The PIDS Algorithm

Version 1.0

Frank Langbein Sonja Schirmer Katharina Organtzoglou

June 17, 1995

1 Introduction

This document describes the `PIDS` algorithm and its application. `PIDS` is an algorithm to simplify expressions which are given as nested lists. The simplification is done by a set of rules which can be applied to the expressions. However the rules just describe how the expression can be rewritten, the real simplification is driven by a merit function. The purpose of `PIDS` is to simplify mathematical expressions. There might be other applications, though. In this article we just give a few examples for mathematical expressions. Note that the whole algorithm and this document is still under development. Huge parts of this document are still a part of the void.

2 Simplification as Graph Search

Assume we have an expression given as nested list and a set of rules. We are searching a somewhat 'simple' expression which could be derived from the original expression by applying the rules. This process could be regarded as a graph search. The vertices of the graph are all possible expressions and the edges are defined by the rules. We start at some expression and try to find another one, which we call 'simple' or at least 'simpler' in some sense. The problem of this search is, that we do not exactly know our goal. The only thing we can do is to follow some or all edges in the graph and evaluate all found states. In the general case this becomes a pretty complex problem, especially

if we consider 'real world' problems. If we want to follow all paths through the graph, even the easiest problem might take days to be computed. If we follow just one path, the result might not at all be simple. So we have to find a way to look only at the most promising paths and forget about the rest. This might still lead to expressions which are not the ideal solution, but we hope to find at least an acceptable one. Another problem is to tell the algorithm, what a simple expression is. Here we need some function which tells us how simple an expression is. This however still requires us to have the expression and we can never be sure to have to ultimate solution. We will have a look at certain graph searching algorithms which at least partly solve this problems. Note that if we want to do real simplification, we always have to search a solution. If we only provide a rewrite mechanism, we can't get more than a simple 'beautifier' of expressions. First we define what we understand by expressions and how our rules are represented.

3 An ADT for Expressions

3.1 Definition

We define an abstract data type E to represent arbitrary expressions for the use in the simplification algorithms based on the graph search. The idea behind these expressions are mathematical ones, which are represented in prefix (lisp) notation. The expression given to the algorithm is a nested list, i.e. a list whose elements could be primitives, lists or empty. A definition in EBNF would be

```
   expr   =  ( "nil" | list )
   list   =  "(" element { " " element } ")"
element   =  ( primitive | list )
```

where `primitive` is the set of all the primitives of our expressions. There might be further conditions where certain primitives could be placed. For mathematical expressions for instance the first element of each list is an operator and the rest of the elements are the arguments of this operator. The operator also defines how many arguments are allowed. In the following we assume that all expressions are given correctly. We don't add a syntax check. Note that `expression` is just the syntax we use to represent the expression, it has not to be stored in the given way.

Now we want to define an ADT E to store `expr` in a suitable way. We do not just store the expression, but also allow to add a set of windows to the expression. A window marks a part of the expression, either a primitive or a complete list. We add this feature to make it easier to detect a simplified expression. As long as there are windows in an expression, the expression can be simplified further. Note that simplify does not mean make an expression simpler, but to apply a rule. Actually a window will tell us where in the expression we still have to apply some rule. If an expression contains more than one window, we have to define a special window, which tells us for which part of the expression we should try to find a rule and apply it. We call this window in the following the *deepest window*. What window this will be, is mainly defined by the implementation of the ADT E. The only thing we require is that there are no further windows

in the sub-expression of the deepest window. So we only apply rules to sub-expressions which have already simplified elements. Once an expression has no more windows, the expression can't be changed any further by the rules.

We have to add a mechanism to add, remove and change the position of the windows. To make the implementation fast and simple, we only allow to change the position of the deepest window. We can add windows at all primitives of an expression or at all lists, which only contain primitives. These are exactly the positions were we will start to apply rules. The deepest window could be moved down or up. If we move the deepest window down, we add windows to all elements of the list the deepest window points to. Moving up means to add a window to the list which contains the element the deepest window points to. If there is no such list, i.e. we are at the top list in the expression, the deepest window is removed from the expression.

