product)} \\
\quad \mid t_1 \simeq t_2 & \text{(bisimulation)} & &
\end{array}
$$

$$
\begin{array}{llllll}
v \in \texttt{Value} ::= vt & \text{(type)} & \mid \lambda x.t & \text{(lambda)} & \mid \star & \text{(axiom)} \\
\quad \mid \langle t_1, t_2 \rangle & \text{(pair)} & \mid i & \text{(integer)} & &
\end{array}
$$

$$
\begin{array}{llll}
t \in \texttt{Term} ::= x & \text{(variable)} & \mid \texttt{ifpair}(\boxed{t_1}, t_2, t_3) & \text{(pair test)} \\
\quad \mid v & \text{(value)} & \mid \texttt{let}\ x, y = \boxed{t_1}\ \texttt{in}\ t_2 & \text{(spread)} \\
\quad \mid \boxed{t_1}\ t_2 & \text{(application)} & \mid \texttt{fix}(\boxed{t}) & \text{(fixpoint)}
\end{array}
$$

$$
\begin{array}{lll}
(\lambda x.F)\ a & \mapsto & F[x\backslash a] \\
\texttt{fix}(v) & \mapsto & v\ \texttt{fix}(v)
\end{array}
\qquad
\begin{array}{lll}
\texttt{let}\ x, y = \langle t_1, t_2 \rangle\ \texttt{in}\ F & \mapsto & F[x\backslash t_1; y\backslash t_2] \\
\texttt{ifpair}(\langle t_a, t_b \rangle, t_1, t_2) & \mapsto & t_1 \\
\texttt{ifpair}(v, t_1, t_2) & \mapsto & t_2, \text{ if } v \text{ is not a pair}
\end{array}
$$

2 Background

Nuprl has a rich type theory, called Constructive Type Theory (CTT), including equality types, a hierarchy of universes, W types, quotient types [16], set types, union and (dependent) intersection types [31], image types [46], PER types [6], approximation and computational equivalence types [50], and partial types [53; 18; 21]. CTT types are interpreted as Partial Equivalence Relations (PERs) on closed terms, by Allen's semantics [2]. CTT differs from other similar constructive type theories, such as the ones implemented by Agda [13; 1], Coq, or Idris [14; 30], primarily because it is an extensional type theory (i.e., propositional and definitional equality are identified [26]) with types of partial functions [53; 18; 21].[4] The rich expressiveness of Nuprl renders type checking undecidable, but in practice this is mitigated by type inference and type checking heuristics implemented as tactics.

Nuprl's programming language is an untyped (à la Curry), lazy λ-calculus with pairs, a fixpoint operator, etc. Fig. 1 presents a subset of Nuprl's syntax and small-step operational semantics [5; 47], which will be used in this paper. A term is either (1) a variable; (2) a value (also called a canonical form); or (3) a non-canonical term. A non-canonical term t has *principal arguments*—marked using boxes in Fig. 1—which are terms that have to be evaluated to canonical forms before t can be reduced further. For example the application $f\ a$, which we often write $f(a)$, diverges if f diverges. The bottom part of Fig. 1 presents part of Nuprl's small-step operational semantics. We omit the rules that reduce principal arguments such as: if $t_1 \mapsto t_2$ then $t_1(u) \mapsto t_2(u)$.

Nuprl's types are interpreted as partial equivalence relations (PERs) on closed terms [2; 3; 21]. The PER semantics can be seen as an inductive-recursive definition of: (1) an inductive relation $T_1 {\equiv} T_2$ that expresses type equality; (2)

[4] For example, the fixpoint $\texttt{fix}((\lambda x.x))$ diverges. It is nonetheless a member of, among others, the partial type $\overline{\mathbb{Z}}$, which is the type of integers and diverging terms. The type $\overline{\mathbb{Z}}$ can be seen as the integer type of ML-like programming languages such as OCaml.

a recursive function $a{\equiv}b{\in}T$ that expresses equality in a type. For example, one case in the definition of $T_1{\equiv}T_2$ states that (i) T_1 computes to $\boldsymbol{\Pi} x_1{:}A_1.B_1$; (ii) T_2 computes to $\boldsymbol{\Pi} x_2{:}A_2.B_2$; (iii) $A_1{\equiv}A_2$; and (iv) for all closed terms t_1 and t_2 such that $t_1{\equiv}t_2{\in}A_1$, $B_1[x_1\backslash t_1]{\equiv}B_2[x_2\backslash t_2]$. We say that a term t *inhabits* or *realizes* a type T if t is equal to itself in the PER interpretation of T, i.e., $t{\equiv}t{\in}T$. It follows from the PER interpretation of types that an equality type of the form $a = b \in T$ is true (i.e. inhabited) iff $a{\equiv}b{\in}T$ holds. [8; 47].

Howe [29] wrote about this PER semantics: “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”. This is no longer true thanks to the recent formalization of CTT in Coq [8; 47; 49]. The implementation includes: (1) Nuprl’s computation system; (2) Howe’s computational equivalence relation, and a proof that it is a congruence; (3) a definition of the PER semantics of CTT; (4) definitions of Nuprl’s derivation rules, and their soundness proofs w.r.t. the PER semantics[5] (5) and a proof of Nuprl’s consistency. This formalization allows for a safe and mechanical way to verify the soundness of existing as well as new rules. This paper, in turn, builds upon and extends this formalization.

