A Mechanisation of Computability Theory in HOL

Vincent Zammit

Computer Laboratory, University of Kent, United Kingdom

Abstract. This paper describes a mechanisation of computability theory in HOL using the Unlimited Register Machine (URM) model of computation. The URM model is first specified as a rudimentary machine language and then the notion of a computable function is derived. This is followed by an illustration of the proof of a number of basic results of computability which include various closure properties of computable functions. These are used in the implementation of a mechanism which partly automates the proof of the computability of functions and a number of functions are then proved to be computable. This work forms part of a comparative study of different theorem proving approaches and a brief discussion regarding theorem proving in HOL follows the description of the mechanisation.

1 Introduction

The theory of computation is a field which has been widely explored in mathematical and computer science literature [4, 12, 13] and several approaches to a standard model of computation have been attempted. However, each exposition of the theory centres on the basic notion of a computable function, and as such, one of the main objectives of a mechanisation of computability in a theorem prover is the formal definition of such functions. The mechanisation illustrated in this paper includes also the proof of a number of basic results of the theory and the implementation of conversions and other verification tools which simplify further development of the mechanisation.

This work is part of a comparative study of an LCF [9] style theorem proof assistant (namely the HOL system [5, 6]), and a non-LCF style theorem proving environment based on constructive type theory (such as the ALF system [1, 7] and the Coq proof assistant [3]). 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 verification styles are also compared to the way proofs of mathematical results are represented in texts. It is expected that this comparative study will contribute to the identification of possible enhancements to the theorem proving styles. Although it is not the scope of this paper to give full details of this comparative study, a brief discussion regarding theorem proving in HOL is given after the description of the implementation.

This particular mechanisation of the theory is based on the URM model [11] of computation and much of the implementation is based on the definitions and results described in [4]. The next section gives a brief discussion on URM computability, however it is strongly suggested that the interested reader consults the literature ([4, 12, 13]). The rest of the paper illustrates the actual mechanisation and includes the specification of the notion of a computable function in HOL, the proof of a number of results of computability and a mechanism for constructing and proving the computability of functions.

2 The URM Model of Computation

An Unlimited Register Machine (URM) consists of a countably infinite set of registers, usually referred to as the memory or store, each containing a natural number. The registers are numbered R_0, R_1, $\ldots$, R_n, $\ldots$, and the value stored in R_n is specified by r_n. A URM executes a finite program constructed from the following four different types of instructions:

Zero: `ZR` n sets r_n to 0.
Successor: `SC` n increments r_n.
Transfer: `TF` n m copies r_n to R_m.
Jump: `JP` n m p jumps to the pth instruction (starting from 0) of the program if $r_n = r_m$.

A program counter keeps track of the current point in program execution, and the configuration of a URM is given by a pair (p, r) consisting of the program counter and the current store. A configuration is said to be *initial* if the program counter is set to the index of the first instruction, and it is said to be *final* if the program counter exceeds the index of the last instruction.

Given a program P and an initial configuration c_0, a *computation* is achieved by executing the instructions of the program one by one altering the URM configuration at each step. An execution step of a URM with a final configuration has no effect on the current configuration. A computation is thus an infinite sequence of configurations $\langle c_0, c_1, c_2, \ldots\rangle$ and is denoted by $P\langle c_0\rangle$, or simply by $P(r)$ where $c_0 = (0, r)$. The store r is usually represented by a sequence of register values $(r_0, r_1, \ldots)$ and a finite sequence $(r_0, r_1, \ldots, r_n)$ represents the store where the first $n+1$ registers are given by the sequence and the rest contain the value 0, which is the initial value held in each register. We use the notation $P\langle c\rangle \rightarrow_n c'$ to express that P alters the URM state from c to c' in n steps.

A computation is said to *converge* if it reaches a final configuration, otherwise it is said to *diverge*. The *value* of a convergent computation is given by the contents of the first register R_0 of the final configuration.

The computation of a program can be used to define an n-ary partial function by placing the parameters in the first n registers of a cleared[1] URM store and then executing the program returning the contents of the first register as the function value. Formally, a program P is said to compute an n-ary function f

[1] all registers containing 0.

if, for every $a_0, \ldots, a_{n-1}$ and v, $P(a_0, \ldots, a_{n-1})$ converges to v if and only if $f(a_0, \ldots, a_{n-1}) = v$. This definition implies that $P(a_0, \ldots, a_{n-1})$ diverges if and only if $f(a_0, \ldots, a_{n-1})$ is undefined. A function is said to be *URM-computable* if there is a program which computes it.

