Towards Checking Proof-Checkers

Robert S. Boyer, Gilles Dowek

Towards Checking Proof-Checkers

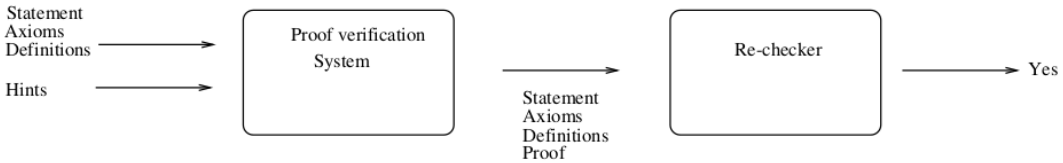
Robert S. Boyer * **Gilles Dowek** †

It is a common practice in software engineering to use one's own tools. For instance, compilers for programming languages are usually written in a compiled programming language. The past thirty years has seen the construction of many *proof development systems*, which in some cases permit one to develop a program together with a proof of its correctness with respect to some specification. The methodology of developing proved programs is intended to become a standard way of programming computers, so proof development systems themselves ought ultimately to be developed using these techniques. A novel issue that arises with proving the correctness of proof development systems is *cross-verification*, i.e., proving the correctness of one system with another. In this way, if one trusts one system, one may trust the other. In this paper we describe an implementation of the Calculus of Constructions [9, 10] in the logic of Nqthm, also known as the Boyer-Moore system [5], and we describe an initial foray into proving theorems about this implementation. Our work is complementary to the deep work described in [3] [2] [13] as our goal is not to prove meta-theoretical properties of the Calculus of Constructions (normalization, confluence, etc.), but rather use these meta-theoretical properties to prove the correctness of an actual implementation of a type-checker, one that can run under any LISP interpretor.

1 Double Checkers and Proof-objects

Less complex by far than a whole proof development system is a *proof-checker*, a program that takes as an input a proposition P and a proof p and checks that the proof p is indeed a proof of P. Formal proofs are typically tedious to write, so proof development systems are often much more complex than mere proof-checkers, and are thus much more difficult to specify and prove correct. Nevertheless, the correctness of a proof is independent of the proof development process used to build it and once obtained such a proof can always be re-checked by a mere proof-checker. Thus, in order to construct a safe proof development system, we do not need to prove the correctness of the entire proof development system but merely that of a proof-checker for the same logic.

We get the following schema:



To following such a schema, we need to have a well-defined proof-language, which is used as output by the proof development system and as input by the proof-checker. For these purposes,

*University of Texas, Austin TX 78712-1188, USA, boyer@cli.com
†CLI, 1717 West Sixth Street, Ste. 290, Austin TX 78703-4776, USA
and INRIA, B.P. 105, 78153 Le Chesnay Cedex, France, Gilles.Dowek@inria.fr

languages based on Heyting semantics and the Curry-Howard isomorphism, such as the Calculus of Constructions, are good candidates.

As we will not necessarily check the correctness of the whole proof development system, but only the proof-checker, we have to make sure that the re-checked proof is indeed a proof of the proposition P. So we have to be able to read the statement (as well as the axioms, definitions, etc.) produced by the proof development system and used by the proof-checker. We need therefore to write a pretty-printer for the intermediate language.

In the investigations discussed in this paper, we have used the system COQ [11] as our proof development system, and we have specified and implemented a proof-checker for the Calculus of Constructions (the logic of COQ) in Nqthm.

2 The Calculus of Constructions

2.1 Basic Formalism

The basic idea underlying systems based on Heyting semantics and the Curry-Howard isomorphism (such as the Calculus of Constructions) is that a proof of a proposition is a functional object. For instance a proof of a proposition of the form $A \Rightarrow B$ is a function that maps every proof of A to a proof of B. The type of this function is isomorphic to the proved proposition, so types and proposition are identified, as are objects and proofs. In order to express all the proofs, the term language has to be an extension of typed lambda-calculus supporting functions from terms to types (dependent types), functions from types to terms (polymorphic types) and functions from types to types (type constructors). A smooth presentation is obtained when we take only one syntactical category for terms and types.

The basic judgement in this formalism is written $\Gamma \vdash t : T$ which is read t *has type* T *in the context* Γ. The context Γ contains the types of the free variables of t and T. An additional judgement, written Γ *is well-formed*, indicates that the context Γ is valid, i.e., the type of each variable x declared in Γ is a term well-typed in the part of the context declared in the left of x.

This calculus is defined in the following way.

Definition: Term

The set of terms is the smallest set such that:

- $Prop$ is a term,
- $Kind$ is a term,
- if x is an identifier then x is a term,
- if t and t' are terms then $(t\ t')$ is a term,
- if t and t' are terms and x is an identifier then $[x : t]t'$ is a term,
- if t and t' are terms and x is an identifier then $(x : t)t'$ is a term.

The terms $Prop$ and $Kind$ are two predefined types, $Prop$ is the type of the types (and of the propositions) and $Kind$ is the type of the term $Prop$. The terms $(t\ t')$ are *applications*, the terms $[x : t]t'$ are λ-*abstractions*, and the terms $(x : t)t'$ are *products*. The product $(x : t)t'$ is a

generalization of the type $t \to t'$. The notation $t \to t'$ is used for $(x : t)t'$ when x does not occur free in t'.

At first in this presentation, we shall ignore variable renaming problems. We subsequently treat this matter precisely, after we introduce the de Bruijn notation for bound variables.

Definition: Context

A *context* Γ is a list of pairs (written $x : T$) where x is an identifier and T a term, i.e., the set of contexts is the smallest set such that:

- $[\,]$ is a context,
- if Γ is a context, x an identifier and T a term then $\Gamma[x : T]$ is a context.

