# Evaluation Techniques as a Part of the Verification Process

Dirk Eisenbiegler and Ramayya Kumar
Forschungszentrum Informatik
Haid–und–Neustraße 10-14 76131 Karlsruhe, Germany
e–mail: eisen@fzi.de, kumar@fzi.de

August 12, 1994

**Abstract**

Verifying an implementation means proving that the implementation meets a given formal specification. For small sized implementations, exhaustive simulation can be an appropriate way to obtain the proof. But with the complexity of the implementations growing larger and larger, the number of cases to be considered increases exponentially and exhaustive simulation is not suitable any more.

In this paper implementations will be represented within a functional programming language. The evaluation of typed $\lambda$–terms corresponds to the simulation of the implementation. Besides conventional, nonsymbolic evaluation, advanced evaluation techniques will be presented dealing with symbolic evaluation and termination detection. Evaluation as described in this paper can be both an advanced means for simulation and also a part of a verification tactic.

## 1 Introduction

The approach presented in this paper is based on a simple functional programming language called PML. PML is a subset of HOL used for the representation of arbitrary computable functions.

PML is a rather poor language based on the typed $\lambda$–calculus. In this paper PML functions will always be represented as HOL–terms. But with some slight syntactical modifications, a PML program represented as HOL terms can also be turned into a program of a functional programming language such as ML. Within an ML interpreter, PML programs can be run just as any other ML program. The HOL representation of PML will be used to prove certain properties of PML functions (verification) or to convert the PML functions into equivalent but optimized PML functions.

The execution of PML programs (evaluation of PML terms) can also be performed by a conversion within HOL. The result of such an evaluation will be a proven theorem stating that the term to be evaluated is equivalent to a specific result. The execution of PML programs within an ML interpreter will also determin the same result but without proving this. For pure simulation purposes it is convenient to use the ML interpreter rather than an evaluation conversion within HOL, since the ML interpreter runs much faster.

Interpreters of functional programming languages expect a term consisting of constructors, constants and functions. Usually free variables are not allowed. If the result is defined (i.e. $f(x)\downarrow$) the evaluation process will stop after a certain time returning a term exclusively consisting of constructors. Whenever the result is not defined (i.e. $f(x)\uparrow$), the evaluation process will not terminate.

During the verification process it can be helpful to also evaluate terms comprising free variables. In contrast to conventional evaluation, results of symbolic evaluations are not unambiguous. The results of the evaluation of two equivalent terms will be equivalent but they may be unequal. It is not possible any more to prove the equivalence of two terms by just evaluating them and afterwards checking whether the results are equal. It will be discussed, how this problem can be overcome.

Another extension of interest for the verification process is the detection of termination and nontermination. Conventional evaluation algorithms terminate iff the term is defined (i.e. $f(x)\downarrow$). An evaluation algorithm that *always* detects whether or not the term will terminate, is not computable (halting problem). But it is possible to write an algorithm, that besides computing the result whenever it is defined (i.e. $f(x)\downarrow$) also *sometimes* detects nontermination. Such an algorithm becomes the more powerful, the more nontermination situations he can detect. It will be discussed, how algorithms for nontermination detection may look like.

The paper is structured as follows: First an introduction to the formalization methodology of PML is given in section 2. Section 3 explains the reasons for this approach. In section 4 it is explained, which are the basic HOL transformations that are needed to perform an evaluation. The order of these basic equivalence transformations has no influence on the correctness of the result but changing the order may speed up the algorithm.

Such optimizations are discussed in section 5 . Section 6 deals with symbolic evaluations and section 7 is concerned with the detection of nontermination.

## 2 A Brief Introduction to PML

The functional programming language PML is used for formalizing $\mu$–recursive functions. It allows the user to define data types, constants and functions. PML is closely related to HOL: PML data types correspond to HOL–style data types and the constant and function definitions of PML correspond to constant definition of HOL. PML is a subset of HOL. Its expressive power is restricted but the considered functions are all computable.