Finally we need a way to change the expression. We only apply rules at the deepest window. So we allow to replace the sub-expression at the deepest window by another sub-expression (given in the `expr` syntax). Further we allow to move the deepest window up, down or leave it were it was before the sub-expression was exchanged.

The complete specification of E is given by the following list. C is the set of expressions given in `expr` syntax. Further we denote the set of list of elements from the sets $A_1, \ldots, A_n$ as $(A_1, \ldots, A_n)$.

1. `ECreate`: $C \to E$
   The function value `ECreate(e,c)` is an element e of E which represents an expression $c \in C$. There are no windows in e.

2. `Bottom`: $E \to E$
   The function value `EBottom(e)` is the expression e containing a window for each list in e which has just primitive elements. Existing windows will not be removed. If there is already a window at an operator of the type mentioned above, there will be no new window added. This operation may change the deepest window.

3. `EBottomArgs`: $E \to E$
   The function value `EBottomArgs(e)` is the expression e containing a window for each primitive in e. Existing windows will not be removed. If there is already a window at a primitive, there will be no new window added. This operation may change the deepest window.

4. `EDown` : $E \to E$
   The function value `EDown(e)` is the expression e where the deepest window is removed and a new window for each element in the former deepest window is added to e.

5. `EUp` : $E \to E$
   The function value `EUp(e)` is expression e where the deepest window is removed and a new window is created for the list which contained the element the former deepest window pointed to.

6. `ENoWindows` : $E \to B$
   The function value `ENoWindows(e)` is *true* if e is empty and *false* otherwise.

7. `EExp` : $E \to C$
   The function value `EExp` is the sub-expression in the deepest window (in `expr` syntax).

8. `EReplace` : $E \times (C, I\!R) \to E$
   The function value `EReplace(e,(c,r))` is the expression e where the sub-expression in the deepest window is replaced by c. If r is positive the deepest window is moved up (by `EUp`), if r is negative the deepest window is moved down (by `EDown`). If r is zero, the position of the deepest window is not changed.

3.2 Implementation

We provide a very simple implementation of the E ADT. It's mainly for test purposes and it should be improved later. This implementation is for simplifying mathematical expressions. So each list of `expr` contains an operator as first element and a certain number of additional elements which are the arguments of the operator. Since in mathematical expressions the first element of a list is always an operator, we don't add windows to this element. Further `EDown` does not add windows to elements of the list which are primitives and `EBottom` only adds windows to lists which do only contain primitives, there are no windows added to primitives. `EBottomArgs` adds windows to all primitives in the list, which are no operators. We assume that the argument of `ECreate` is already is some kind of normal form. Later we might add some function that normalizes the given expression.