Systems such as Coq, Agda, or Idris are only closed under computation, meaning that one can only substitute a term for another term if they compute to the same value. Nuprl in contrast is closed under the more general notion of Howe’s computational equivalence $\sim$, which was proven to be a congruence [27]. In general, computing and reasoning about computation in Nuprl involves reasoning about Howe’s computational equivalence relation. However, as we will see in the sequel, the current definition of the relation is not adequate to handle evolving libraries of definitions, and therefore evolving computation systems (see Sec. 4). We commonly use Howe’s computational equivalence to reduce expressions inside sequents by proving that terms are computationally equivalent and using the fact that $\sim$ is a congruence. For that, Nuprl provides the type $t_1 \simeq t_2$, which is the theoretical counterpart of the metatheoretical relation $t_1 \sim t_2$. Also, to reason about any term in the computation system, Nuprl provides the `Base` type, which is the type of all closed terms of the computation system with $\sim$ as its equality.

3 Computing With User Definitions

User definitions in Nuprl are more general than definitions used in other theorem provers such as Coq or HOL, since they include undefined abstractions. In accordance with refinement-based methodologies, Nuprl supports using a abstraction before it is actually defined. The existence of undefined abstractions offers major advantages, such as providing a “natural” way of defining recursive functions. The ‘fix’ operator, for example, can be defined using abstractions simply by $\texttt{fix}(f) == f(\texttt{fix}(f))$, which indicates that the term on the left of the $==$ symbol unfolds to the term on the right. What is more, unconstraint

[5] See https://github.com/vrahli/NuprlInCoq/blob/master/RULES for a list of Nuprl’s inference rules along with pointers to their soundness proofs.

definitions allow for a simple, user-friendly way to support mutual recursion. A user can simply write down the definitions one after the other. The unfolding of such definitions behaves precisely as their computational semantics entails.

User definitions in Nuprl are slightly different than those of similar systems such as Coq because their parameters can include some information regarding their binding structures. This is especially useful in untyped languages such as Nuprl because it provides partial information regarding the meaning of the parameters of a definition. This is achieved using *generalized variables*, which are sometimes called *second-order variables* in the Nuprl literature.

Example. Let us provide a simple example of such a definition. In Nuprl, a list is either a pair of an element and a list, or the constant $\star$, which indicates the end of the list. The map function on a list could be defined as follows:

$$\mathtt{map}(x.f[x];L) == \mathtt{ifpair}(L, \mathtt{let}\ h,t = L\ \mathtt{in}\ \langle f[h], \mathtt{map}(x.f[x];t)\rangle, L)$$

In this definition the parameter L is a standard variable, i.e., a generalized variable of arity 0, while $x.f[x]$ indicates that f is a generalized variable of arity 1, which indicates that `map`'s first argument is meant to be a function. The expression $f[h]$ is the application of the generalized (second-order) variable f to the term h. Note that `map` occurs in the right-hand-side of the definition, even though it is not yet defined (however, the system does not generate a fixpoint from this definition).

Next, we need to incorporate user definitions into the computation system. In terms of computation, the only relevant fragment of the library is its definition entries, which we here call a *definition library*. First, we parameterize Nuprl's small step operational semantics by a definition library. Then, we extend it with a new case that reduces abstractions by looking up the abstraction in the definition library and then substitutes the arguments in the term that it found in the library (we illustrate this mechanism through an example below). To handle user definitions that make use of generalized variables[6], we extend the substitution mechanism to support *generalized substitution*, which we illustrate below (see sosub's definition in `https://github.com/vrahli/NuprlInCoq/blob/master/terms/sovar.v` for more details).

In addition to having the advantage of conveying more information about the binding structure of abstractions, generalized variables also offers the following advantage: without them it is sometimes necessary to instantiate definitions with functions and then β-reduce some applications, while the same can be performed by a single generalized substitution. We next illustrate this by describing how to reduce the following instance of `map`: $\mathtt{map}(x.x+1;[1,2,3])$.

[6] A generalized parameter is of the form $x_1,\dots,x_n.X[x_1;\cdots;x_n]$, which indicates that X can be substituted by a term that mentions the variables $x_1,\dots,x_n$. We write X for the variable $X[]$.

Retrieval. We first retrieve the generalized term, which we call T, that the abstraction maps to in the current definition library, i.e.:

$$\texttt{ifpair}(L, \texttt{let } h, t = L \texttt{ in } \langle f[h], \texttt{map}(x.f[x]; t)\rangle, L)$$

Substitution. The abstraction is turned into a generalized substitution by extracting its arguments. Formally, a generalized substitution is a list of triples each consisting of a standard variable, f, a list of standard variables, $x_1, \cdots, x_n$, and a standard term, t (intuitively, f stands for $\lambda x_1, \cdots, x_n.t$). Given the above example, we build a generalized substitution that essentially maps f to $\lambda x.(x+1)$ and L to $[1, 2, 3]$. Formally, the generalized substitution is (which we call *sub*):

$$[(f, [x], x + 1), (L, [], [1, 2, 3])]$$

Unfolding. This generalized substitution *sub* is then applied to the generalized term T to generate a standard term, i.e. one with no generalized variables. To do this we recurse through T, until we reach either f or L. When we reach the generalized variable $f[h]$, we first apply *sub* to the list of terms to which f is applied to: here to the singleton list $[h]$, where h is a generalized variable with no subterms. Because h does not occur in *sub*, this step has the effect of building the standard variable h from the generalized variable h. Next, we extract the pair $([x], x + 1)$ from *sub*. Then, we build the standard substitution $[(x, h)]$ from the list of variables $[x]$ (from the above pair) and the list of terms $[h]$. Finally, we apply this standard substitution to $x + 1$ to obtain $h + 1$. The end result is:

$$\texttt{ifpair}([1, 2, 3], \texttt{let } h, t = [1, 2, 3] \texttt{ in } \langle h + 1, \texttt{map}(x.x + 1; t)\rangle, [1, 2, 3])$$

4 Semantics

The generalized notion of user definitions in Nuprl, and in particular the presence of undefined abstractions, while useful in many ways, also poses some challenges in the formal treatment of libraries. In this section we describe some examples and the ways in which they were solved.