Conventional functional programming languages such as ML allow the user to define recursive functions by a set of recursive equations, i.e. equations where the function symbol that is to be defined may also appear within the right hand sides. Since function definitions of PML correspond to constant definitions of HOL, such definitions are not allowed. A PML function definition must consist of a single equation and the function symbol that is to be defined must not appear on the right hand side. Although recursive equations are not allowed, recursion can also be expressed in PML.

There are two means for describing recursive PML functions: one fore primitive recursive functions and another for $\mu$–recursive functions.

### 2.1 Primitive Recursion

There are certain basic functions corresponding to data types: For every data type xyz there is a corresponding basic function called PRIMREC_xyz. These functions can be used to express primitive recursion over the corresponding data types. They are defined automatically whenever a data type is added.

The natural numbers for example can be defined as a HOL–style data type:

```
num = Zero | Suc of num
```

From this definition, the function PRIMREC_num is automatically derived as follows:

$$\vdash \mathsf{PRIMREC\_num}\; f\; c\; \mathsf{Zero} = c \tag{1}$$

$$\vdash \mathsf{PRIMREC\_num}\; f\; c\; (\mathsf{Suc}\; n) = f\; n\; (\mathsf{PRIMREC\_num}\; f\; c\; n) \tag{2}$$

Based on this basic function, arbitrary primitive recursive functions over $\mathsf{num}$ can be derived by means of HOL constant definitions. In ML, primitive recursive functions such as the sum $+$ over two natural numbers, would be expressed by a set of equations:

$$\begin{aligned} +(\mathsf{Zero}, b) &= b \\ +(\mathsf{Suc}\, a, b) &= \mathsf{Suc}(+(a, b)) \end{aligned}$$

Such a set of equations with the functor being defined appearing on the right hand side, is not a valid PML function definition. In PML the same function can be expressed by:

$$+(a, b) = \mathsf{PRIMREC\_num}\, (\lambda x.\, \mathsf{Suc})\, a\, b$$

In the examples below, it is preassumed that the functions $+$ (sum of two natural numbers), $*$ (product of two natural numbers) and $\mathsf{sqr}$ (the square of a natural number) and the constants $0, 1, 2, \ldots$ have already been defined as described in the equations (3) through (8). To improve the readability $+$ and $*$ will also be used in infix notation with the usual binding priorities.

$$\vdash 0 = \mathsf{Zero} \tag{3}$$

$$\vdash 1 = \mathsf{Suc}\; 0 \tag{4}$$

$$\vdash 2 = \mathsf{Suc}\; 1 \tag{5}$$

$$\vdash +(a, b) = \mathsf{PRIMREC\_num}\, (\lambda x.\, \mathsf{Suc})\, a\, b \tag{6}$$

$$\vdash *(a, b) = \mathsf{PRIMREC\_num}\, (\lambda x\, y.\, +(y, a))\, \mathsf{Zero}\, b \tag{7}$$

$$\vdash \mathsf{sqr}\; a = a * a \tag{8}$$

## 2.2 $\mu$–Recursion

There is also a means for describing $\mu$–recursion: the basic function $\mathsf{WHILE}$. In contrast to ML, the result of a PML function is always specified in an explicit manner — a specific value is given even when it is undefined (i.e. $f(x)\uparrow$). The data type $\mathsf{('a)partial}$ is used to express results of $\mu$–recursive functions.

```
partial = Defined of 'a | Undefined
```

$f(x) = \mathsf{Undefined}$ stands for $f(x)\uparrow$ and $f(x) = (\mathsf{Defined}\; y)$ means that $f(x)$ is defined (i.e. $f(x)\downarrow$) and that the result is y.

$\mathsf{WHILE}$ performs a loop. The term $(\mathsf{WHILE}\, g\, f\, x)$ calculates the result of the iteration of the function $f$ starting with an initial state $x$. The iteration is performed until a state $y$ is reached that does not fulfill the predicate $g$. If such a state $y$ exists then $(\mathsf{WHILE}\; g\; f\; x)$ is equivalent to $(\mathsf{Defined}\; y)$ else it is equivalent to $\mathsf{Undefined}$. [1]

