Proof by Computation in the Coq system

Martijn Oostdijk and Herman Geuvers

Computer Science Department, Eindhoven University of Technology

Abstract

In informal mathematics, statements involving computations are seldom proved. Instead, it is assumed that readers of the proof can carry out the computations on their own. However, when using an automated proof development system based on type theory, the user is forced to find proofs for all claimed propositions, including computational statements.

This paper presents a method to automatically prove statements from primitive recursive arithmetic. The method replaces logical formulas by boolean expressions. A correctness proof is constructed, which states that the original formula is derivable, if and only if the boolean expression equals `true`. Because the boolean expression reduces to `true`, the conversion rule yields a trivial proof of the equality. By combining this proof with the correctness proof, we get a proof for the original statement.

1 Introduction

This paper presents a method to automatically prove statements from first order primitive recursive arithmetic, in the context of type theoretical proof systems [1]. This is done by replacing proof obligations by computations. For example, the proposition Prime(61) can be verified by a computer program which checks all potential divisors of 61. By doing these computations, it can be seen that there are no proper divisors of 61. From this, it is concluded that 61 is prime.

In informal mathematical proofs, propositions like Prime(61) are seldom proved. They are not considered to be "mathematically interesting" and verification is normally left to the reader. However, when constructing formal proofs using an automated proof system based on type theory, such as the Coq *proof assistant* [9], the user is forced to find proofs for all claimed propositions, including propositions like Prime(61). The ability to prove these propositions automatically, allows users of these systems to concentrate on formalizing the important, mathematically interesting parts of a theory.

Preprint submitted to Elsevier Preprint *14 April 2000*

The method presented here is based on two main ideas. The first idea, called *computational reflection* in [7] (dating back to original work by [8], who calls it *reflection*) or *two level approach* in [3] is to interpret a class of propositions on three different levels: a syntactical level, a propositional level, and a computational level. The syntactical level makes it possible to relate the computational level to the propositional level by proving that a decision algorithm (on the computational level) indeed has the intended effect (on the propositional level). The second idea, called *Poincaré's principle* in [2], states that propositions which can be verified by a computation are easy; i.e., no proof is required. This principle is incorporated in Coq through the so-called conversion rule: types that are computationally equal (convertible) are not distinguished. The Poincaré's principle is crucial for the use of computational reflection in theorem provers, as it allows to replace a large proof-object (laborious to generate) by a small proof-object plus a computation (mechanical).

In our case the combination of these two ideas allows us to replace a proposition from primitive recursive arithmetic (the propositional level) with a computation (the computational level) involving characteristic functions of primitive recursive predicates. The latter can be resolved using the conversion rule. Proving that this replacement is indeed allowed, involves lifting the original proposition to the syntactic level and translating it to the computational and propositional levels. It is proved that these two translations conform with each other: the translation to the computational level evaluates to `true` if and only if the translation to the propositional level is provable.

The intention of the paper is to present the result without assuming detailed knowledge of type theory or proof-assistants based on type theory. To meet that condition, the paper is organized as follows. First we give a general introduction to proof-assistants based on type theory, briefly discussing the philosophy and the technology. Then we introduce a type theory for higher order predicate logic and we show by example how mathematical reasoning may be formalized in this system. We extend this system with (a restricted form of) inductive types. The system we thus obtain is a subsystem of the type theory that is implemented in the proof-assistant Coq. In the last section we show how we have defined a decision procedure for primitive recursive arithmetic inside Coq.

2 Proof Assistants based on type theory

In type theory one interprets formulas and proofs via the well-known 'formulas-as-types' and 'proofs-as-terms' embedding, originally due to Curry, Howard and De Bruijn. Under this interpretation, a formula is viewed as the type of

its proofs. Hence, a statement in type theory of the form

$$M : A$$

can be read in two ways:

- M is an *element of the set* denoted by A,
- M is a *proof of the formula* denoted by A.

In the case that M denotes a proof, one can (in general) really construct a natural deduction style derivation out of the proof term M. Whether this is possible depends on the specific type theory, but for many well-known logics an *isomorphic* typed λ-calculus has been defined: there is a bijection between natural deductions in the logic and proof terms in the typed λ-calculus. We shall illustrate this correspondence between logic and typed λ-calculus later by some examples. The main consequences of this approach towards theorem proving are that

- Proof checking is Type checking,
- Interactive Theorem Proving is the interactive construction of a term of a given type.

The Proof Assistant Coq is an interactive theorem prover based on type theory: the implemented typed λ-calculus is a version of constructive higher order logic with powerful inductive types. The system Coq provides the user with powerful tactics to interactively construct a proof term. In this construction process, the system guarantees the type correctness. An important distinction to be made – which is a basic philosophy behind type theoretic provers like Coq – is the one between

- Checking an alleged proof: this is easy, comparable with checking the *syntactic correctness* of a computer program,
- Constructing a proof for a given formula: this is hard (undecidable in general), comparable with constructing a program which satisfies a specification.

In type theoretic provers, the first task is performed by a *type checking algorithm*, the second task is performed interactively with the user.

2.1 Correctness of Proof Assistants

An important issue in automated theorem proving in general is the question of *correctness* of the implemented system. Or, phrased differently: how can we be sure that a formula that has been proven by the Proof Assistant (PA) is

really true? We may sometimes not be convinced that all the powerful tactics that a PA provides are sound and it occasionally turns out that a PA contains a bug. In type theoretic PAs, this issue of reliability is solved to some extent, because the PA does not only tell the user that the theorem has been proved, but it also provides a *proof term* that can either be *type checked* by the user (using his own – relatively easy to write – type checking algorithm) or it can be exported to some natural language style proof that can be read by other humans. The feature of having proof terms that can be checked *independently* by a relatively *small* and *easy* algorithm, is also known as the *De Bruijn criterion* (see [2]), named after the founding father of the Automath project. In this project the first PAs based on type theory were implemented (in fact they were proof checkers instead of proof assistants).

So, on the one hand the De Bruijn criterion gives a higher degree of reliability to PAs. On the other hand, however, this criterion makes it harder to implement very powerful proof tactics (like resolution), because the system will always have to construct a complete proof term that can be (type) checked easily in a small underlying system. In this paper we show that it is possible to add powerful proof tactics to Coq and at the same time comply with the De Bruijn criterion. This is done by applying the so called 'two level approach' ([2]), also known as the 'reflection principle' ([7]). The basic idea of that approach is to code a specific syntactic class of formulas into an inductive type form. We write $[\![-]\!]$ for the decoding function giving for every formula $a : \texttt{form}$ a proposition $[\![a]\!]$. A given (powerful) proof procedure can (in the simplest case) then be defined as a function F of type $\texttt{form}{\to}\texttt{form}$. Now, if we can prove this procedure to be correct *inside* Coq, i.e. if we prove $\forall a{:}\texttt{form}([\![a]\!] \leftrightarrow [\![Fa]\!])$, then we can replace a proof obligation $[\![a]\!]$ by a proof obligation $[\![Fa]\!]$ (which will in general be easier).

In this paper we illustrate the method sketched above by looking at the formulas of primitive recursive arithmetic (PRA). We define a function $(\![-]\!)$ (comparable with the F above) that computes true or false for every closed formula of PRA (using a characteristic function) and we prove that $(\![-]\!)$ is correct (i.e. $(\![-]\!)$ preserves derivability: $\forall a{:}\texttt{form}([\![a]\!] \leftrightarrow ((\![a]\!) = \texttt{true})))$. Hence, if we want to check, e.g. whether Prime(61) holds, we have to find a term a of type form such that $[\![a]\!]$ is convertible with Prime(61) and we have to verify that $(\![a]\!) = \texttt{true}$. The latter is done by just computing $(\![a]\!)$: the outcome is either true or false.

3 A type theory for higher order predicate logic with inductive types

In this section we define a part of the type system that is implemented in Coq. We will not attempt to give a general introduction into Coq, but restrict to that part of Coq that is necessary for our proof development. First we introduce the system λPREDω, a type theory in which one can (faithfully) interpret higher order predicate logic. Then we extend this system with inductive types, to obtain the system λPREDω^{ind}.

Before giving the precise definition, we make some introductory remarks to guide the intuition.

