Defining Recursive Functions in HOL[‡]

W.Ploegaerts*, L.Claesen†, H.De Man†

IMEC vzw.
Kapeldreef 75, B-3001 Leuven, BELGIUM
Phone +32/16/281525

Abstract

The limited support for the definition of recursive functions and the generation of induction schemes is a major shortcoming of the HOL system when applied for hardware verification. In this paper a theory of primitive recursive functions and a tool that minimizes the proof effort required for the definition of a wide class of primitive recursive functions is presented. Due to its flexibility and degree of automation it offers a useful extension of the HOL system.

1 Introduction

The correctness of hardware designs is one of the major unsolved problems design engineers currently have to cope with. Several synthesis tools such as the CATHEDRAL system [1] offer a means for a nearly automatic design of very complex circuits. As error-free software still does not exist, bugs or even conceptual errors in the CAD tools are likely to occur. The functional correctness of the generated designs is to be distrusted, and verification is essential. Formal hardware verification, in principle a methodology leading to a full-proof correct design, offers a valuable alternative to classical verification methods such as simulation. The latter can only partially validate the correctness of a design.

Though the idea of formal verification has been around for several years now, it has not been accepted in the world of the design engineers. There are three important reasons for this:

1. the lack of real-world results proving that the approach works for anything but toy-circuits

2. the large effort required for both the formal description of a design and the construction of the proof itself [2]

3. most importantly, the large gap between the way of thinking required for formal verification and the way design engineers think [3]: design engineers are no logicians and most of the time even refuse to try to be so

In the HOL hardware verification community, the latter issue is of growing importance. Several approaches to help bridge this gap can be found in literature. Some examples are the semantic embedding of hardware description languages in HOL [4] [5]. This aims to provide theorem proving tools for reasoning about hardware design in a formalism the designers feel comfortable with. Other approaches are the inclusion of design hints in the verification process to guide the proof [6] and the hierarchical approach to hardware verification based on the idea of abstraction [7] [8] [9].

All these issues are of major importance for the hardware verification community as it provides means to manage the inherent complexity of the proofs. However, they still do not make the HOL system acceptable for the design engineers. These are looking for "canned solutions" and do not want to be bothered with the theoretical matters a HOL user gets involved with. The latter is partly due to the nature of the HOL system itself. Due to the lack of automation and support currently available, the user can hardly benefit from the high expressive power inherent to the higher order logic: facts that are intuitively clear often require a major proof effort in the HOL system [3]. Although the logic is a good formalism for specifying and verifying hardware [10] [11], systems such as the Boyer-Moore system [12] based on a less expressive logic but offering a higher degree of automation might easier be accepted by the design engineers.

The lack of support for the definition of recursive functions and the generation of induction schemes is a major shortcoming of the HOL system in this respect. Induction, the means of function abstraction and recursive function definitions are the main potential advantages the theorem prover based approach offers compared with an classic (and fully automatic) approach to hardware verification. The HOL system though hardly offers any support for the definition of recursive functions and the generation of induction strategies. Even the modeling phase therefore becomes a burden to the user: he is interested in the verification of a design and *not* in the tedious, often highly theoretical HOL proofs inherent to definition of recursive functions and induction schemes.

The goal of the work described in this paper is to help the user bridge the gap between the intuitive understanding of the well-foundedness of a recursion and the definition of a recursive function in HOL. A tool

*Research assistant with the Belgian National Fund for Scientific Research

†Professor at K.U.Leuven

‡Work partly funded by the ESPRIT 2 BRA CHEOPS(3215)

is provided such that the related proof effort is minimised. The tool offers support both for the proof of the well-foundedness of the recursive scheme and for the definition of a specific recursive function.

In section 2 the existing tools and theories pertaining this subject matter that are currently available in HOL (Version 1.12) will be discussed. In section 3 the theory on which the tool is based will be presented. This is followed in section 4 by a discussion of the logic functions provided. In section 5 several examples will demonstrate the flexibility, the power and the limitations of the new tool.

2 Recursive definitions in HOL

In this section the definition of recursive functions in HOL is discussed. The tools and theories provided with the HOL system are presented. Their limitations in an hardware verification environment will be clarified.