The URM model of computation is proved to be equivalent to the numerous alternative models such as the Turing machine model, the Gödel-Kleene partial recursive functions model, and Church's lambda calculus [4, 11] in the sense that the set of URM computable functions is identical to the set of the functions computed by any other model.

3 Mechanisation of URM Computability

An Unlimited Register Machine can be regarded as a simple machine language and as such its formal specification in HOL is similar to that of real world architectures [14].

3.1 The URM Instruction Set

A URM store can be represented as a function from natural numbers to natural numbers and configurations as pairs consisting of a natural number signifying the program counter and a store,

```
store   == :num → num
config  == :num × store
```

The syntax of the URM instruction set is specified through the definition of the type `:instruction` using the type definition package [8] of HOL

```
instruction ::= ZR num
              | SC num
              | TF num → num
              | JP num → num → num
```

and programs are defined as lists of instructions.

The semantics of the instruction set is then specified through the definition of a function `exec_instruction: instruction -> config -> config` such that given an instruction i and a configuration c, `exec_instruction` i c returns the configuration achieved by executing i in configuration c.

```
⊢def (∀n c. exec_instruction (ZR n) c
       = (SUC (FST c),(λx. (x = n) → 0 | (SND c x)))) ∧
     (∀n c. exec_instruction (SC n) c
       = (SUC (FST c),
             (λx. (x = n) → (SUC (SND c n)) | (SND c x)))) ∧
     (∀n m c. exec_instruction (TF n m) c
       = (SUC (FST c),(λx. (x = m) → (SND c n) | (SND c x)))) ∧
     (∀n m p' c. exec_instruction (JP n m p') c
       = (((SND c n = SND c m) → p' | (SUC (FST c))),SND c))
```

The execution of a number of steps of a URM program is then given by the primitive recursive function `EXEC_STEPS: num -> program -> config -> config`, such that `EXEC_STEPS` n P c_0 = c_1 if and only if $P\langle c_0\rangle \rightarrow_n c_1$

```
⊢def (∀P c. EXEC_STEPS 0 P c = c) ∧
     (∀n P c. EXEC_STEPS (SUC n) P c
       = EXEC_STEPS n P (EXEC_STEP P c))
```

where `EXEC_STEP: program -> config -> config` represents one step execution of a given program,

```
⊢def ∀P c. EXEC_STEP P c
       = ((Final P c) → c | (exec_instruction (EL (FST c) P) c))
```

and the predicate `Final: program -> config -> bool` holds for final configurations.

3.2 Computations

A finite list of natural numbers is transformed into an initial URM configuration by the function `set_init_conf: (num list) -> config`, and `CONVERGES: program -> (num list) -> num -> bool` and `DIVERGES: program -> (num list) -> bool` represent converging and diverging computations respectively. It is shown that a program converges to a unique value unless it diverges.

```
⊢ ∀P l. (∃!v. CONVERGES P l v) ∨ DIVERGES P l
```

3.3 Computable Functions

Since the functions which are considered are not necessarily total, a polymorphic type of *possibly undefined values* is defined. Elements of this type are either undefined or have a single value:

```
'a PP  ::=  Undef
        |   Value 'a
```

The domain of functions is then chosen to be the type of possibly partial numbers. Since the functions have different arities, the codomain is chosen to be the type of lists of numbers, where the length of the list represents the function's arity:

```
pfunc == :num list → num PP
```

A program computes a function if and only if it converges to the value of the application of the function whenever this is defined,

```
⊢def ∀n P f. COMPUTES n P f
       = (∀l v. (LENGTH l = n) ⇒
           (CONVERGES P l v = (f l = Value v)))
```

such that `COMPUTES` n P f holds if P computes the n-ary function f. Finally, a function is computable if there is a program which computes it.

```
⊢def ∀n f. COMPUTABLE n f = (∃P. COMPUTES n P f)
```

3.4 Manipulating URM Programs

The proof that a particular function is computable usually involves the construction of a URM program which computes it. The URM instruction set is rudimentary and it would be impractical as a general purpose programming language without a mechanism for concatenating program segments, and without a number of program modules performing simple but often used tasks.

