A Proof of the S_n^m Theorem in Coq

Vincent Zammit

March 1997

Abstract

This report describes the implementation of a mechanisation of the theory of computation in the Coq proof assistant which leads to a proof of the S_n^m theorem. This mechanisation is based on a model of computation similar to the partial recursive function model and includes the definition of a computable function, proofs of the computability of a number of functions and the definition of an effective coding from the set of partial recursive functions to natural numbers. This work forms part of a comparative study of the HOL and Coq proof assistants.

1 Introduction

This report illustrates the mechanisation in Coq of the theory of computation leading to the S_n^m theorem. This work is a case study using Coq and is part of a comparative study of the theorem proof assistants Coq and HOL. The definitions and proofs of even the most trivial results of computability tend to be of a very technical nature much similar to the proofs of theorems one finds in mathematical texts, and thus this theory offers an extensive case study for the analysis of the two approaches of mechanical verification.

The implementation illustrated in this report is based on a model of computation similar to the definition of partial recursive functions found in the literature on computation (see for instance [3, 10, 12].) The next section introduces the definition of partial recursive functions and section 3 gives a brief overview of the Coq theorem prover. A model of computation based on partial recursive functions and its formalisation in Coq is then given in section 4. In section 5, the key notion of a computable function is defined, and several functions are proved to be computable according to this definition. The result given in this section are then used in section 6 in the definition of an effective coding of partial recursive functions and the proof of the S_n^m theorem. Conclusions are finally given in the last section of this report.

A different mechanisation of the theory of computation has also been implemented in HOL. This mechanisation is based on the URM model of computation and includes a proof that partial recursive functions are URM computable. This mechanisation is illustrated separately in [13]. The results of the comparative study will be published in [14].

2 Partial Recursive Functions

The set of partial recursive functions is defined in the literature (see for instance [3, 10]) as the smallest set of n-ary partial functions on natural numbers which contains the three basic types of functions:

- the zero functions: $\forall n, x_0, \ldots, x_{n-1}.\mathcal{Z}_n(x_0, \ldots, x_{n-1}) = 0$,
- the successor function: $\forall x_0.\mathcal{S}(x_0) = x_0 + 1$,
- and the projection functions: $\forall n, i < n, x_0, \ldots, x_{n-1}.\mathcal{U}_n^i(x_0, \ldots, x_{n-1}) = x_i$;

and which is closed under the operations of:

Substitution Given a k-ary function f, and k n-ary functions $\vec{g} = (g_0, \ldots, g_{k-1})$, the substitution $f \hat{\circ} \underline{g}$ is defined as the function which maps a vector $\vec{x} = (x_0, \ldots, x_{n-1})$ into the application of f on the result of the application of the functions in $\vec{g}$ on $\vec{x}$:

$$\begin{aligned} &f\hat{\circ}\underline{g}(x_0, \dots, x_{n-1}) = \\ &\quad f(g_0(x_0, \dots, x_{n-1}), \dots, g_{k-1}(x_0, \dots, x_{n-1})). \end{aligned}$$

Recursion Given an n-ary base case function β and an $(n+2)$-ary recursion step function σ, the $(n+1)$-ary primitive recursive function $\mathcal{R}(\beta;\sigma)$ is defined as follows:

$$\begin{aligned} \mathcal{R}(\beta;\sigma)(0, x_0, \dots, x_{n-1}) &= \beta(x_0, \dots, x_{n-1}) \\ \mathcal{R}(\beta;\sigma)(x+1, x_0, \dots, x_{n-1}) &= \sigma(x, \mathcal{R}(\beta;\sigma)(x, x_0, \dots, x_{n-1}), x_0, \dots, x_{n-1}). \end{aligned}$$

Minimalisation Given an $(n+1)$-ary function f, its unbounded minimalisation is the n-ary function given by:

$$\mu_x(f(x, x_0, \dots, x_{n-1}) = 0) = \begin{cases} \text{the least } x \textit{ s.t.} & f(x, x_0, \dots, x_{n-1}) = 0, \text{ and for} \\ & \text{all } x' \leq x,\ f(x', x_0, \dots, x_{n-1}) \\ & \text{is defined} \\ \text{undefined} & \text{if no such } x \text{ exists.} \end{cases}$$

It is shown that the set of partial recursive functions is equal to the set of computable functions defined according to any proposed model of computation [3, 10, 12]; and a mechanisation in HOL [5, 6] of the result that any partial recursive function is computable according to the URM model of computation is illustrated in [13]. The mechanisation described in this report is based on a model of computation which is very similar to the above definition of partial recursive functions. As a result proofs that particular functions are computable are relatively straightforward.

3 An Overview of Coq

The Coq system is an implementation in CAML of the Calculus of Inductive Constructions (CIC) [1], a variant of type theory related to Martin-Löf's Intuitionistic Type Theory [7, 8] and Girard's polymorphic λ-calculus F_ω [4]. Terms in CIC are typed and types are also terms. Such a type theory can be treated as a logic through the *Curry-Howard isomorphism* (see [11, 8] for introductions of the Curry-Howard isomorphism) where propositions are expressed as types. For instance, a conjunction $A \wedge B$ is represented by a product type $A \times B$, and an implication $A \Rightarrow B$ is represented by a function type $A \rightarrow B$. Also, a term of type τ can be seen as a proof of the proposition represented by τ, and thus theorems in the logic are nonempty types. For example, the function

$$\texttt{curry} = \lambda f.\lambda x.\lambda y.f(x, y)$$

which has type $((A \times B) \rightarrow C) \rightarrow A \rightarrow B \rightarrow C$ is a proof of the theorem $((A \wedge B) \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$. Objects which have the same normal form according to $\beta\delta\iota$-conversion are called convertible, and are treated as the same term by the logic. δ-conversion involves the substitution of a constant by its defining term and ι-conversion is automation of inductive definitions. The CIC implemented in Coq differs from that of LEGO [9] by having two sorts of universes, an impredicative universe for sets in which functions are computable, and a predicative universe for types and propositions in which functions (predicates) need not be computable (decidable).

Due to the Curry-Howard isomorphism, theorem proving corresponds to the construction of well typed terms and the core inference engine of Coq is basically a type checking algorithm of CIC terms. Terms whose type is a theorem are usually called proof objects and are stored in Coq theories. The Coq system provides the specification and proof language Gallina in which users perform the actual interactive theorem proving. Gallina constructs include commands for specifying definitions and for tactic based theorem proving and Coq users can extend the Gallina language by implementing new contructs in CAML. The files which Gallina accepts during theorem proving are usually called scripts (or proof scripts).

4 Partial Recursive Functions as a Model of Computation

The following section illustrates the syntax and semantics of $\mathcal{PRF}$.