$$\neg(g\;x) \vdash \mathsf{WHILE}\; g\; f\; x = \mathsf{Defined}\; x \tag{9}$$

$$g\;x,\quad f\;x = \mathsf{Undefined} \vdash \mathsf{WHILE}\; g\; f\; x = \mathsf{Undefined} \tag{10}$$

$$g\;x,\quad f\;x = \mathsf{Defined}\; y \vdash \mathsf{WHILE}\; g\; f\; x = \mathsf{WHILE}\; g\; f\; y \tag{11}$$

$$\vdash (\mathsf{WHILE}\; g\; f\; x = \mathsf{Undefined}) = \neg(\exists n y. f^n(x) = \mathsf{Defined}\; y \wedge (g\; y)) \tag{12}$$

# 3 HOL, PML and ML

The HOL system already provides means for defining ML–style functions: data type declarations, constant definitions and primitive recursive functions over single data types. Using these three facilities, arbitrary primitive recursive functions can be defined. All these functions are total and they are all computable and can easily be evaluated.

$\mu$–recursive functions that are not primitive recursive cannot be defined this way. Therefore these facilities do not provide a sufficient means for formalizing a functional programming language. This is why the function WHILE has been introduced. WHILE provides a means for expressing $\mu$–recursion.

Inspite of WHILE, another adequate $\mu$–recursive function such as the $\mu$–operator could have been used to express $\mu$–recursion. The function WHILE has been chosen because of technical reasons: It can easily be implemented in ML and it allows the user to program loops that can efficiently be evaluated within a conventional ML interpreter.

The way of expressing recursive functions in PML completely defers from conventional ML programs. There are no recursive equations, i.e. equations with the function symbol appearing on the right hand side. Starting from the primitive recursive functions PRIMREC_xyz and the $\mu$–recursive function WHILE, everything is built up by means of constant definitions. This is why writing PML programs is more difficult than writing ML programs. But it is rather difficult to prove consistency when using recursive equations for describing nested primitive or even $\mu$–recursive functions. PML functions

[1]The constant WHILE has not really been introduced as described in (9) ... (11) but by a conventional constant definition (see [EiSK93] for details). This set of equations has been derived from the original definitions.

are always build up by constant definitions which definitely does preserve consistency.

There is another difference between PML and ML functions: PML functions are always total, i.e. an explicit value is defined for every input — even when the function does not terminate. Two PML functions are equivalent iff the functions terminate/do not terminate for the same inputs and the results are the same when they terminate. ML–style $\mu$–recursive functions that where defined using recursive equations are partial (they do not assign an explicit output when the function does not terminate). This is why it is not possible to prove, that two functions that terminate/do not terminate for the same inputs and the results are the always the same when they terminate are equivalent.

# 4 Evaluation

Evaluating a term means converting it into an equivalent term exclusively consisting of constructors. It must be provided that the term to be evaluated does not comprise any free variables. The type of the term to evaluated must be a compound data type. Terms with a type comprising the type operator "$\rightarrow$" are not allowed. The term $(\lambda x.\mathsf{sqr}(\mathsf{sqr}\ x))$ for example is not allowed since it is of type $\mathsf{num} \rightarrow \mathsf{num}$. In general the result of such terms is not unambiguous.

There are four basic transformations for performing an evaluation:

1. expanding constants
2. $\beta$–reduction
3. evaluation of PRIMREC–functions
4. evaluation of WHILE–functions

Since every PML function definition and every PML constant definition corresponds to a constant definition in HOL, PML constants and functors are nothing but abbreviations. All PML constant and function definitions can be transformed into the form $\vdash c = t$ where $c$ is the constant and $t$ is a closed term. HOL constants that have been defined by a constant definition can always be substituted by the term they stand for. Since there are no recursive equations with the functor to be defined appearing on the right hand side and since mutual recursive function definitions are not allowed (constants have to be defined one by one and constants have to be defined before they are used), the complete expansion of constants and functors can always be performed in a finite number of steps.

To express functions within PML, $\lambda$–abstraction is used. When applying them to terms $\beta$–redices arise. $\beta$–redices can be reduced by $\beta$–reduction. The $\beta$–reduction substitutes the local parameters by the the term the function has been applied to.