The expressions will be stored as nested lisp lists. The windows are added as lists, which contain the keyword `window` as first element and the sub-expression as second one, for instance `(window (* a b))`. Note that this forbids an operator with the name `window`. The deepest window is the first window found, which has no windows in the expression it points to, while doing a pre-order traversing of the expression.

```
;; expr.el - expression ADT
;;
;; Copyright (C) 1994-1995 Frank C Langbein
;;                         <langbein@mathematik.uni-stuttgart.de>
;;
;; This program is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published by the Free
;; Software Foundation; either version 2 of the License, or (at your option)
;; any later version.
;;
;; This program is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
;; or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
;; for more details.
;;
;; You should have received a copy of the GNU General Public License along
;; with this program; if not, write to the Free Software Foundation, Inc.,
;; 675 Mass Ave, Cambridge, MA 02139, USA.

(defun ECreate (c)
```

```
  c)

(defun EBottom (expr)
  (if (atom expr)
      expr
    (let ((argl (cdr expr))
          (newargl nil)
          (mark t))
      (while argl
        (if (atom (car argl))
            (setq newargl (append newargl (list (car argl))))
          (setq newargl (append newargl (list (EBottom (car argl))))
                mark nil))
        (setq argl (cdr argl)))
      (setq newargl (cons (car expr) newargl))
      (if mark
          (list 'window newargl)
        newargl))))

(defun EBottomArgs (expr)
  (if (atom expr)
      (list 'window expr)
    (let ((argl (cdr expr))
          (newargl nil))
      (while argl
        (setq newargl (append newargl (list (EBottomArgs (car argl))))
              argl (cdr argl)))
      (setq newargl (cons (car expr) newargl)))))

(defun EDown (expr)
  (car (EDownAtDeepestWindow expr)))

(defun EDownAtDeepestWindow (expr)
  (if (atom expr)
      (list expr nil)
    (let ((argl (cdr expr))
          (deep nil)
          (window nil)
          (newargl nil)
          r)
      (if (eq (car expr) 'window)
          (setq deep t))
      (while argl
        (setq r (EDownAtDeepestWindow (car argl)))
        (if (car (cdr r))
            (setq deep nil
                  window t
                  newargl (append newargl (list (car r)) (cdr argl))
                  argl nil)
          (setq argl (cdr argl)
                newargl (append newargl (list (car r))))))
      (cond (deep
             (list (EDownExecute (car (cdr expr))) t))
            (window
```

```
            (list (cons (car expr) newargl) t))
           (t
            (list (cons (car expr) newargl) nil))))))

(defun EDownExecute (expr)
  (if (atom expr)
      expr
    (let ((argl (cdr expr))
          (newargl nil)
          (r t))
      (while argl
        (if (consp (car argl))
            (setq newargl (append newargl (list
                                           (list 'window (car argl))))
                  r nil)
          (setq newargl (append newargl (list (car argl)))))
        (setq argl (cdr argl)))
      (setq expr (cons (car expr) newargl))
      (if r
          (list 'window expr)
        expr))))

(defun EUp (expr)
  (car (EUpAtDeepestWindow expr)))

(defun EUpAtDeepestWindow (expr)
  (if (atom expr)
      (list expr nil)
    (let ((argl (cdr expr))
          (deep nil)
          (window nil)
          (up nil)
          (newargl nil)
          r)
      (if (eq (car expr) 'window)
          (setq deep t))
      (while argl
        (setq r (EUpAtDeepestWindow (car argl)))
        (cond ((or (eq (car (cdr r)) 'found)
                   (eq (car (cdr r)) 'remove))
               (setq deep nil
                     window (car (cdr r))
                     newargl (append
                              (append newargl (list (car r))) (cdr argl))
                     argl nil))
              ((eq (car (cdr r)) 'up)
               (setq deep nil
                     up t
                     newargl (append
                              (append newargl (list (car r))) (cdr argl))
                     argl nil))
              (t
               (setq argl (cdr argl)
                     newargl (append newargl (list (car r)))))))))
```

```
      (cond (deep
             (list (car (cdr expr)) 'up))
            (up
             (list (list 'window (cons (car expr) newargl)) 'remove))
            (window
             (if (and (eq window 'remove)
                      (eq (car expr) 'window))
                 (list (car newargl) 'found)
               (list (cons (car expr) newargl) 'found)))
            (t
             (list (cons (car expr) newargl) nil))))))

(defun ENoWindows (expr)
  (cond ((atom expr)
         expr)
        ((eq (car expr) 'window)
         nil)
        (t
         (setq expr (cdr expr))
         (let ((r t))
           (while (and expr r)
             (setq r (ENoWindows (car expr))
                   expr (cdr expr)))
           r))))

(defun EExp (expr)
  (if (atom expr)
      nil
    (let ((argl (cdr expr))
          (r nil))
      (while argl
        (setq r (EExp (car argl))
              argl (cdr argl))
        (if r
            (setq argl nil)))
      (cond (r
             r)
            ((eq (car expr) 'window)
             (car (cdr expr)))
            (t
             nil)))))

(defun EReplace (expr c)
  (let ((expr (car (EReplaceAtDeepestWindow expr (car c)))))
    (cond ((equal (car (cdr c)) 0)
           expr)
          ((> (car (cdr c)) 0)
           (EUp expr))
          ((< (car (cdr c)) 0)
           (EDown expr)))))

(defun EReplaceAtDeepestWindow (expr c)
  (if (atom expr)
      (list expr nil)
```

```
(let ((argl (cdr expr))
      (deep nil)
      (window nil)
      (newargl nil)
      r)
  (if (eq (car expr) 'window)
      (setq deep t))
  (while argl
    (setq r (EReplaceAtDeepestWindow (car argl) c))
    (if (car (cdr r))
        (setq deep nil
              window t
              newargl (append (append newargl (list (car r))) (cdr argl))
              argl nil)
      (setq argl (cdr argl)
            newargl (append newargl (list (car r))))))
  (cond (deep
         (list (list 'window c) t))
        (window
         (list (cons (car expr) newargl) t))
        (t
         (list (cons (car expr) newargl) nil))))
```