4.1 The Syntax of $\mathcal{PRF}$

The syntax of the language $\mathcal{PRF}$ is defined such that each language construct corresponds to one of the three basic functions or to one of the three operators which build up partial recursive functions:

```
prf ::= Zero: prf
      | Succ: prf
      | Proj: nat → prf
      | Sub:  prf → prf → nat → nat → prf
      | Rec:  prf → prf → prf
      | Min:  prf → prf
```

It should be noted that a particular $\mathcal{PRF}$ program represents a different partial function for each arity. For example, although `Succ` is defined in order to represent the successor function $\mathcal{S}$, it also represents the n-ary functions which return the successor of the first number of their input:

$$\lambda(x_0, x_1, \ldots, x_{n-1}).\mathcal{S}(x_0).$$

The semantics of $\mathcal{PRF}$ programs is described in detail in the next section.

The only major difference between the above syntax and that of the partial recursive functions described in the previous section is the construct `Sub` which denotes the substitution of a single function rather than of a vector or list of functions. However it is shown (Sec. 4.3) that this construct can be used to define a function `Subl: prf → list prf → prf` whose syntax and behaviour correspond to the substitution of a list of functions into a function.

This syntax is defined in Coq as the inductive set `prf`.

4.2 The Semantics of $\mathcal{PRF}$

As it was stated above, programs in $\mathcal{PRF}$ are defined in order to compute n-ary partial functions which map natural numbers to natural numbers; and the semantics of such programs is given through the definition of a relation $\downarrow \subseteq (\mathcal{PRF} \times \mathit{list}\ \mathbb{N} \times \mathbb{N})$. Given a program p, a list l and a natural number x such that $(p, l, x) \in \downarrow$, we say that p *converges to* x having the list l as its input, and we use the notation $p\langle l\rangle \downarrow x$. Since a program p in $\mathcal{PRF}$ corresponds to some partial function $f_p^{(n)}$ for each arity n, the behaviour of p is defined such that

$$p\langle l\rangle \downarrow x \ \text{ if } f_p^{(n)} v_l^{(n)} = x,$$

where $v_l^{(n)}$ denotes the vector with n components which corresponds to the list l. If the length of l, denoted by $\sharp l$, is greater or equal to n, then $v_l^{(n)}$ consists of the first n elements of l, otherwise it consists of all the elements in l followed by $(n - \sharp l)$ zeros.

The predicate `converges_to: prf → list nat → nat → Prop` is inductively defined in Coq to represent the relation $\downarrow$, and thus in this text $\downarrow$ is no longer assumed to be a subset of the tuple $(\mathcal{PRF} \times \mathit{list}\ \mathbb{N} \times \mathbb{N})$, and the symbol '$\downarrow$' is used only in the notation $p\langle l\rangle \downarrow x$ as an abbreviation for `converges_to` $p\ l\ v$.

The semantics of $\mathcal{PRF}$ is illustrated below:

Zero For any list l, `Zero` converges to 0.

$$\frac{}{\texttt{Zero}\langle l\rangle \downarrow 0}$$

Successor Given any non-empty list $x : l$, `Succ` converges to the successor of x. In order that `Succ` converges for all lists, its semantics is defined such that it converges to `(S 0)` if its input is an empty list.

$$\frac{}{\texttt{Succ}\langle [\,]\rangle \downarrow \texttt{(S 0)}} \qquad \frac{}{\texttt{Succ}\langle x : l\rangle \downarrow \texttt{(S } x\texttt{)}}$$

Projections Given a list l, the projection `Proj` i converges to `zel` i l. The function `zel` is defined such that `zel` i l returns the $(i+1)$th element in l if $i < \sharp l$, otherwise it returns 0.

$$\frac{}{(\texttt{Proj}\ i)\langle l\rangle \downarrow (\texttt{zel}\ i\ l)}$$

Substitution The substitution of two functions `Sub` f g n m is based on the notion of function composition $f' \circ g'$ where the output of g' is given as the input of f'. However in the substitution `Sub` f g n m part of the input of g as well as its output is given as the input of f, If l is the input of g then the sublist having the m elements starting at offset n of the list l is given as input to g together with the output x where $g\langle l\rangle \downarrow x$. The program `Sub` f g n m converges to some value y given the input list l, if and only if:

- The program g converges to some value x for the input $l = [x_0, x_1, \ldots, x_w]$,
- f converges to y given the input $[x'_n, x'_{n+1}, \ldots, x'_{n+m-1}] ++ [x]$, where

$$\begin{array}{rcll} x'_i & = & x_i & \text{if } i <= w \\ & = & 0 & \text{if } i > w. \end{array}$$

The semantics of substitution is given by the rule:

$$\frac{g\langle l\rangle \downarrow x \quad f\langle \texttt{pcombine}\ n\ m\ l\ x\rangle \downarrow y}{(\texttt{Sub}\ f\ g\ n\ m)\langle l\rangle \downarrow y}$$

where `pcombine` n m l x represents the list made up by appending x at the end of the sublist of l containing the m elements starting at offset n:

$$\texttt{pcombine}\ n\ m\ l\ x = [\texttt{zel}\ n\ l, \texttt{zel}\ (n+1)\ l, \ldots, \texttt{zel}\ (n+m-1)\ l, x]$$

We use the notation $f \circ^n_m g$ as an abbreviation for `Sub` f g n m.

Recursion The behaviour of `Rec` β σ corresponds to the definition of the recursion operator described in section 2. If the empty list is given as the input of `Rec` β σ then it is treated as the singleton list containing 0 such that if β and σ are total functions then so is `Rec` β σ.

$$\frac{\beta\langle[]\rangle \downarrow x}{(\texttt{Rec}\ \beta\ \sigma)\langle[]\rangle \downarrow x} \qquad \frac{\beta\langle l\rangle \downarrow x}{(\texttt{Rec}\ \beta\ \sigma)\langle 0 : l\rangle \downarrow x}$$

$$\frac{(\texttt{Rec}\ \beta\ \sigma)\langle h : l\rangle \downarrow r \quad \sigma\langle h : r : l\rangle \downarrow x}{(\texttt{Rec}\ \beta\ \sigma)\langle(\texttt{S}\ h) : l\rangle \downarrow x}$$

Minimalisation The behaviour of the minimalisation `Min` f is also defined such that it corresponds to the definition of the minimalisation operator described in section 2.

$$\frac{\texttt{minl}\ (\lambda h.\texttt{converges\_to}\ f\ h : l)\ x}{(\texttt{Min}\ f)\langle l\rangle \downarrow x}$$

The term $(\lambda h.\texttt{converges\_to}\ f\ h : l)$ denotes the binary relation between natural numbers such that h relates with y if $f\langle h : l\rangle \downarrow y$. The predicate `minl: (nat → nat → Prop) → nat → Prop` is defined such that given a relation R and a natural number n, and using the notation $x \sim_R y$ to denote that $(R\ x\ y)$ holds, then `minl` R n holds if $n \sim_R 0$ holds and that for all $m < n$ there exists a $j > 0$ such that $m \sim_R j$.