An operator $\widehat{++}$ can be defined such that, given two programs P_1 and P_2, the computation of $P_1 \widehat{++} P_2$ is given by the individual computation of the two programs. This is achieved by first adding the length of P_1 to the destination of the jumps in P_2 and then appending the two programs together using the normal list concatenation function. The destination of the jumps in P_2 need to be altered since URM jump instructions are absolute, rather than relative. However, in order that the required property is achieved, the programs must be in *standard form*, in the sense that the destinations of all their jumps are less than or equal to the length of the program. This is required so that the program counter of any final configuration is equal to the length of the program; in particular the program counter of a final configuration of P_1 is equal to its length and thus the next instruction executed in the combined computation of $P_1 \widehat{++} P_2$ is the first one in P_2. This does not constitute any restrictions since it is proved that any program can be transformed into standard form by setting the destination of out of range jumps to the length of the program. Moreover, it can be proved that $P_1 \widehat{++} P_2$ diverges if one of the component programs diverges.

The transformation of programs into standard form is given by the function `SF: program -> program` and the proof that for any program P, its behaviour is equivalent to `SF` P is done by first showing that after a single step of the execution of both programs the resulting configurations are equivalent. Two configurations are equivalent either if they are the same, or both are final and have the same store. This result is then extended for any number of execution steps and finally it is proved that

```
⊢ ∀P l v. CONVERGE (SF P) l v = CONVERGE P l v
```

by showing that if one program converges in a number of steps then the other converges in the same number of steps.

The concatenation operator $\widehat{++}$ is defined in HOL as the function `SAPP: program -> program -> program`, and since it is often required to concatenate more than two programs, a function `SAPPL: program list -> program` which concatenates a given list of programs is defined as well.

The following three simple program modules which are used quite often in the construction of general URM programs:

- `SET_FST_ZERO` n stores the value 0 in the registers $(R_0, R_1, \ldots, R_n)$.
- `TRANSFER_FROM` p n stores $(r_p, r_{p+1}, \ldots, r_{p+n-1})$ into $(R_0, R_1, \ldots, R_{n-1})$.
- `TRANSFER_TO` p n store $(r_0, r_1, \ldots, r_{n-1})$ into $(R_p, R_{p+1}, \ldots R_{p+n-1})$.

are defined as follows:

$\vdash_{def}$ ∀n. SET_FST_ZERO n = GENLIST ZR (SUC n)

$\vdash_{def}$ ∀p n. TRANSFER_FROM p n = GENLIST (λx. TF (p + x) x) n

$\vdash_{def}$ ∀p n. TRANSFER_TO p n
      = REVERSE (GENLIST (λx. TF x (p + x)) n)

where GENLIST and REVERSE are defined in the List theory of HOL. The programs yielded by these functions are proved to converge and to convey their expected behaviour by induction on the number of steps of execution of the programs.

Another program module, which is given by $[P\ p_s \xrightarrow{n} p_v]$, or by the term PSHIFT $P\ p_s\ n\ p_v$, is defined. This program module executes P, taking its n parameters from the memory segment at offset p_s rather than from the first n registers. Also, this program stores the value of the computation in R_{p_v} rather than the first register. This is defined by:

```
⊢def ∀P ps n pv. PSHIFT P ps n pv
      =  SAPPL [SET_FST_ZERO (MAXREG P);
                TRANSFER_FROM p n;
                P;
               [TF 0 pv]]
```

where MAXREG P is the maximum register used by P and is denoted by $\rho(P)$. The proofs that $[P\ p_s \xrightarrow{n} p_v]$ diverges if and only if P diverges, and that if the former converges it yields the expected configuration, are done by applying the result that the computation of programs constructed by $\widehat{++}$ is made up from the computations of the constructing programs.

4 Constructing Computable Functions

In this section we show that a number of basic functions are computable and that the family of computable functions is closed under the operations of substitution, recursion and minimalisation. These results yield a mechanism for constructing computable functions and automatically proving their computability. Moreover, the set of functions which are constructed by the above operations, which is called the set of partial recursive functions, is equal to the set of computable functions [4, 10]. Thus particular functions can be proved to be computable by proving their equality to some partial recursive function. However this process is not decidable; nevertheless a number of symbolic animation tactics [2] have been implemented which simplify the proof of theorems stating such an equality.

The verification of the closure properties involves the construction of a URM program which is proved to compute the function constructed by the particular operation being considered. Due to space limitations, only the proof of the closure under recursion is illustrated in detail.

4.1 The Basic Functions

The following three basic functions are considered:

1. The zero functions (each of different arity) which return 0 for any input: $\forall n, x_0, \ldots, x_{n-1}.\mathcal{Z}(x_0, \ldots, x_{n-1}) = 0$,
2. the successor function which increments its input by one: $\forall x_0.\mathcal{S}(x_0) = x_0 + 1$,
3. and projections, which return a particular component from a given vector: $\forall n, i < n, x_0, \ldots, x_{n-1}.\mathcal{U}_n^i(x_0, \ldots, x_{n-1}) = x_i$.