4 The Rule Set ADT

4.1 Definition

We also need a set of rules which defines the edges of the graph. Therefore we define an ADT which represents a set of such rules. A rule consists of a left and a right side. The left side is the condition which tells us when we can apply the right side. Note that we represent the left side as a function $x : C \rightarrow$BOOLEAN. The set of all these functions is X. The right side of a rule is represented by a function $f : C \rightarrow C$. The set of these functions is F. A rule set is basically a collection of those rules. We also provide a way to find all rules in the rule set which could be applied to a given expression.

The notation $([A])$ represents the set of lists of elements in A. The type of the rule set ADT is denoted by S. Note that the following provides a function that can add more rules with the same left side. This is to enable a faster implementation for the inserting and the search procedures.

1. `RSEmpty` : $. \rightarrow RS$
   The function value of `RSEmtpy` is an empty set of rules.

2. `RSIsEmpty` : $RS \rightarrow B$
   The function value of `RSIsEmpty(r)` is *true* if r is empty otherwise it is *false*.

3. `RSInsert` : $RS \times X \times ([F]) \rightarrow RS$
   The function value of `RSInsert(r,x,([f])` is the set of rules r with additional rules, which have the left side x and the right side represented by the list $([f])$.

4. `RSSearch` : $RS \times C \rightarrow ([F])$
   The function value of `RSSearch(r,c)` is a list of maps $f : C \rightarrow C$ which could be applied to expression c according to the rules in r.

To make this more efficient, we should provide some mechanism to express a hierarchy of rules. The left sides of the rules may be logical expressions combined by 'and'. If we split up these expressions and create a decision tree, `RSSearch` could be implemented faster.

4.2 Implementation

Again a very simple implementation. The set of rules is represented as list of lists, which contain x as first element and $([f])$ as second one.

```
;; ruleset.el - ruleset ADT
;;
;; Copyright (C) 1994-1995 Frank C Langbein
;;                         <langbein@mathematik.uni-stuttgart.de>
;;
;; This program is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published by the Free
;; Software Foundation; either version 2 of the License, or (at your option)
;; any later version.
;;
;; This program is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
;; or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
;; for more details.
;;
;; You should have received a copy of the GNU General Public License along
;; with this program; if not, write to the Free Software Foundation, Inc.,
;; 675 Mass Ave, Cambridge, MA 02139, USA.

(defun RSEmpty ()
  (list nil))

(defun RSIsEmpty (rs)
  (equal (car rs) nil))

(defun RSInsert (rs x fl)
  (append (list (list x fl)) rs))

(defun RSSearch (rs expr)
  (let ((res nil))
    (while (not (RSIsEmpty rs))
      (let ((r (car rs)))
        (if (funcall (car r) expr)
            (setq res (append res (car (cdr r)))))
        (setq rs (cdr rs))))
    res))
```

5 The Simplification Algorithm

We have reached the point where we should discuss the graph searching algorithms. First will provide a few standard algorithms for graph searching which finally lead to the `PIDS` algorithm. For details about these algorithm refer to *Artificial Intelligence, 2nd Edition* by Elaine Rich and Kevin Knight, McGraw Hill, 1991, part I, chapter 2 and 3. All algorithms use a merit function, which evaluates a vertex of the graph and a set of rules. Both will be described later. When we are talking about a state in the following, we mean a vertex of our graph.