4.1 Monotonicity

Extensions by definitions is a well-known methodology that usually forms a monotonic extension of the original logic. That is, by adding new definitions one cannot falsify already establish theorems. However, Nuprl's definitions are more general than the standard logical ones, or even the ones used in other theorem provers. Because undefined abstractions are valid expressions that can occur in definitions, lemmas, and proofs, one has to be careful when adapting Nuprl's semantics to account for user definitions, in order to avoid the undesirable situation where adding a new definition makes a true lemma false.

The main difficulty is that due to the presence of undefined abstractions, Nuprl's computation system, and in turn, Howe's computational equivalence relation, are inherently non-monotonic. This is since all undefined abstractions are computationally equivalent to $\perp$, because according to the computation system defined in Sec. 3, they are stuck expressions, i.e. expressions which are not a value but cannot be further reduced (for example $1(0)$ is stuck because 1 not a function). By definition, Howe's computational equivalence relation equates diverging expressions such as $\perp$ (defined as $\texttt{fix}(\lambda x.x)$), and stuck expressions such as $1(0)$ because both kinds share the property that they do not reduce to values. Another example of a stuck expression, critical to the work here, is an instance of an abstraction that is not defined in the current library. Thus, if $\texttt{foo}()$ is an abstraction which is not defined in the library *lib*, then $\texttt{foo}()$ is computationally equivalent to $\perp$ w.r.t. *lib*, but it is not computationally equivalent to $\perp$ w.r.t. some extension of the library that defines $\texttt{foo}()$ to be, say, 0, because 0 and $\perp$ are not computationally equivalent.

Nuprl's main derivation rule for reasoning about computations relies on Howe's computational equivalence relation. For example, to prove that $((\lambda x.x)\ 0)$ reduces to 0, we prove that the proposition $((\lambda x.x)\ 0) \simeq 0$ is true by invoking the derivation rule which says that $a \simeq b$ is true if a computes to b in one computation step. As mentioned in Sec. 2, a type of the form $a \simeq b$ is true if the metatheoretical statement $a \sim b$ is true. Therefore, in order to address the above issue, one possible solution is to adapt Howe's computational equivalence relation to make it monotone. However, this would only establish monotonicity at the computation level, and it is unclear whether this property will propagate up to the level of the semantics of sequents and rules. Instead, we here take a more direct approach to obtaining monotonicity by adapting the semantics at the level of sequents to account for the dynamic process of extensions by definitions, and most critically, to account for the unstable nature of undefined abstractions.[7]

4.1.1 Old Sequent Semantics

Sequents in Nuprl are of the form $h_1, \ldots, h_n \vdash T\ \lfloor \textbf{ext}\ t \rfloor$. The term t is a member of the type T, which in this context is called the *extract* or *evidence* of T. An hypothesis h is of the form $x : A$, where the variable x is referred to as the name of the hypothesis and A its type. Such a sequent states, among other things, 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:

$$(S_1 \wedge \cdots \wedge S_n) \Rightarrow S$$

Extracts are programs that are computed by the system once a proof is complete using rules such as the standard function introduction rule:

$$(H, x : A \vdash B\ \lfloor \textbf{ext}\ b \rfloor) \Rightarrow (H \vdash x{:}A \to B\ \lfloor \textbf{ext}\ \lambda x.b \rfloor)$$

[7] See https://github.com/vrahli/NuprlInCoq/blob/master/per/sequents_lib.v for details.

This rule says that to prove $x{:}A \to B$, it is enough to prove B assuming the extra hypothesis $x : A$. Users do not have to enter extracts; those are automatically computed from the leaves of a proof up to the root of the proof. In the above example, once the system has extracted b for the subgoal, it generates $\lambda x.b$ as extract for the main goal. Note that it is sometimes desirable to have control over the extracts generated by the system. Therefore, Nuprl provides the following rule to enforce a user-provided extract t:

$$(H \vdash t \in T \ \lfloor\textbf{ext}\ e\rfloor) \Rightarrow (H \vdash T \ \lfloor\textbf{ext}\ t\rfloor)$$

This rule turns the goal into a membership goal, to let the user prove that t is indeed a member of T. Once a proof is complete, Nuprl builds an abstraction that points to the extract of the proof. A similar procedure is invoked in systems such as Coq, where lemmas and definitions are interchangeable: once a lemma has been established it can be used as a definition which unfolds to its extract.

Several definitions for the truth of sequents occur in the Nuprl literature [17; 21; 31; 8]. They were all shown to be equivalent [7, Sec. 5.1]. Since our results are invariant to the specific semantics, we do not repeat these definitions here. Instead, we take `trueSequent` to be any of the possible definitions of the truth of a sequent. The sequent semantics standardly induces the notion of validity of a rule of the form $(S_1 \wedge \cdots \wedge S_n) \Rightarrow S$ in the following way:

$$\texttt{trueSequent}(S_1) \wedge \cdots \wedge \texttt{trueSequent}(S_n) \to \texttt{trueSequent}(S)$$

4.1.2 Modified Sequent Semantics

The library of Nuprl has always had an implicit impact on the semantics of sequents. This is because the sequent semantics is build on top of the PER semantics, which is based on the computation system. In order to unfold abstractions as part of computations, this dependency had to be make explicit. Thus, we first adapt the semantics of types so as to also depend on a library of definitions.