2.1 Recursive definitions

To preserve the consistency in the HOL system, HOL theories should only be extended by *definitional extension* [13]: new constants and types are to be defined in terms of existing ones. The definition in HOL of a function fun, characterised by a predicate P, consists out two parts:

1. *existence proof* of the function fun, this involves the construction of a function with the intended behavior (and possibly the use of the choice operator) and results in the theorem $\exists fun.P\ fun$;

2. *constant specification*: the symbol fun is selected as the name of the function for which the property P holds, the behavior of the function fun is specified by $P\ fun$; this is implemented by the logic function "new_specification";

The problems related to the definition of recursive functions in HOL are twofold: there is the purely theoretical issue related to the existence of recursive functions, and a technical matter related to the modeling of the existence proof in the HOL system. The first problem is related to the question why a specific recursive definition is well-founded. This problem cannot be solved in its full generality. Therefore, the definition of recursive function can only be automated for a restricted class of functions. For all other definitions, a tedious proof is required due to the inherent complexity of the related existence proofs. The HOL system provides limited support and automation pertaining this subject matter. The tools and theories supplied with the system will now be discussed.

2.2 Automatic definitions

The logic functions based on the type definition tool [13] offer a means for the automatic definition of recursive functions. The class of functions that can be defined automatically is restricted: only primitive recursive functions defined by cases on the type constructors used for the definition of a new type are directly supported [13]. Every recursive type is abstractly characterized by a theorem of the form:

2.1

$$\begin{aligned}\vdash \forall f_1 \cdots f_m.\ \exists!\ \mathsf{fn}{:}op \longrightarrow *.\\ \forall x_1^1 \cdots x_1^{k_1}.\ \mathsf{fn}(C_1\ x_1^1 \ \ldots\ x_1^{k_1}) =\\ f_1\ (\mathsf{fn}\ x_1^1) \ldots (\mathsf{fn}\ x_1^{k_1})\ x_1^1 \ \ldots x_1^{k_1}\\ \vdots\\ \forall x_m^1 \cdots x_m^{k_m}.\ \mathsf{fn}(C_m\ x_m^1 \ \ldots\ x_m^{k_m}) =\\ f_m\ (\mathsf{fn}\ x_m^1) \ldots (\mathsf{fn}\ x_m^{k_m})\ x_m^1 \ \ldots x_m^{k_m}\end{aligned}$$

In theorem 2.1, C_i for $i : 1 \cdots m$ are the constructors used for the definition of the type ":op" [13]. With the logic function prove_recursive_function_exists the existence of any recursive function, both complete and partial functions, defined by cases on these constructors C_i can be derived from the theorem 2.1. This logic function can be used with theorems syntactically similar to theorem 2.1 for the definition of recursive functions by cases on constructors other than the one automatically derived from the type definition. This is illustrated with an example from the theory list. The type ":(*) list" is characterised by the list_Axiom:

2.2

$$\vdash \forall x\ f.\ \exists!\mathsf{fn}.(\mathsf{fn}[] = x) \wedge (\forall h\ t.\ \mathsf{fn}(\mathsf{CONS}\ h\ t) = f(\mathsf{fn}\ t)h\ t)$$

This allows recursive definitions such as the length of a list:

2.3

$$\vdash_{def} (\mathsf{LENGTH}[] = 0) \wedge (\forall h\ t.\ \mathsf{LENGTH}(\mathsf{CONS}\ h\ t) = \mathsf{SUC}(\mathsf{LENGTH}\ t))$$

However, one could also consider an equivalent definition of the length of a list by recursion on the last element of the list:

2.4

$$\vdash_{def} (\mathsf{LENGTH2}[] = 0) \wedge (\forall h\ t.\ \mathsf{LENGTH2}\ (\mathsf{APPEND}\ t\ [h]) = \mathsf{SUC}(\mathsf{LENGTH2}\ t))$$

To define this function LENGTH2 with the logic function prove_recursive_function_exists, the following theorem should be provided:

2.5

$$\vdash \forall x\ f.\ \exists!\mathsf{fn}.(\mathsf{fn}[] = x) \wedge (\forall h\ t.\ \mathsf{fn}(\mathsf{APPEND}\ t\ [h]) = f(\mathsf{fn}\ t)t\ h)$$

This theorem can be derived from theorem 2.2. Even for this simple case, the proof is rather complex and no support is provided.