(1) The language of higher order predicate logic is a typed language. In λPREDω there are 'first order sets', which are of type `Set` and there are higher order sets, which are of type `Type`. These 'universes' `Set` and `Type` are called *sorts*.
(2) In λPREDω, formulas like $\varphi \supset \psi$ and $\forall x{:}A.\varphi$ will become types. However, these 'propositional' types are not the same as the set types (like e.g. `nat`). Hence there is another 'universe', `Prop`, containing the 'propositional' types. So, all formulas are of type `Prop` in λPREDω.
(3) `Prop` itself is a higher order set type, so `Prop` : `Type`.
(4) For A a first order set (i.e. A : `Set`), the set of predicates on A is represented as $A \rightarrow$`Prop`, the type of functions from A to `Prop`. If $P : A \rightarrow$`Prop` and $a : A$, then Pa : `Prop`. the intended meaning is that 'a belongs to P' if the formula Pa can be proved.
(5) Natural deductions are represented as typed λ-terms. The discharging of hypotheses is done by λ-abstraction. The modus ponens rule is interpreted via application.
(6) A formula is provable if we can find a proof of it. That is in λPREDω, if φ : `Prop`, then 'φ is provable' if we can find a term M such that $M : \varphi$.

The *derivable judgements* of λPREDω are of the form

$$\Gamma \vdash M : A,$$

where Γ is a *context* and M and A are terms. A context is of the form $x_1{:}A_1, \ldots, x_n{:}A_n$, where $x_1, \ldots, x_n$ are variables and $A_1, \ldots, A_n$ are terms. In a context the variables that occur in M and A are given a type. If, in the judgment $\Gamma \vdash M : A$, the term A is a 'propositional type' (i.e. $\Gamma \vdash A$: `Prop`), we view M as a *proof* of A. If the term A is a 'set type' (i.e. $\Gamma \vdash A$: `Set` or $\Gamma \vdash A$: `Type`), we view M as an *element* of the set A.

Finally, there is another *sort* `Type`^{s}, that contains just `Set`. It is there to allow

declarations of the form x:Set in the context, declaring a new set, which is in our formalism only possible if Set itself has a type.

Definition 1 *The typed λ-calculus λPREDω, representing higher order predicate logic, is defined as follows. The set of* pseudoterms T *is defined by*

$$\mathsf{T} ::= \texttt{Prop} \mid \texttt{Set} \mid \texttt{Type} \mid \texttt{Type}^s \mid \mathsf{V} \mid (\Pi \mathsf{V}{:}\mathsf{T}.\mathsf{T}) \mid (\lambda \mathsf{V}{:}\mathsf{T}.\mathsf{T}) \mid \mathsf{T}\,\mathsf{T}.$$

Here, V *is a set of variables. The set of* sorts, S *is* $\{\texttt{Prop}, \texttt{Set}, \texttt{Type}, \texttt{Type}^s\}$. *A* context *is a sequence* $x_1{:}A_1, \ldots, x_n{:}A_n$*, where the* $\vec{x}$ *are in* V *and the* $\vec{A}$ *are in* T.

The typing rules, that select the well-typed *terms from the pseudo-terms, are as follows. Here, s ranges over the set of sorts* S.

$$(axiom) \vdash \texttt{Prop} : \texttt{Type} \quad \vdash \texttt{Set} : \texttt{Type}^s$$

$$(var) \quad \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$$

$$(weak) \quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$$

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$$
$$\text{if } (s_1, s_2) \in \{(\texttt{Set}, \texttt{Set}), (\texttt{Set}, \texttt{Type}), (\texttt{Type}, \texttt{Type}), (\texttt{Prop}, \texttt{Prop}), (\texttt{Set}, \texttt{Prop}), (\texttt{Type}, \texttt{Prop})\}$$

$$(\lambda) \quad \frac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$$

$$(app) \quad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$$

$$(conv) \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \qquad \text{if } A =_\beta B$$

In the rules (var) and (weak) it is always assumed that the newly declared variable is fresh, that is, it has not yet been declared in Γ. *The equality in the conversion rule (*$conv_\beta$*) is the* β*-equality on the set of pseudo-terms* T.

A pseudo-term A is typable *if there is a context Γ and a pseudo-term B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable. The set of typable terms of λPREDω is denoted by* $\mathsf{TERM}(\lambda\text{PRED}\omega)$.

The only type-forming operator in this language is the Π, which comes in three flavors, depending on the type of the *domain* (the A in $\Pi x{:}A.B$) and the type of the *range* (the B in $\Pi x{:}A.B$). Intuitively, a Π-type should be read as a set of functions. If we depict the occurrences of x in B explicitly by writing $B(x)$, the intuition is:

$$\Pi x{:}A.B(x) \approx \prod_{a \in A} B(a) = \{f \mid \forall a \in A[f\ a \in B(a)]\}.$$

So, $\Pi x{:}A.B$ is the *dependent function type* of functions taking a term of type A as input and delivering a term of type B in which x is replaced by the input. We therefore immediately recover the ordinary function type as a special instance.

Notation 2 • *In case $x \notin \mathrm{FV}(B)$, we write $A{\to}B$ for $\Pi x{:}A.B$. We call this a* non-dependent *function type.*
- *We omit brackets by letting them associate to the right. So $A{\to}B{\to}C$ denotes $A{\to}(B{\to}C)$.*

By examples we list all instances of the Π-type that can be encountered in λPREDω.

Examples 3 *(1) Using the combination (*$\mathtt{Set}$*,*$\mathtt{Set}$*), we can form the function type $A{\to}B$ for $A, B{:}\mathtt{Set}$. Furthermore, it also extends to higher order function types like $(A{\to}B){\to}A$, the type of functions taking functions from A to B as input and returning a value of type A.*
If $\Gamma \vdash A{:}\mathtt{Set}$ and $\Gamma, x{:}A \vdash B{:}\mathtt{Set}$, then $x \notin \mathrm{FV}(B)$ in λPREDω, so all types formed by ($\mathtt{Set}$*,*$\mathtt{Set}$*) are non-dependent function types.*

(2) Using the combination ($\mathtt{Set}$*,*$\mathtt{Type}$*) we can form types of unary predicates and binary relations: if $A : \mathtt{Set}$, then $A{\to}\mathtt{Prop} : \mathtt{Type}$ and $A{\to}A{\to}\mathtt{Prop} : \mathtt{Type}$.*
If $\Gamma \vdash A{:}\mathtt{Set}$ and $\Gamma, x{:}A \vdash B{:}\mathtt{Type}$, then $x \notin \mathrm{FV}(B)$ in λPREDω, so all types formed by ($\mathtt{Set}$*,*$\mathtt{Type}$*) are non-dependent types.*

(3) Using the combination ($\mathtt{Type}$*,*$\mathtt{Type}$*) we can form higher order predicate types: if $A : \mathtt{Set}$, then $(A{\to}\mathtt{Prop}){\to}\mathtt{Prop} : \mathtt{Type}$, the type of predicates over unary predicates over A. All types formed by (*$\mathtt{Type}$*,*$\mathtt{Type}$*) are non-dependent types.*

(4) Using the combination ($\mathtt{Prop}$*,*$\mathtt{Prop}$*), we can form the propositional type $\varphi{\to}\psi$ for $\varphi, \psi{:}\mathtt{Prop}$. This is to be read as an* implicational formula.
All types formed by ($\mathtt{Prop}$*,*$\mathtt{Prop}$*) are non-dependent types.*

(5) Using the combination ($\mathtt{Set}$*,*$\mathtt{Prop}$*), we can form the dependent propositional type $\Pi x{:}A.\varphi$ for $A{:}\mathtt{Set}$, $\varphi{:}\mathtt{Prop}$. This is to be read as a* universally

quantified formula *over* A.
If $\Gamma \vdash A{:}\texttt{Type}$ *and* $\Gamma, x{:}A \vdash \varphi{:}\texttt{Prop}$*, then it* can *occur that* $x \in \mathrm{FV}(\varphi)$ *in* λPREDω*. An example is* $\Pi x{:}A.Px{\to}Px : \texttt{Prop}$ *(in the context* $A{:}\texttt{Set}, P{:}A{\to}\texttt{Prop}$*).*

*(6) Using the combination (*Type,Prop*), we can do quantification over higher order domains, like in* $\Pi P{:}A{\to}A{\to}\texttt{Prop}.\varphi$*. In general, if* $B : \texttt{Type}$ *and* $\varphi : \texttt{Prop}$*, then* $\Pi P{:}B.\varphi : \texttt{Prop}$*.*
This type is (in general) a dependent type. An example is $\Pi P{:}A{\to}\texttt{Prop}.Px{\to}Px : \texttt{Prop}$ *(in the context* $A{:}\texttt{Set}, x{:}A$*).*

We will not define formal interpretations from higher order predicate logic to λPREDω and back. We motivate λPREDω by listing some examples of typing statements.