A relation `allsucs` is first defined such that `allsucs` R n holds if for all $m \leq n$, there exists some k such that $m \sim_R (\texttt{S}\ k)$

$$\frac{0 \sim_R (\texttt{S}\ k)}{\texttt{allsucs}\ f\ 0} \qquad \frac{(\texttt{S}\ m) \sim_R (\texttt{S}\ k) \quad \texttt{allsucs}\ R\ m}{\texttt{allsucs}\ R\ (\texttt{S}\ m)}$$

and then `minl` is defined by the following two rules:

$$\frac{0 \sim_R 0}{\texttt{minl}\ R\ 0} \qquad \frac{(\texttt{S}\ n) \sim_R 0 \quad \texttt{allsucs}\ R\ n}{\texttt{minl}\ R\ (\texttt{S}\ n)}$$

> If R is a single-valued relation then `minl` $R\ n$ holds if n is the smallest number such that $n \sim_R 0$ and for all $m \leq n$, there exists a unique j (depending on m) such that $m \sim_R j$. The predicate `minl` R is single-valued if R is a single-valued relation; and for any two relations, R and Q, if they are equivalent $(\forall x, y.(x \sim_Q y) \Leftrightarrow (x \sim_R y))$ then so are the predicates `minl` Q and `minl` R. The first result is proved by rule induction on `minl` and the second one is proved by applying the principle of mathematical induction on the proposition $(\forall n.\texttt{minl}\ Q\ n \Leftrightarrow \texttt{minl}\ R\ n)$ assuming that Q and R are equivalent.

The language $\mathcal{PRF}$ is proved to be deterministic (i.e. the relation `converges_to` is single-valued) and in the next section it is shown how a function `Subl: prf → list prf → prf` which corresponds to the substitution of a list of functions is constructed in terms of `Sub`. It should be noted that the set `prf` cannot be defined such that it contains the construct `Subl` because the term `prf` does not occur strictly positively in `prf → list prf → prf` (see [2] page 74).

4.3 Substitution of a List of Functions

Given a program f and a list of programs $gl = [g_0, g_1, \ldots, g_{k-1}]$, `Subl` $f\ gl$ should be a program such that for all lists of natural numbers l and values $(x_0, x_1, \ldots, x_{k-1})$, $(\texttt{Subl}\ f\ gl)\langle l \rangle \downarrow y$ if:

- for all $i < k$, $g_i\langle l \rangle \downarrow x_i$, and
- $f\langle [x_0, x_1, \ldots, x_{k-1}] \rangle \downarrow y$.

Given f and gl, the value y can be computed by substituting (using `Sub`) the programs $g_0, g_1, \ldots, g_{k-1}$ into each other so that the output of each program g_m is passed, together with its input, to the next program g_{m+1}. Thus the input of each program g_i is made up of the original input of g_0 and all the outputs of the previous programs g_j for $j < i$; finally the outputs of all the programs in gl are given as the input to f. If for any program p, one can find a number n_p such that the behaviour of p depends only on the first n_p elements of the input list, then the required substitution can be constructed by:

$$\texttt{Subl}\ f\ gl = (\cdots(f \circ^{n_g}_{k-1} g_{k-1}) \circ^{0}_{n_g+k-2} \cdots \circ^{0}_{n_g+1} g_1) \circ^{0}_{n_g} g_0$$

where n_g is $\max(n_{g_0}, \ldots, n_{g_{n-1}})$ so that the behaviour of each program g_i is not altered by appending the outputs of the substituting programs appended at the end of its input.

The value of n_p can be given by a function $\eta(p)$, which we call the *natural arity* of p. This function is defined in Coq by:

```
⊢def natarity Zero = 0
              Succ = (S 0)
              Proj i = (S i)
              Sub f g n m = max (natarity g) (n + m)
              Rec b s = max (S (natarity b) (pred (natarity s))
              Min f = pred (natarity f)
```

and its significance is given by the theorem

```
⊢ ∀p l l0.
    (length nat l) = (natarity p) ⇒
    ∀x. (converges_to p l x) ⇔
        (converges_to p (l ++ l0) x)
```

The function `Subl` is then defined recursively

```
⊢_def Subl_in f m [] n          = Sub f Zero 0 0
              f m [g] n         = Sub f g m n
              f m g_1 : g_2 : gl n = Sub (Subl_in f m (g_2 : gl) (S n)) g_1 0 (m + n)
```

```
⊢_def Subl f gl = (Subl_in f (maxarity gl) gl 0)
```

```
⊢_def maxarity [g_0, ..., g_x] = max (natarity g_0 , ..., natarity g_x)
```

and it is proved that its behaviour is as required.

```
⊢ ∀f gl l x.
   (converges_to (Subl f gl) l x) ⇔
   (∃xl. (mapR prf nat (λxl. converges_to g l) gl xl) ∧
         (converges_to f xl x))
```

The relation `mapR` corresponds to the standard map function over lists, in the sense that given a relation R and two lists $l = [x_0, x_1, \ldots, x_{n-1}]$ and $l' = [x'_0, x'_1, \ldots, x'_{n'-1}]$, `mapR` $R\ l\ l'$ holds (or, $l \sim_{(\texttt{mapR}\ R)} l'$) if and only if $\sharp l = \sharp l'$ and all corresponding pairs of elements in l and l' are related to each other, i.e. $\forall i < n.x_i \sim_R x'_i$.

$$\frac{}{[] \sim_{(\texttt{mapR}\ R)} []} \qquad \frac{a \sim_R b \quad k \sim_{(\texttt{mapR}\ R)} l}{(a : k) \sim_{(\texttt{mapR}\ R)} (b : l)}$$

5 $\mathcal{PRF}$ Computability

In this section we illustrate how $\mathcal{PRF}$ programs are used as a model of computation through the definition of the notion of a computable function. The type of functions which are considered for computability are represented as single-valued relations between vectors of natural numbers and natural numbers. We first describe how vectors have been defined in Coq, then we illustrate the definition of a computable function and how particular functions can be proved to be computable.