A more general case is the definition of functions simultaneously recursing in different arguments. This cannot be handled by the current implementation of prove_recursive_function_exists. This type of definitions pops up frequently in the specification of hardware devices. One example is the functional definition of a two-input n-bit bitwise-and function:

2.6

$\vdash_{def}$ (AND [] [] = []) ∧
(AND (CONS x X) (CONS y Y) =
CONS (x ∧ y) (AND X Y))

Following the philosophy of the type definition package, the existence of the function AND requires the proof of theorem 2.7:

2.7

⊢ ∀a f1 f2 f3.
∃! fn.
(fn[][] = a) ∧
(∀x X. fn(CONS x X)[] = f1(fn X[])x X) ∧
(∀y Y. fn[](CONS y Y) = f2(fn[]Y)y Y) ∧
(∀x X y Y. fn(CONS x X)(CONS y Y) =
f3(fn X Y)x X y Y)

The proof of theorem 2.7 is fairly complex. The definition of a three-input n-bit bitwise-and function would require yet another theorem to be proven.

This discussions suggests the lack of flexibility of the definition tool. The class of functions that can be defined automatically is restricted by the availability of theorems like 2.1. Only functions recursing in a single argument are currently supported. The HOL user thus has two possible tracks to follow:

- cast all definitions in a form that is supported
- prove the dedicated theorem of type 2.1

In the field of hardware verification, the first approach is not attractive. The definition of recursive functions is mainly a means for modeling. It is therefore not acceptable for the formal model in HOL to be completely different from the informal one that is generally used. Such a formal model increases the proof effort to be spent when building a theory because of the gap between intuition and formal model. Moreover, it makes the approach lose all credit in the eyes of the design engineers. On the other hand, the second approach seems to be even a greater burden: it leads the user to a complex existence proof, which has nothing to do with the subject matter he is interested in. The necessity to carry out these proofs are an unintentional and often frustrating confrontation with the HOL system.

2.3 Other theories and tools

As the existing definition tool is too restrictive and not flexible enough, a more general approach must be found. The HOL system (Version 1.12) contains several contributions offering an elegant theoretical basis for a theory of recursive functions:

- The Library well-order (T.Kalker): based on the proof that every set can be well-ordered, the principle of transfinite induction is derived. A conversion that transforms a term into the appropriate existence theorem related to the transfinite induction theorem is provided.
- The Library fixpoints (M.Gordon): the fixpoint operator is defined and the validity of computation induction is proved.
- The Contribution CPO (A.J.Camilleri): a theory of complete partial orderings and fixed-points that is intended to lead to the fixed point theorem which will allow the definitions of recursive operators.

Unfortunately, none of these theories have been worked out up to a level high enough that they can easily be used for making the recursive definitions. The gap between the provided theories and their practical use can only be bridged by a large proof effort, to a great extent dealing with domain-theoretical issues. Due to its highly theoretical nature, this is again too much of a burden for the HOL user interested in hardware verification.

2.4 Discussion

The current status of the support for the definition of recursive functions can be summarised as follows:

- Automatic definition without a large proof effort is only possible for a limited class of recursive functions.
- If an automatic definition is not possible, the proof effort is not directly related to the correctness of the recursion itself; it mainly involves a tricky syntactic transformation of theorems similar to the output of type definition tool.
- The theories offering a more general solution to the problem are still too theoretical to be of direct practical use. The main problems the user is confronted with are domain-theoretical.

This yields some requirements for a tool for automating the definition of recursive functions:

- *automation* for a large class of functions
- *flexibility* in the choice of recursive schemes and the number of arguments the function simultaneously recurses in
- *minimal proof obligation*

3 A theory of Recursive functions

In this section, the theory of recursive functions that forms the basis of the new definition tool is presented. First an informal discussion introduces the basic theorem of the theory, followed by its formalization in the HOL system.