Examples 4 *(1)* $\texttt{nat}{:}\texttt{Set}, 0{:}\texttt{nat}, >{:}\texttt{nat}{\to}\texttt{nat}{\to}\texttt{Prop} \vdash \lambda x{:}\texttt{nat}.x{>}0 : \texttt{nat}{\to}\texttt{Prop}$.
Here we see the use of λ*-abstraction to define predicates.*

(2) $\texttt{nat}{:}\texttt{Set}, 0{:}\texttt{nat}, S{:}\texttt{nat}{\to}\texttt{nat}$
$\vdash \Pi P{:}\texttt{nat}{\to}\texttt{Prop}.(P0){\to}(\Pi x{:}\texttt{nat}.(Px{\to}P(Sx))){\to}\Pi x{:}\texttt{nat}.Px : \texttt{Prop}$.
This is the formula for induction written down in λPREDω *as a term of type* Prop*.*

(3) $A{:}\texttt{Set}, R{:}A{\to}A{\to}\texttt{Prop} \vdash \Pi x, y, z{:}A.Rxy{\to}Ryz{\to}Rxz : \texttt{Prop}$.
Transitivity of R*.*

(4) $A{:}\texttt{Set} \vdash \lambda R, Q{:}A{\to}A{\to}\texttt{Prop}.\Pi x, y{:}A.Rxy{\to}Qxy :$
$(A{\to}A{\to}\texttt{Prop}){\to}(A{\to}A{\to}\texttt{Prop}){\to}\texttt{Prop}$.
Inclusion of relations.

(5) $A{:}\texttt{Set} \vdash \lambda x, y{:}A.\Pi P{:}A{\to}\texttt{Prop}.(Px{\to}Py) : A{\to}A{\to}\texttt{Prop}$.
This relation is also called 'Leibniz equality' and is usually denoted by $=_L$ *or* $=_A$ *if we want to denote the domain type explicitly.*

(6) $A{:}\texttt{Set}, x, y{:}A \vdash \lambda r{:}x =_A y.\lambda P{:}A{\to}\texttt{Prop}.r(\lambda z{:}A.Pz \supset Px)(\lambda q{:}Px.q) :$
$x =_A y{\to}y =_A x$.
The proof of the fact that Leibniz equality is symmetric.

Just as in higher order predicate logic, it is possible to define the usual intuitionistic connectives and constants &, $\vee$, False, True, $\neg$ and $\exists$ in λPREDω. However, in presence of inductive types, one usually also defines the connectives inductively (as is also standard in Coq). We therefore do not give the higher order definitions of the connectives here, but take them as being defined inductively. We discuss the connectives briefly in the next Section.

3.1 Inductive Types

A basic notion in logic and set theory is induction: when a set is defined inductively, we understand it as being 'built up from the bottom' by a set of basic constructors. Elements of such a set can be decomposed in 'smaller

elements' in a well-founded manner. This gives us the principles of *proof by induction* and *function definition by recursion*.

If we want to add inductive types to our type theory, we have to add a definition mechanism that allows us to introduce a new inductive type, by giving the name and the constructors of the inductive type. The theory should automatically generate a scheme for proof-by-induction and a scheme for primitive recursion. It turns out that this can be done very generally in type theory, including very many instances of induction. Here we shall use a variant of the inductive types that are present in the system Coq [9] and that were first defined in [5].

We illustrate the rules for inductive types in $\lambda\text{PRED}\omega^{\text{ind}}$ by first treating the (very basic) example of natural numbers nat. We would like the user to be able to write something like

$$\begin{aligned}\texttt{Inductive nat} : \texttt{Set} :=& \\ & 0 : \texttt{nat} \\ & \mid S : \texttt{nat}{\to}\texttt{nat}.\end{aligned}$$

to obtain elimination principles that allow to define functions over nat by (higher order) primitive recursion and to prove properties over nat by induction. This amounts to the derivability of the following rules. ($\texttt{Rec}_{\texttt{nat}}(f_1, f_2)$ denotes *some* term containing f_1 and f_2 as subexpressions.)

$$(\text{elim}_1)\ \frac{\Gamma \vdash A : \texttt{Set/Type} \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : \texttt{nat}{\to}A{\to}A}{\Gamma \vdash \texttt{Rec}_{\texttt{nat}}(f_1, f_2) : \texttt{nat}{\to}A}$$

$$(\text{elim}_2)\ \frac{\Gamma \vdash P : \texttt{nat}{\to}\texttt{Prop} \quad \Gamma \vdash f_1 : P0 \quad \Gamma \vdash f_2 : \Pi x{:}\texttt{nat}.Px{\to}P(Sx)}{\Gamma \vdash \texttt{Rec}_{\texttt{nat}}(f_1, f_2) : \Pi x{:}\texttt{nat}.Px}$$

The rule (elim_1) allows the definition of functions by primitive recursion. The rule (elim_2) allows proofs by induction. To make sure that the $\texttt{Rec}_{\texttt{nat}}(-,-)$ functions compute in the correct way, we should have the following reduction (computation) rules.

$$\begin{aligned}\texttt{Rec}_{\texttt{nat}}(f_1, f_2)0 &\to_\iota f_1 \\ \texttt{Rec}_{\texttt{nat}}(f_1, f_2)(St) &\to_\iota f_2 t(\texttt{Rec}_{\texttt{nat}}(f_1, f_2)t)\end{aligned}$$

In Coq, these terms $\texttt{Rec}_{\texttt{nat}}(f_1, f_2)$ can be constructed, using a well-founded fixed point construction. (See [9] for details.) It is also possible to take the (elim) rules as primitives (adding a Rec constant) and define everything in terms of Rec, but this approach is not taken in the type system of Coq.

However, given the definition of nat above, Coq generates itself three defined constants Nat_rec, Nat_rect and Nat_ind, representing Rec above. In particular, the constant Nat_ind is of type

$$\Pi P{:}\texttt{nat}{\to}\texttt{Prop}.(P0){\to}(\Pi x{:}\texttt{nat}\,Px{\to}P(Sx)){\to}\Pi x{:}\texttt{nat}.Px.$$

One usually defines a function in Coq by giving an equational specification. Given the following equations ($h(x, fx)$ is a term with sub-terms x and fx and no other occurrences of f)

$$\begin{aligned} f0 &= g \\ f(Sx) &= h(x, fx), \end{aligned}$$

Coq generates a term $\texttt{Rec}\,(g, h)$ that satisfies these equations (for f). This amounts to specifying a function by primitive recursion. The situation is more general: Coq also generates a solution for f specified by the equations

$$\begin{aligned} f00 &= g_1 \\ f0(Sy) &= g_2(y), \\ f(Sx)0 &= g_3(x) \\ f(Sx)(Sy) &= g_4(x, y, f(x, y)) \end{aligned}$$

and more general for functions that are specified by giving a set of equations where the left hand sides cover all possible patterns and the recursive calls on the right hand side are on 'strictly smaller' expressions (according to some some syntactic ordering on terms). The precise syntax is as follows. (We define equality on natural numbers, as a binary function from nat to bool.)

```
Fixpoint b_eq [n,m:nat]: bool :=
   Cases n m of
      0     0     => true
    | 0     (S y) => false
    | (S x) 0     => false
    | (S x) (S y) => (b_eq x y)
   end.
```

In Coq, this defines a function, like the f above (where now, $g_2 = g_3 = \lambda n{:}\texttt{nat}.\texttt{true}$ and $g_4(x, y, f(x, y)) = f(x, y)$). The conditions under which such a pattern defines a function are that the left-hand sides should cover all possible patterns and that the recursive call on the right hand side is on structurally smaller expressions.