5.1 Vectors

A vector of a set A is defined by the inductively defined set:

```
vector A ::= Vnil: (vector A 0)
           | Vcons: (n: nat) → A → (vector A n) → (vector A (S n))
```

We use the notation () to represent the empty vector `Vnil` A and $(x, \vec{v}^{(n)})$, or simply $(x, \vec{v})$, to represent `Vcons` $A\ n\ x\ \vec{v}$. Since the set `vector` $A\ n$ depends on n, in general an expression of type `vector` $A\ e_1$ cannot be defined to have the type `vector` $A\ e_2$ even if it can be proved that $e_1 = e_2$. For example although for all n, m, one can prove that $(n+m) = (m+n)$, a vector of type `vector` $A\ (n+m)$ cannot be used as having type `vector` $A\ (m+n)$. However a function `Change_arity` has been defined such that given a vector $\vec{v}$`:(vector` $A\ n$`)` and a proof t of $(n = m)$, then `Change_arity` $n\ m\ t\ A\ \vec{v}$ has type `(vector` $A\ m$`)`; and it is proved that:

```
⊢ Change_arity_eq =
    ∀n (t: (n=n)) A v⃗. (Change_arity n n t A v⃗) = v⃗
```

For instance, if $\vec{v}^{(n+m)}$ is a vector of type `vector nat` $(n+m)$ then the term

```
Change_arity (n + m) (m + n) (plus_sym n m) nat v⃗^(n+m)
```

has type `vector nat` $(m+n)$; and by rewriting with any theorem of type $((n+m) = (m+n))$ and then by `Change_arity_eq`, one can substitute the above term with $\vec{v}^{(n+m)}$ in any expression. The theorem `plus_sym` represents the commutativity of addition and has type $\forall n, m.(n+m) = (m+n)$.

The head, tail and any element of a vector is given by the relations:

$$\frac{}{\texttt{Vhd}\ A\ (\texttt{S}\ n)\ (h,\vec{t})\ h} \qquad \frac{}{\texttt{Vtl}\ A\ (\texttt{S}\ n)\ n\ (h,\vec{t})\ \vec{t}}$$

$$\frac{}{\texttt{Vel}\ A\ 0\ (\texttt{S}\ n)\ (h,\vec{t})\ h} \qquad \frac{\texttt{Vel}\ A\ i\ n\ \vec{t}\ x}{\texttt{Vel}\ A\ (\texttt{S}\ i)\ (\texttt{S}\ n)\ (h,\vec{t})\ x}$$

as well as by functions:

- $\texttt{vhd:}\ (A\texttt{: Set}) \rightarrow (n\texttt{: nat}) \rightarrow (\texttt{vector}\ A\ (\texttt{S}\ n)) \rightarrow A$,
- $\texttt{vtl:}\ (A\texttt{: Set}) \rightarrow (n\texttt{: nat}) \rightarrow (\texttt{vector}\ A\ (\texttt{S}\ n)) \rightarrow (\texttt{vector}\ A\ n)$ and
- $\texttt{vel:}\ (A\texttt{: Set}) \rightarrow (i\texttt{, }n\texttt{: nat}) \rightarrow (\texttt{Hl: } i < n) \rightarrow (\texttt{vector}\ A\ n) \rightarrow A$.

In general, properties of vectors are easier to prove if they are specified using the above relations, although terms written using the respective functions are more readable and are sometimes more useful in the proof of theorems requiring rewriting. Thus, both sets of definitions are implemented in Coq and are proved to be equivalent, so that either one is used in making the mechanisation in Coq more elegant and less laborious.

Finally a function $\texttt{vzel:}\ (i\texttt{, }n\texttt{: nat}) \rightarrow (\texttt{vector nat}\ n) \rightarrow \texttt{nat}$ which corresponds to the function `zel` over lists is also defined, and vectors are mapped into lists and vice-versa through the functions

- $\texttt{listify:}\ (A\texttt{: Set}) \rightarrow (n\texttt{: nat}) \rightarrow (\texttt{vector}\ A\ n) \rightarrow (\texttt{list}\ A)$ and
- $\texttt{vectrify:}\ (A\texttt{: Set}) \rightarrow (l\texttt{: list}\ A) \rightarrow (\texttt{vector}\ A\ (\texttt{length}\ A\ l))$.

5.2 Partial Functions

The type of partial functions of arity n mapping vectors into natural numbers, `pfunc:nat` → `Type` is defined as the dependent product type of single-valued relations between vectors and numbers. This is given by the definition of the dependent record[1]

```
⊢def pfunc arity := mk_pfunc
       { reln      : (Rel (vector nat arity) nat);
         One_valued: (one_valued2 (vector nat arity) nat reln)}
```

where `Rel` A B is the type of the relations between the sets A and B.

The field `reln` represents a relation between vectors having `arity` components and natural numbers, and the field `One_valued` is a proof that `reln` is single-valued.

The type of all partial functions is then defined as the dependent product

```
⊢def pfuncs ::= Pfuncs: (n: nat) → (pfunc n) → pfuncs
```

A function $g\texttt{:}(\texttt{vector nat}\ n) \rightarrow \texttt{nat}$ defined in Coq can be used to specify an object of type `pfunc` n since f obviously describes a single-valued relation R such that $\forall \vec{v}, x. \vec{v} \sim_R x \Leftrightarrow g(\vec{v}) = x$. This is achieved through a function $\texttt{pfuncize:(arity: nat)} \rightarrow \texttt{((vector nat arity)} \rightarrow \texttt{nat)} \rightarrow \texttt{(pfunc arity)}$.

5.3 Computable Functions

A $\mathcal{PRF}$ program p is said to compute an n-ary partial function $f\texttt{:}(\texttt{pfunc}\ n)$ if for all vectors $\vec{v}$ and natural number x, the relation in f holds if and only if the program p converges to x with input $l_{\vec{v}} = \texttt{listify nat}\ n\ \vec{v}$.

```
⊢def computes p n f =
  ∀v⃗ x. (reln n f v⃗ x) ⇔
          (converges_to p (listify nat n v⃗) x)
```

[1] A record f of type `pfunc` n, is constructed by `mk_pfunc` R H where R has type `Rel (vector nat` n`) nat` and H has type `(one_valued2 (vector nat` n`) nat` R`)`. The functions `reln` and `One_valued` select R and H respectively from f.

and a function is said to be computable if there is some program which computes it.

$\vdash_{def}$ `computable` n f `=` $\exists p$. `computes` p n f

A predicate P is proved to be decidable by showing that its characterstic function f_P given by:

$$\begin{aligned} f_P(\vec{v}) &= 1, \text{if } P(\vec{v}) \\ &= 0, \text{otherwise} \end{aligned}$$

is computable.