These functions are defined in HOL as follows:

```
⊢def ∀l. ZERO l = Value 0

⊢def ∀l. SUCC l
       = ((LENGTH l = 1) → (Value (SUC (HD l))) | Undef)

⊢def ∀i n l. PROJ i n l = ((i < n) → (Value (ZEL i l)) | Undef)
```

and are proved to be computable by showing that the programs `[ZR 0]` and `[SC 0]` compute $\mathcal{Z}$ and $\mathcal{S}$ respectively; and that the projection $\mathcal{U}_n^i$ is computed by `[TF` i `0]` for $i < n$ and since it is undefined for $i \geq n$ it is computed by `[JP 0 0 0]`. The function `ZEL` i l returns the $(i+1)$th element of l if i is less than the length of l; otherwise it returns 0.

4.2 Substitution

The substitution of k n-ary functions $\underline{g} = (g_0, \ldots, g_{k-1})$ into a k-ary function f gives the n-ary function produced by applying f on the results of the applications of $\underline{g}$. That is,

$$f\hat{\circ}\underline{g}(x_0, \ldots, x_{n-1}) = \\ f(g_0(x_0, \ldots, x_{n-1}), \ldots, g_{k-1}(x_0, \ldots, x_{n-1})).$$

This is defined in HOL as the function `FSUBS: pfunc -> pfunc list -> pfunc`.

```
⊢def ∀f l. APPLY f l
       = ((ALL_EL DEFINED l) → (f (MAP VALUE l)) | Undef)

⊢def ∀f gl l. FSUBS f gl l = APPLY f (MAP (λg. g l) gl)
```

In order to prove that computable functions are closed under substitution, it is required to show that given the programs $P_f, P_{g_0}, \ldots, P_{g_{k-1}}$ which compute the functions $f, g_0, \ldots, g_{k-1}$ respectively, a program $P_{f\hat{\circ}\underline{g}}$ can be constructed which computes $f\hat{\circ}\underline{g}$. Such a program is shown in Fig. 1. The program parameters are first transferred to some memory location at offset p_s. The programs P_{g_i} for $i < k$ are then executed one at a time storing their results into another memory segment starting at p_v, and finally P_f is executed on the results. The value of p_s is chosen to be $\max(n, k, \rho(P_f) + 1, \max(\rho(P_{g_0}), \ldots, \rho(P_{g_{k-1}})) + 1)$ so that the contents of this memory segment is not altered during the program execution. Similarly, p_v is set to $p_s + n$.

start: TRANSFER_TO p_s n	store parameters in $(r_{p_s}, \ldots, r_{p_s+n-1})$
inner: $[P_{g_0}\ p_s \xrightarrow{n} p_v]$	
$\vdots$	for each $i < k$ execute P_{g_i}
$[P_{g_i}\ p_s \xrightarrow{n} (p_v + i)]$	storing its result in R_{p_v+i}
$\vdots$	
$[P_{g_{k-1}}\ p_s \xrightarrow{n} (p_v + k - 1)]$	
outer: $[P_f\ p_v \xrightarrow{n} 0]$	execute P_f on the values returned by the P_{g_i}'s

Fig. 1. The program $P_{f \circ \underline{g}}$

4.3 Recursion

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

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

and is specified in HOL as the function `FREC: pfunc -> pfunc -> pfunc`.

```
⊢def (∀basis step l. RECURSION basis step 0 l = basis l) ∧
     (∀basis step n l.
      RECURSION basis step (SUC n) l =
      (let r = RECURSION basis step n l in
      (DEFINED r) → (step (CONS n (CONS (VALUE r) l)))
                  | Undef))

⊢def ∀basis step l.
      FREC basis step l =
      ((l = []) → (basis [])
               | (RECURSION basis step (HD l) (TL l)))
```