The term T is called the type of the variable x.

Definition: Substitution

Let t and u be terms and x an identifier. The term $t[x \leftarrow u]$ is defined by induction over the structure of t as:

- $Prop[x \leftarrow u] = Prop$,
- $Kind[x \leftarrow u] = Kind$,
- $x[x \leftarrow u] = u$,
- $y[x \leftarrow u] = y$, when y is an identifier different from x,
- $(t\ t')[x \leftarrow u] = (t[x \leftarrow u]\ t'[x \leftarrow u])$,
- $([y : t]\ t')[x \leftarrow u] = ([y : t[x \leftarrow u]]t'[x \leftarrow u])$,
- $((y : t)\ t')[x \leftarrow u] = ((y : t[x \leftarrow u])t'[x \leftarrow u])$.

Note that the definition of the substitution function is effective. This function is simple enough that no non-effective definition seems more intuitive than this one. Let us reiterate that we are ignoring variable renaming problems at this point.

Definition: β-reduction

The β-reduction relation $\rhd$ is the smallest relation such that $([x : T]t\ u) \rhd t[x \leftarrow u]$ and that is reflexive and transitive and is also a congruence with respect to term structure, i.e., the smallest relation such that:

- $([x : T]t\ u) \rhd t[x \leftarrow u]$,
- $t \rhd t$,
- if $t \rhd u$ and $u \rhd v$ then $t \rhd v$,
- if $t \rhd u$ and $t' \rhd u'$ then $(t\ t') \rhd (u\ u')$,
- if $t \rhd u$ and $t' \rhd u'$ then $[x : t]t' \rhd [x : u]u'$,
- if $t \rhd u$ and $t' \rhd u'$ then $(x : t)t' \rhd (x : u)u'$.

Definition: β-convertibility

The relation $\equiv$ is the smallest equivalence relation that contains β-reduction, i.e., the smallest relation such that:

- if $t \rhd u$ then $t \equiv u$,
- if $t \equiv u$ then $u \equiv t$,
- if $t \equiv u$ and $u \equiv v$ then $t \equiv v$.

Definition: Typing rules

The relations Γ *is well-formed* and t *has type* T *in* Γ (for which we use the notation $\Gamma \vdash t : T$) are the smallest relations such that:

- $[\,]$ is well-formed,
- if $\Gamma \vdash T : s$ and $s \in \{Prop, Kind\}$ then $\Gamma[x : T]$ is well-formed,
- if Γ is well-formed then $\Gamma \vdash Prop : Kind$,
- if Γ is well-formed and $x : T \in \Gamma$ then $\Gamma \vdash x : T$,
- if $\Gamma \vdash T : s$, $\Gamma[x : T] \vdash U : s'$, $s \in \{Prop, Kind\}$ and $s' \in \{Prop, Kind\}$ then $\Gamma \vdash (x : T)U : s'$,
- if $\Gamma \vdash T : s1$, $\Gamma[x : T] \vdash U : s2$, $\Gamma[x : T] \vdash t : U$, $s1 \in \{Prop, Kind\}$ and $s2 \in \{Prop, Kind\}$ then $\Gamma \vdash [x : T]t : (x : T)U$,
- if $\Gamma \vdash t : (x : T)U$ and $\Gamma \vdash u : T$ then $\Gamma \vdash (t\ u) : U[x \leftarrow u]$,
- if $\Gamma \vdash t : T$, $\Gamma \vdash U : s$ and $T \equiv U$ and $s \in \{Prop, Kind\}$ then $\Gamma \vdash t : U$.

2.2 Proof-checker

A *proof-checker* is a program that takes a context Γ, and two terms t and T and gives back the boolean value *true* if $\Gamma \vdash t : T$ and *false* otherwise. The claim that a proof-checker exists, is constructively speaking, the claim that the ternary predicate $\Gamma \vdash t : T$ is decidable.

2.3 Example

Let us consider a type T, three elements of type T: a, b, and c, a relation R over the elements of type T, an axiom *trans* stating that the relation R is transitive, and two axioms $ax1$ (resp. $ax2$) stating that the elements a and b (resp. b and c) are related by R, i.e., the context
$\Gamma = [T : Prop; a : T; b : T; c : T; R : T \to T \to Prop;$
$trans : (x : T)(y : T)(z : T)((R\ x\ y) \to (R\ y\ z) \to (R\ x\ z)); ax1 : (R\ a\ b); ax2 : (R\ b\ c)]$
In this context the term $(trans\ a\ b\ c\ ax1\ ax2)$ has type $(R\ a\ c)$ (i.e., the term $(trans\ a\ b\ c\ ax1\ ax2)$ is a proof of the proposition $(R\ a\ c)$).

2.4 Definitions and Lemmas

In mathematical developments, *definitions* are useful; they permit us to associate a name x with a term t of type T and then to use x in place of t. When the term t is seen as a proof, its type, the definition $x := t : T$, is called a *lemma*.

The Calculus of Constructions can be enhanced with definitions and lemmas in the following way.

In the definition of the notion of *context* we add the rule:

- if Γ is a context, x an identifier and t and T terms then $\Gamma[x := t : T]$ is a context.

β-reduction and β-convertibility are replaced by $\beta\delta$-reduction and $\beta\delta$-convertibility. These relations are now parameterized by a context.