Given a single-valued relation R_f which constructs the partial function f: `pfunc` n, a proof that f is computable involves the construction of a $\mathcal{PRF}$ program p: `prf` such that:

$$\forall \vec{v}, x. p\langle \texttt{listify } \vec{v}\rangle \downarrow x \Leftrightarrow \vec{v} \sim_{R_f} x.$$

This proof is relatively straightforward if R_f is specified through some Coq function g_f (in the sense that f `= pfuncize` n g_f) since obviously

$$\forall \vec{v}, x. \vec{v} \sim_{R_f} x \Leftrightarrow \texttt{reln } n\ f\ \vec{v}\ x \Leftrightarrow g_f(\vec{v}) = x$$

and the equivalence

$$\forall \vec{v}, x. p\langle \texttt{listify } \vec{v}\rangle \downarrow x \Leftrightarrow g_f(\vec{v}) = x$$

can be proved by showing that:

$$\forall \vec{v}. p\langle \texttt{listify } \vec{v}\rangle \downarrow g_f(\vec{v}).$$

The required equivalence follows by applying the fact that R_f is total and that `computes_to` is a single-valued relation.

The following tables list a number of functions which are proved to be computable.

5.4 Basic Functions

Function name	Definition	$\mathcal{PRF}$ program
Undefined	$Undef(\vec{v}) = \perp$	`Diverges_all = Min Succ`
Zero	$\mathcal{Z}(\vec{v}) = 0$	`Zero`
Successor	$\mathcal{S}(x) = x + 1$	`Succ`
Projections	$\mathcal{U}_n^i(x_0, \ldots, x_{n-1}) = x_i$	`Proj` i, if $i < n$ `Diverges_all`, if $i \geq n$

5.5 Rearrangement

The function `Rarr:` `prf` $\rightarrow$ `list nat` $\rightarrow$ `prf` is defined by

$\vdash_{def}$ `Rarr` f `[`i_0`,` i_1`, ...,` i_{n-1}`] =`
    `Subl` f `[Proj` i_0`, Proj` i_1`, ..., Proj` i_{n-1}`]`

such that if we define the list $l_i = [i_0, i_1, \ldots, i_{n-1}]$ then, if $f\langle[x_0, x_1, \ldots, x_{m-1}]\rangle \downarrow r$ then `Rarr` f l_i converges to the same value r if it is given the list which is made up by rearranging the elements in $[x_0, x_1, \ldots, x_{m-1}]$ according to the values in l_i:

$$\texttt{Rarr } f\ l_i\langle[x_{i_0}, x_{i_1}, \ldots, x_{i_{n-1}}]\rangle \downarrow r.$$

5.6 Arithmetic

Identity	$\iota_{\mathbb{N}}(x) = x$	`Identity = Proj 0`

Constants	$C_n(\vec{v}) = n$	`Constant 0 = Zero`
		`Constant (S` n`)`
		`= Subl Succ [Constant` n`]`

Addition	$0 + x_1 = x_1$
	$\mathcal{S}(x_0) + x_1 = \mathcal{S}(x_0 + x_1)$
		`Add = Rec Identity`
		`Rarr Succ [1]`

Multiplication	$0 \times x_1 = 0$
	$\mathcal{S}(x_0) \times x_1 = x_1 + (x_0 \times x_1)$
		`Multiply = Rec Zero`
		`Rarr Add [2, 1]`

Power	$x_1^0 = 1$
	$x_1^{\mathcal{S}(x_0)} = x_1 \times (x_1^{x_0})$
		`Power' = Rec (Constant 1)`
		`Rarr Multiply [2, 1]`
		`Power = Rarr Power' [1, 0]`

		Note that `Power'`$\langle[x_0, x_1]\rangle \downarrow x_1^{x_0}$

Predecessor	$\mathrm{pred}(0) = 0$
	$\mathrm{pred}(\mathcal{S}(x)) = x$
		`Pred = Rec Zero`
		`Identity`

Subtraction	$x_1 - 0 = x_1$
	$x_1 - \mathcal{S}(x_0) = \mathrm{pred}(x_1 - x_0)$
		`Subtract' = Rec Identity`
		`(Rarr Pred [1])`
		`Subtract = Rarr Subtract' [1, 0]`

Difference	$\mathrm{diff}(0, x_1) = x_1$
	$\mathrm{diff}(\mathcal{S}(x_0), 0) = \mathcal{S}(x_0)$
	$\mathrm{diff}(\mathcal{S}(x_0), \mathcal{S}(x_1)) = \mathcal{S}(x_0)$
		`Difference`
		`= Subl Add [Subtract, Subtract']`

		Since $\mathrm{diff}(x, y) = (x - y) + (y - x)$

5.7 Boolean Operations

Conditional	if 0 then x_1 else $x_2 = x_2$
	if $\mathcal{S}(x_0)$ then x_1 else $x_2 = x_1$
		`Cond = Rec (Proj 1)`
		`(Proj 2)`

Boolean identity	$\iota_2(0) = 0$
	$\iota_2(\mathcal{S}(x)) = 1$
		`Bid = Subl Cond [Identity,`
		`Constant 1, Zero]`

Negation	$\neg_{\mathbb{N}}(0) = 1$
	$\neg_{\mathbb{N}}(\mathcal{S}(x)) = 0$
		`Neg = Subl Cond [Identity,`
		`Zero, Constant 1]`

Conjunction	$0 \wedge_{\mathbb{N}} x_1 = 0$
	$\mathcal{S}(x_0) \wedge_{\mathbb{N}} x_1 = \iota_2(x_1)$
		`Conj = Subl Cond [Proj 0,`
		`Rarr Bid [1],`
		`Zero]`

Disjunction	$0 \vee_{\mathbb{N}} x_1 = \iota_2(x_1)$
	$\mathcal{S}(x_0) \vee_{\mathbb{N}} x_1 = 1$
		`Disj = Subl Cond [Proj 0,`
		`Constant 1,`
		`Rarr Bid [1]]`

5.8 Predicates on Natural Numbers

Is zero	$\text{is0}(0) = 1$ $\text{is0}(\mathcal{S}(x)) = 0$	`Is0 = Neg`
Non zero	$\text{non0}(0) = 0$ $\text{non0}(\mathcal{S}(x)) = 1$	`Non0 = Bid`
Equality	$0 =_{\mathbb{N}} 0 = 1$ $0 =_{\mathbb{N}} \mathcal{S}(x_1) = 0$ $\mathcal{S}(x_0) =_{\mathbb{N}} 0 = 0$ $\mathcal{S}(x_1) =_{\mathbb{N}} \mathcal{S}(x_1) = x_0 =_{\mathbb{N}} x_1$	`Equal = Subl Is0 [Difference]` Since $x =_{\mathbb{N}} y = \text{is0}(\text{diff}(x,y))$
Inequality	$0 \neq_{\mathbb{N}} 0 = 0$ $0 \neq_{\mathbb{N}} \mathcal{S}(x_1) = 1$ $\mathcal{S}(x_0) \neq_{\mathbb{N}} 0 = 1$ $\mathcal{S}(x_1) \neq_{\mathbb{N}} \mathcal{S}(x_1) = x_0 \neq_{\mathbb{N}} x_1$	`Different = Subl Non0 [Difference]` Since $x \neq_{\mathbb{N}} y = \text{non0}(\text{diff}(x,y))$
Less than	$0 <_{\mathbb{N}} 0 = 0$ $0 <_{\mathbb{N}} \mathcal{S}(x_1) = 1$ $\mathcal{S}(x_0) <_{\mathbb{N}} 0 = 0$ $\mathcal{S}(x_1) <_{\mathbb{N}} \mathcal{S}(x_1) = x_0 <_{\mathbb{N}} x_1$	`Less = Subl Non0 [Subtract']` Since $x <_{\mathbb{N}} y = \text{non0}(x - y)$
Greater than	$x_0 >_{\mathbb{N}} x_1 = x_1 <_{\mathbb{N}} x_0$	`Greater = Rarr Less [1, 0]`
Less or equal	$x_0 \leq_{\mathbb{N}} x_1 = \neg_{\mathbb{N}}(x_0 >_{\mathbb{N}} x_1)$	`Less_eq = Subl Neg [Greater]`
Greater or equal	$x_0 \geq_{\mathbb{N}} x_1 = \neg_{\mathbb{N}}(x_0 <_{\mathbb{N}} x_1)$	`Greater_eq = Subl Neg [Less]`