For a technical reason concerning the evaluation of WHILE-expressions the derived function while is introduced by:

$$\begin{aligned}\vdash\ & \mathsf{while}\ c\ y\ g\ f\ x = \\ & \quad c => \\ & \quad\quad (\mathsf{PRIMREC\_partial}\ y\ (\lambda z.\mathsf{WHILE}\ g\ f\ z)\ \mathsf{Undefined})\ | \\ & \quad\quad (\mathsf{Defined\ x})\end{aligned} \tag{13}$$

The following equations can be derived:

$$\vdash \mathsf{WHILE}\ g\ f\ x = \mathsf{while}\ (g\ x)\ (f\ y)\ g\ f\ x \tag{14}$$

$$\vdash \mathsf{while\ F}\ y\ g\ f\ x = \mathsf{Defined}\ x \tag{15}$$

$$\vdash \mathsf{while\ T\ Undefined}\ g\ f\ x = \mathsf{Undefined} \tag{16}$$

$$\vdash \mathsf{while\ T}\ (\mathsf{Defined}\ z)\ g\ f\ x = \mathsf{WHILE}\ g\ f\ z \tag{17}$$

Basic functions can be evaluated by rewriting. Terms that are exclusively consisting of constructors and basic functions (PRIMREC–functions and WHILE) can be evaluated by rewriting using all the definitions of the PRIMREC–functions (such as (1) and (2)) and the equations (14) through (17).

Summary: The whole evaluation of PML–terms can be performed by $\beta$–reduction and rewriting. The set of the equations needed for the rewriting is fixed. It consists of the equations of the basic functions (the PRIMREC–functions and WHILE) and the definitions of the derived functions and constants. The evaluation terminates, when neither an equation nor a $\beta$–reduction can be applied. When this state is reached, then there is nothing left but constructors. Such an evaluation will always terminate except that there is a non–terminating loop, i.e. the expression to be evaluated is equivalent to Undefined. Fig. 1 gives an example for an evaluation of a PML term.

# 5 Optimizing the Evaluation

In a certain state of the evaluation, there usually are several basic transformations that can alternatively be applied. The result of the evaluation will

$$\boxed{1 * 0 + 1}$$

$$\downarrow \text{ rewriting with (4)}$$

$$\boxed{(\mathsf{Suc}\ 0) * 0 + (\mathsf{Suc}\ 0)}$$

$$\downarrow \text{ rewriting with (3)}$$

$$\boxed{(\mathsf{Suc\ Zero}) * \mathsf{Zero} + (\mathsf{Suc\ Zero})}$$

$$\downarrow \text{ rewriting with (7)}$$

$$\boxed{(\mathsf{PRIMREC\_num}\ (\lambda x\ y.\ +(y, \mathsf{Suc\ Zero}))\ \mathsf{Zero\ Zero}) + (\mathsf{Suc\ Zero})}$$

$$\downarrow \text{ rewriting with (1)}$$

$$\boxed{\mathsf{Zero} + (\mathsf{Suc\ Zero})}$$

$$\downarrow \text{ rewriting with (6)}$$

$$\boxed{\mathsf{PRIMREC\_num}\ (\lambda x.\ \mathsf{Suc})\ \mathsf{Zero}\ (\mathsf{Suc\ Zero})}$$

$$\downarrow \text{ rewriting with (2)}$$

$$\boxed{(\lambda x.\ \mathsf{Suc})\ \mathsf{Zero}\ (\mathsf{PRIMREC\_num}\ (\lambda x.\ \mathsf{Suc})\ \mathsf{Zero\ Zero})}$$

$$\downarrow \text{ rewriting with (1)}$$

$$\boxed{(\lambda x.\ \mathsf{Suc})\ \mathsf{Zero\ Zero}}$$

$$\downarrow \beta\text{–reduction}$$

$$\boxed{\mathsf{Suc\ Zero}}$$

Figure 1: Evaluation of a PML-Term

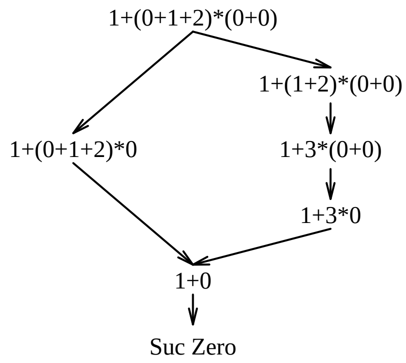


Figure 2: Variants of Evaluating a Term

always be the same no matter how the evaluation is performed, i.e. which basic transformations are chosen in certain states of the evaluation. Distinct subterms can even be evaluated in parallel. Although the evaluation will always have the same result, the decision of which basic transformation to apply next is significant as to the evaluation speed and the amount of data needed during the evaluation.