Definition: $\beta\delta$-reduction

The $\beta\delta$-reduction relation $\rhd_\Gamma$ is the smallest relation such that:

- $([x : T]t\ u) \rhd_\Gamma t[x \leftarrow u]$,
- if $x := t : T \in \Gamma$ then $x \rhd_\Gamma t$,
- $t \rhd_\Gamma t$,
- if $t \rhd_\Gamma u$ and $u \rhd_\Gamma v$ then $t \rhd_\Gamma v$,
- if $t \rhd_\Gamma u$ and $t' \rhd_\Gamma u'$ then $(t\ t') \rhd_\Gamma (u\ u')$,
- if $t \rhd_\Gamma u$ and $t' \rhd_\Gamma u'$ then $[x : t]t' \rhd_\Gamma [x : u]u'$,
- if $t \rhd_\Gamma u$ and $t' \rhd_\Gamma u'$ then $(x : t)t' \rhd_\Gamma (x : u)u'$.

Definition: $\beta\delta$-convertibility

The $\beta\delta$-convertibility relation $\equiv_\Gamma$ is the smallest equivalence relation that contains $\beta\delta$-reduction, i.e., the smallest relation such that:

- if $t \rhd_\Gamma u$ then $t \equiv_\Gamma u$,
- if $t \equiv_\Gamma u$ then $u \equiv_\Gamma t$,
- if $t \equiv_\Gamma u$ and $u \equiv_\Gamma v$ then $t \equiv_\Gamma v$.

Definition: Typing rules

Finally, we modify the last typing rule in:

- if $\Gamma \vdash t : T$, $\Gamma \vdash U : s$, $T \equiv_\Gamma U$ and $s \in \{Prop, Kind\}$ then $\Gamma \vdash t : U$,

and we add two typing rules

- if $\Gamma \vdash t : T$ then $\Gamma[x := t : T]$ is well-formed,
- if Γ is well-formed and $x := t : T \in \Gamma$ then $\Gamma \vdash x : T$.

In this enhanced formalism, we can add to the context Γ above the lemma $(R\ a\ c)$ and get the well-formed context $\Gamma[lem := (trans\ a\ b\ c\ ax1\ ax2) : (R\ a\ c)]$.

In this formalism a proof-checker can be merely a program that takes a context Γ as input and answers whether this context is well-formed or not; indeed, we have $\Gamma \vdash t : T$ if and only if the context $\Gamma[x := t : T]$ is well-formed.

3 A Cursory Overview of the Logic of Nqthm

For a complete description of the Nqthm logic, we refer the reader to Chapter 4 of [5]. We now make a few vague remarks that we hope will permit those unfamiliar with this logic to read the subsequent formulas written in the logic. The logic of Nqthm is a quantifier-free first order logic with equality. The syntax is Lisp-like. The basic theory includes axioms defining the following:

- the Boolean constants `t` and `f`, corresponding to the true and false truth values.
- equality. `(equal x y)` is `t` or `f` according to whether `x` is equal to `y`.
- an if-then-else function. `(if x y z)` is `z` if `x` is `f` and `y` otherwise.

The logic of Nqthm contains two 'extension' principles under which the user can introduce new concepts into the logic with the guarantee of consistency.

- *The Shell Principle* allows the user to add axioms introducing 'new' inductively defined 'abstract data types.' Natural numbers, ordered pairs, and symbols are axiomatized in the logic by adding shells:
  - *Natural Numbers.* The nonnegative integers are built from the constant `0` by successive applications of the constructor function `add1`. The function `numberp` recognizes natural numbers. The function `sub1` returns the predecessor of a non-`0` natural number.
  - *Symbols.* The data type of symbols, e.g., `'kind`, is built using the primitive constructor `pack` and `0`-terminated lists of ASCII codes. The symbol `'nil`, also abbreviated `nil`, is used to represent the empty list.
  - *Ordered Pairs.* Given two arbitrary objects, the function `cons` builds an ordered pair of these two objects. The function `listp` recognizes ordered pairs. The functions `car` and `cdr` return the first and second component of such an ordered pair. Lists of arbitrary length are constructed with nested pairs. Thus (`list` $arg_1\ \ldots\ arg_n$) is an abbreviation for (`cons` arg_1 ... (`cons` arg_n `nil`)).
- *The Definitional Principle* allows the user to define new functions in the logic. For recursive functions, there must be an ordinal measure of the arguments that can be proved to decrease in each recursion, which, intuitively, guarantees that one and only one function satisfies the definition. Many functions are added as part of the basic theory by this definitional principle.

The rules of inference of the logic are those of propositional logic and equality with the addition of mathematical induction.

Commands to the theorem prover include

- `(dcl` fn `(x y))`, which declares fn to be an undefined function of two arguments.
- `(defn` fn `(x y)` body`)`, which defines the function fn to take two arguments, x and y, and to return body as the value of (fn x y).
- `(add-axiom` name `(types ...)` formula`)`, which adds formula as an axiom, storing it under name and suggesting how best to use the formula in proofs with the hints types ....
- `(prove-lemma` name `(types ...)` formula`)`, which adds formula as a proved lemma after proving it, storing it under name and suggesting how best to use the formula in proofs with the hints types ....

4 Inductive Definitions

4.1 Inductive Definitions

Most of the definitions of section 2 are *inductive definitions*, i.e., definitions of sets (or predicates) of the following form:
A is the smallest subset of B such that, if $x_1, ..., x_{n_1}$ *are element of A then* $f_1(x_1, ..., x_{n_1})$ *is an element of A, ... and if* $x_1, ..., x_{n_p}$ *are elements of A then* $f_p(x_1, ..., x_{n_p})$ *is an element of A.*
The existence of such a set is given by Tarski's fixed-point theorem, by considering the function from $\mathcal{P}(B)$ to $\mathcal{P}(B)$ that associates to the set X the set

$$F(X) = \{f_i(a_1, ..., a_{n_i}) \mid 1 \leq i \leq p \wedge a_1, ..., a_{n_i} \in X\}.$$

This function is obviously increasing for the order $\subset$ in $\mathcal{P}(B)$. The set A is defined as its least fixed point. This least fixed-point is the set

$$A = \bigcap_{X \in C} X$$

Where $C = \{X \in \mathcal{P}(B) \mid \forall 1 \leq j \leq p \; \forall x_1, ..., x_{n_j} \in X(f_j(x_1, ..., x_{n_j}) \in X)\}$.
It is also the case that

$$A = \bigcup_{i \in \mathcal{N}} F^i(\emptyset)$$

Finally, an element a is in A if and only if there exists a sequence $a_1, ..., a_k$ such that $a_k = a$ and for each i there exists an f_j and elements $b_1, ..., b_{n_j}$ of $\{a_1, ..., a_{i-1}\}$ such that $a_i = f_j(b_1, ..., b_{n_j})$ [1].