It is understood that the additional ι-reduction is also included in the *conversion*-rule (conv), where we now have '$A =_{\beta\iota} B$' as a side-condition. The subscript in $\texttt{Rec}_{\texttt{nat}}$ will be omitted, when clear from the context.

An example of the use of $\texttt{Rec}$ is in the definition of addition, $\texttt{add}$, which can be defined by $\texttt{add} := \texttt{Rec}\,(\lambda x{:}\texttt{nat}.x)(\lambda x{:}\texttt{nat}.\lambda f{:}\texttt{nat}\to\texttt{nat}.\lambda y{:}\texttt{nat}.S(fy))$. But we can equivalently define it by an equational specification

$$\begin{aligned} \texttt{add}\,0y &= y \\ \texttt{add}\,(Sx)y &= S(\texttt{add}\,xy). \end{aligned}$$

It is also possible to define predicates and relations by primitive recursion, by just taking $\texttt{Prop}$ or $\texttt{nat}\to\texttt{Prop}$ for A in (elim_1). An example is the relation 'less than or equal', $-\leq-$, which can be defined equationally as follows.

$$\begin{aligned} 0\leq y &= \texttt{True}, \\ (Sx)\leq 0 &= \texttt{False}, \\ (Sx)\leq(Sy) &= x\leq y. \end{aligned}$$

An example of the use of (elim_2) is the proof of $\Pi x{:}\texttt{nat}.x\leq x$ (by induction). Say that $\texttt{triv}$ is the (canonical) term (proof) of type $\texttt{True}$. Combining this with $\vdash \lambda x{:}\texttt{nat}.\lambda h{:}(x\leq x).h : \Pi x{:}\texttt{nat}.(x\leq x)\to((Sx)\leq(Sx))$ and applying $\texttt{Rec}$ we obtain

$$\vdash \texttt{Rec}\,\texttt{triv}(\lambda x{:}\texttt{nat}.\lambda h{:}(x\leq x).h) : \Pi x{:}\texttt{nat}.(x\leq x).$$

Another well-known example is the type of lists over a domain D. It is defined as follows.

$$\begin{aligned} &\texttt{Inductive list} : \texttt{Set} := \\ &\qquad \texttt{nil} : \texttt{list} \\ &\qquad \mid \texttt{cons} : \texttt{list}\to D\to\texttt{list} \end{aligned}$$

with the following derivable rules.

$$(\text{elim}_1)\ \frac{\Gamma\vdash A : \texttt{Set/Type} \quad \Gamma\vdash f_1 : A \quad \Gamma\vdash f_2 : \texttt{list}\to D\to A\to A}{\Gamma\vdash \texttt{Rec}_{\,\texttt{list}} f_1 f_2 : \texttt{list}\to A}$$

$$(\text{elim}_2)\ \frac{\Gamma\vdash P : \texttt{list}\to\texttt{Prop} \quad \begin{array}{l}\Gamma\vdash f_1 : P\texttt{nil} \\ \Gamma\vdash f_2 : \Pi x{:}\texttt{list}.\Pi d{:}D.Px\to P(\texttt{cons}\,xd)\end{array}}{\Gamma\vdash \texttt{Rec}_{\,\texttt{list}} f_1 f_2 : \Pi x{:}\texttt{list}.Px}$$

The rule (elim_1) allows the definition of functions by primitive recursion. The rule (elim_2) allows proofs by induction. The following reduction rules for $\texttt{Rec}_{\,\texttt{list}}$ hold, to make sure that the functions compute in the correct way.

$$\begin{aligned}\texttt{Rec}_{\texttt{list}} f_1 f_2 \texttt{nil} &\to_\iota f_1\\ \texttt{Rec}_{\texttt{list}} f_1 f_2 (\texttt{cons}\, t d) &\to_\iota f_2 t d (\texttt{Rec}_{\texttt{list}} f_1 f_2 t)\end{aligned}$$

Of course, there is a more general pattern behind these two examples. The types `nat` and `list` are examples of so called *algebraic inductive types*. In an algebraic inductive type, the types of the constructors (like `nil` and `cons`) are of the form $A_1 \to \cdots \to A_n \to \mu$, where μ is the type to be defined, and all the A_i are either equal to μ or do not contain μ as a sub-term. In Coq there is a much stronger schema for defining inductive types, allowing constructors of higher type and constructors with a dependent type. Furthermore the stronger schema allows to define inductive *predicates*, as opposed to just types. Then one can define, e.g. the relation $\leq$ inductively by saying it is the least binary relation over `nat` such that $\Pi x{:}\texttt{nat}.0 \leq x$ and $\Pi x, y{:}\texttt{nat}.(x \leq y) \to (Sx \leq Sy)$ hold. (Note that this definition of $\leq$ is different from – but equivalent to – the binary *recursive function* $\leq$ on `nat` given before.) As this is meant to be an introduction, we restrict our general theoretical exposition to the algebraic inductive types. In the formalization of the primitive recursive predicates, we use one inductive type that is not algebraic, namely the type `form`, which has two constructors of higher type:

```
f_all: nat -> (nat -> form) -> form
f_ex:  nat -> (nat -> form) -> form
```

The general scheme for such inductive types is rather complicated, although quite natural. We will not give it but treat such inductive types by some examples.

The extension of $\lambda\text{PRED}\omega$ with algebraic inductive types, $\lambda\text{PRED}\omega^{\text{ind}}$, is defined by adding the following scheme.

$$\begin{aligned}&\texttt{Inductive}\ \mu : \texttt{Set} :=\\ &\qquad \texttt{constr}_1 : \sigma^1_1 \to \cdots \sigma^1_{m_1} \to \mu\\ &\qquad \vdots\\ &\qquad \texttt{constr}_n : \sigma^n_1 \to \cdots \sigma^n_{m_n} \to \mu\end{aligned}$$

where the $\sigma^i_j : \texttt{Set}$ are all either μ or do not contain μ as a sub-term. We want to abstract over the occurrences of μ, so we denote $\sigma^i_1[X/\mu] \to \cdots \to \sigma^i_{m_i}[X/\mu] \to X$ by $\tau^i(X)$. (So $\tau^i(X)$ is the type scheme $\sigma^i_1 \to \cdots \sigma^i_{m_i} \to \mu$ in which all μ have been replaced by the variable X.)

We take the elimination rules (rules (elim_1) and (elim_2) from the `nat` example) as primitives. To define the elimination schemes in general we look at the list

of σ^i_j in τ^i that are equal to μ. Say that for τ^1, $\sigma^1_{j_1}, \ldots, \sigma^1_{j_k}$ are the types that are equal to μ. Then we define for $A : \texttt{Set/Type}$, $\hat{\tau}^1(A)$ as follows.

$$\hat{\tau}^1(A) := \sigma^1_1 \rightarrow \cdots \rightarrow \sigma^1_{m_1} \rightarrow \underbrace{A \rightarrow \cdots \rightarrow A}_{k} \rightarrow A.$$

The first elimination rule is now as follows.

$$(\text{elim}_1)\ \frac{\Gamma \vdash A : \texttt{Set/Type} \quad \Gamma \vdash f_1 : \hat{\tau}^1(A) \ \cdots \ \Gamma \vdash f_n : \hat{\tau}^n(A)}{\Gamma \vdash \texttt{Rec}_{\,\mu} f_1 \cdots f_n : \mu \rightarrow A}$$

It can easily be verified that the (elim$_1$)-rules of `nat` and `list` satisfy this general pattern.

For the reduction rule of the general pattern, we abbreviate $\texttt{Rec}_{\,\mu} f_1 \cdots f_n$ to $\texttt{Rec}\,\vec{f}$. The reduction rule is

$$\texttt{Rec}_{\,\mu} f_1 \cdots f_n (\texttt{constr}_i t_1 \cdots t_{m_i}) \rightarrow_\iota f_i t_1 \cdots t_{m_i} (\texttt{Rec}\,\vec{f} t_{j_1}) \cdots (\texttt{Rec}\,\vec{f} t_{j_k})$$

Let us now turn to the general pattern of the second elimination rule. Again we look at the list of σ^i_j in τ^i which are equal to μ. Say that for τ^1, $\sigma^1_{j_1}, \ldots, \sigma^1_{j_k}$ are the types that are equal to μ. Then we define for $P : \mu \rightarrow \texttt{Prop}$, $\hat{\tau}^1(P)$ as follows.

$$\hat{\tau}^1(P) := \Pi x_1{:}\sigma^1_1. \cdots \Pi x_{m_1}{:}\sigma^1_{m_1}. P x_{j_1} \rightarrow \cdots \rightarrow P x_{j_k} \rightarrow P(\texttt{constr}_1 x_1 \cdots x_{m_1}).$$

So, note that we have different definitions for $\hat{\tau}(A)$ (if $A : \texttt{Set/Type}$) and $\hat{\tau}(P)$ (if $P : \mu \rightarrow \texttt{Prop}$).