Fig. 2 sketches two ways to evaluate $1{+}(0{+}1{+}2){*}(0{+}0)$. The evaluation on the left hand side starts with the evaluation of $0 + 0$. Since the result is 0, it turns out that the term is independent from the result of the evaluation of $0 + 1 + 2$ so that this subterm is not evaluated. The evaluation on the right hand side is slower since it starts with the (unnecessary) evaluation of $0 + 1 + 2$ calculating a result that will not be needed in further steps.

### 5.1 Strict Evaluation and Lazy evaluation

Good evaluation algorithms must try to minimize the number of basic operations by making a good choice, which basic operation is to be performed next. There will never be a perfect evaluation algorithm since very often the best order cannot be determined in advance but depends on the results of subterm evaluations.

Example: The evaluation of $t_1 * t_2$ can either start with the evaluation of $t_1$ or with $t_2$. When either $t_1$ or $t_2$ is equivalent to 0 then the optimal algorithm would have to start with the one that is equivalent to 0. The evaluation of the other term may be omitted. But this decision cannot be made unless $t_1$ and $t_2$ (or at least one of them) have already been evaluated.

There are two basic evaluation techniques: strict evaluation and lazy evaluation. The strict evaluation algorithm is based on the principle: evaluate the parameters first. A term $(a\ b)$ is evaluated by first evaluating $b$ to $b'$ and then evaluating $(a\ b')$. The lazy evaluation algorithm is based on the principle: a subterm is not evaluated unless it is needed. The term $(a\ b)$ is evaluated by first reducing $a$ to $a'$. The term $b$ will only be evaluated if it occurs in $a'$ else it is not. The lazy evaluation algorithm minimizes the number of basic function evaluations, but there is an additional requirement of time and memory for the organization. It depends on the term to be evaluated, which algorithm is more efficient.

### 5.2 Precomputation

During the evaluation it may happen, that subterms have to be evaluated which have already been evaluated before. Reusing results of former evaluations can reduce the expense of the evaluation. During the evaluation process, results of subterm evaluations could be stored so that if in further evaluation steps these subterms can be looked up rather than being calculated again. Storing and handling such results leads to an additional requirement of time and data. An advantage can be achieved only if the result being stored will definitely be reused at least once.

It is also possible to do some calculations even before the evaluation starts. When a function with a small number of possible input values is used rather frequently, it may be convincing to calculate all the results and store them in a table. For a function $\mathsf{f} : (bool * bool * bool) \rightarrow bool$ for example there are only eight cases that have to be considered: $(F, F, F)$, $(F, F, T)$, $\ldots$ $(T, T, T)$. Before the evaluation starts, a table with eight entries can be calculated holding all possible input values and the corresponding results. Whenever a term such as $f(F, T, F)$ appears during the evaluation, the entry $(F, T, F)$ can be looked up rather than being calculated.

## 6 Symbolic Evaluation

In previous sections, it has always been preassumed that there are no free variables in the terms that are to be evaluated. This section attends to symbolic evaluation, i.e. the evaluation of terms comprising free variables. The basic transformations needed for a symbolic evaluation are the same as those for the conventional evaluation: constant expansion, $\beta$–reduction and the evaluation of the basic functions.