Even if the functions $f_1, ..., f_p$ are recursive, the defined set A may not be recursive (e.g., the set of theorems of arithmetic can be inductively defined, although it is not recursive), but it is recursively enumerable.

4.2 Inductive Definitions as Specifications

The inductive definition given above is a quite natural specification for the predicate *well-formed*. But, of course, since inductive definitions are not effective, it is not an implementation. (More generally, defining a predicate using quantification over an infinite domain may lead to a clear specification but not necessarily to an obvious implementation.) We want to give an effective definition of a predicate *check* (as a LISP program) and prove that *check* implements the predicate

well-formed, i.e., that these two predicates are extentionally equivalent. More precisely we wish to prove the *soundness* of the implementation:

$$\forall \Gamma \ ((\mathit{check}\ \Gamma) \Rightarrow (\mathit{well\text{-}formed}\ \Gamma))$$

and its *completeness*:

$$\forall \Gamma \ ((\mathit{well\text{-}formed}\ \Gamma) \Rightarrow (\mathit{check}\ \Gamma))$$

4.3 An example of a Specification and a Program

Before we continue with the implementation of the Calculus of Constructions, let us give as an illustration a tiny example of a predicate, defined both as an inductive predicate and as a program.

Definition The predicate *even* is the smallest predicate that contains 0 and $n+2$ if it contains n.

Definition The predicate `ev` is defined by the following algorithm in the Lisp-like logic of Nqthm:

```
(defn ev (x)
 (if (numberp x)
     (if (equal x 0) t (if (equal x 1) f (ev (sub1 (sub1 x)))))
     f))
```

The *soundness* of the implementation is:

$$\forall x \ ((\mathit{ev}\ x) \Rightarrow (\mathit{even}\ x))$$

and the *completeness* is:

$$\forall x \ ((\mathit{even}\ x) \Rightarrow (\mathit{ev}\ x))$$

4.4 Inductive Definitions in a First Order Setting

Inductive definitions in the foregoing style (the least set such that ...) are typically expressed either within a first order logic containing axioms for set theory or in a high-order logic. One might be attempted to express the inductive definition of *even* in the first-order logic of Nqthm thus:

```
(add-axiom even0 () (even 0))
(add-axiom even_plus_two () (implies (even x) (even (add1 (add1 x)))))
```

But these axioms only express that the set of even numbers contains 0 and $n+2$ if it contains n (positive part of the definition), but not the fact that it is the smallest set verifying these properties (negative part). So, for instance the statement *(not (even 1))* is not provable from these axioms. Roughly speaking the positive part of an inductive definition (the fact that the defined set verifies the given properties) is useful to prove the soundness of an implementation and the negative part (the fact that it is the smallest among these sets) is useful to prove its completeness. For instance with the two axioms above we can prove the soundness of the implementation:

$$\forall x \ ((\mathit{ev}\ x) \Rightarrow (\mathit{even}\ x))$$

but not its completeness:

$$\forall x \ ((\mathit{even}\ x) \Rightarrow (\mathit{ev}\ x))$$

An approach to handling completeness in the Nqthm logic is to define the predicate `ev` as above and then prove that it verifies the two conditions:

```
(prove-lemma corr ()
     (and (ev 0)
          (implies (ev x) (ev (add1 (add1 x))))))
```

and then that every predicate `fn` that verifies these two properties contains the predicate `ev`. We first declare `fn` to be an undefined function of one argument:

```
(dcl fn (x))

(add-axiom fn-prop ()
   (and (fn 0) (implies (fn x) (fn (add1 (add1 x))))))

(prove-lemma comp () (implies (ev x) (fn x)))
```

The second order quantification *every predicate* `fn` is simulated in this first order setting by extending the language with a new predicate symbol `fn`. Although it is very powerful, this technique sometimes leads to surprisingly long and uncomfortable proofs.

Another solution is to add as axioms more properties of the specified predicate. For instance, for the definition of `even` one can add the axioms:

```
(add-axiom even1 () (not (even 1)))
(add-axiom even_minus_two () (implies (even (add1 (add1 x))) (even x)))
```

Alternatively one can add the axiom:

```
(add-axiom even_inv () (implies (even x) (or (equal x 0)
                                             (and (leq 2 x)
                                                  (even (sub1 (sub1 x))))))))
```

The generality of this approach (i.e., the possibility of expressing the negative part of an inductive definition by a first order statement) is not yet understood by us [1].

In this paper we have only proved the soundness of our implementation (and not its completeness), so we have merely stated as axioms the positive part of the inductive definitions. Nevertheless, although we have only worked on the soundness half, we have failed to prove three lemmas presented below, which are useful in the soundness proof but which require the negative part of the inductive definition.

[1] Note that Clark's completion axiom [8] expresses the negative part of an inductive definition in some cases, but not in general. Indeed the axioms
$(even\ 0)$
$\forall x((even\ x) \Rightarrow (even\ (S\ (S\ x))))$
$\forall x((even\ x) \Rightarrow ((x = 0) \vee \exists y((even\ y) \wedge (x = (S\ (S\ y))))))$ (completion axiom)
characterize a unique set.