The second elimination rule is now as follows.

$$(\text{elim}_2)\ \frac{\Gamma \vdash P : \mu \rightarrow \texttt{Prop} \quad \Gamma \vdash f_1 : \hat{\tau}^1(P) \ldots \Gamma \vdash f_n : \hat{\tau}^n(P)}{\Gamma \vdash \texttt{Rec}_{\,\mu} f_1 \cdots f_n : \Pi x{:}\mu. P x}$$

Again, it can easily be verified that the (elim$_2$)-rules of `nat` and `list` satisfy this general pattern.

For the dependent case we have the same ι-reduction rule as for the non-dependent case:

$$\texttt{Rec}_{\,\mu} f_1 \cdots f_n (\texttt{constr}_i t_1 \cdots t_{m_i}) \rightarrow_\iota f_i t_1 \cdots t_{m_i} (\texttt{Rec}\,\vec{f} t_{j_1}) \cdots (\texttt{Rec}\,\vec{f} t_{j_k}).$$

Example 5 *The inductive type of booleans,* `bool`*, can be defined as follows.*

$$\begin{aligned}&\texttt{Inductive bool} : \texttt{Set} :=\\&\qquad\quad \texttt{true} : \texttt{bool}\\&\qquad\quad |\texttt{false} : \texttt{bool}\end{aligned}$$

This generates the following derivation rules.

$$(elim_1)\ \frac{\Gamma \vdash A : \texttt{Set/Type} \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : A}{\Gamma \vdash \texttt{Rec}_{\texttt{bool}} f_1 f_2 : \texttt{bool}{\to}A}$$

$$(elim_2)\ \frac{\Gamma \vdash P : \texttt{bool}{\to}\texttt{Prop} \quad \Gamma \vdash f_1 : P\texttt{true} \quad \Gamma \vdash f_2 : P\texttt{false}}{\Gamma \vdash \texttt{Rec}_{\texttt{bool}} f_1 f_2 : \Pi x{:}\texttt{bool}.Px}$$

The rewrite rule for $\texttt{Rec}_{\texttt{bool}}$ *is as follows.*

$$\begin{aligned}\texttt{Rec}_{\texttt{bool}} f_1 f_2 \texttt{true} &\to_\iota f_1,\\ \texttt{Rec}_{\texttt{bool}} f_1 f_2 \texttt{false} &\to_\iota f_2.\end{aligned}$$

So, $\texttt{Rec}_{\texttt{bool}}$ *represents the 'if–then–else–' function. More precisely, if* $t, q : A$ *and* $b : \texttt{bool}$*, then* `if` b `then` t `else` $q : A$ *is represented by* $\texttt{Rec}_{\texttt{bool}} tqb$*.*

The scheme defined so far is for *algebraic inductive types*. We now give an example of an inductive type that is more complicated than `nat` and `list`, because it uses higher types in one of the constructors. We want to define the type `tree` of countably branching trees with labels in D. (So a term of type `tree` represents a tree where the nodes and leaves are labeled with a term of type D and where at every node there are countably many subtrees.) The definition of `tree` is as follows.

$$\begin{aligned}&\texttt{Inductive tree} : \texttt{Set} :=\\&\qquad\quad \texttt{leaf} : D{\to}\texttt{tree}\\&\qquad\quad |\texttt{join} : D{\to}(\texttt{nat}{\to}\texttt{tree}){\to}\texttt{tree}\end{aligned}$$

Here, `leaf` creates a tree consisting of just a leaf, labeled by a term of type D. The constructor `join` takes a label (of type D) and an infinite (countable) list of trees to create a new tree. The (elim$_1$) rule is as follows.

$$(\mathrm{elim}_1)\ \frac{\Gamma \vdash A : \texttt{Set} \ \ \Gamma \vdash f_1 : D{\to}A \ \ \Gamma \vdash f_2 : D{\to}(\texttt{nat}{\to}\texttt{tree}){\to}(\texttt{nat}{\to}A){\to}A}{\Gamma \vdash \texttt{Rec}_{\texttt{tree}} f_1 f_2 : \texttt{tree}{\to}A}$$

$\texttt{Rec}_{\texttt{tree}}$ has the following reduction rule.

$$\begin{aligned}\texttt{Rec}_{\texttt{tree}} f_1 f_2 (\texttt{leaf}\, d) &\rightarrow_\iota f_1 d\\ \texttt{Rec}_{\texttt{tree}} f_1 f_2 (\texttt{join}\, dt) &\rightarrow_\iota f_2 dt (\lambda x{:}\texttt{nat}.\texttt{Rec}_{\texttt{tree}} f_1 f_2 (tx))\end{aligned}$$

It is an interesting exercise to define all kinds of standard functions on tree, like the function that takes the nth subtree (if it exists and take $\texttt{leaf}\, d_0$ otherwise) or the function that decides whether a tree is infinite (or just a single leaf).

For the type tree, we obtain the following (elim_2) rule.

$$(\text{elim}_2)\ \frac{\begin{array}{ll}\Gamma \vdash P : \texttt{tree}{\rightarrow}\texttt{Prop} & \Gamma \vdash f_2 : \Pi d{:}D.\Pi t{:}\texttt{nat}{\rightarrow}\texttt{tree}.\\ \Gamma \vdash f_1 : \Pi d{:}D.P(\texttt{leaf}\, d) & \qquad (\Pi n{:}\texttt{nat}.P(tn)){\rightarrow}P(\texttt{join}\, dt)\end{array}}{\Gamma \vdash \texttt{Rec}_{\texttt{tree}} f_1 f_2 : \Pi x{:}\texttt{tree}.Px}$$

We can explain this rule as follows: a tree is a well-founded object, but a tree may be created by joining countably many trees (indexed over nat) into a new one. This is done via the join constructor, which takes a list of trees ($t : \texttt{nat}{\rightarrow}\texttt{tree}$) and a label ($d : D$) to create another tree ($\texttt{join}\, dt$). Now, if we want to prove a property P for all trees, we have to show that P is preserved under the join constructor, i.e. we have to prove

$$(\forall n{:}\texttt{nat}.P(tn)) \rightarrow P(\texttt{join}\, dt).$$

for all $d : D$ and for all $t : \texttt{nat}{\rightarrow}\texttt{tree}$.

The reduction rule for $\texttt{Rec}_{\texttt{tree}}$ associated with this second elimination scheme is the same as before.

4 The method

This Section presents a method to mechanically prove a proposition φ from first order primitive recursive arithmetic in the Coq system. The method uses a three level interpretation of φ. The proposition is viewed on a syntactical, on a propositional, and on a computational level. The syntactical level is represented by the inductive type form, the propositional level by the type-sort Prop, and the computational level by the inductive type bool. The Prop and bool types are already present in Coq; the form type is defined in subsection 4.1.

Trivial propositions are trivial because they belong to a class of propositions that can be proved in a general mechanical fashion. In the `Coq` system there are three ways to deal with these trivial propositions: *ad hoc*, using *tacticals*, and using *reflection*. The reflection style of dealing with trivial propositions is the method we are interested in here.

In the ad hoc style of proving trivial propositions, φ is formalized on the propositional level as an expression of type `Prop`. The user provides a proof by hand by applying tactics to the current goal, until it is resolved. Advantages of the ad hoc style: It is usually the most efficient way if there is only one proposition to be proved. (One doesn't first have to define general tacticals, or to set up a general theory.) Moreover, if one is considering just one specific (type of) proposition, usually more clever tricks can be used to speed up the proving. For example in [6] (pp. 148–156), to prove primality of certain numbers, one first proves a result from algebra which is then applied. Disadvantage: An ad hoc proof works only once (to prove that specific proposition).