3.1 The well-foundedness of recursive definitions

In this work an intuitive approach, has been opted for, instead of one that starts from the existing mathematical theories. A recursive definition for a function is, roughly speaking, a definition wherein values of the function for given arguments are directly related to values of the same function for "simpler" arguments or to values of "simpler" functions [14]. The notion

"simpler" is to be specified in the chosen characterisation, usually taken as the simplest of all (e.g. the constant function). The latter are the "base cases" of the recursion. In the remainder the set BC, say of type "*set", will be taken as the set of all base cases for a given recursion. The predicate IS_BC of type "*→bool" is the related predicate defining the set BC. Such a recursive definition uniquely defines a function f of type "*→**" if two conditions are satisfied. First, it must be defined for every element x of type "*". how $f\ x$ is decomposed in its simpler parts. Secondly, for all x a base case must be reached after a finite number of decompositions. Consider the class of recursive definitions restricted to primitive functions of arity one, having M base cases BC_i and defined by cases on N constructors C_j. Then, a function f can be defined by cases on the constructors and the base cases:

$$\begin{cases} f\ BC_1 = \alpha_1 \\ \vdots \\ f\ BC_M = \alpha_M \\ f\ (C_1\ x) = f_1\ (f\ x)\ x \\ \vdots \\ f\ (C_N\ x) = f_N\ (f\ x)\ x \end{cases}$$

The proof obligation for the existence of the function f is that every x of type "*" can be constructed in a finite amount of steps starting from one of the base cases. As the number of base cases and constructors are variable, neither the proof obligation nor the existence theorem itself can be formalised in a closed formula. The proof strategy can thus not be a simple transformation of a pre-proven theorem that captures the correctness in its full generality; instead it has to be based on conversions. Past experience has shown that the fulfilment of the proof obligation leads to rather complex proofs as a case split on the different base cases and constructors is necessary. A different approach has therefore been chosen.

As all constructors are functions that are one_one, a *destructor* function D can be defined by cases on the constructors

$$\begin{cases} D\ (C_1\ x) = x \\ \vdots \\ D\ (C_N\ x) = x \end{cases}$$

Using this destructor function, the definition of the function f can be transformed into

$$f\ x = (IS\_BC\ x)\ \Rightarrow\ g\ x\ \mid\ h\ (f\ (D\ x))\ x \qquad (1)$$

where for all i

$$g\ BC_i = \alpha_i \qquad (2)$$

For the destructor style of recursive definitions, the proof obligation can easily be formalised. The function f exists if for every x of type "*" a finite number of applications of the destructor D will eventually result in a base case. The proof obligation can thus be expressed as follows:

$$\forall x.\exists k.IS\_BC\ (D^k\ x) \qquad (3)$$

The combination of formulae 1 and 3 yields the following theorem:

3.1

$$\begin{aligned} &\forall D\ IS\_BC\ . \\ &\quad (\forall x.\exists k.IS\_BC\ (D^k\ x)) \Rightarrow \\ &\quad (\forall g\ h.\exists f.f\ x = \\ &\qquad (IS\_BC\ x)\ \rightarrow\ g\ x\ \mid\ h\ (f\ (D\ x))\ x) \end{aligned}$$

Theorem 3.1is the basis of the definition tool that has been implemented. An analysis of 3.1 shows that for this class of recursive definitions, the existence of a specific function is fully determined by the properties of the destructor and the set of base cases. Compared with the constructor style of recursive definitions and the input format of the definition tool provided with the system, it has the following advantages

- Closed formula: both the well-foundedness of recursive scheme and the existence recursively defined function itself have a fixed format, a general proof strategy consists of a strategy to rewrite a specific definition to the fixed format of 3.1.
- There are no limitations on the destructor and the set of base cases (as long as the implicant can be proven)
- Theorem 3.1 does not depend on any theoretical concept, the proof obligation to be fulfiled is understandable for any HOL user.
- The reason for the existence of a specific function is fully captured in the implicant of 3.1. The proof can thus be given separately from the actual definition (divide and conquer). Special support for this proof can be implemented

In the next section the formalisation of the theorem in HOL will be discussed.

3.2 Functions of arity one

Let the function fpw stand for the function power defined in the following way:

3.2

```
⊢def  (∀f x. fpw f 0 x = x)  ∧
      (∀f n x. fpw f (SUC n) x = f (fpw f n x))
```

Then, the implicant of theorem 3.1 can be written as:

```
∀x. ∃k . IS_BC (fpw D k x)
```

A predicate IS_BC and a destructor function D for which this hold are called a *recursive pair*.