But let us define the empty set E as the smallest set such that if x is in E then x is in E. In this case, the axioms
$\forall x((E\ x) \Rightarrow (E\ x))$
$\forall x((E\ x) \Rightarrow \exists y((E\ y) \wedge (x = y)))$ (completion axiom)
do not characterize a unique set.

5 Normalization

The kernel of the implementation of our proof-checker is the normalization function that permits us to decide if two terms are convertible. In the recursive fragment of the Nqthm logic, only total functions can be defined, so one would need to prove the normalization of reduction to define it. The following problems arise: (1) the reduction function does not terminate on all terms, but only on well-typed terms and it is not possible in the Nqthm logic to define a function restricted in such a way as to be applicable only to some terms, (2) even on typed terms, the ordinal of the normalization function is far above ε_0 and therefore this function cannot be defined in the recursive fragment of the Nqthm logic. It is possible to define any partial recursive function in the Nqthm logic via the function `EVAL$`, an interpreter for the partial recursive functions. However, reasoning about functions defined via `EVAL$` within Nqthm is much more difficult than reasoning about recursively defined functions.

The current solution we have taken in our implementation is to give a bound c to the normalization function, such that it stops after c reduction steps. This way, we get weaker soundness and completeness statements.
Soundness:

$$\forall \Gamma\ (\exists c\ (\mathit{check}\ \Gamma\ c) \Rightarrow (\mathit{well\text{-}formed}\ \Gamma))$$

Completeness:

$$\forall \Gamma\ ((\mathit{well\text{-}formed}\ \Gamma) \Rightarrow (\exists c\ \mathit{check}\ \Gamma\ c))$$

Such a parameter as c above, sometimes called a 'clock' parameter, is frequently employed by users of Nqthm in the verification of computing systems, giving semantics to a system by first defining a 'single-stepper' and defining then a function that runs the single-stepper a given number of steps. See, for example, [6].

6 λ-calculus with Nameless Dummies

In the presentation of the calculus above, we have ignored variable renaming problems. These problems cannot be ignored in a formal specification and implementation. We have therefore used the notion of terms with nameless dummies introduces by de Bruijn [7]. In these terms, variables have no names and each occurrence of a variable is represented by a positive integer: the relative depth of its binder.

For instance the term $[T : \mathit{Prop}][f : (T \to T) \to T](f\ [y : T](f\ [x : T]y))$ is expressed by $[\mathit{Prop}][(1 \to 2) \to 2](1\ [2](2\ [3]2))$, so is the term $[U : \mathit{Prop}][g : (U \to U) \to U](g\ [a : U](g\ [b : U]a))$.

The definitions above are transformed into:

Definition: Term

The set of term is the smallest set such that:

- Prop is a term,
- Kind is a term,
- if n is a positive integer then n is a term,
- if t and t' are terms then $(t\ t')$ is a term,

- if t and t' are terms then $[t]t'$ is a term,
- if t and t' are terms then $(t)t'$ is a term.

Definition: Context

A *context* Γ is a list of which the elements are either terms (variable declarations) or pairs of terms (constant declarations), i.e., the set of contexts is the smallest set such that:

- $[\,]$ is a context,
- if Γ is a context and T a term then $\Gamma[T]$ is a context,
- if Γ is a context and t and T are terms then $\Gamma[t:T]$ is a context.

Definition: Substitution (and lift by one while you are at it)

- $Prop[n \leftarrow u] = Prop$,
- $Kind[n \leftarrow u] = Kind$,
- $n[n \leftarrow u] = \uparrow_1^{n-1} u$,
- $p[n \leftarrow u] = p$ (if $p < n$),
- $p[n \leftarrow u] = p - 1$ (if $n < p$),
- $(t\ t')[n \leftarrow u] = (t[n \leftarrow u]\ t'[n \leftarrow u])$,
- $[t]t'[n \leftarrow u] = [t[n \leftarrow u]]t'[n+1 \leftarrow u]$,
- $(t)t'[n \leftarrow u] = (t[n \leftarrow u])t'[n+1 \leftarrow u]$,

where the lifting function $\uparrow_n^k$ is defined by:

- $\uparrow_n^k Prop = Prop$,
- $\uparrow_n^k Kind = Kind$,
- $\uparrow_n^k p = p$ (if $p < n$),
- $\uparrow_n^k p = p + k$ (if $p \geq n$),
- $\uparrow_n^k (t\ t') = (\uparrow_n^k t\ \ \uparrow_n^k t')$,
- $\uparrow_n^k [t]t' = [\uparrow_n^k t]\ \uparrow_{n+1}^k t'$,
- $\uparrow_n^k (t)t' = (\uparrow_n^k t)\ \uparrow_{n+1}^k t'$.

Note that introducing de Bruijn indices makes the definition of substitution less intuitive since we need to use the lifting operator $\uparrow_n^k$. We have here an example in which the specification is as tricky as a program, and we could fail to express it correctly. An interesting problem is to find a specification of the substitution function with de Bruijn indices as intuitive as the one with explicit names.

Definition: $\beta\delta$-reduction

- $([T]t\ u) \rhd_\Gamma t[1 \leftarrow u]$
- if $t : T$ is the n^{th} element of Γ then $n \rhd_\Gamma \uparrow_1^n t$,
- $t \rhd_\Gamma t$,
- if $t \rhd_\Gamma u$ and $u \rhd_\Gamma v$ then $t \rhd_\Gamma v$,
- if $t \rhd_\Gamma u$ and $t' \rhd_\Gamma u'$ then $(t\ t') \rhd_\Gamma (u\ u')$,
- if $t \rhd_\Gamma u$ and $t' \rhd_\Gamma u'$ then $[t]t' \rhd_\Gamma [u]u'$,
- if $t \rhd_\Gamma u$ and $t' \rhd_\Gamma u'$ then $(t)t' \rhd_\Gamma (u)u'$.