5.1 Simple Search

We assume you know about breadth and depth first searching in graphs. Applying this to our problem would in both cases create all possible simplifications before we can stop. In principle both methods have to fail, since we don't know our goal state. We might consider the following simple algorithm

1. Create a list VISITED with the initial state as member.
2. Create a list WORK with the initial state as member.
3. Create an empty list SIMP.
4. Loop until WORK is empty
   (a) Remove an element from WORK and make it the current state.
   (b) Generate all expressions we can derive from our set of rules from the current state and store those not in VISITED in SUCC.
   (c) Store all members of SUCC in VISITED
   (d) Store all elements of SUCC in SIMP which do not contain windows and store them in WORK otherwise.
5. All elements in SIMP are simplified. Choose one or more elements from SIMP to be the goal state using the merit function.

Depending on how we choose/store elements of WORK, we get breadth or depth searching. We can prevent the loop from traversing the whole graph. Therefore we also have to check if there is already an element in SIMP which we regard as simple enough using the merit function.

5.2 Agenda Driven Search

The mathematics discovery program AM by Lenat used a different kind of search, since the goal state was not known. The program should just look around and try to find something interesting. This lead to the agenda driven search method as provided in Elaine Rich's book

1. Do until a goal state is reached or the agenda is empty:

(a) Choose the most promising task from the agenda. Notice that this task can be represented in any desired form. It can be thought of as an explicit statement of what to do next or simply as an indication of the next node to be expanded.

(b) Execute the tasks by devoting to it the number of resources determined by its importance. The important resources to consider are time and space. Executing the tasks will probably generate additional tasks (successor nodes). For each of them, do the following

   i. See if it is already on the agenda. If so, then see if this same reason for doing it is already on its list of justifications. If so, ignore this current evidence. if this justification was not already present, add it to the list. If the task was not on the agenda, insert it.

   ii. Compute the new tasks rating, combining the evidence from all its justifications. Not all justifications need have equal weight. It is often useful to associate with each justification a measure of how strong a reason it is. The measures are then combined at this step to produce an overall rating for the task.

We will now try to use this general technique for our purposes. This first leads to

5.3 The Interest Driven Search - IDS

We have a merit function, a set of rules and an expression to be simplified. In order not to search the whole graph, we try to evaluate the expressions and always take the most promising one for the next expansion by the given rules.

1. Store the initial expression and its evaluation by the merit function in the list WORK and add a window structure to this expression by `EBottom`.

2. Create a list VISITED containing the initial expression.

3. Create an empty list SIMP.

4. Loop while WORK is not empty and there is no expression in SIMP which is already simple enough (which is indicated by a small value of our merit function).

   (a) Remove the best expression from WORK, i.e the one with the smallest merit function value and make it the current expression.

   (b) Generate a list of all rules which could be applied to the deepest window of the current expression. For each rule do the following

      i. Apply the rule to the function (this includes moving the deepest window and rewriting the sub-expression in the deepest window).

      ii. If the result is already in VISITED, continue with the next rule, otherwise add the result to VISITED.

      iii. If there is no window left in the result, check if it is already in SIMP. If so, drop the result, otherwise add it to SIMP

      iv. If there is at least one window left, check if it is already in WORK. If so, drop it, otherwise add it to WORK.

6 The Pre-estimated Interest Driven Search - PIDS

To make the search faster, we do not just evaluate the expressions, but also the rules. Therefore we have to add values to the rules that tell us how promising it is that the rule actually creates a simpler expression. Further we split the set of rules. There is one set which is used for searching, denoted by RULES, and another one which is always applied when its left side is satisfied, denoted by BRULES.

1. Create an empty list WORK and SIMP.
2. Apply BRULES to the initial expression.
3. Generate all right sides of the rules in RULES which could be applied to the deepest window in the expression. Create for each function an element of WORK, containing the expression, the merit value of the function modified by the merit value of the rule and the function.
4. Create an empty list SIMP.
5. Loop until WORK is empty or there is a 'good-enough' expression in SIMP.
   (a) Take the most promising element from WORK, apply the function to the expression, apply the BRULES and evaluate the result with the merit function.
   (b) If there is no deepest window left in the result, store it together with its merit value in SIMP, if it is not already in SIMP.
   (c) If there are windows left, generate a set of functions which could be applied to the deepest window of the result. Create for each function an element of WORK, containing the expression, the merit value of the function modified by the merit value of the rule and the function, if it is not already in WORK or was visited earlier.