3.3

```
⊢def ∀D IS_BC. IS_REC_PAIR (D,IS_BC) =
            (∀x. ∃k. IS_BC (fpw D k x))
```

Two examples of recursive pairs are:

⊢ IS_REC_PAIR(TL,λl.l = []) ,

⊢ IS_REC_PAIR(PRE,λn.n = 0)

The basic theorem of the theory on recursive functions (3.1) states that every recursive pair defines a well-founded recursion

3.4

```
⊢ ∀D IS_BC.
  (IS_REC_PAIR (D,IS_BC)) ⇒
  (∀g f. ∃fun:*→ ** . ∀x.
    (fun x = (IS_BC x) →  g x | f (fun (D x)) x))
```

3.3 Functions of a higher arity

Two types of functions with a higher arity can be distinguished: those that recurse in a single argument and functions recursing in several arguments.

When the function has two arguments, say x of type ":*1" and y of type ":*2", and when the function recurses on x according to the recursive pair (D,IS_BC), then theorem 3.4 can be extended:

3.5

```
⊢ ∀D IS_BC.
  (IS_REC_PAIR (D,IS_BC)) ⇒
  (∀g f h. ∃fun:*1→ *2→ ** . ∀x y.
    (fun x y = (IS_BC x) →  g x y |
                        f (fun (D x) (h x y)) x y))
```

Note that the function h is universally quantified. The second argument of the recursive function thus has no effect on the well-foundedness of the definition. The existence of f only depends on the recursive pair.

For the definition of functions recursing in *n* arguments and having m non-recursive arguments, the types of the variables x and y are instantiated with the appropriate product types. For every variable z of type ":*1# ... #*n" the following holds:

3.6

```
⊢ ∀z. ∃z1 ... zn . z = (z1,...,zn)
```

Theorem 3.5 can be transformed into (omiting the types) the following:

3.7

```
⊢ ∀D IS_BC.
  (IS_REC_PAIR (D,IS_BC)) ⇒
  (∀g f h. ∃fun. ∀x1 ... xn y1 ... ym .
   (fun (x1,...,xn) (y1,...,ym) =
    (IS_BC (x1,...,xn) →
      g (x1,...,xn) (y1,...,ym) |
      f (fun (D (x1,...,xn)) (h (x1,...,xn) (y1,...,ym)))
                          (x1,...,xn) (y1,...ym)))
```

The destructor is a function of the following form:

3.8

```
D = λ(x1,...,xn). (D1 (x1,...,xn), ... , Dn (x1,...xn))
```

The definition tool will rewrite a specific recursive definition into the format of theorem 3.7. Tuples and uncurried functions are extensively. This is not common practice in HOL and the available support is limited. A set of conversions dealing with tuples therefore has been developed.

4 The definition tool

The definition of recursive functions based on theorem 3.7 consists of two parts:

1. Fulfiling the proof obligation: given a destructor and a set of base cases, prove that they form a recursive pair.
2. Existence proof: rewrite a given definition to the form of 3.7 to prove its existence

The tool can handle only recursive function definitions that can be transformed into the syntactic format of either theorem 3.4 or of theorem 3.7.

4.1 The proof obligation

The fulfilment of the proof obligation

4.1

```
IS_REC_PAIR (D,IS_BC)
```

is left to the user of the tool. This issue cannot be solved in general and thus cannot be automated. However, support is essential to make the tool useful for the HOL user. The tool currently supports two different cases:

1. *Combination of recursive pairs.*

   Given a list of recursive pairs,

   ```
   [⊢ IS_REC_PAIR (D1,IS_BC1) ;
    ⊢ IS_REC_PAIR (D2,IS_BC2) ;
          ...                 ;
    ⊢ IS_REC_PAIR (Dn,IS_BCn)]
   ```

   the combination of the destructors and the disjunction of the base case predicates also form a recursive pair

   ```
   ⊢ IS_REC_PAIR (λ(x1,..,xn).(D1 x1, ... , Dn xn),
                  λ(x1,..,xn).(IS_BC1 x1 ∨ ... ∨ IS_BCn xn))
   ```

   The tool provides a logic function, COMBINE_IS_REC_PAIR_THM. This function generates from a given list of recursive pair the related combined recursive pair.