5.9 First Occurence

Given a partial function f, (first f) is defined as the first natural number n such that $f(n) > 0$ and for all $m \leq n$, $f(m)$ is defined. If no such number exists, (first f) is undefined. If f_P is the characteristic function of some unary predicate P, then (first f_P) returns the first number n such that $P(n)$ holds.

First that	first f $=$ `minl` $(\lambda m, n.\text{is0}(f(n)) = m)$	`First_that` p `= Min (Subl Is0 [`p`])`

5.10 Division

The quotient and remainder of a division operator can be defined by the partial functions div $: \mathbb{N}^2 \rightarrow \mathbb{N}$ and mod $: \mathbb{N}^2 \rightarrow \mathbb{N}$ such that:

$$\begin{aligned} \text{div}\,(n,m) &= q, \text{if } \exists r < m.qm + r = n \\ \text{mod}\,(n,m) &= r, \text{if } r < m \text{ and } \exists q.qm + r = n \end{aligned}$$

Otherwise, one can define total, primitive recursive functions:

$$\begin{aligned} \text{modt}\,(0,m) &= 0 \\ \text{modt}\,(\mathcal{S}(n),m) &= 0, \text{if modt}\,(n,m) + 1 = m \\ &= \text{modt}\,(n,m) + 1, \text{if modt}\,(n,m) + 1 \neq m \end{aligned}$$

$$\begin{aligned} \text{divt}\,(0,m) &= 0 \\ \text{divt}\,(\mathcal{S}(n),m) &= \text{divt}\,(n,m) + 1, \text{if modt}\,(n,m) + 1 = m \\ &= \text{divt}\,(n,m), \text{if modt}\,(n,m) + 1 \neq m \end{aligned}$$

such that $\text{divt}\,(n,0) = 0$ and $\text{modt}\,(n,0) = n$.

The partial functions div and mod are defined in Coq as the predicates `div` and `mod` respectively, and the above total functions as the functions `modt` and `divt`. The predicates are then used to specify the partial functions `pf_div:pfunc 2` and `pf_mod:pfunc 2`. The program `Divide:prf`

$\vdash_{def}$ `Divide = Subl Pred [First (Subl Less [Proj 1, Rarr Multiply [0, 2]])]` computes `pf_div` by calculating the predecessor of the first q such that $n < qm$, given a list $n : m : l$ as input. Also, since

$$\mathrm{mod}\,(n, m) = n - (m \times \mathrm{div}\,(n, m))$$

the program

```
⊢def Mod = Subl Subtract [Proj 0, Subl Multiply [Proj 1, Divide]]
```

computes `pf_mod`.

6 Coding Programs and the S^m_n Theorem

6.1 Coding Domains

A coding of a set A can be obtained by defining two *effective* and injective functions $\alpha : A \to \mathbb{N}$ and $\beta : \mathbb{N} \to A$ such that α is total, α and β are inverses of each other and the predicate n *is in the range of* α is decidable. By the term 'effective', it is meant that the function is computable in some informal sense, and such notion is not defined in the implementation in Coq; although if one defines a function f: $A \to$ `nat`, it can be assumed that f is an effective mapping. Also given two functions f: $A \to B$ and g: $B \to A$, the predicate g *is the inverse of* f is defined such that it holds if and only if $\forall a.g(f(a)) = a$. Since functions in Coq are necessarily well formed, if g is the inverse of f, then f is injective and g is surjective. The predicate b *is in the range of* f is then given by $\exists a.f(a) = b$. This predicate is specified in Coq by `in_range: (A, B: Set) -> (A -> B) -> B -> Prop`, and the range of f is decidable if

$$\forall b.\{\texttt{in\_range}\ f\ b\}\ \texttt{+}\ \{\neg\texttt{in\_range}\ f\ b\}.$$

The range of a function is obviously decidable if the function is surjective.

6.2 Coding Pairs

The bijection $\pi : \mathbb{N}^2 \to \mathbb{N}$ which maps pairs of natural numbers into natural numbers is represented in Coq by the curried function `pi: nat → nat → nat` defined as follows:

```
⊢def pi  0      0     = 0
        (S n)  0     = S (S (pi n 0 + n)
         n    (S m)  = S (pi n m + n + m)
```

The inverse of π is given by the two functions $\pi_1^{-1} : \mathbb{N} \to \mathbb{N}$ and $\pi_2^{-1} : \mathbb{N} \to \mathbb{N}$ defined mutually inductively in Coq by

```
⊢def pi1  0      = 0
         (S n)  = if_equal nat (pi2 n) 0 0 (S (pi1 n))

     with

     pi2  0      = 0
         (S n)  = if_equal nat (pi2 n) 0 (S (pi1 n)) (pred (pi2 n))
```

where the term `if_equal` $A\ n\ m\ a\ b$ is equal to a_A if $n =_{nat} m$, and is equal to b_A if $n \neq_{nat} m$; and it is proved that

```
⊢ ∀n. pi (pi1 n) (pi2 n)) = n

⊢ ∀n m. pi1 (pi n m) = n

⊢ ∀n m. pi2 (pi n m) = m
```

As a result, these functions are used to define the bijective functions `vf_pi: (vector nat 2) -> nat` and `vf_invpi: nat -> (vector nat 2)` which represent an effective coding of the set `vector nat 2`. The relation `pfuncize vf_pi` is computable, and so are the objects of type `pfunc (S 0)` which represent `pi1` and `pi2`.