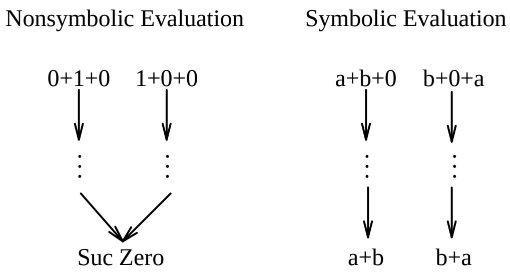


Figure 3: Symbolic Evaluation

## 6.1 Motivation

Symbolic evaluation is an extended means for simulation. A class of inputs, rather than one single input, is executed at once. The term $(a + 0) * (0 * b + 1) + b$ can be evaluated (reduced) to $a + b$. Symbolic evaluation is a substitute for a large number of single conventional evaluations with all possible instantiations of the free variables.

Obviously, the term $(a + 0) * (0 * b + 1) + b$ can be converted to $a + b$. But since PML functions (basic functions as well as derived functions) need not be injective (just as in any other functional programming language) the result of a symbolic evaluation need not be unambiguous. The result of $(a + 0) * (0 * b + 1) + b$ may as well be $b + a$ or $a + b + 0$. A nonsymbolic evaluation of two equivalent terms leads to two equal terms, since the terms consist of injective constructors only. In contrast to nonsymbolic evaluation, the symbolic evaluation of two equivalent terms may lead to two results that are equivalent but not equal. This turns out to be a disadvantage of symbolic evaluation. It is not possible any more to compare two terms just by evaluating them.

## 6.2 Meta–Constructors

In the introductory example $(a + 0) * (0 * b + 1) + b$ has been evaluated to $a + b$. In contrast to the previously sketched evaluation algorithm, not all possible basic transformations have been performed: $+$ is a derived function and could be expanded. Since the derived functions of PML are nothing but abbreviations, all derived functions may be eliminated from the result of a symbolic evaluation such that all functions within the result are but basic functions. But there seems to be no sense in expanding all derived functions

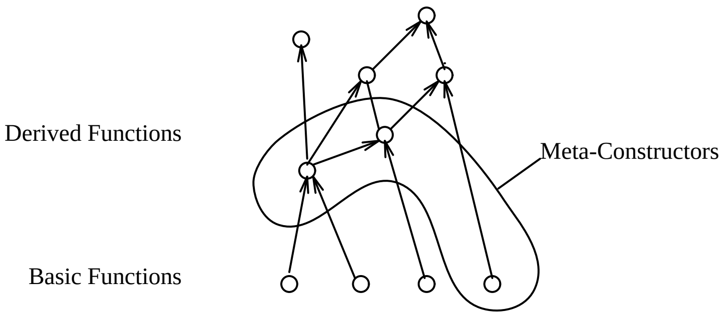


Figure 4: Defining a Set of Meta–Constructors

since it does not lead to an unambiguous representation. Expanding $a + b$ may lead to

$$\mathsf{PRIMREC\_num}\ (\lambda x\ y.\ \mathsf{Suc}\ y)\ b\ a$$

and evaluating an equivalent term may lead to

$$\mathsf{PRIMREC\_num}\ (\lambda x\ y.\ \mathsf{Suc}\ y)\ a\ b$$

It is not possible to eliminate all functors from the result of a symbolic evaluation. An arbitrary set of functions can be chosen, that shall not be expanded. These functions shall be called *meta–constructors*. Meta–constructors may be basic functions or derived functions. All functions to be considered in evaluation shall either be meta–constructors or functions that have been derived from meta-constructors (see fig. 4). When performing a symbolic evaluation the result will exclusively consist of free variables, constructors and meta–constructors.

In contrast to constructors, meta–constructors must not be injective and so the result must not be unambiguous. To obtain an unambiguous representation, the symbolic evaluation can be succeeded by a normalization step where different equivalent terms are transformed into equal terms. For such a normalization it is necessary to built a set of converging transformations upon the meta–constructors. A set of converging functions can be found by proving some equations and deriving a converging equation set using Knuth-Bendix completion.