Example. Prior to this work, the integer type $\mathbb{Z}$ was interpreted by the following 4-ary predicate `INT`:

$$\texttt{INT}(\tau, T, T', \phi) = T \Downarrow \mathbb{Z} \wedge T' \Downarrow \mathbb{Z} \wedge (\forall t, t'.\ t\ \phi\ t' \iff \exists i.\ t \Downarrow i \wedge t' \Downarrow i)$$

This states that T and T' are equal types of the type system τ if they both compute to the integer type $\mathbb{Z}$ ($a \Downarrow b$ denotes that a computes to b); and moreover ϕ is the partial equivalence relation of the integer type, i.e., t and t' are equal members of $\mathbb{Z}$ if they both compute to some integer i. The other type constructors were similarly interpreted by 4-ary predicates that described when two types are equal and uniquely defined their PERs.

The new 5-ary interpretation of the integer type is now parameterized by a library, as follows:

$$\begin{aligned}&\texttt{INT}(\mathit{lib}, \tau, T, T', \phi)\\ &= T \Downarrow_{\mathit{lib}} \mathbb{Z} \wedge T' \Downarrow_{\mathit{lib}} \mathbb{Z} \wedge (\forall t, t'.\ t\ \phi\ t' \iff \exists i.\ t \Downarrow_{\mathit{lib}} i \wedge t' \Downarrow_{\mathit{lib}} i)\end{aligned}$$

where $a \Downarrow_{\mathit{lib}} b$ says that a reduces to b w.r.t. the definition library lib.

Consequently, we parameterize the `trueSequent` predicate by a library of definitions: $\texttt{trueSequent}(S, \mathit{lib})$. Then, we define a new predicate that captures the notion of monotonicity (i.e. that a true lemma w.r.t. some library stays true in any extension of the library) as follows:

$$\texttt{monTrueSequent}(S, \mathit{lib}) = \forall \mathit{lib}'.\texttt{extends}(\mathit{lib}', \mathit{lib}) \to \texttt{trueSequent}(S, \mathit{lib}')$$

where $\texttt{extends}(\mathit{lib}', \mathit{lib})$ says that every abstraction defined in lib has the same definition in lib'. That is, lib' is allowed to have a larger set of abstractions than lib. The induced semantics says that to prove that a sequent is valid, we now have to prove that it is valid in all possible extensions of the library.

Accordingly, the semantics of a rule R of the form $(S_1 \wedge \cdots \wedge S_n) \Rightarrow S$ is naturally generalized to:

$$\begin{aligned}&\texttt{monTrueRule}(R)\\ &= \forall \mathit{lib}.\ (\texttt{monTrueSequent}(S_1, \mathit{lib}) \wedge \cdots \wedge \texttt{monTrueSequent}(S_n, \mathit{lib}))\\ &\qquad\quad \to\ \texttt{monTrueSequent}(S, \mathit{lib})\end{aligned}$$

This new semantics states that Nuprl's rules are immune to the open-ended nature of the system [28].

It is then straightforward to translate the proofs that Nuprl's rules are sound w.r.t. the old semantics to obtain a soundness result w.r.t. the new monotonic semantics. This shows that properties about the current state of a library that were ever used in Nuprl proofs are always preserved under extensions of that library. The fact that none of the rules had to be changed in order to comply with the new semantics provides another evidence for the adequacy of the modified semantics, which, in turn, enables us to prove the monotonicity property according to which all the actions on the library described in Sec. 5 preserve its validity.

Theorem 1 (Monotonicity).

$$\begin{aligned}&\forall \mathit{lib}, S.\ \texttt{monTrueSequent}(S, \mathit{lib})\\ &\qquad \to\ \forall \mathit{lib}'.\texttt{extends}(\mathit{lib}', \mathit{lib}) \to \texttt{monTrueSequent}(S, \mathit{lib}')\end{aligned}$$

The open-ended nature of the `Base` type[8] in Nuprl entails that when proving a lemma that mentions variables in `Base`, the underlying PER semantics says

[8] As mentioned in Sec. 2, `Base` is the the type of all closed terms of the computation system with Howe's computational equivalence relation as its equality. Therefore, it is open-ended insofar as the terms are open-ended.

that it has to be proven for all syntactically correct terms, even those that have not been defined yet. We strive for a similar behavior w.r.t. the undefined abstractions. However, a naive semantics parameterized by a library without any further constraints, would not enforce anything on undefined abstractions. Thanks to this new monotonic semantics, proving a lemma now requires proving that it holds for all possible instantiations of the undefined abstractions, which yields the desired constraints on the undefined abstractions. The key difference between undefined abstractions in the library and variables in `Base` manifests in the instantiation of a lemma. Undefined abstractions behave globally in the sense that one can simply fill in the abstraction, while variables in `Base` behave locally in the sense that one has to instantiate the lemma itself.

Undefined abstractions also seem to bear some resemblance to the notion of open-terms and to the widely used programming languages concept of global variables. This is mainly due to the fact that claims concerning them amount to claims about possible closed instantiations of them. However, there are some key differences, such as the fact that an undefined abstraction can only be instantiated once and from that point on it remains constant.

Circling back to the discussion in the beginning of this section regarding the non-monotonic nature of the computation system, it should be noted that while we chose to modify the semantics at the level of sequents rather than Howe's computational equality, this change to the semantics also percolates down to reasoning about computations. It constrains the theory in such a way that we can only prove that two undefined abstractions are computationally equal if they are syntactically equal. This is because two undefined abstractions could always be defined differently in two different extensions of the library. Thus, it is no longer true that undefined abstractions are computationally equivalent to $\bot$.