2. *Existence of a monotonically decreasing "measure".*

   A monotonically decreasing measure is a function of type ":*→ num" having the property that every base case is mapped to 0 and that for all other elements of type ":*" the function value of the element is larger than the value of the destructed element [12].

```
4.2
⊢def ∀D IS_BC.IS_MEAS (D,IS_BC) =
       ∃Meas. ∀x. (IS_BC x) → (Meas x = 0) |
                              (Meas (D x) < Meas x)
```

   It can be proven that every monotonically decreasing measure defines a recursive pair

```
4.3
⊢ ∀D IS_BC.
    IS_MEAS (D,IS_BC) ⇒
      IS_REC_PAIR(D,IS_BC)
```

   The tool provides a logic function, MEAS_TO_REC_PAIR. This function starts from a measure and proves that the destructor and the base case predicate form a recursive pair. Several input formats are allowed. This will be illustrated in the examples.

4.2 The existence proof

The existence proof is implemented in the function prove_function_exists:

```
4.4
prove_function_exists :
 (thm -> term -> term list -> thm)

prove_function_exists
   ⊢ IS_REC_PAIR (D , IS_BC)
   "f x = IS_BC x → g x | h (f (D x)) x"
   ["x:*"]
```

The first argument is a theorem stating that a given destructor D and a given base case predicate IS_BC form a recursive pair. The second argument is term that defines the function f. The same destructor and base case predicate as in the recursive pair must be used. The last argument is a list that contains the names of the arguments in which the function f recurses. The function prove_function_exists proves the existence of a function f as specified in the second argument.

```
4.5
⊢ ∃f. ∀x. f x = IS_BC x → g x | h ( f (D x)) x
```

From this theorem the function f can be defined by means of a new_specification.

5 Examples

In this section several examples of the use of the definition tool will be provided. These examples demonstrate the flexibility and the power of the tool. It also shows the limitations: only primitive recursive functions that can be cast into the required syntactical format can be defined.

5.1 Functions of arity one

A first example is the definition of the factorial function.

```
5.1
fac = "fac n = ((n = 0) → 1 | n * (fac(n − 1)))"
```

This function definition is legitimate, since for every natural number n the recursion will eventually end in the base case $n = 0$. The proof uses the approach of the monotonically decreasing measure. The measure is the identity function $\lambda n.n$. The base case predicate is $\lambda n.n = 0$. The proof obligation to be fulfiled is:

```
5.2
thm1 = ⊢ ∀n. (n = 0) ∨ (n − 1) < n
```

The following HOL session shows how the definition of the factorial function proceeds.

```
5.3
let rp1 = MEAS_TO_REC_PAIR thm1;;
rp1 = ⊢ IS_REC_PAIR((λn. n − 1),(λn. n = 0))

let Fac = "Fac n = (n=0) → 1 | n*(Fac (n−1))";;

let FAC_exist = prove_function_exists rp1 Fac ["n:num"];;
FAC_exist =
⊢ ∃Fac. ∀n. Fac n = ((n = 0) → 1 | n * (Fac(n − 1)))

new_specification 'FAC' ['constant','Fac'] FAC_exist;;
⊢ ∀n. Fac n = ((n = 0) → 1 | n * (Fac(n − 1)))
```

A less trivial example, which cannot be handled with the existing tools for defining recursive functions, is the definition of a logarithm-like function for naturals. As the current implementation of the tool does not support the definition of partial functions. This causes a problem with the logarithm at 0. This can either be taken to be an arbitrary value of type ":num" or it can be taken to be 0. The second option has been chosen for. Let div2 stand for the natural division by 2 in the following term that will be used for the definition of the function log2

```
5.4
log2 = "log2 n = ((n=0) ∨ (n=1)) → 0 |
                        (log2 (div2 n)) + 1"
```

This recursive definition is legitimate, since for every natural number n, a finite number of division by 2 will eventually result in 0 or 1. The proof uses the concept of the monotonically decreasing measure. This reduces the proof obligation to the following:

5.5
```
⊢ ∀n. (((n = 0) ∨ (n = 1)) → (n − 1 = 0) |
                    ((div2 n) − 1) < (n − 1))
```

The function MEAS_TO_REC_PAIR transforms theorem 5.5 into this:

5.6
```
rp4 =
⊢ IS_REC_PAIR((λn. div2 n),(λn. (n = 0) ∨ (n = 1)))
```

From theorem 5.6 the existence of the logarithmic function can be derived:

5.7
```
prove_function_exists rp4 log2 ["n:num"];;
⊢ ∃log2. ∀n. log2 n =
    (((n = 0) ∨ (n = 1)) → 0 | (log2(div2 n)) + 1)
```

5.2 Functions of a higher arity, recursing in a single argument

The MAP function of the theory list written in the destructor style is given by:

5.8
```
Map =
"Map f X = ((X = []) → [] |
            CONS(f (HD X))(Map f (TL X)))"
```

This function recurses in a single argument, i.e. the list X. The function f is a free variable and theorem 3.5 is used for the definition of Map. Starting from:

5.9
```
rp3 = ⊢ IS_REC_PAIR((λX. TL X),(λX. X = []))
```

the existence theorem can be generated:

5.10
```
prove_function_exists rp3 Map ["X:* list"];;
⊢ ∃Map. ∀X f. Map f X = ((X = []) → [] |
            CONS(f (HD X))(Map f (TL X)))
```

The third argument of prove_function_exists states that the list X is the only recursive argument and thus that f is not a recursive argument.

5.3 Functions recursing in several arguments

The definition of functions recursing in several arguments will be illustrated by the definition of a simple bitstring adder. Bitstrings are modeled by a list of booleans. The most significant bit first (MSB) representation of bitstrings is used, as this is common practice in hardware description. This means that the last bit of a given bitstring denotes the least significant bit. The addition of two bitstrings therefore recurses on the last elements of the lists that represent the bitstrings. The list destructing functions LAST and BUTLAST (5.11) instead of HD and TL have to be used.

The following non-recursive functions are used

5.11
```
XOR  ⊢ ∀x y. x xor y = ¬x ∧ y ∨ x ∧ ¬y
ADDB ⊢ ∀c x y. addb c x y = x xor (y xor c)
COUT ⊢ ∀c x y. cout c x y = x ∧ y ∨ y ∧ c ∨ c ∧ x
HADD ⊢ ∀c x. hadd c x = c xor x
HCOUT ⊢ ∀c x. hcout c x = c ∧ x
LAST ⊢ ∀l y. LAST(APPEND l[y]) = y
BUTLAST ⊢ ∀l y. BUTLAST(APPEND l[y]) = l
```

The following recursive pair is used as a starting point:

5.12
```
rp =
⊢ IS_REC_PAIR((λx. BUTLAST x),(λx. x = []))
```

From theorem 5.12 the following recursive pair can be derived:

5.13
```
let add_thm = COMBINE_IS_REC_PAIR_THMS [rp;rp];;
add_thm =
⊢ IS_REC_PAIR
    ((λ(x,x'). (BUTLAST x,BUTLAST x')),
    (λ(x,x'). (x = []) ∨ (x' = [])))
```

Theorem 5.13 states that a recursive definition, simultaneously destructing two lists by removing their last element, and that ends if one of both lists is empty, is legitimate. The base case of this recursive scheme has to handle the case where one or both of the lists is empty. First, an intermediate function is defined with the techniques described in section 5.2. This function defines the addition of a bitstring with a single bit.

5.14

```
SINGLE_ADD =
⊢ ∀X ci.
   single_add ci X =
   ((X = []) →
    [ci] |
    APPEND
     (single_add(hcout ci(LAST X))(BUTLAST X))
     [hadd ci(LAST X)])
```

This function is used in the definition of the addition itself:

5.15

```
let ADD_term =
  "add ci X Y = ((X=[]) ∨ (Y=[])) →
       ((X = []) →  single_add ci Y  |
                    single_add ci X) |
       APPEND
         (add (cout ci (LAST X) (LAST Y))
              (BUTLAST X)
              (BUTLAST Y))
         [addb ci (LAST X) (LAST Y)]"
```

is defined with prove_function_exists followed by a new_specification

5.16

```
⊢def ∀X Y ci.
  add ci X Y =
    (((X = []) ∨ (Y = [])) →
    ((X = []) →  single_add ci Y | single_add ci X) |
    APPEND
     (add(cout ci(LAST X)(LAST Y))
        (BUTLAST X)
        (BUTLAST Y))
     [addb ci(LAST X)(LAST Y)])
```

Higher order functions recursing in several arguments can also be defined. An example is the definition of a two-argument MAP function:

5.17

```
⊢def ∀f X Y .
  MAP2 f X Y =
    (X = []) ∨ (Y = []) →  [] |
              CONS (f (HD X) (HD Y))
                   (MAP2 f (TL X) (TL Y))
```

The definition of MAP2 can be used for a non-recursive definition of the function AND of 2.6.