In the tacticals style of proving, φ is also formalized as an expression of type `Prop`. The user describes a general decision procedure for a certain class of propositions using tacticals. Tacticals combine tactics into proof procedures (new tactics). Advantages of the tactical style: It is a very general method that can save a lot of work (compared to the ad hoc style), especially when many 'similar' propositions have to be proven. The method yields a proof term that usually corresponds rather closely to the proof term that would have been found by using the ad hoc style. The decision algorithm is described on the meta level, which gives quite a lot of flexibility. However, this can also be a drawback, as the user will have to be able to program in the meta language (or in the tactical language if that is provided). Disadvantages: Can be very slow: all the steps have to be executed in the proof assistant, which requires a lot of unification and type checking.

In the reflection style of proving trivial propositions, φ is not formalized directly as an expression of type `Prop`. Rather, φ is formalized on the syntactical level as an expression p of a new type `form`, where `form` characterizes the class of propositions we are dealing with. Translations, $[\![-]\!]$ from `form` to `Prop` and $(\![-]\!)$ from `form` to `bool`, are used to obtain formalizations of p on the other two levels, such that $[\![p]\!] = \varphi$. These translations, as well as a translation from `bool` to `Prop` are defined in subsection 4.2. The important thing to note here is that the size of p is linear in the size of φ.

Eventually, what is needed is a proof-object inhabiting φ. This proof-object is constructed by combining two proof-objects. First, the proof-object `ok` inhabits the correctness theorem, which states that for all terms q of type `form`: $[\![q]\!]$ holds, if and only if (`istrue` $(\![q]\!)$) holds. Second, an inhabitant of (`istrue` $(\![p]\!)$) is sought for. This is easy: The boolean expression $(\![p]\!)$ reduces to

`true` (and then (`istrue true`) is inhabited) or it reduces to `false` (and then (`istrue false`) is not inhabited). The construction of these proof-objects is presented in subsection 4.3.

Advantages of the reflection method are: The size of the proof-object of type φ is *linear* in the size of φ itself and it is *trivial to construct*. (Note that a proof-object can – in general – be arbitrarily complex in terms of the size of the problem φ.) Almost all of the 'proof' is in the computation – which can be arbitrarily complex – but this is hidden in the type checking algorithm. That the proof-object is trivial conforms with the idea that proofs by computation are trivial and that computations should not contribute to the proof-object. Furthermore, reflection is a very general method, solving a class of problems instead of one problem. Disadvantages: Can be very slow: due to the generality of the method, the generated decision algorithms follow a general (non-optimized) pattern. For example the algorithm for checking primality is a characteristic function that is generically extracted from the definition of Prime. This is far more inefficient then, e.g. the special primality algorithm used in [6] (pp. 148–156). On the other hand a generic method for solving a large class of propositions will always be slow, compared to ad hoc clever tricks. Another disadvantage is that the user needs to syntactically characterize the class of propositions and provide the translations and the correctness proof.

4.1 Languages

Primitive recursive arithmetic (PRA) can be seen as a language of formulas. Formulas from this language are either basic formulas or compound formulas.

Basic formulas are built using the relations $<$, $=$, and $>$, from arithmetical terms. Arithmetical terms are either number constants, or number variables, or the result of applying a primitive recursive function prescription to other arithmetical terms.

Compound formulas are built using connectives or using bounded quantifiers. Connectives are $\neg$, $\wedge$, $\vee$, and $\rightarrow$. Bounded first order quantifiers are $\forall_<$ and $\exists_<$. These bind a number variable. The upper bound is an arithmetical term. The division and primality properties are examples which can be expressed in this language.

Example 6 *The division and primality predicates are primitive recursive.*

$$\begin{aligned} \text{Divides}(n,m) &= \exists k < m+1\,[k * n = m] \\ \text{Prime}(n) &= \forall d < n\,[\text{Divides}(d,n) \rightarrow d = 1] \wedge n > 1 \end{aligned}$$

The language of primitive recursive arithmetic is formalized in Coq as the inductive type form. Notice that the terms from which basic formulas are built are just objects of type nat. It is not necessary to treat these terms syntacticly, since both $[\![-]\!]$ and $(\!|-|\!)$ will translate them similarly. Note that the choice of not treating terms syntactically has a consequence: the formulas (the p of type form) are not really from PRA, but an extension thereof, namely where the base terms are the terms of type nat in Coq (instead of the terms generated from $\mathbb{N}$ by just application of primitive recursive functions). Formalizing this slight extension of PRA is more convenient, as it removes the extra syntactic level. Notice also the use of higher order function types in the type of the quantifier constructors f_all and f_ex. This allows binding of variables using the object level lambda abstraction.

Definition 7 *The language of primitive recursive arithmetic as formalized in* Coq.

```
Inductive form: Set :=
      f_lt:  nat -> nat -> form
    | f_le:  nat -> nat -> form
    | f_eq:  nat -> nat -> form
    | f_ge:  nat -> nat -> form
    | f_gt:  nat -> nat -> form
    | f_not: form -> form
    | f_and: form -> form -> form
    | f_or:  form -> form -> form
    | f_imp: form -> form -> form
    | f_all: nat -> (nat -> form) -> form
    | f_ex:  nat -> (nat -> form) -> form.
```

Notation 8 Coq-*notation for lambda- and Pi abstraction. We write*

[x:A]B *for* $\lambda x{:}A.B$

(x:A)B *for* $\Pi x{:}A.B$

The automatically generated induction principle form_ind ($\texttt{Rec}_{\texttt{form}}$ of the previous section) has the following type.

form_ind:
$\forall P$: form $\to$ Prop.
$(\forall n, m$: nat.$(P\,($f_lt $n\,m))) \to$
$(\forall n, m$: nat.$(P\,($f_le $n\,m))) \to$
$(\forall n, m$: nat.$(P\,($f_eq $n\,m))) \to$
$(\forall n, m$: nat.$(P\,($f_ge $n\,m))) \to$
$(\forall n, m$: nat.$(P\,($f_gt $n\,m))) \to$

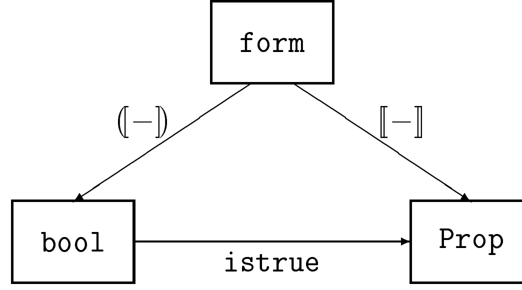

Fig. 1. The different languages and translations.

$(\forall\varphi\colon \texttt{form}.(P\,\varphi) \to (P\,(\texttt{f_not}\,\varphi))) \to$
$(\forall\varphi\colon \texttt{form}.(P\,\varphi) \to \forall\psi\colon \texttt{form}.(P\,\psi) \to (P\,(\texttt{f_and}\,\varphi\,\psi))) \to$
$(\forall\varphi\colon \texttt{form}.(P\,\varphi) \to \forall\psi\colon \texttt{form}.(P\,\psi) \to (P\,(\texttt{f_or}\,\varphi\,\psi))) \to$
$(\forall\varphi\colon \texttt{form}.(P\,\varphi) \to \forall\psi\colon \texttt{form}.(P\,\psi) \to (P\,(\texttt{f_imp}\,\varphi\,\psi))) \to$
$(\forall n\colon \texttt{nat}.\forall\Phi\colon \texttt{nat} \to \texttt{form}.(\forall m\colon \texttt{nat}.(P\,(\Phi m))) \to (P\,(\texttt{f_all}\,n\,\Phi))) \to$
$(\forall n\colon \texttt{nat}.\forall\Phi\colon \texttt{nat} \to \texttt{form}.(\forall m\colon \texttt{nat}.(P\,(\Phi m))) \to (P\,(\texttt{f_ex}\,n\,\Phi))) \to$
$\forall\varphi\colon \texttt{form}.(P\,\varphi)$

So, `form_ind` states that if a predicate P on `form` is closed under the constructors of the inductive type `form` (`f_lt`, `f_le` etcetera), then P holds for all terms of type `form`. Note the cases for `f_all` and `f_ex`: closure of P under `f_all` says that if P holds for all instances of Φ ($\forall m : \texttt{nat}.(P(\Phi m))$), then P holds for ($\texttt{f_all}\ n\,\Phi$).