Remark 2. Our main focus in this paper was to allow *building up* a library of knowledge with the guarantee of preserving validity. Nevertheless, another key property of libraries is that of conservativity, i.e. that adding new definitions does not increase the provability power over the original library. The presence of undefined abstractions again makes this a non-trivial statement. This is because an abstraction that occurs undefined in a given library can be defined in an extension of that library, thus permitting new results concerning the previously undefined object. To see this, consider the following example: Let *lib* be a library containing the sole definition: $\texttt{foo}() == \texttt{bar}()$. Clearly, under *lib*, one cannot prove $\texttt{foo}() = 4 \in \mathbb{Z}$. However, extending the library by adding the definition: $\texttt{bar}() == 4$ enables us to prove $\texttt{foo}() = 4 \in \mathbb{Z}$. While the example above shows that general extensions by definitions are not conservative, it also hints at the fact that the source of non-conservativity is the ability to define an undefined abstraction. If all abstractions are fully 'covered' by the original library, then any extension of the library is conservative. Formally establishing this requires further development such as formalizing notions of a library covering a sequent, dealing with cyclic definitions and such, and is so left for a consecutive paper.

4.2 Name-invariance

As part of the formal treatment of libraries and undefined abstractions, another key property one should look for is name-invariance.[9] By this we mean that renaming some entries in a library results in the same theory modulo the same renaming. This property is essential in any programming paradigm, for example, to allow users to change the names of their definitions and lemmas at any time. For instance, nominal logic has recently been explored as a convenient framework for handling names [24; 48]. A key feature of that framework is its indifference to any specific choice of names. To support renaming, we have implemented a renaming operation, which is similar to the name-swapping operation in nominal logic. A renaming is implemented as a pair of operator names, and to rename an operator name, we simply swap the name w.r.t. the renaming. It is straightforward to prove that this swapping operator preserves operations on terms, such as computing the free-variables of a term, or the substitution operation.

Formally, we show that Nuprl is name-invariant by proving the following:

Theorem 3 (name-invariance).

$$\forall r : \texttt{Renaming}.\ \texttt{monTrueSequent}(S, \mathit{lib}) \rightarrow \texttt{monTrueSequent}(r(S), r(\mathit{lib}))$$

where $r(e)$ applies the renaming r to the entity e.

The most involved part of the proof of the above theorem is establishing that the renaming preserves computations, Howe's computational equality relation, and the PER semantics of types.

5 Library

Thus far we have only considered definition libraries, that is, libraries whose entries are abstractions. We next provide the more general notion of a library that also contains facts. Note that this section provides an overview of the main concepts and methods of the formal implementation. The full formalization can be found at: https://github.com/vrahli/NuprlInCoq/blob/master/rules/proof_with_lib_non_dep.v.

5.1 Generalized Libraries

A *rigid* library is a list of definitions and proofs. A proof entry consists of a proof name *name* (simply a string in the implementation), and a Nuprl proof *prf*.

Proofs are inductively defined as trees of instances of proof rules. In the inductive definition of the type of proof steps (essentially proof rule names) each proof rule has a constructor. For example, the constructor 'proof_step_hypothesis' takes a variable x as argument, and is used to prove a sequent of the form $G, x : A, J \vdash A\ \lfloor\mathbf{ext}\ x\rfloor$. The 'proof_step_lemma' constructor differs from the

[9] See https://github.com/vrahli/NuprlInCoq/blob/master/rules/name_invariance.v for details.

other ones bcause it does not refer to any specific proof rule. Instead it takes a lemma name as argument, and is used to build a proof from one stored in the current library by referring to the name of the proof.

As a part of our formal treatment of a library, we also implement the concept of a *soft* library, which is a rigid library augmented by a list of incomplete proofs. Digital libraries often contain incomplete information, and soft libraries are intended to capture such natural behavior of libraries. As opposed to rigid libraries which contain knowledge that cannot be changed, a soft library contains incomplete and evolving knowledge. We sometimes simply call a soft library, a library. As opposed to a proof tree of a fact, an incomplete proof, which we also call a pre-proof, might have holes, i.e., leaves for which no proof tree has been provided yet. Pre-proofs are defined similarly to complete proofs except that (1) sequents, which we call pre-sequents in this context, do not contain extracts—extracts are built only once a proof is complete; and (2) the pre-proof type has an additional constructor for a hole in a proof, which allows one to make a proof hole that can later be filled with an another pre-proof, and eventually with an actual complete proof. It is then straightforward to define a partial function that turns incomplete proofs into complete ones whenever possible, i.e., whenever the incomplete proof does not have holes.

5.2 Script

A *script* is a sequence of standard operations on the library. We have implemented support for writing scripts instead of directly writing (pre-)proof trees and definition entries. Formally, a script is a list of commands of the following types (the script language can be of course extended to support additional operations):

– Adding a new definition. (Note that redefining abstractions is prohibited.)
– Starting a new proof.
– Updating an incomplete proof at a given address in the tree. If the address points to a hole, then the hole is updated according to the proof step, i.e., replaced by the rule instance corresponding to the proof step.
– Completing a proof by: (1) verifying that there are no more holes in it, (2) building its extract, and (3) committing the proof to the library, i.e., moving it from the soft part of the library to its rigid part.
– Renaming a user definition throughout the library as discussed in Sec. 4.2.

Next we provide some details about the way proof extracts are handled in our framework. As mentioned in Sec. 4.1.1, Nuprl provides a way to extract programs from proof trees, such that if p is a proof that T is true, then the extract e extracted from p is guaranteed to be a member of T. This allows us to build an abstraction in the library for each proven lemma that unfolds to the extract of the proof. The abstraction is simply the name of the proof. When building the extract of a proof p, if we reach a leaf that corresponds to an already proven lemma p', which is stored in our library, we simply return the corresponding abstraction that unfolds to the extract of p'.