To implement this algorithm, we have to change the rule sets ADT. Further we define a new ADT for handling the WORK list.

6.1 A new ADT for rule sets

We do no longer just evaluate expressions but also rules. This values must be given, though and are not computed by a function. The following redefinition of the rule set ADT *RS* will allow us to store this values with the rules. We evaluate rules with non-negative real numbers. A rule with evaluation 0 has the lowest possible 'use'. Higher values represent higher use. We will use *RN* to denote the new ADT.

1. `RNEmpty` : $\rightarrow RN$
   The function value of `RNEmtpy` is an empty set of rules.
2. `RNIsEmpty` : $RN \rightarrow B$
   The function value of `RNIsEmpty(r)` is *true* if r is not empty otherwise it is *false*.

3. `RNInsert` : $RN \times X \times ([(F, I\!R_0^+)]) \to RN$
   The function value of `RNInsert(r,x,([(f,v)])` is the set of rules r with additional rules represented by the list $([(f, v)])$, which could be applied to an expression e if $x(e)$ is true. The members of $([(f, r)])$ are maps $f : C \to C$ which represent the right side of a rule and associated non-negative real numbers which represent the 'value' of the rule. $x : C \to B$ is a map which represents the left side of the rules.

4. `RNSearch` : $RN \times C \to ([(F, I\!R_0^+)])$
   The function value of `RNSearch(r,c)` is a list of maps $f : C \to C$ and non-negative real numbers v which evaluate the corresponding function which could be applied to expression c according to the set of rules r.

We give an implementation, which is based on the rule set ADT RS from above.

```
;; rulexset.el - new ruleset ADT
;;
;; Copyright (C) 1994-1995 Frank C Langbein
;;                         <langbein@mathematik.uni-stuttgart.de>
;;
;; This program is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published by the Free
;; Software Foundation; either version 2 of the License, or (at your option)
;; any later version.
;;
;; This program is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
;; or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
;; for more details.
;;
;; You should have received a copy of the GNU General Public License along
;; with this program; if not, write to the Free Software Foundation, Inc.,
;; 675 Mass Ave, Cambridge, MA 02139, USA.

(defun RNEmpty ()
  (list nil))

(defun RNIsEmpty (rs)
  (equal (car rs) nil))

(defun RNInsert (rs x fl)
  (append (list (list x fl)) rs))

(defun RNSearch (rs expr)
  (let ((res nil))
    (while (not (RNIsEmpty rs))
      (let ((r (car rs)))
        (if (funcall (car r) expr)
            (setq res (append res (car (cdr r)))))
        (setq rs (cdr rs))))
    res))
```

6.2 An ADT for simplification lists

`PIDS` does no longer just apply a rule. It will first store all possible rules that could be applied to an expression. Their merit value of the expression and the evaluation of the rule will decide which expression is considered to be the best choice for further examination. How this is done is defined by this ADT, which we will denote by S. The special feature of this list, is that an element can only be inserted once. Once it has been inserted, it cannot be inserted again, even if it was removed meanwhile. This prevents us from inserting elements, we already evaluated.

1. `SEmpty` : $. \rightarrow S$
   The function value `SEmpty` is an empty simplification list, i.e. a list which does not contain any entries and allows to insert any entry.

2. `SIsEmpty` : $S \rightarrow B$
   The function value `SIsEmpty(l)` is *true* if l is empty and *false* otherwise. l is considered to be empty if there are no values left in the list. This does not mean that the list still accepts every member as entry in the list.

3. `SAddList`:$S \times E \times I\!R_0^+ \times ([(F, I\!R_0^+)]) \rightarrow S$
   The function value `SAddList(s,e,n,([(f,v)]))` is the rule list s containing also an entry for each (f, v). This entry consists of the expression e, the simplification function f and an evaluation computed from the functions merit value n and the rules merit v. If this entry was already inserted in the list, it is not inserted again - even if it was already removed again by `SGetBest`. the list.