Example: The set of meta–constructors consists of $+$ and $*$. After the evaluation, the result will exclusively consist of constructors ($\mathsf{Zero}$ and $\mathsf{Suc}$),

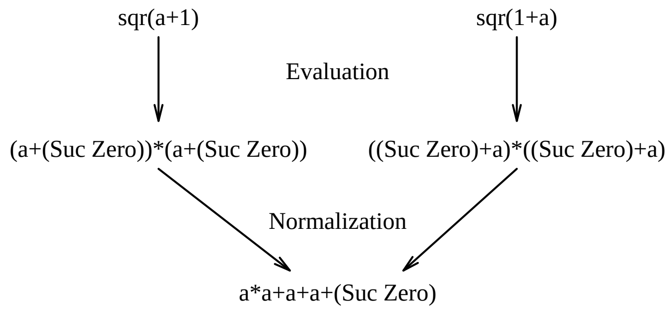


Figure 5: Symbolic Evaluation and Normalization

free variables and the meta–constructors $+$ and $*$. The following equations can manually be derived:

$$
\begin{array}{rclcrcl}
a * (b + c) & = & a * b + a * c & \quad & (\mathsf{Suc}\ a) + b & = & \mathsf{Suc}(a + b) \\
a + \mathsf{Zero} & = & a & & a + (\mathsf{Suc}\ b) & = & \mathsf{Suc}(a + b) \\
\mathsf{Zero} * a & = & \mathsf{Zero} & & (\mathsf{Suc}\ a) * b & = & b + (a * b) \\
a + b & = & b + a & & a * (\mathsf{Suc}\ b) & = & b + (a * b) \\
a * b & = & b * a & & & &
\end{array}
$$

An arbitrary term–ordering can be defined and by means of Knuth–Bendix a converging equation set can automatically be derived, which can be used to perform the normalization (see fig. 5).

# 7 Termination/Nontermination Detection

In PML the result of a non–terminating function has an explicit value called $\mathsf{Undefined}$ and the result of a terminating function is $(\mathsf{Defined}\ x)$ rather than $x$. Conventional evaluation algorithms will always terminate when the result is $(\mathsf{Defined}\ x)$ but will not terminate when the result is $\mathsf{Undefined}$. An evaluation algorithm that always decides whether a $\mu$–receive function is defined and that calculates the result whenever it is defined, would be beyond the limits of computability. But it is possible to extent conventional evaluation algorithms. Such an extended algorithm calculates the result whenever the result is defined, and in some cases it also halts even though the result is undefined returning the result $\mathsf{Undefined}$. There will always be some situations left where the result is undefined but the algorithm does not terminate.

Primitive recursive functions do always terminate. Nontermination can only occur when dealing with $\mu$–recursive functions and $\mathsf{WHILE}$ is the only basic means for describing $\mu$–recursion. Theorem (12) states that the result of a loop is Undefined whenever

$$\neg(\exists ny. f^n(x) = \mathsf{Defined}\ y \wedge (g\ y)) \tag{18}$$

is fulfilled. This problem is not computable, i.e. in general it cannot automatically be decided whether this formula holds or not. Although there is no general solution, algorithms can be constructed that produce results for certain situations.

Whenever the assumption (18) is not fulfilled, a nonsymbolic evaluation will find the result using the equations (14) trough (17). But when it is not, then such an evaluation will not terminate. This kind of evaluation can be improved by an additional feature: When evaluating a loop, the states the evaluation algorithm has already passed through are recorded and from time to time the algorithm takes a look a the list of states and tries to prove that the evaluation will never terminate.

When dealing with symbolic evaluation, the result may be ($\mathsf{Defined}\,x$) for certain instances and Undefined for others. Proving that a an algorithm is completely correct means proving that for every possible input $x$ the result will always be ($\mathsf{Defined}\ y$) with $x$ and $y$ standing in a relation described by the program specification. A successfully applied symbolic evaluation would do a big part of the correctness proof.

There are two problems that shall now be discussed:

- How can nontermination be detected during a nonsymbolic evaluation?
- How to prove that a loop terminates for all inputs of a function?

## 7.1 Detecting Nontermination

A general scheme to detect nontermination has already been sketched: recording the states when evaluating a WHILE loop and trying to derive a prove for nontermination from the states the loop has already passed trough.