5.3 Validity

We say that a rigid library is valid if all its entries are valid w.r.t. previous entries. A soft library is said to be valid if (1) its rigid part is valid, and (2) its incomplete part is valid w.r.t. the rigid part.

The validity predicate on rigid libraries is recursively defined so that the empty rigid library is valid, and an extension by a new entry is valid only when the entry itself is valid w.r.t. the proof context that consists of the rest of the library. A definition entry is valid w.r.t. a proof context if it does not shadow, i.e. redefines, any definition from its context. A proof entry is valid w.r.t. a proof context essentially if (a) its name does not shadow any definition from its context (a proof is also considered as a definition that points to the extract of the proof); (b) it is well-formed, meaning that each of its nodes corresponds to the instance of a valid proof rule, most notably the nodes that refer to other proofs have to refer to proofs that are in the context (in which case we know the root sequent of the proof is true). The validity predicate on soft libraries is defined similarly, with the difference that holes are considered as valid proof steps.

Using the monotonicity property described in Sec. 4.1, we have proven that all the actions of a script preserve the validity of the library to which they are applied. Given the fact that the empty library is clearly valid, this entails that applying any script results in a valid library. Consequently, our formalization can produce certificates of validity for libraries constructed in this manner.

6 Proof Checker

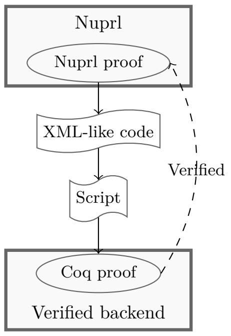
A first step towards building a verified version of Nuprl was taken in [8], where Nuprl's type theory was formalized in Coq. The second step, which we discuss here, is to formally define the concept of a library and implement functionalities to manipulate it. This is essential as Nuprl proofs contain computation steps that can unfold user definitions, but also to ensure that any operation on the library yields a valid library. The formal treatment of libraries now provides us with the guarantee that standard operations on the library preserve its validity.

The next step is to provide a convenient frontend user interface to this backend implementation in order to make this verified version of Nuprl usable. This could either be done by building a new interface to our formalization, or it could be done by using the current version of Nuprl as the frontend.

Regarding the first approach, in addition to building a convenient visual interface, one needs to at least provide support for tactics. This is left for future work. However, we illustrate in `https://github.com/vrahli/NuprlInCoq/blob/master/rules/proof_with_lib_non_dep_example1.v` how our verified theorem prover backend can be used to conduct simple proofs.

Let us now turn to the second approach by demonstrating how to use the Coq formalization to automatically verify Nuprl proofs (illustrated in the figure to the right). A Nuprl proof is a tree of inference rule instances. The Nuprl system already provides functionalities to export such proofs to an XML-like format. We have written an OCaml program that takes such an XML-like file, and converts it into a script as described in Sec. 5.2. This script can then be evaluated using Coq in order to build the corresponding proof within the Coq framework. See `https://github.com/vrahli/NuprlInCoq/blob/master/nuprl_checker/README.md` for details.

This OCaml program could also be used in the first approach mentioned as a way to transfer the vast amount of knowledge that has been accumulated over the years in Nuprl into this new verified theorem prover. This, on its own, seems to be a worthy large scale project for future work.



The examples we have explored demonstrate that this automation is fesable, but it remains to develop the mechanism in general since not all of Nuprl's rules are currently supported. Also, performance issues should be addressed. For example, the largest example we considered contains 4 abstractions and 355 proof steps. It takes about 15 seconds on a i7-5600U laptop to check the proof in Coq, and about 19 seconds to check the proof and produce a certificate that it is valid. Therefore, we plan to explore ways of improving the running time, e.g. by extracting the Coq code to OCaml.

7 Related Work

While to the best of our knowledge our formalization is the first to provide a usable formally verified theorem prover backend for a dependent type theory, several other logical/programming systems have been implemented and verified using proof assistants. Below we discuss some of these systems.

Coq. Barras [9] formalized Werner's [54] set theoretic semantics for a fragment of Coq, which includes the Peano numbers but excludes all other inductive types, in Coq. Werner's semantics assumes the existence of inaccessible cardinals to give denotations to the predicative universes of Coq as sets. In a previous work, [10], a deep embedding of a fragment of Coq that excludes universes and inductive types was provided, thereby obtaining a certified typechecker for this fragment.

HOL Light. In [25] Harrison 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 (augmented by an axiom that provides a larger universe of sets).

CakeML. Myreen et al. extended the above-mentioned work to build a verified implementation of HOL Light [45; 32; 33] using their verified ML-like programming language – CakeML [34]. CakeML is fully verified in HOL4 down to machine code. In [32] HOL's principles for defining new types and new constants were validated for both a stateless version of HOL, and a stateful version, which was proven correct by translating it into the stateless one. In [33], the correctness of a stateful implementation was directly verified, where sequents depend on a separate "theory". The authors proved a version of monotonicity for theories, which followed from the fact that they do not allow redefinitions. In this work, in contrast, to obtain monotonicity we had to account for a restricted form of redefinitions, namely that of instantiating a previously undefined abstraction.

HOL. In [37], the authored improved on previous works [36; 35] and resolved an open problem by establishing conservativity and meta-safety for HOL and HOL based provers over initial HOL. One challenge there was to deal with the intricate interplay between constant and type definitions. In Nuprl there is no notion of extension by type definition, as those are all a part of the initial system (since Nuprl provides a uniform subset type constructor). Moreover, Nuprl's constant definitions slightly differ from that of HOL due to their open-ended nature. Nonetheless, fully formalizing conservativity and meta-safety for constant definitions in Nuprl is, as noted, left for future work.