The predicates from example 6 can now be expressed as functions with codomain `form`.

Example 9 *The division and primality predicates as formalized in* Coq.

```
Definition f_Divides: nat -> nat -> form :=
   [n,m:nat] (f_ex (S m) [k:nat](f_eq (mult k n) m)).

Definition f_Prime: nat -> form :=
   [n:nat]
      (f_and (f_gt n (1))
      (f_all n [d:nat] (f_imp (f_Divides d n) (f_eq d (1))))).
```

4.2 Translations

Three translations are defined on the types `form`, `bool`, and `Prop`. First, $[\![-]\!]$ maps terms of type `form` to terms of type `Prop`. Second, $(\![-]\!)$ maps terms of type `form` to terms of type `bool`. Third, `istrue` maps terms of type `bool` to terms of type `Prop`. The three translations are depicted in Figure 1.

4.2.1 *The translation* $[\![-]\!]$: `form` $\to$ `Prop`

The translation $[\![-]\!]$ takes as input a formula p of type `form` and it produces a proposition of type `Prop`. Because `form` is an inductive type, $[\![-]\!]$ can be defined by recursion by specifying a translation for each of the `form`-constructors. In describing recursive functions, we will not use the `Rec` notation that we introduced in the definition of the type system $\lambda\text{PRED}\omega^{\text{ind}}$. Instead we use a `Coq` like notation, which uses pattern matching to deconstruct an element of an inductive type. Moreover, Coq has special syntactic sugar for defining recursive functions by a `Fixpoint` command. Arbitrary fixpoints are however not allowed: the recursive calls should be done on structurally smaller terms. This conforms precisely with the functions definable by the (elim) schemes that we have given before. (In the following, the definitions using `Fixpoint` can all be translated to functions defined by `Rec`.)

Definition 10 *The translation* $[\![-]\!]$ *as formalized in* `Coq`.

$$
\begin{aligned}
[\![\texttt{f_lt}\ t_1\,t_2]\!] &= \texttt{lt}\,t_1\,t_2\\
[\![\texttt{f_le}\ t_1\,t_2]\!] &= \texttt{le}\,t_1\,t_2\\
[\![\texttt{f_eq}\ t_1\,t_2]\!] &= t_1\,\texttt{=}\,t_2\\
[\![\texttt{f_ge}\ t_1\,t_2]\!] &= \texttt{ge}\,t_1\,t_2\\
[\![\texttt{f_gt}\ t_1\,t_2]\!] &= \texttt{gt}\,t_1\,t_2\\
[\![\texttt{f_not}\ p]\!] &= \texttt{\~{}}[\![p]\!]\\
[\![\texttt{f_and}\ p\,q]\!] &= [\![p]\!]\texttt{/\textbackslash}[\![q]\!]\\
[\![\texttt{f_or}\ p\,q]\!] &= [\![p]\!]\texttt{\textbackslash/}[\![q]\!]\\
[\![\texttt{f_imp}\ p\,q]\!] &= [\![p]\!]\texttt{->}[\![q]\!]\\
[\![\texttt{f_all}\ t\,h]\!] &= \texttt{(}x\texttt{:}\,\texttt{nat)}\,\texttt{(lt}\,x\,t\texttt{)}\ \texttt{->}\ [\![h\,x]\!]\\
[\![\texttt{f_ex}\ t\,h]\!] &= \texttt{Ex}\ \texttt{[}x\texttt{:}\,\texttt{nat]}\,\texttt{((lt}\,x\,t\texttt{)}\ \texttt{/\textbackslash}\ [\![h\,x]\!]\texttt{)}
\end{aligned}
$$

4.2.2 *The translation* $(\![-]\!)$: `form` $\to$ `bool`

The translation $(\![-]\!)$ takes as input a formula p of type `form` and it produces a boolean expression of type `bool`. Because `form` is an inductive type, $(\![-]\!)$ can be defined by specifying a translation for each of the `form`-constructors.

Definition 11 *The translation* $(\!|-|\!)$ *as formalized in* Coq.

$$
\begin{aligned}
(\!|\texttt{f_lt}\ t_1\, t_2|\!) &= \texttt{b_lt}\ t_1\, t_2\\
(\!|\texttt{f_le}\ t_1\, t_2|\!) &= \texttt{b_le}\ t_1\, t_2\\
(\!|\texttt{f_eq}\ t_1\, t_2|\!) &= \texttt{b_eq}\ t_1\, t_2\\
(\!|\texttt{f_ge}\ t_1\, t_2|\!) &= \texttt{b_ge}\ t_1\, t_2\\
(\!|\texttt{f_gt}\ t_1\, t_2|\!) &= \texttt{b_gt}\ t_1\, t_2\\
(\!|\texttt{f_not}\ p|\!) &= \texttt{b_not}\ (\!|p|\!)\\
(\!|\texttt{f_and}\ p\, q|\!) &= \texttt{b_and}\ (\!|p|\!)\, (\!|q|\!)\\
(\!|\texttt{f_or}\ p\, q|\!) &= \texttt{b_or}\ (\!|p|\!)\, (\!|q|\!)\\
(\!|\texttt{f_imp}\ p\, q|\!) &= \texttt{b_imp}\ (\!|p|\!)\, (\!|q|\!)\\
(\!|\texttt{f_all}\ t\, h|\!) &= \texttt{b_all}\ t\ [x\colon \texttt{nat}]\, (\!|h\, x|\!)\\
(\!|(\texttt{f_ex}\ t\, h|\!) &= \texttt{b_ex}\ t\ [x\colon \texttt{nat}]\, (\!|h\, x|\!)
\end{aligned}
$$

The boolean versions of the basic relations are defined by:

Definition 12 *Boolean inequalities as formalized in* Coq.

```
Fixpoint b_le [n,m:nat]: bool :=
   Cases n m of
      O     O     => true
    | O     (S y) => true
    | (S x) O     => false
    | (S x) (S y) => (b_le x y)
   end.

Definition b_lt := [n,m:nat](b_le (S n) m).
Definition b_ge := [n,m:nat](b_le m n).
Definition b_gt := [n,m:nat](b_lt m n).
```

Definition 13 *Boolean equality as formalized in* Coq.

```
Fixpoint b_eq [n,m:nat]: bool :=
   Cases n m of
      O     O     => true
    | O     (S y) => false
    | (S x) O     => false
    | (S x) (S y) => (b_eq x y)
   end.
```

The computational versions of the connectives are defined by:

Definition 14 *Boolean versions of the connectives as defined in* Coq*.*

```
Definition b_not := [x:bool](if x then false else true).
Definition b_and := [x,y:bool](if x then y else false).
Definition b_or  := [x,y:bool](if x then true else y).
Definition b_imp := [x,y:bool](if x then y else true).
```

The computational version of the bounded universal quantifier is defined by translating it into a large conjunction. The computational version of the bounded existential quantifier is defined by translating it into a large disjunction.

Definition 15 *Boolean version of the bounded universal quantifier as formalized in* Coq*.*

```
Fixpoint b_all [b:nat]: (nat -> bool) -> bool :=
   [f:nat->bool]
      Cases b of
         O       => true
       | (S m)  => (b_and (f m) (b_all m f))
      end.
```

Definition 16 *Boolean version of the bounded existential quantifier as formalized in* Coq*.*

```
Fixpoint b_ex [b:nat]: (nat -> bool) -> bool :=
   [f:nat->bool]
      Cases b of
         O       => false
       | (S m)  => (b_or (f m) (b_ex m f))
      end.
```

4.2.3 The translation `istrue`: `bool` $\rightarrow$ `Prop`

The translation `istrue` takes as input a boolean expression and lifts it to the propositional level:

Definition 17 *The translation* `istrue` *as formalized in* Coq*.*

```
Definition istrue := [x:bool](if x then True else False).
```

Given a formula p of type `form`, the objective is to construct a proof-object inhabiting $[\![p]\!]$. This is done in two steps. First, it is shown that the diagram in Figure 1 commutes. Next, it is shown, using the conversion rule, that (`istrue` $(\!|p|\!)$) is inhabited. The combination of these two steps yields the desired proof-object.