Definition: $\beta\delta$-convertibility

- if $t \rhd_\Gamma u$ then $t \equiv_\Gamma u$,
- if $t \equiv_\Gamma u$ then $u \equiv_\Gamma t$,
- if $t \equiv_\Gamma u$ and $u \equiv_\Gamma v$ then $t \equiv_\Gamma v$.

Definition: Typing rules

- [ ] is well-formed,
- if $\Gamma \vdash T : s$ and $s \in \{Prop, Kind\}$ then $\Gamma[T]$ is well-formed,
- if $\Gamma \vdash t : T$ then $\Gamma[t : T]$ is well-formed,
- if Γ is well-formed then $\Gamma \vdash Prop : Kind$,
- if Γ is well-formed and T is the n^{th} element of Γ then $\Gamma \vdash n :\uparrow_1^n T$,
- if Γ is well-formed and $t : T$ is the n^{th} element of Γ then $\Gamma \vdash n :\uparrow_1^n T$,
- if $\Gamma \vdash T : s$, $\Gamma[T] \vdash U : s'$, $s \in \{Prop, Kind\}$ and $s' \in \{Prop, Kind\}$ then $\Gamma \vdash (T)U : s'$,
- if $\Gamma \vdash T : s1$, $\Gamma[T] \vdash U : s2$, $\Gamma[T] \vdash t : U$, $s1 \in \{Prop, Kind\}$ and $s2 \in \{Prop, Kind\}$ then $\Gamma \vdash [T]t : (T)U$,
- if $\Gamma \vdash t : (T)U$ and $\Gamma \vdash u : T$ then $\Gamma \vdash (t\ u) : U[1 \leftarrow u]$,
- if $\Gamma \vdash t : T$, $\Gamma \vdash U : s$ and $T \equiv_\Gamma U$ and $s \in \{Prop, Kind\}$ then $\Gamma \vdash t : U$.

7 Formal Specification

We are now ready to give the (positive part of the) formal specification of the function *well-formed*. Here is an informal sketch of the mapping from the syntax of the Calculus of Constructions described above to the corresponding objects in the Nqthm logic.

- *Prop* and *Kind* are represented as the symbols `'prop` and `'kind`.
- The variable n (i.e., the positive integer n, a de Bruijn index) is represented as itself.
- $(t1\ t2)$ is represented as `(list 'apply t1 t2)`.
- $[t1]t2$ is represented as `(list 'lambda t1 t2)`.
- $(t1)t2$ is represented as `(list 'product t1 t2)`.
- A context is represented as a list. The empty context is represented as `nil`. Definitional elements of a context are represented with `(list 'constant ty te)` and other elements are represented with `(list 'variable ty)`.

We start with the lifting function $\uparrow_n^k$ and the substitution function.

```
(defn lift (n k c)
  (cond ((numberp c) (if (lessp c n) c (plus c k)))
        ((listp c)
         (list (car c)
               (lift n k (cadr c))
               (lift (if (equal (car c) 'apply) n (add1 n)) k (caddr c))))
        (t c)))

(defn subst (d n c)
  (cond ((numberp c) (cond ((equal c n) (lift 1 (sub1 n) d))
                           ((lessp c n) c)
                           (t (sub1 c))))
        ((listp c)
         (list (car c)
               (subst d n (cadr c))
               (subst d (if (equal (car c) 'apply) n (add1 n)) (caddr c))))
        (t c)))
```

We then axiomatize the (positive part of the) reduction and convertibility relations. We first declare the function `red` to be a function of three arguments. `(red env t1 t2)` expresses that `t1` reduces to `t2` in environment `env`.

```
(dcl red (env t1 t2))

(add-axiom red-beta (rewrite)
  (red env (list 'apply (list 'lambda u1 u2) u3) (subst u3 1 u2)))

(add-axiom red-delta (rewrite)
  (implies (equal (nth env n) (list 'constant ty te))
           (red env n (lift 1 n te))))
```

```
(add-axiom red-lambda (rewrite)
  (implies (and (red env t1 u1) (red (cons (list 'variable t1) env) t2 u2))
           (red env (list 'lambda t1 t2) (list 'lambda u1 u2))))

(add-axiom red-product (rewrite)
   (implies (and (red env t1 u1)
                 (red (cons (list 'variable t1) env) t2 u2))
            (red env (list 'product t1 t2) (list 'product u1 u2))))

(add-axiom red-apply (rewrite)
  (implies (and (red env t1 u1) (red env t2 u2))
           (red env (list 'apply t1 t2) (list 'apply u1 u2))))

(add-axiom red-refl (rewrite)
  (red env t1 t1))

(add-axiom red-trans (rewrite)
  (implies (and (red env t1 t2) (red env t2 t3)) (red env t1 t3)))
```

`(equiv en t1 t2)` expresses that `t1` is convertible to `t2` in environment `env`.

```
(dcl equiv (env t1 t2))

(add-axiom equiv-red (rewrite)
  (implies (red env t1 t2) (equiv env t1 t2)))

(add-axiom equiv-sym (rewrite)
  (implies (equiv env t1 t2) (equiv env t2 t1)))

(add-axiom equiv-trans (rewrite)
  (implies (and (equiv env t1 t2) (equiv env t2 t3))
           (equiv env t1 t3)))
```