4. `SGetBest` : $S \rightarrow (S, E, F)$
   The function value `SGetBest(s)` is a list containing the most promising expression e, the corresponding f and the simplification list s without this simplification.

A possible implementation is given by the following code. We store S as a list of two lists. The first one represents the members of S and the 2nd one contains all entries once inserted in the list.

```
;; slist.el - simplification lists ADT
;;
;; Copyright (C) 1994-1995 Frank C Langbein
;;                         <langbein@mathematik.uni-stuttgart.de>
;;
;; This program is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published by the Free
;; Software Foundation; either version 2 of the License, or (at your option)
;; any later version.
;;
;; This program is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
;; or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
;; for more details.
;;
;; You should have received a copy of the GNU General Public License along
```

```
;; with this program; if not, write to the Free Software Foundation, Inc.,
;; 675 Mass Ave, Cambridge, MA 02139, USA.

(defun SEmpty ()
  (list nil nil))

(defun SIsEmpty (s)
  (equal (car s) nil))

(defun SAddList (s expr merit rulex)
  (let ((w (car s))
        (a (car (cdr s))))
    (while rulex
      (let* ((r (car rulex))
             (eval (- merit (car (cdr r))))
             (el (list eval expr (car r))))
        (if (< eval 0)
            (setq eval 0))
        (if (not (ShelpIsIn a el))
            (setq w (ShelpInsert w el)
                  a (ShelpInsert a el)))
        (setq rulex (cdr rulex))))
    (list w a)))

(defun ShelpIsIn (l el)
  (let ((r nil))
    (while l
      (if (equal (car l) el)
          (setq r t
                l nil)
        (setq l (cdr l))))
    r))

(defun ShelpInsert (s el)
  (let ((hs s))
    (setq s nil)
    (if (SIsEmpty hs)
        (setq s (list el))
      (while hs
        (cond ((< (car (car hs)) (car el))
               (setq s (append s (list (car hs)))
                     hs (cdr hs))
               (if (equal hs nil)
                   (setq s (append s (list el)))))
              ((and (equal (car (car hs)) (car el))
                    (equal (car (cdr (car hs))) (car (cdr el))))
               (setq s (append s hs)
                     hs nil))
              ((equal (car (car hs)) (car el))
               (setq s (append s (list (car hs)))
                     hs (cdr hs))
               (if (equal hs nil)
                   (setq s (append s (list el)))))
              (t
```

```
              (setq s (append s (list el) hs)
                    hs nil))))))
  s)

(defun SGetBest (s)
  (let ((b (car (car s))))
    (setq s (list (cdr (car s)) (car (cdr s))))
    (list s (car (cdr b)) (car (cdr (cdr b))))))
```