Given the programs P_β and P_σ which compute the functions β and σ respectively, the program $P_{\mathcal{R}(\beta;\sigma)}$ shown in Fig. 2 computes the recursive function $\mathcal{R}(\beta;\sigma)$. The value of p_c is chosen to be $\max(\rho(P_\beta)+1, \rho(P_\sigma)+1, n+2)$ so that the registers starting at p_c are not used by P_β and P_σ; and the values of p_v, p_s and p_x are chosen to be $p_c + 1$, $p_v + 1$ and $p_s + n$ respectively. The register R_{p_c} is used to store a counter for the number of times the inner loop

```
start: TRANSFER_TO p_v (n+1)  Store (x, x_0, ..., x_{n-1}) in (R_{p_v}, R_{p_s}, ..., R_{p_s+n-1})
       TF 0 p_x               Set r_{p_x} to the value x
       [P_β p_s --n--> p_v]   Execute P_β
loop:  JP p_c p_x final       While r_{p_c} < x
       [P_σ p_c --n+2--> p_v]    Execute P_σ
       SC p_c                    Increment the counter r_{p_c}
       JP 0 0 loop
final: TF p_v 0               Return the final value r_{p_v}
```

Fig. 2. The program $P_{\mathcal{R}(\beta;\sigma)}$

is executed. The value of each recursion step is stored at R_{p_v} and the memory segment $(R_{p_s}, \ldots, R_{p_s+n-1})$ is used to store the function's parameters, which are transferred by the first step of the program. The register R_{p_x} stores the depth of the recursion such that the inner loop is repeated r_{p_x} times after the code computing the base case function, $[P_\beta\ p_s \xrightarrow{n} p_v]$, stores $\beta(x_0, \ldots, x_{n-1})$ into r_{p_v}. The final value of p_v is then transferred into the first register R_0.

This program can be divided into three parts, which we call P_{start}, P_{loop} and P_{final}. These are represented in HOL by the terms

```
Pstart = SAPPL [TRANSFER_TO p_v (n + 1);
                [TF 0 p_x];
                PSHIFT P_β p_s n p_v]

Ploop = let Ps = PSHIFT P p_c (n + 2) p_v in
        APPEND (SAPP [JP p_c p_x (3 + LENGTH Ps)]
                     Ps)
                     [SC p_c;
                      JP 0 0 0]

Pfinal = [TF p_v 0]
```

and $P_{\mathcal{R}(\beta;\sigma)}$ is then given by

```
SAPPL [^Pstart; ^Ploop; ^Pfinal]
```

This program is then proved to compute the recursive function by considering whether the base case function β and the step function σ are defined:

– If $\beta(x_0, \ldots, x_{n-1})$ is defined then

- P_β converges, and so does $[P_\beta\ p_s \xrightarrow{n} p_v]$. As a result P_{start} converges to a final configuration containing the value of $\beta(x_0, \ldots, x_{n-1})$ (which is equal to $\mathcal{R}(\beta;\sigma)(0, x_0, \ldots, x_{n-1})$) in the first register, the parameters $(x_0, \ldots, x_{n-1})$ stored in $(R_{p_s}, \ldots, R_{p_s+n-1})$ and r_{p_x} set to the depth x;
- If, also the step function σ is defined for all values of $i \leq x$ then
  - all programs $[P_\sigma\ p_c \xrightarrow{n+2} p_v]$ converge for each value of the recursion counter r_{p_c}, and hence P_{loop} converges, placing the final value of the application of σ (which is equal to $\mathcal{R}(\beta;\sigma)(x, x_0, \ldots, x_{n-1})$) in (R_{p_v});
  - and finally P_{final} stores the value of $\mathcal{R}(\beta;\sigma)(x, x_0, \ldots, x_{n-1})$ into the first register. Thus, whenever $\mathcal{R}(\beta;\sigma)$ is defined, $P_{\mathcal{R}(\beta;\sigma)}$ converges to the required value.
- On the other hand, if σ is undefined for some non-zero value $i \leq x$ then
  - the program $[P_\sigma\ p_c \xrightarrow{n+2} p_v]$ diverges when $r_{p_c} = i - 1$, thus P_{loop} diverges and so does $P_{\mathcal{R}(\beta;\sigma)}$. However, if the step function is undefined then $\mathcal{R}(\beta;\sigma)$ is undefined as well. Hence, in this particular case, the function is undefined and the program diverges (as expected).

– Now, if the base case function β is undefined then
  - P_β diverges. As a result, all the programs constructed from it using the $\widehat{++}$ operator diverge. In particular the programs $[P_\beta\ p_s \xrightarrow{n} p_v]$, P_{start} and $P_{\mathcal{R}(\beta;\sigma)}$. Also, given that β is undefined, then so is $\mathcal{R}(\beta;\sigma)$, and even in this final case the recursive function is undefined and the program diverges.