At last we axiomatize the (positive part of the) typing predicate and the well-formedness predicate.
`(types env te ty)` expresses that `te` has type `ty` in environment `env`.
`(well-formed env)` expresses that the environment `env` is well-formed.

```
(dcl types (env term type))

(dcl well-formed (env))

(add-axiom empty (rewrite)
  (well-formed nil))

(add-axiom declaration (rewrite)
  (implies (and (well-formed env) (member s '(prop kind)) (types env ty s))
           (well-formed (cons (list 'variable ty) env))))

(add-axiom sort (rewrite)
  (implies (well-formed env) (types env 'prop 'kind)))
```

```
(add-axiom variable (rewrite)
  (implies (and (well-formed env) (equal (nth env n) (list 'variable ty)))
           (types env n (lift 1 n ty))))

(add-axiom product (rewrite)
  (implies (and (member s1 '(prop kind))
                (member s2 '(prop kind))
                (types env ty1 s1)
                (types (cons (list 'variable ty1) env) ty2 s2))
           (types env (list 'product ty1 ty2) s2)))

(add-axiom abstraction (rewrite)
  (implies (and (member s1 '(prop kind))
                (member s2 '(prop kind))
                (types env ty s1)
                (types (cons (list 'variable ty) env) ty2 s2)
                (types (cons (list 'variable ty) env) t2 ty2))
           (types env (list 'lambda ty t2) (list 'product ty ty2))))

(add-axiom application (rewrite)
  (implies (and (types env t1 (list 'product u1 u2)) (types env t2 u1))
           (types env (list 'apply t1 t2) (subst t2 1 u2))))

(add-axiom conversion (rewrite)
  (implies (and (types env te ty1) (types env ty2 s) (equiv env ty1 ty2))
           (types env te ty2)))

(add-axiom constdecl (rewrite)
  (implies (types env te ty)
           (well-formed (cons (list 'constant ty te) env))))

(add-axiom constant (rewrite)
  (implies (and (well-formed env) (equal (nth env n) (list 'constant ty te)))
           (types env n (lift 1 n ty))))
```

8 Program

We start the implementation with the normalization function.

```
(defn apply-list (x l)
  (if (nlistp l) x (apply-list (list 'apply x (car l)) (cdr l))))

(defn normal (flg env term stack clock)
  (cond
   ((equal flg 'list)
    (if (nlistp term)
        nil
      (cons (normal t env (car term) nil clock)
            (normal 'list env (cdr term) nil clock))))
   (t (cond
       ((zerop clock) (apply-list term stack))
```

```
      ((listp term)
       (let ((op (car term)) (c1 (cadr term)) (c2 (caddr term)))
         (cond
          ((equal op 'apply)
           (normal t env c1 (cons c2 stack) clock))
          ((equal op 'product)
           (apply-list
            (list op
                  (normal t env c1 nil clock)
                  (normal t (cons (list 'variable c1) env) c2 nil clock))
            stack))
          ((equal op 'lambda)
           (if (nlistp stack)
               (list op
                     (normal t env c1 nil clock)
                     (normal t (cons (list 'variable c1) env) c2 nil
                             clock))
             (normal t env (subst (car stack) 1 c2) (cdr stack)
                     (sub1 clock))))
          (t f))))
      ((numberp term)
       (let ((val (nth env term)))
         (if (and val (equal (car val) 'constant))
             (normal t env (lift 1 term (caddr val)) stack (sub1 clock))
           (apply-list term (normal 'list env stack nil (sub1 clock))))))
      ((equal term 'prop) (apply-list 'prop stack))
      ((equal term 'kind) (apply-list 'kind stack))
      (t f))))
  ((ord-lessp (cons (add1 clock) (count term)))))
```

Then we implement the function that computes a type of a term.

```
(defn check-term (env term clock)
  (cond
   ((numberp term)
    (let ((val (nth env term)))
      (if val (lift 1 term (cadr val)) f)))
   ((equal term 'prop) 'kind)
   ((or (nlistp term) (not (equal nil (cdddr term)))) f)
   (t (let ((op (car term)) (c1 (cadr term)) (c2 (caddr term)))
        (cond
         ((equal op 'apply)
          (let ((typ1 (check-term env c1 clock))
                (typ2 (check-term env c2 clock)))
            (if (and typ1 typ2)
                (let ((ntyp1 (normal t env typ1 nil clock))
                      (ntyp2 (normal t env typ2 nil clock)))
                  (if (and (equal 'product (car ntyp1))
                           (equal ntyp2 (cadr ntyp1)))
                      (subst c2 1 (caddr ntyp1))
                    f))
              f)))
```

```
        ((equal op 'lambda)
         (let ((typ1 (check-term env c1 clock))
               (typ2 (check-term (cons (list 'variable c1) env) c2 clock)))
           (if (and (member typ1 '(prop kind))
                    typ2
                    (not (equal typ2 'kind)))
               (list 'product c1 typ2)
             f)))
        ((equal op 'product)
         (let ((typ1 (check-term env c1 clock))
               (typ2 (check-term (cons (list 'variable c1) env) c2 clock)))
           (if (and (member typ1 '(prop kind)) (member typ2 '(prop kind)))
               typ2
             f)))
        (t f))))))
```

At last, here is the function that checks that a context is well-formed.

```
(defn check-item (env item clock)
  (let ((nature (car item)))
    (if (equal nature 'constant)
        (let ((typ (cadr item)) (val (caddr item)))
          (let ((ty2 (check-term env val clock)))
            (if (and (equal nil (cdddr item))
                     (check-term env typ clock)
                     ty2
                     (equal (normal t env ty2 nil clock)
                            (normal t env typ nil clock)))
                (cons item env)
              f)))
      (if (equal nature 'variable)
          (let ((typ (cadr item)))
            (if (and (equal nil (cddr item))
                     (member (check-term env typ clock) '(prop kind)))
                (cons item env)
              f))
        f))))

(defn check (env clock)
(if (nlistp env)
    (equal 'nil env)
    (and (check (cdr env) clock) (check-item (cdr env) (car env) clock))))
```

This implementation is very influenced by [12].