5.4 Discussion

The examples show how the tool can be used for defining primitive recursive functions in HOL. The class of recursive functions that can be defined is restricted by the syntactic format of 3.7: only definitions that can be re-cast in this format can be defined once the proof obligation has been fulfiled.

The tool splits the definition of a recursive function in three parts:

1. prove of the well-foundedness of the recursion
2. prove of the existence of the function
3. constant specification

The first part is left to the user, but support is provided. It involves the proof that a given destructor and a given base case predicate form a recursive pair.

The second part is fully automated: any function that recurses according to a given recursive pair and that can be reduced into the form of 3.7 can be defined.

The third part uses the logic function new_specification.

Because of the support the tool provides, and because of the simplicity of the proof obligation, the tool offers a high degree of flexibility.

The tool has two major disadvantages. First, only complete functions can be defined. Secondly, this style of definitions cannot be used for simple rewriting since the HOL rewrite strategy will loop.

6 Induction

Every recursive pair also defines an induction scheme.

6.1

```
⊢ ∀Q (D:*→ *) IS_BC. (IS_REC_PAIR (D,IS_BC)) ⇒
   (∀x. (IS_BC x) ⇒  Q x) ⇒  (∀x. (Q (D x) ⇒  Q x))
        ⇒  (∀x. Q x)
```

By proper instantiations, the theorem can be used for an induction on any number of arguments as long as the proof obligation has been fulfiled. The proof obligation is the same as for recursive pairs. This means that every recursive scheme can be used an induction scheme. A function for the generation of the appropriate induction schemes will be provided in the future.

7 Conclusion

In this document, a theory on recursive functions has been presented. The theory forms the basis of a tool that minimizes the proof effort for the definition of wide class of recursive functions and the derivation of induction schemes. The basic idea is the concept of "recursive pairs", stating that a given destructor and a base case predicate define a valid recursion. Once this proof obligation has been fulfiled, the function definition can be derived automatically. In all application domains where extensive use of recursive functions and induction is made, the tool offers a useful extension of the HOL system.

References

[1] H.De Man et al., *Cathedral-II: A silicon compiler for digital signal processing*, IEEE Design and Test of Computers, December 1986, Vol. 3, No. 6, pp 73-85.

[2] B.Brock, W.A.Hunt,Jr, *Report on the Formal Specification and Partial Verification of the VIPER Microprocessor*, Tech.Rep. 46, Jananuary 15, 1990, Computational Logic, Inc.

[3] P.Lowenstein, *Expereriences Using a Theorem Prover for Hardware Verification*, 1991 International Workshop on Formal Methods in VLSI Design, January 1991, Miami.

[4] R.Boulton et al., *The* HOL *Verification of* ELLA *Designs*, 1991 International Workshop on Formal Methods in VLSI Design, January 1991, Miami.

[5] A.Gordon, *The definition of* TINY-SILAGE, University of Cambridge Computer Laboratory, August 1991.

[6] S.Kalvala, *A Methodology for Integrating Hardware Design and Verification*, 1991 International Workshop on Formal Methods in VLSI Design, January 1991, Miami.

[7] J.J.Joyce, *Multi-Level Verification of Microprocessor Based Systems*, Ph.D. thesis, Computer Laboratory, Cambridge University, December 1989, Report No 195.

[8] T.F.Melham, *Abstraction Mechanisms for Hardware Verification*, in: G.Birtwistle and P.Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988, pp. 267-291.

[9] P.J.Windley, *Abstract Hardware*, 1991 International Workshop on Formal Methods in VLSI Design, January 1991, Miami.

[10] M.J.C.Gordon, *Why Higher-Order Logic is a good formalism for Specifying and Verifying Hardware*, in:G.Milne and P.Subrahmanyam, eds., Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Conference on VLSI, North-Holland, 1986, pp.153-177.

[11] J.J.Joyce, *More Reasons Why Higher-Order Logic is a good formalism for Specifying and Verifying Hardware*, 1991 International Workshop on Formal Methods in VLSI Design, January 1991, Miami.

[12] R.S.Boyer and J.S.Moore, *A Computational Logic*, Academic Press, Inc., 1979.

[13] M.J.C.Gordon et al., *The HOL System: DESCRIPTION*, Cambridge Research Centre, SRI International, December 1989, Version 1 (for HOL88.1.10).

[14] H.Rogers, Jr., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.