Using the induction principle generated by the inductive definition of `form`, we can construct a correctness proof `ok` of the translations.

$$\texttt{ok} : \forall p \colon \texttt{form}.[\![p]\!] \leftrightarrow (\texttt{istrue}\ (\!|p|\!))$$

The proof-object `ok` shows that the diagram in Figure 1 commutes. In general only the implication from right to left is needed. However, in the proof of the correctness theorem the other direction is very useful in some of the induction cases.

The translation $(\!|-|\!)$ is constructed in such a way, that for closed terms p of type `form` that represent a provable proposition, it holds that

$$(\!|p|\!) \twoheadrightarrow_{\beta\iota} \texttt{true}$$

and therefore

$$(\texttt{istrue}\ (\!|p|\!)) \twoheadrightarrow_{\beta\iota} \texttt{True}$$

where $\twoheadrightarrow_{\beta\iota}$ is the `Coq` reduction relation. From the conversion rule, it now follows that any inhabitant of `True` is also an inhabitant of (`istrue` $(\!|p|\!)$). Clearly, `True` is inhabited by the unit term `triv`, and therefore (`istrue` $(\!|p|\!)$) is inhabited.

By combining the inhabitant of (`istrue` $(\!|p|\!)$) with `ok`, we get an inhabitant of $[\![p]\!]$, which is what we were looking for.

It would be nice if we also had an inverse of $[\![-]\!]$. In that case the user could write down the goal as an expression φ of type `Prop` and have the system translate it to an expression p of type `form`. This inverse translation cannot be expressed within the object language. Some programming in the implementation language of `Coq` would be required to implement this translation. An alternative would be to use the extensible grammar mechanism of `Coq` to make the syntactical level look the same as the propositional level.

5 Results and discussion

The language of primitive recursive arithmetic can be elegantly formalized in the Coq system using inductive types. As a matter of fact, the inductive type `form` contains a bit more than the formulas of PRA, namely the ones where we take the terms of type `nat` in Coq as base terms. The formalization is used to automatically prove propositions of primitive recursive arithmetic.

Even though primitive recursive arithmetic is a limited language, many trivial propositions that are tedious to prove by hand can be expressed in it. By having the Coq proof assistant to prove these automatically, the user can concentrate on the real, important, and mathematically interesting problems. We believe that the methods discussed in this paper contribute to the user-friendliness of systems like Coq. It is possible to extend the method to include other predicates and functions on `nat` (or even other logical connectives). Suppose we have a relation R typable in Coq, so $R : \texttt{nat}^n \to \texttt{Prop}$. Moreover suppose that R is computable in Coq, so there is a term $f_r : \texttt{nat}^n \to \texttt{bool}$ that computes R. Then we can extend our method to include R as a predicate by adding a constructor $r : \texttt{nat}^n \to \texttt{form}$ in the definition of `form` and by constructing a term q such that

$$q : \Pi \vec{x}{:}\texttt{nat}^n . R\vec{x} \leftrightarrow \texttt{istrue}(f_r \vec{x}).$$

The proof term q states (in Coq) that R is computable by f_r; it is used in the construction of the new proof term `ok` for this extension of `form`. We can depict the situation as follows.

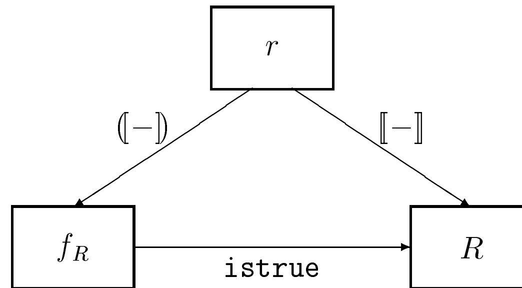

Fig. 2. Extension of the method with computable predicate R.

As to the efficiency of the procedure: The procedure described here is not very fast. To check (Prime 61) takes several minutes on a fast Unix workstation, even though the proof-object is a λ-term of only 10 lines of code and the total size of the theory development is only 300 lines of code. (See [13].) There are three reasons why this method is slow. First, the addition and multiplication functions operate on the standard unary numbers (generated by the constructors `O` and `S`). Things would be faster had we used binary versions of these functions on the computational level [10]. However, the correctness proof will

become more complicated if on the propositional level the same definitions of addition and multiplication are used. The use of these inefficient definitions is desirable because a lot of theory development depends on the unary defined natural numbers. The second reason is that computations are interpreted in Coq which in turn is interpreted in a functional language. This is not the most efficient setting for large computations. Third, the procedure is very general, meaning that it cannot take into account clever tricks to avoid computations. This results in slow algorithms. For example to check (Prime 61) all numbers between 1 and 61 are tested as divisors of 61 instead of only the numbers up-to $\sqrt{61}$.

The method of computational reflection is not new, [7] gives an overview and history of reflection and contrasts it with the LCF (tacticals) approach. (We have briefly contrasted the reflection method with other approaches in Section 4.) In NuPrl a reflection mechanism and a library with many different applications was implemented [8]. In [4] computational reflection is applied in Coq to first order theories of algebraic structures such as monoids and rings. In [3] application of the reflection principle to decide equational theories is studied.

In [12] a similar technique was used to generate proofs for statements of PRA; there are however some differences with the internal method described in this paper. The method in [12] uses an external program. This program takes as input a string containing a formula φ of PRA and produces output which can be read by the Lego [11] proof system. The output produced in this way contains the formula φ of type `Prop`, a characteristic term χ_φ of type `bool` and Lego tactics which will construct a proof-object $\mathtt{ok}_\varphi$ of type $\varphi \leftrightarrow (\mathtt{istrue}\ \chi_\varphi)$. The present method uses one correctness proof `ok`, which can be instantiated with a formula φ of PRA by applying it to φ since φ is of type `form` which is now part of the object language.

Applying the method to other theories requires modifications to the type `form` as well as to the translations $[\![-]\!]$ and $(\![-]\!)$ introduced in section 4.2, and to the proof-object `ok` from section 4.3.

Acknowledgments

This work has benefited much from discussions with Henk Barendregt and Thijs Cobben. Furthermore we want to express our gratitude to the anonymous referees for their valuable comments.

References

[1] BARENDREGT, H (1992), Lambda Calculi with Types, *in* "Handbook of Logic in Computer Science, Volume II".

[2] BARENDREGT, H. AND BARENDSEN, E. (1997), Autarkic Computations in Formal Proofs, Computing Science Institute, University of Nijmegen.

[3] BARTHE G. AND RUYS, M. AND BARENDREGT H. (1996), A Two-Level Approach towards lean Proof-Checking.

[4] BOUTIN, S. (1997), Using reflection to build efficient and certified decision procedures.

[5] COQUAND, TH. AND PAULIN-MOHRING, CH. (1990), Inductively defined types, In P. Martin-Löf and G. Mints editors. *COLOG-88 : International conference on computer logic, LNCS 417.*

[6] ELBERS H. J. (1998), Connecting Informal and Formal Mathematics, PhD. thesis, Eindhoven University of Technology.

[7] HARRISON, J. (1995), Metatheory and Reflection in Theorem Proving: a Survey and Critique, Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre.

[8] HOWE, D. (1988) Computational Metatheory in Nuprl, The Proceedings of the Ninth International Conference of Automated Deduction, eds. E. Lusk and R. Overbeek, LNCS 310, pp. 238–257.

[9] HUET, G. ET AL. (1997), The Coq Proof Assistant, Reference Manual, Version 6.1, INRIA-Rocquencourt — CNRS-ENS Lyon.

[10] HUISMAN, M. (1997), Binary addition in Lego, Technical Report CSI-R9716, Computing Science Institute, University of Nijmegen.

[11] LUO Z. AND POLLACK R. (1992) (1993,1994), LEGO Proof Development System: User's Manual, Department of Computer Science, University of Edinburgh.

[12] OOSTDIJK, M. (1996), Proof by Calculation, Master's thesis 385, Universitaire School voor Informatica, University of Nijmegen.

[13] OOSTDIJK, M. AND GEUVERS, H. (1998), Coq proof development files, `http://www.win.tue.nl/ martijno/work/reflection/`.