9 Unproved Lemmas

We failed to prove three lemmas that are needed to prove soundness and that require the negative part of the inductive definitions.

```
(add-axiom subject-reduction (rewrite)
```

```
  (implies (and (types env t1 ty) (red env t1 t2))
           (types env t2 ty)))

(add-axiom type-type (rewrite)
  (implies (types env te ty)
           (or (equal ty 'kind) (types env ty 'prop) (types env ty 'kind))))

(add-axiom red-kind (rewrite)
  (equal (red env 'kind te) (equal te 'kind)))
```

10 Soundness

Given the foregoing axioms, definitions, and unproved lemmas, we can now check with Nqthm our soundness result:

```
(implies (check l clock) (well-formed l))
```

Events suitable for driving Nqthm to check this result are given in [4].

11 An Example

Using this system, we have checked a proof of Tarski's fixed point theorem. The object was obtained by using the system COQ. Nqthm can prove (in fact simply by running code for `check`) that `check` returns non-`F` on this formal proof object, given a clock argument of 7, and thus by the theorem above, it follows that that the proof object is well-formed.

12 Proving The Soundness of a Enhanced Conversion Function

When we want to apply a function f of type $A \to B$ to a value of type A' we need to check that the terms A and A' are convertible. In the program above we normalize the terms A and A' and we check that their normal forms are equal. When $A = F(a)$ and $A' = F(a')$ where F is a constant declared in the context $F := t$ and a and a' are convertible terms, we normalize the terms $(t\ a)$ and $(t\ a')$ and check that the normal forms are equal. In [12] Huet has proposed a more evolved method, in which when we get the same constant as head-symbol of the terms A and A' we first try to check that the arguments of this function are pairwise convertible and only when this test fails expand the constants. Keeping the specification the same, we have proved the soundness of another program using this method.

Conclusions and Speculations

Thus far in our investigation, we have formally specified and implemented in the Nqthm logic a proof-checker for the Calculus of Constructions and we have proved the soundness of the implementation, assuming three unproved lemmas.

Several problems remain unsolved.

- Our specification of substitution is not very intuitive, since we define it through the lifting operator $\uparrow_n^k$.
- We need to find a way to express the negative part of the inductive definitions to be able to complete the proofs of the three admitted lemmas and to prove the completeness of our implementation.
- We need to understand how to define the normalization function without a 'clock' argument, for which we need to be able to express a function that terminates only when its arguments belong to some class (here, well-typed terms) and prove that in our definition this function is only used with such arguments. Because the class in question is characterized by the function we are trying to define, this problem is nontrivial for Nqthm.
- We need to understand how to strengthen Nqthm in order to be able to prove the termination of this unclocked normalization function, something non-trivial since, by Gödel's second incompleteness theorem, it can be proved neither in the Calculus of Constructions nor in weaker systems, such as the recursive fragment of the Nqthm logic. For example, given a concrete representation for a suitably large ordinal and given a primitive recursive 'less-than' relation on this representation, we could easily 'wire' the Nqthm system to permit induction up to that ordinal. However, we do not currently know of such an ordinal and representation.
- As an alternative to attempting to formalize an unclocked-version of the normalization function, we might instead reformulate the *well-formed* predicate to take both an environment and a finite sequence of reduction operations to perform. That is, we could retreat from defining a 'decision procedure' for well-formedness to being satisfied with merely defining a proof-checker for well-formedness.
- As another alternative, we could define `check` via `EVAL$`, an interpreter for the partial recursive functions.

References

[1] P. Aczel, An introduction to Inductive Definitions, *Handbook of Mathematical Logic*, J. Barwise (Ed.), North-Holland, 1977, pp. 739-782.

[2] Th. Altenkirch, A Formalization of the strong normalization proof for system F in LEGO, *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science 664, 1993, pp. 13-28.

[3] S. Berardi, Girard Normalization Proof in LEGO, Informal proceedings of the first workshop on logical frameworks, Antibes, 1990, pp. 65-75.

[4] R. Boyer, G. Dowek, Towards Checking Proof-Checkers, Manuscript, 1992.

[5] R. Boyer, J. Moore, A Computational Logic Handbook, Academic Press, 1988.

[6] R. Boyer, Y. Yu, Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor, *Automated Deduction – CADE-11, Lecture Notes in Computer Science 607*, D. Kapur (Ed.), Springer-Verlag, 1992, pp. 416-430.

[7] N.G. de Bruijn, Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem, *Indagationes Mathematicae*, 34, 5, 1972, pp. 381-392.

[8] K. L. Clark, Negation as Failure, *Logic and Data Bases*, H. Gallaire and J. Minker (Eds.), Plenum Press, New York, 1978, pp. 293-322.

[9] Th. Coquand, Une Théorie des Constructions, *Thèse de troisième cycle*, Université Paris VII, 1985.

[10] Th. Coquand, G. Huet, The Calculus of Constructions, *Information and Computation*, 76, 1988, pp. 95-120.

[11] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, Ch. Paulin-Mohring, B. Werner, The Coq Proof Assistant User's Guide, V5.8, INRIA-Rocquencourt, ENS-Lyon, 1993.

[12] G. Huet, The Constructive Engine, *A Perspective in Theoretical Computer Science*, Commemorative Volume for Gift Siromoney, R. Narasimhan (Ed.), World Scientific Publishing, 1989.

[13] J. McKinna, R. Pollack, Pure Type Systems Formalized, *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science 664, 1993, pp. 289-305.