Moreover, by nesting π (and π_1^{-1} and π_2^{-1}) it is possible to define effective codings for any size of vectors. The function

$$\xi(w, x, y, z) = \pi(\pi(w, x)), (\pi(y, z)))$$

with inverse projections ξ_1^{-1}, ξ_2^{-1}, $\xi_3^{-1}$and $\xi_4^{-1}$is used in the following section.

6.3 Coding Programs

An effective coding of $\mathcal{PRF}$ programs can be given by the functions $\gamma : \mathcal{PRF} \to \mathbb{N}$ and $\mathcal{P} : \mathbb{N} \to \mathcal{PRF}$:

$$\begin{array}{rlcl}
\gamma : & \texttt{Zero} & \mapsto & 0 \\
 & \texttt{Succ} & \mapsto & 1 \\
 & \texttt{Proj}\ i & \mapsto & i \times 4 + 2 \\
 & \texttt{Sub}\ f\ g\ n\ m & \mapsto & \xi(\gamma(f), \gamma(g), n, m) \times 4 + 3 \\
 & \texttt{Rec}\ f\ g & \mapsto & \pi(\gamma(f), \gamma(g)) \times 4 + 4 \\
 & \texttt{Min}\ f & \mapsto & \gamma(f) \times 4 + 5
\end{array}$$

$$\begin{array}{rclll}
\mathcal{P} : & 0 & \mapsto & \texttt{Zero} & \\
 & 1 & \mapsto & \texttt{Succ} & \\
 & n+2 & \mapsto & \texttt{Proj}\ d, & \text{if } m = 0 \\
 & & \mapsto & \texttt{Sub}\ \mathcal{P}(\xi_1^{-1}(d))\ \mathcal{P}(\xi_2^{-1}(d))\ \xi_3^{-1}(d)\ \xi_4^{-1}(d), & \text{if } m = 1 \\
 & & \mapsto & \texttt{Rec}\ \mathcal{P}(\pi_1^{-1}(d))\ \mathcal{P}(\pi_2^{-1}(d)), & \text{if } m = 2 \\
 & & \mapsto & \texttt{Minl}\ \mathcal{P}(d), & \text{if } m = 3 \\
 & & & \text{where } d = \operatorname{div}(n+2, 4) & \\
 & & & \qquad\quad m = \operatorname{mod}(n+2, 4) &
\end{array}$$

We use the notation $\mathcal{P}_n$ to denote the program $\mathcal{P}(n)$. The function γ is clearly primitive recursive and is defined in Coq by the function `Godel: prf` $\to$ `nat`. Also, since $\mathcal{P}$ is applied recursively to values which are always less than the original value (note that for all n, $\pi_1^{-1}n \leq n$ and $\pi_2^{-1}n \leq n$), then it is well formed and is defined as follows:

```
⊢def Prog     0          = Zero
             (S 0)       = Succ
             (S (S n))   = (λd, r.
         if_equal prf r 0 (Proj d)
         if_equal prf r 1 (Sub (calc_Prog n (pi1 (pi1 d)) (Prog n))
                               (calc_Prog n (pi2 (pi1 d)) (Prog n))
                               (pi1 (pi2 d)) (pi2 (pi2 d)))
         if_equal prf r 2 (Rec (calc_Prog n (pi1 d) (Prog n))
                               (calc_Prog n (pi2 d) (Prog n)))
                          (Min (calc_Prog n d (Prog n))))
              (divt n 4) (modt n 4)

with

calc_Prog 0      r p = p
          (S m)  r p = if_equal prf r m (Prog m) (calc_Prog m r p)
```

such that

```
⊢ ∀n m p. (m < n) ⇒ (calc_Prog n m p) = Prog m
```

and

$\vdash$ $\forall n\ m\ p.\ (n \leq m) \Rightarrow$ (calc_Prog $n\ m\ p$) = p

The function Prog is proved to be the inverse of Godel by induction on the structure of prf; and Godel is proved to be the inverse of Prog by strong mathematical induction[2] on n and by case analysis over the mutually exclusive cases:

$$\begin{aligned}&\forall n.(n=0) \vee (n=1) \vee ((n \geq 2) \wedge \\ &\quad ((\mathrm{mod}\,(n,4)=0) \vee (\mathrm{mod}\,(n,4)=1) \vee \\ &\quad (\mathrm{mod}\,(n,4)=2) \vee (\mathrm{mod}\,(n,4)=3)))\end{aligned}$$

The functions Godel and Prog constitute an effective coding for the set prf. Note that all the functions used in the definition of these two functions are proved to be computable; the computability of these functions is required for the proof of the S_n^m theorem.

We also define the function $\phi_e^{(n)}$ as the n-ary function which is computed by the program $\mathcal{P}_e$. Since γ is surjective, any n-ary computable function is *equivalent* to some function $\phi_e^{(n)}$. This is represented in Coq by the partial function pf_compute_Prog: pfunc n e which is constructed from the single-valued relation fcompute_Prog:

$\vdash_{def}$ fcompute_Prog n e = $\lambda\vec{v}$. (converges_to (Prog e) (listify nat n $\vec{v}$))

6.4 The S_n^m Theorem

The S_n^m theorem, also called the *parametrisation* theorem, states that for any $(m+n)$-ary function $\phi_e^{(m+n)}$, one can find an equivalent n-ary function $\phi_s^{(n)}$, such that s can be computed from m, n, e and the first m parameters of $\phi_e^{(m+n)}$. In other words, for all m, n there is a total computable $(m+1)$-ary function s_n^m such that:

$$\forall e, \vec{x}, \vec{y}. \phi_e^{(m+n)}(\vec{x}, \vec{y}) = \phi_{s_n^m(e,\vec{x})}^{(n)}(\vec{y}).$$

Given the numbers m, n and e, and the vector $\vec{x} = (x_0, x_1, \ldots, x_{m-1})$, then the function $\phi_{s_n^m(e,\vec{x})}^{(n)}$ can be computed by the program constructed by substituting the m constant programs Constant x_0, Constant x_1, ..., Constant x_{m-1}, and the projections Proj 0, Proj 1, ..., Proj $n-1$ into the program coded by e, $\mathcal{P}_e$. This program is defined in Coq by:

$\vdash_{def}$ smnprf m n e xl
    = Subl (Prog e) ((constants [xl_0, xl_1, ... , xl_{m-1}]) ++ (projections n)

where xl is the list listify $\vec{x}$ and xl_i is the $(i+1)$th element in xl. The functions constants and projections are the lists of $\mathcal{PRF}$ programs defined such that:

constants [x_0, x_1, ..., x_{m-1}] = [Constant x_0, Constant x_1, $\vdots$, Constant x_{m-1}]

projections n = [Proj 0, Proj 1, $\vdots$, Proj $n-1$]

The function s_n^m is then defined as the function pf_smnprf: (pfunc m)