Milawa. Myreen and Davis formally proved the soundness of the Milawa theorem prover [44] (an ACL2-like theorem prover) which runs on top of their verified Lisp runtime called Jitawa [43].

Buisse and Dybjer. In [15] a categorical model of an intensional Martin-Löf type theory was partially formalized in Agda. As far as we know, this formalization has not led to a verified theorem prover implementation.

KeYmaera X. The core of the KeYmaera X theorem prover [23] was verified in both Coq and Isabelle in [12]. They verified existing pen-and-paper formalizations and extended the metatheory to include features used in practice. Moreover, based on these formalizations, they developed verified prover kernels inside those two provers, and left generating verified standalone kernel for future work.

First-order logic. From a formal proof of completeness of first-order logic in Isabelle/HOL, Ridge and Margetson derived an algorithm that can be used to test the validity of formulas [52].

8 Future Work

In this paper we implemented the notion of a library, and incorporated it into the formalization of Nuprl in Coq. Nuprl's semantics has been adapted in order to validate two key properties: monotonicity and name-invariance. The former allowed us to prove that defining an undefined abstraction preserves the validity of the library, and the latter is a critical feature in any real system. We also added support to build up libraries by adding new definitions and proving lemmas. This resulted in a verified backend of (a variant of) Nuprl. Finally, we showed how to use this backend as a proof checker for proofs generated by Nuprl.

An overarching goal of this line of work is turning this verified variant of Nuprl into a full fledged theorem prover. This requires extending the Coq implementation in order to handle all of Nuprl (currently it only supports a portion of it).

Another line of work initiated in this paper, but is yet far from completion, is to formalize and verify other desired library properties such as the ones put forward by Allen et al. in their vision for the behavior of formal libraries [4; 39].

Further work is also required in order to turn Nuprl into a completely open-ended system, by allowing for the addition of new computation rules to a library. Currently we have the ability to add new terms by defining undefined abstractions, and new proofs. We would like to also be able to add new primitive terms together with their corresponding computation rules.

Abstractions play an important role in the construction of large-scale software and appear in many programming languages. The formal treatment of libraries and systems that can manipulate undefined abstractions presented here opens the door for investigation of several standard programming paradigms, which is left for further study. This includes: incremental programming [19], design-by-contract programming [41; 42; 38] and parametricity [51].

References

[1] *Agda Wiki*. http://wiki.portal.chalmers.se/agda/pmwiki.php.

[2] Stuart F. Allen. “A Non-Type-Theoretic Definition of Martin-Löf’s Types”. In: *LICS*. IEEE Computer Society, 1987, pp. 215–221.

[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, and Lori Lorigo. *FDL: A Prototype Formal Digital Library*. Tech. rep. 2004-1941. Cornell University, 2002.

[5] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. “Innovations in computational type theory using Nuprl”. In: *J. Applied Logic* 4.4 (2006). http://www.nuprl.org/, pp. 428–469.

[6] Abhishek Anand, Mark Bickford, Robert L. Constable, and Vincent Rahli. “A Type Theory with Partial Equivalence Relations as Types”. Presented at TYPES 2014. 2014.

[7] Abhishek Anand and Vincent Rahli. *Towards a Formally Verified Proof Assistant*. Tech. rep. http://www.nuprl.org/html/Nuprl2Coq/. Cornell University, 2014.

[8] Abhishek Anand and Vincent Rahli. “Towards a Formally Verified Proof Assistant”. In: *ITP 2014*. Vol. 8558. LNCS. Springer, 2014, pp. 27–44.

[9] Bruno Barras. “Sets in Coq, Coq in Sets”. In: *Journal of Formalized Reasoning* 3.1 (2010), pp. 29–48.

[10] Bruno Barras and Benjamin Werner. *Coq in Coq*. Tech. rep. INRIA Rocquencourt, 1997.

[11] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. http://www.labri.fr/perso/casteran/CoqArt. SpringerVerlag, 2004.

[12] Brandon Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp, and André Platzer. “Formally verified differential dynamic logic”. In: *CPP 2017*. ACM, 2017, pp. 208–221.

[13] Ana Bove, Peter Dybjer, and Ulf Norell. “A Brief Overview of Agda - A Functional Language with Dependent Types”. In: *TPHOLs 2009*. Vol. 5674. LNCS. http://wiki.portal.chalmers.se/agda/pmwiki.php. Springer, 2009, pp. 73–78.

[14] Edwin Brady. “IDRIS —: systems programming meets full dependent types”. In: *PLPV 2011*. ACM, 2011, pp. 43–54.

[15] Alexandre Buisse and Peter Dybjer. “Towards Formalizing Categorical Models of Type Theory in Type Theory”. In: *Electr. Notes Theor. Comput. Sci.* 196 (2008), pp. 137–151.

[16] Robert L. Constable. “Constructive Mathematics as a Programming Logic I: Some Principles of Theory”. In: *Fundamentals of Computation Theory*. Vol. 158. LNCS. Springer, 1983, pp. 64–77.

[17] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[18] Robert L. Constable and Scott F. Smith. “Computational Foundations of Basic Recursive Function Theory”. In: *Theoretical Computer Science* 121.1&2 (1993), pp. 89–112.

[19] William Cook and Jens Palsberg. "A Denotational Semantics of Inheritance and Its Correctness". In: *SIGPLAN Not.* 24.10 (Sept. 1989), pp. 433–443.

[20] *The Coq Proof Assistant*. http://coq.inria.fr/.

[21] Karl Crary. "Type-Theoretic Methodology for Practical Programming Languages". PhD thesis. Ithaca, NY: Cornell University, Aug. 1998.