– Thus
  1. $P_{\mathcal{R}(\beta;\sigma)}$ converges to $\mathcal{R}(\beta;\sigma)(0, x_0, \ldots, x_{n-1})$ whenever the latter is defined; and
  2. $P_{\mathcal{R}(\beta;\sigma)}$ diverges whenever $\mathcal{R}(\beta;\sigma)(0, x_0, \ldots, x_{n-1})$ is undefined.

– So, $P_{\mathcal{R}(\beta;\sigma)}$ computes $\mathcal{R}(\beta;\sigma)$ proving that the latter is computable.

The same method of considering whether the constituting functions of an operation are defined or not, is used in the proofs that substitution and minimalisation of computable functions yield functions which are also computable.

4.4 Minimalisation

The unbounded minimalisation of an $(n+1)$-ary function f is the n-ary function given by:

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

This is formalised in HOL by the following definition:

```
⊢def ∀f l.
      FMIN f l =
      (let Z x = f (CONS x l) = Value 0 in
       let n = FIRST_THAT Z in
       (Z n ∧
        (∀m. m ≤ n ⇒
          DEFINED (f (CONS m l)))) → (Value n) | Undef)
```

where FIRST_THAT R returns the first natural number n such that $R(n)$ holds, if such an n exists, or any particular value otherwise.

```
⊢def ∀P. FIRST_THAT P = (εn. P n ∧ (∀m. P m ⇒ n ≤ m))
```

Given that P_f computes f, the program $P_{\mu_x(f(x)\mapsto 0)}$ shown in Fig. 3 computes the minimalisation function $\mu_x(f(x) \mapsto 0)$. This is done by executing P_f until it returns the value 0. The register at position p_c is used as a counter which is incremented each time P_f is executed and is returned by $P_{\mu_x(f(x)\mapsto 0)}$ if it terminates. The value of p_c is set to $\max(\rho(P)+1, n+1)$, so that it will not be used by P_f, also the parameters are stored at the memory segment starting at p_s which is set to p_c+1. The value of p_0 is chosen to be p_s+n and is not altered during program execution, so that $r_{p_0} = 0$.

start:	TRANSFER_TO p_s n	Store parameters in $(R_{p_s}, \ldots, R_{p_s+n-1})$
fetch:	$[P_f\ p_c \xrightarrow{n+1} 0]$	Execute P_f
	JP 0 p_0 final	Until it returns 0
	SC p_c	Otherwise, increment r_{p_c}
	JP 0 0 fetch	Jump back to fetch
final:	TF p_c 0	Return r_{p_c}

Fig. 3. The program $P_{\mu_x(f(x)\mapsto 0)}$

4.5 Proving the Computability of Particular Functions

The operations mentioned above make up the language of partial recursive functions and are sufficient to build up the family of computable functions. A HOL conversion simulating the application of function terms constructed using these constructs is implemented. The three basic functions are automated by simply

Function Description	Notation	Partially recursive equivalent
Parameter rearrangement	$f \circ \upsilon_{i_0,\ldots,i_{n-1}}$	$f \hat{\circ} (\underline{\lambda j.\mathcal{U}_n^{i_j}})$
Identity	ι	$\mathcal{U}_1^0$
One	1_c	$\mathcal{S} \hat{\circ} (\mathcal{Z})$
Addition	$+_c$	$\mathcal{R}(\iota; \mathcal{S} \circ \upsilon_1)$
Multiplication	$\times_c$	$\mathcal{R}(\mathcal{Z}; +_c \circ \upsilon_{1,2})$
Factorial	fact_c	$\mathcal{R}(1_c; \times_c \hat{\circ} (\mathcal{S}, \iota))$
Predecessor	pred_c	$\mathcal{R}(\mathcal{Z}; \mathcal{U}_2^0)$
Subtraction	$-'_c$	$\mathcal{R}(\iota; \text{pred}_c \circ \upsilon_1)$
	$-_c$	$-'_c \circ \upsilon_{1,0}$
Power	e'_c	$\mathcal{R}(1_c; \times_c \circ \upsilon_{1,2})$
	e_c	$e_c \circ \upsilon_{1,0}$
Conditional	if_c	$\mathcal{R}(\mathcal{U}_4^3; \mathcal{U}_2^0)$
Check if 0[a]	is0	$\text{if}_c \hat{\circ} (\iota, \mathcal{Z}, 1_c) \circ \upsilon_{0,0,0}$
Check if non 0	non0	$\text{if}_c \hat{\circ} (\iota, 1_c, \mathcal{Z}) \circ \upsilon_{0,0,0}$
Difference	$\vert -_c (x_0, x_1) \vert$	$+_c \hat{\circ} (-'_c, -_c)$
Equality	$=_c$	$\text{is0} \hat{\circ} (\lambda (x_0, x_1). \vert -_c (x_0, x_1) \vert)$
Inequality	$\neq_c$	$\text{is0} \hat{\circ} (=_c)$
Conjunction	$\wedge_c$	$\text{non0} \hat{\circ} (\times_c)$
Disjunction	$\vee_c$	$\text{non0} \hat{\circ} (+_c)$
Minimal inverse	f^{-1}	$\mu_y((\neq_c \hat{\circ} (f, \iota) \circ \upsilon_{0,0})(y) \mapsto 0)$