$\vdash_{def}$ vf_smnprf m n $\vec{v}$)
  = (Godel (smnprf m n (vhd nat m v) (listify nat m (vtl nat m v))))

$\vdash_{def}$ pf_smnprf = λm,n.(pfuncize (S m) (vf_smnprf m n))

This function is obviously total since it is defined using the function pfuncize. Also, by proving the following theorems expressing the behaviour of constants and projections

$\vdash$ $\forall l\ lm$. (mapR prf nat (λg.(converges_to g l)) (constants lm) lm)

$\vdash$ $\forall l\ n$. $\sharp l$ = $n \Rightarrow$
    (mapR prf nat (λg.(converges_to g l)) (projections n) l)

[2] $\forall n.P(n)$ can be deduced from $\forall n.(\forall m.m < n \Rightarrow P(m)) \Rightarrow P(n)$. This principle is given by the theorem lt_wf_ind proved in Wf_nat.v.

the function `pf_smnprf` m n is proved to be as required:

$$\begin{array}{l} \vdash \lambda m\ n\ e\ \vec{x}\ \vec{y}\ k\ z. \\ \quad \texttt{(reln (S } m\texttt{) (pf\_smnprf } m\ n\texttt{) (Vcons nat } m\ e\ \vec{x}\texttt{) } k\texttt{)} \Rightarrow \\ \quad \texttt{(reln (plus } m\ n\texttt{) (pf\_compute\_Prog (plus } m\ n\texttt{) } e\texttt{) } (\vec{x},\vec{y})\texttt{ z)} \Leftrightarrow \\ \quad \texttt{(reln } n\texttt{ (pf\_compute\_Prog } n\ k\texttt{) } \vec{y}\ z\texttt{)} \end{array}$$

Since the functions used in the definition of `Godel` are proved to be computable, the function `pf_smnprf` m n is also computable. The proof of this results is done as follows:

1. Since for all c, $\eta(\texttt{Constant}\ c) = 0$; and for all i, $\eta(\texttt{Proj}\ i) = i$, it can be shown that

   $$\vdash \forall m\ n.\ \texttt{maxarity (pf\_smnprf } m\ n\texttt{) = } n$$

2. For any list $l = [y_0, y_1, \ldots, y_{k-1}]$, and natural numbers n, n', there exists some $\mathcal{PRF}$ program which computes the function

   $$\gamma(\texttt{Subl\_in}\ \mathcal{P}_e\ n\ \texttt{[Proj}\ y_0\texttt{, Proj}\ y_1\texttt{, }\ldots\texttt{, Proj}\ y_{k-1}\texttt{]}\ n').$$

   for any number e. This is proved by induction on l.

3. For all c, the function $\gamma(\texttt{Constant}\ c)$ is computable. This result is proved by mathematical induction on c and is needed in the proof of the next step.

4. For any list of projections programs l_p, and for any list of natural numbers l, and numbers n, n', there is some $\mathcal{PRF}$ program which computes the function

   $$\gamma(\texttt{Subl\_in}\ \mathcal{P}_e\ n\ \texttt{((constants}\ l\texttt{)}\ {+}{+}\ l_p\texttt{)}\ n').$$

   for any number e. This is proved by mathematical induction on the length of l, the base case being step 2 above.

5. The required theorem is a generalisation of the previous step, where

   - $l_p = [\texttt{Proj}\ 0, \texttt{Proj}\ 1, \ldots, \texttt{Proj}\ n-1]$
   - $n' = \texttt{maxarity (pf\_smnprf}\ m\ n\texttt{)} = n$ (by step 1)
   - l is the tail of the input of `pf_smnprf` m n

   $$\vdash \forall m\ n.\ \texttt{computable (S}\ m\texttt{) (pf\_smnprf}\ m\ n\texttt{))}$$

The proof of the S^m_n theorem, as well as all the proofs implemented in the mechanisation in Coq, does not involve the axiom of the excluded middle. Other theorems in the theory of computation are also expected to be constructive, although however, the literature of computability does contain theorems whose proof requires classical reasoning. An example of this is the proof of the existence of an uncomputable function given in Cutland [3].

7 Conclusions

The mechanisation illustrated above includes the defintion of computable function, the proof of the computability of a number of particular functions, an effective coding of partial recursive functions on natural numbers, and finally the proof of the S^m_n theorem. The proofs of the theorems derived in this implementation tend to be quite elaborate and involve the consideration of details often omitted in proofs given in mathematical texts. However, this mechanisation shows that the Coq theorem prover is a robust system and is suitable for the mechanisation of mathematical and 'real world' theories.

An advantage of using a theorem prover based on a powerful type theory, like the calculus of constructions in Coq, over a theorem prover which is based on a simpler logic (for example HOL which is based on a polymorphic version of Church's simple theory of types) is the availability of dependent types. In this report we have seen how dependent types are used in the definition of partial functions as single valued relations (section 5.2) for instance, and in general, mathematical concepts can be naturally defined as dependent objects (e.g., vectors, matrices, etc.). An apparent disadvantage of using Coqover HOL is the difficulty needed in extended the Gallina language. HOL users can implement their own tactics and inference rules easily, however the implementation of a new tactic in Coq requires the non-trivial task of extending the Gallina language with a new construct. The effect of this disadvantage is however relieved by the power of the calculus of constructions as the underlying logic of Coq. In fact, during the implementation described in this report, no need was felt for implementing new tactics which would somehow facilitate the mechanisation considerably. The results of the comparative study of Coqand HOL will be published in more detail in [14].

References

[1] Thierry Coquand and Gérard Huet. The calculus of constructions. Rapport de Recherche 530, INRIA, Rocquencourt, France, May 1986.

[2] C. Cornes et al. The Coq Proof Assistant Reference Manual, Version 5.10. Rapport technique RT-0177, INRIA, 1995.

[3] N.J. Cutland. *Computability: An introduction to recursive function theory.* Cambridge University Press, 1980.

[4] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[5] M. Gordon. HOL a machine oriented formulation of higher order logic. Technical Report TR-68, Computer Laboratory, Cambridge University, July 1985.

[6] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[7] Per Martin-Löf. *Intuitionistic Type Theory.* Bibioplois, Napoli, 1984. Notes of Giowanni Sambin on a series of lectues given in Padova.

[8] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf type theory: an introduction.* Clarendon, 1990.

[9] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[10] H. Rogers. *Theory of recursive functions and effective computability.* McGraw-Hill, 1967.

[11] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[12] G.J. Tourlakis. *Computability.* Reston Publishing Company, 1984.

[13] Vincent Zammit. A mechanisation of computability theory in HOL. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 431–446, Turku, Finland, August 1996. Springer-Verlag.

[14] Vincent Zammit. A comparative study of Coq and HOL. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Bell Labs, New Jersey, US, August 1997. Springer-Verlag.