The evaluation of a WHILE loop can be performed as follows: First the predicated $(g\ x)$ is evaluated where $x$ is the current state. If $(g\ x)$ equals F then $x$ is the result of the loop else the next state $(f\ x)$ is calculated and the evaluation of the loop continues with the new state. Every WHILE loop evaluation passes through a list of states $x, f(x), f^2(x), f^3(x), \ldots$ that all do hold $g$. If the result is defined then the number of these states is finite and there is a smallest $n$ such that $g(f^n(x))$ is not fulfilled. Otherwise this list is infinite.

A nontermination can be proven when a cycle is detected, i.e. when the evaluation passes through the same state for the second time. If the number of all possible states is finite (the type of the state represents a finite set), then the detection of nontermination is computable. If the number of evaluation steps is larger, than the total number of possible states then a cycle obviously has occurred. So if $n$ is the total number of states, then after having calculated at most $n$ states it can be made out whether the loop will ever terminate or not.

When dealing with sequential circuits, the number of states of an implementations is always finite. But the number of states of a circuit is growing exponentially with the size of its memory. For realistic sized circuits the number of states becomes to large for passing through all states.

To handle loops with a large or infinite number of states improved techniques have to be used. A more general criterion for nontermination is: The current state is an element of a subset of states that are all fulfilling $g$ and it can be proven that for every state $x$ within the subset $f(x)$ is also an element of the subset.

How to find such subsets? When designing sequential circuits, control unit and operation unit are often separated. Usually the number of states in the control part is rather small compared with the operation unit. Very often an infinite loop leads to a rather small cycle within the control states. The set of states of the entire circuit where the control state is within this cycle will never be left. Only a small part of the entire circuit must be analyzed to detect the infinite loop. A big part of the state of the operation unit will have no influence on the control flow.

## 7.2 Proving that a Loop Always Terminates

Given a WHILE loop expression with the initial state comprising free variables it has to be proven, that for every instantiation of the free variables the loop will terminate, i.e. there exists a number $n$ such that $\neg(g(f^n(x)))$ is fulfilled. When dealing with a finite number of states, this problem is computable since the number of instantiations is finite and for every instantiation it can be computed whether or not the loop will halt (see previous section).

To handle a large or infinite number of states more powerful techniques have to be used. Often a converging behavior can be detected. If it is possible to find a strictly monoton declining function $h$ mapping the states on natural numbers and if $\neg(g(x))$ holds whenever $h(x)$ is smaller than a constantly given number $m$ then termination can be derived.

## 8 Conclusion

The basic mechanisms for the evaluation of PML terms have been introduced. Since the underlying basic steps have been described in terms of logical transformations within the HOL calculus, correctness of the evaluation mechanisms is guarantied and the techniques described can be used within a formal proof.

Symbolic evaluation has already been used to perform symbolic circuit simulations within HOL. The considered circuits where rather small and although using precomputations, the simulation speed has been comparatively slow — much slower than exhaustive simulations within conventional simulation tools. For larger circuits exhaustive simulation within a conventional simulation tool is not suitable any more. Better results may be achieved using advanced evaluation techniques.

Several distinct evaluation techniques have already been implemented. Still a lot of manual interaction is needed. These algorithms will have to be improved and be integrated within a unique evaluation tool.

## References


[BGGH92] R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experiences with Embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R. Boute, editors, *Conference on Theorem Provers in Circuit Design*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.

[Cami88] J. Camilleri. Executing behavioural definitions in higher order logic. Technical Report 140, University of Cambridge Computer Laboratory, 1988.

[EiSK93] D. Eisenbiegler, K. Schneider, and R. Kumar. A functional approach for formalizing regular hardware structures. In *International Workshop on Higher-Order Theorem Proving and its Applications*, Vancouver, Canada, 1993. (submitted).

[Jone87] S.L.P. Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[KfMA82] A. J. Kfoury, R. N. Moll, and M. A. Arbib. *A Programming Approach to Computability.* Springer, New York, 1982.

[OLLA92] John O'Leary, Mark Linderman, Miriam Leeser, and Mark Aagaard. HML: A hardware description language based on SML. Technical Report EE–CEG–92–7, School of Electrical Engineering, Cornell University, Ithaca, NY 14853, 1992.