[22] Stuart F. Allen, Robert L. Constable, and Douglas J. Howe. "Reflecting the Open-Ended Computation System of Constructive Type Theory". In: *Logic, Algebra and Computation*. Vol. F79. NATO ASI Series. Springer-Verlag, 1990, pp. 265–280.

[23] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. "KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems". In: *CADE-25*. Vol. 9195. LNCS. Springer, 2015, pp. 527–538.

[24] Murdoch Gabbay and Andrew M. Pitts. "A New Approach to Abstract Syntax Involving Binders". In: *LICS 1999*. IEEE Computer Society, 1999, pp. 214–224.

[25] John Harrison. "Towards Self-verification of HOL Light". In: *IJCAR 2006*. Vol. 4130. LNCS. Springer, 2006, pp. 177–191.

[26] Martin Hofmann. "Extensional concepts in intensional type theory". PhD thesis. University of Edinburgh, 1995.

[27] Douglas J. Howe. "Equality in Lazy Computation Systems". In: *LICS 1989*. IEEE Computer Society, 1989, pp. 198–203.

[28] Douglas J. Howe. "On Computational Open-Endedness in Martin-Löf's Type Theory". In: *LICS '91*. IEEE Computer Society, 1991, pp. 162–172.

[29] Douglas J. Howe. "Semantic Foundations for Embedding HOL in Nuprl". In: *Algebraic Methodology and Software Technology*. Vol. 1101. LNCS. Berlin: Springer-Verlag, 1996, pp. 85–101.

[30] *Idris*. http://www.idris-lang.org/.

[31] Alexei Kopylov. "Type Theoretical Foundations for Data Structures, Classes, and Objects". PhD thesis. Ithaca, NY: Cornell University, 2004.

[32] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. "HOL with Definitions: Semantics, Soundness, and a Verified Implementation". In: *ITP 2014*. Vol. 8558. LNCS. Springer, 2014, pp. 308–324.

[33] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. "Self-Formalisation of Higher-Order Logic - Semantics, Soundness, and a Verified Implementation". In: *J. Autom. Reasoning* 56.3 (2016), pp. 221–259.

[34] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: a verified implementation of ML". In: *POPL'14*. ACM, 2014, pp. 179–192.

[35] Ondrej Kuncar and Andrei Popescu. "A Consistent Foundation for Isabelle/HOL". In: *ITP 2015*. Vol. 9236. LNCS. Springer, 2015, pp. 234–252.

[36] Ondrej Kuncar and Andrei Popescu. "Comprehending Isabelle/HOL's Consistency". In: *ESOP 2017*. Vol. 10201. LNCS. Springer, 2017, pp. 724–749.

[37] Ondrej Kuncar and Andrei Popescu. "Safety and conservativity of definitions in HOL and Isabelle/HOL". In: vol. 2. POPL. 2018, 24:1–24:26.

[38] Rustan Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, Apr. 2010, pp. 348–370.

[39] Lori Lorigo. "Information Management in the Service of Knowledge and Discovery". PhD thesis. Cornell University, 2006.

[40] Martin-Löf. "Constructive Mathematics and Computer Programming". In: *6th International Congress for Logic, Methodology and Philosophy of Science*. Noth-Holland, Amsterdam, 1982, pp. 153–175.

[41] Bertrand Meyer. "Applying "Design by Contract"". In: *IEEE Computer* 25.10 (1992), pp. 40–51.

[42] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

[43] Magnus O. Myreen and Jared Davis. "A Verified Runtime for a Verified Theorem Prover". In: *ITP 2011*. Vol. 6898. LNCS. Springer, 2011, pp. 265–280.

[44] Magnus O. Myreen and Jared Davis. "The Reflective Milawa Theorem Prover Is Sound - (Down to the Machine Code That Runs It)". In: *ITP 2014*. Vol. 8558. LNCS. Springer, 2014, pp. 421–436.

[45] Magnus O. Myreen, Scott Owens, and Ramana Kumar. "Steps towards Verified Implementations of HOL Light". In: *ITP'13*. Vol. 7998. LNCS. Springer, 2013, pp. 490–495.

[46] Aleksey Nogin and Alexei Kopylov. "Formalizing Type Operations Using the "Image" Type Constructor". In: *Electr. Notes Theor. Comput. Sci.* 165 (2006), pp. 121–132.

[47] *Nuprl in Coq*. https://github.com/vrahli/NuprlInCoq.

[48] Andrew M. Pitts. "Nominal Logic: A First Order Theory of Names and Binding". In: *TACS 2001*. Vol. 2215. LNCS. Springer, 2001, pp. 219–242.

[49] Vincent Rahli and Mark Bickford. "A nominal exploration of intuitionism". In: *CPP 2016*. Extended version: http://www.nuprl.org/html/Nuprl2Coq/continuity-long.pdf. ACM, 2016, pp. 130–141.

[50] Vincent Rahli, Mark Bickford, and Abhishek Anand. "Formal Program Optimization in Nuprl Using Computational Equivalence and Partial Types". In: *ITP'13*. Vol. 7998. LNCS. Springer, 2013, pp. 261–278.

[51] John C. Reynolds. "Types, Abstraction and Parametric Polymorphism". In: *IFIP Congress*. 1983, pp. 513–523.

[52] Tom Ridge and James Margetson. "A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic". In: *TPHOLs 2005*. Vol. 3603. LNCS. Springer, 2005, pp. 294–309.

[53] Scott F. Smith. "Partial Objects in Type Theory". PhD thesis. Ithaca, NY: Cornell University, 1989.

[54] Benjamin Werner. "Sets in Types, Types in Sets". In: *TACS*. Vol. 1281. LNCS. Springer, 1997, pp. 530–546.