Fig. 4. A list of computable functions

[a] can also be used as a negation operator

rewriting with the appropriate definitions; substitution is automated by first evaluating each of the substituting functions and then evaluating the function into which these functions are substituted. A function defined by recursion is animated by evaluating either the base case function, or the step function recursively. A function constructed by minimalisation, $\mu_x(f(x) \mapsto 0)$ is animated by first proving that if for some i, $f(i) = 0$ and for all $j < i$, $f(j)$ is defined and is greater than 0, then $\mu_x(f(x) \mapsto 0) = i$. By evaluating $f(j)$, for $j = 0, 1, \ldots$ one can construct a thoerem which states that $\forall j'.j' < j \Rightarrow f(j) > 0$ until $j = i$ and thus $f(j) = 0$. This theorem and the evaluation of $f(i)$ are then used to prove that $\mu_x(f(x) \mapsto 0) = i$, thus evaluating the minimalisation function.

This conversion is used to prove that particular functions are equal to some specified partial recursive function. In general an equality to a function constructed by substitution is proved by applying this conversion on the functional application term and then proving the equality of the resulting terms (often by simple rewriting of the definitions); and equality to a function defined by recursion is proved by mathematical induction and then applying this conversion on the base case and induction step subgoals. Since the simulation of minimalisation involves a possibly non-terminating fetching process (when the function is total and never returns 0) the execution of this conversion may diverge, in such case this conversion cannot be used in the required proof; although if the fetching process terminates the proof of the equality of functions defined by minimalisation is otherwise relatively straightforward.

Moreover, since partial recursive functions are constructed from three basic functions which are proved to be computable and by three operators which are proved to preserve computability, the process of proving that such functions are computable can be automated. Given a conversion which automates this mechanism, the proof that a function is computable simply involves showing that it is equal to some partial recursive function. Figure 4 lists a number of functions which are proved to be computable.

5 Theorem Proving in HOL

The proofs of most of the results in this mechanisation tend to be quite elaborate and involve the consideration of details which are often omitted in the proofs given in mathematical texts. This is often the case with mechanical verification since most of the definitions and proofs done by hand and represented in texts omit a number of steps which are considered to be trivial or not interesting to the reader. For example the proof that for every URM program one can construct a program in standard form which has an equivalent behaviour (Sect. 3.4) is considered to be trivial in [4], although it requires a considerate number of lemmas concerning the behaviour of executing URM programs in general and programs transformed into standard form by the function `SF`. Also, in the proofs that computable functions are closed under the operations of substitution, recursion and minimalisation, little or no attention is given in showing that the program constructed to compute the required function diverges whenever the function is undefined, probably because this part of the proof is considered uninteresting. It is to note, however, that such proofs usually offer an interesting challenge in a mechanisation.

However, an advantage of theorem proving in HOL and other LCF style theorem provers is the availability of a flexible general purpose meta-language which can be used in the implementation of program modules which simulate the behaviour of the formal definitions as well as intelligent algorithms which automate parts of the verification process. In this particular implementation, such a mechanism is found to be quite useful in the verification of general computable functions. On the other hand, theorem proving in Coq is usually done by applying tactics through the specification language Gallina. This offers the advantage that unless a number of specialised tactics need to be implemented, the user does not need any knowledge on how the actual terms and theorem are represented in the implementation of the theorem prover and is thus easier to learn than HOL. Another advantage of having a specification language which bridges the user from the meta-language is that proofs can be easily represented as lists, or trees of tactics in a format which can be read by the user. However such an approach has the disadvantage that it reduces the flexibility of the system and discourages the user from implementing his or her own tactics.

Type theories allow the definition of dependent types where a type can be parametrised by other types, and thus offer a more powerful and flexible type definition mechanism than the one available in HOL. For example, the type of functions which are considered for computability are defined as mappings from lists of numbers to possibly partial numbers (Sect. 3.3) and the type itself contains no information regarding the function's arity. In an implementation in Coq, an n-ary partial function is defined as a single valued relation between vectors of n elements and natural numbers. Vectors are defined as the following inductive dependent type:

```
Inductive vector [A: Set]: nat → Set
  := Vnil:  (vector A O)
  |  Vcons: (n: nat)A → (vector A n) → (vector A (S n)).
```

and partial functions as a record with two fields:

```
Record pfunc [arity: nat] : Type := mk_pfunc
  { reln:        (Rel (vector nat arity) nat);
    One_valued: (one_valued (vector nat arity) nat reln) }.
```

The first field `reln` represents the function as a relation and the second field `One_valued` is a theorem which states that `reln` is single valued. Relations are defined by:

```
Definition Rel := [A,B: Set]A → B → Prop.
```

and the predicate `one_valued` by:

```
Definition one_valued
  := [A,B: Set][R: (Rel A B)](a: A)(b1, b2: B)
       (R a b1) → (R a b2) → (b1 = b2).
```

Finally the type of all partial functions is defined as the dependent product

```
Inductive pfuncs: Type
  := Pfuncs: (n: nat)(pfunc n) → pfuncs.
```

Possible enhancements to theorem proving in the HOL system include mechanisms for naming or numbering assumptions in a goal-directed proof, for allowing constant redefinition and the declaration of local definitions. Proofs found in mathematical texts often contain definitions which are only used in showing a number of particular results. Such definitions can be made local to the results which require them. For example, in the proof that computable functions are closed under recursion (Sect. 4.3) the constants `Pstart`, `Ploop` and `Pfinal` are used only in obtaining this particular results. Such constants are defined as local meta-language variable definitions, however a more elegant approach would involve a theory structure mechanism where constants can be defined local to a theory module and made invisible outside their scope.

6 Conclusion

The mechanisation of computability theory discussed above includes the definition of a computable function according to the URM model and the proof of various results of the theory of which we have given particular attention to the closure property of computable functions under the operations of substitution, recursion and minimalisation. This result is used in the implementation of a process which automates the proof of the computability of functions constructed from these operations and a number of basic computable functions. In the future, it is expected that the theory will be extended through the proof of a number of theorems including the denumerability of computable functions, the S_n^m theorem and the universal program theorem.

A mechanisation of computability theory is also being implemented in the Coq proof assistant. This implementation uses the partial recursive function model of computation and will include the proof of the results mentioned in the previous paragraph. These two implementations and other work in Alf are expected to yield a comparative study of the different theorem proving approaches.

7 Acknowledgements

I thank my supervisor, Simon Thompson, for his continuous support and for his comments on the material presented in this paper, as well as the anonymous referees for their constructive comments on the first draft of this paper.

References

1. Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A User's Guide to ALF.* Chalmers University of Technology, Sweden, May 1994.
2. J. Camilleri and V. Zammit. Symbolic animation as a proof tool. In T.F. Melham and J. Camilleri, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 113–127, Malta, September 1994. Springer-Verlag.
3. C. Cornes et al. The Coq Proof Assistant Reference Manual, Version 5.10. Rapport technique RT-0177, INRIA, 1995.
4. N.J. Cutland. *Computability: An introduction to recursive function theory.* Cambridge University Press, 1980.
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. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.
8. T.F. Melham. Using recursive types to reason about hardware and higher order logic. In G.J. Milne, editor, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 27–50, Glasgow, Scotland, July 1988. IFIP WG 10.2, North-Holland.
9. L.C. Paulson. *Logic and computation : interactive proof with Cambridge LCF.* Cambridge tracts in theoretical computer science, 1987.
10. H. Rogers. *Theory of recursive functions and effective computability.* McGraw-Hill, 1967.
11. J.C. Shepherdson and H.E. Sturgis. Computability of recursive functions. Technical Report 10, J. Assoc. Computing Machinery, 1967.
12. R. Sommerhalder and S.C. van Westrhenen. *The theory of computability: programs, machines, effectiveness and feasibility.* Addison-Wesley publishing company, 1988.
13. G.J. Tourlakis. *Computability.* Reston Publishing Company, 1984.
14. P.J. Windley. Specifying instruction-set architectures in HOL: A primer. In T.F. Melham and J. Camilleri, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 440–456, Malta, September 1994. Springer-Verlag.