6.3 An implementation of PIDS

Using all the above ADTs we get the following implementation. Note that by the choice of the function `not-good-enough` and the global variable `return-all-simp`, we generate all (by the choice of `rules` and `brules`) possible simplifications of the given expression (unless we run out of resources).

```
;; pids.el - the PIDS algorithm
;;
;; Copyright (C) 1994-1995 Frank C Langbein
;;                         <langbein@mathematik.uni-stuttgart.de>
;;
;; This program is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published by the Free
;; Software Foundation; either version 2 of the License, or (at your option)
;; any later version.
;;
;; This program is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
;; or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
;; for more details.
;;
;; You should have received a copy of the GNU General Public License along
;; with this program; if not, write to the Free Software Foundation, Inc.,
;; 675 Mass Ave, Cambridge, MA 02139, USA.

(defun PIDS (cexpr rules brules meritf)
  (setq PIDS-counter 0)
  (let* ((expr (apply-basic-rules brules (EBottom (ECreate cexpr))))
         (work (SAddList (SEmpty) expr (funcall meritf expr)
                         (RNSearch rules (EExp expr))))
         fr
         merit
         (simp nil))
    (if (not (ENoWindows expr))
        (while (and (not (SIsEmpty work)) (not-good-enough simp))
          (let* ((bestsimp  (SGetBest work))
                 (oldexpr   (car (cdr bestsimp)))
                 (simplexpr
                  (apply-basic-rules brules (EReplace oldexpr
                                                      (funcall (car (cdr (cdr bestsimp)))
                                                               (EExp oldexpr))))))
            (setq work (car bestsimp))
            (while (and (not (ENoWindows simplexpr))
                        (not (setq fr (RNSearch rules (EExp simplexpr)))))
```

```
            (setq simplexpr (EUp simplexpr)))
          (setq merit (funcall meritf simplexpr))
          (if (ENoWindows simplexpr)
              (setq simp (insert-in-sorted-list simplexpr merit simp))
            (setq work (SAddList work simplexpr merit fr))))
        )
      (setq simp (insert-in-sorted-list expr (funcall meritf expr) simp)))
    (return-one-or-all simp)))

(defun not-good-enough (simp)
  (setq PIDS-counter (+ PIDS-counter 1))
  (cond ((and (not return-all-smp) simp)
         t)
        ((and (> PIDS-counter 400) simp)
         nil)
        ((and simp (< (car (car simp)) 1))
         nil)
        (t
         t)))

(defun apply-basic-rules (rules expr)
  (let (r)
    (while (and (not (ENoWindows expr))
                (setq r (RNSearch rules (EExp expr))))
      (setq expr (EReplace expr (funcall (car (car r)) (EExp expr)))
            r    (RNSearch rules (EExp expr)))))
  expr)

(defun is-in-sorted-list (el merit l)
  (let ((r nil))
    (while l
      (if (equal (car (cdr (car l))) el)
          (setq r t
                l nil)
        (setq l (cdr l))))
    r))

(defun insert-in-sorted-list (el merit l)
  (let ((hl l))
    (setq l nil)
    (if hl
        (while hl
          (if (< (car (car hl)) merit)
              (progn
                (setq l (append l (list (car hl)))
                      hl (cdr hl))
                (if (equal hl nil)
                    (setq l (append l (list (list merit el))))))
            (if (equal (car (car hl)) merit)
                (if (equal (car (cdr (car hl))) el)
                    (setq l (append l hl)
                          hl nil)
                  (progn
                    (setq l (append l (list (car hl)))
```

```
                        hl (cdr hl))
                  (if (equal hl nil)
                      (setq l (append l (list (list merit el)))))))
             (setq l (append l (list (list merit el)) hl)
                   hl nil))))
      (setq l (list (list merit el)))))
  l)

(setq return-all-simp t)
(defun return-one-or-all (l)
  (if return-all-simp
      (let (r)
        (while l
          (setq r (append r (list (car (cdr (car l)))))
                l (cdr l)))
        r)
    (car (cdr (car l)))))
```

7 The Merit Functions

In the following we only consider the simplification of mathematical expressions. The choice of the merit function actually defines what we call simple. There are quite a lot of possibilities to choose such a function. We might for instance count arguments, or operators or consider the relation between arguments and operators. Further we can add weights to operators depending on which operators we prefer to have in our simple expressions. And there are all kinds of combination of those functions. It is obvious that there is not one ideal merit functions, but that there are more depending on what we want to achieve. To make things (very!) easy, we only allow the addition and multiplication. A few simple merit functions are

```
;; meritf.el - merit function for example
;;
;; Copyright (C) 1994-1995 Frank C Langbein
;;                         <langbein@mathematik.uni-stuttgart.de>
;;
;; This program is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published by the Free
;; Software Foundation; either version 2 of the License, or (at your option)
;; any later version.
;;
;; This program is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
;; or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
;; for more details.
;;
;; You should have received a copy of the GNU General Public License along
;; with this program; if not, write to the Free Software Foundation, Inc.,
;; 675 Mass Ave, Cambridge, MA 02139, USA.

(defun meritf-const (expr)
  10)
```

```
(defun meritf-count-arg (expr)
  (if (consp expr)
      (let ((r 0) (l (mapcar 'meritf-count-arg (cdr expr))))
        (while l
          (setq r (+ r (car l))
                l (cdr l)))
        r)
    1))

(defun meritf-count-op (expr)
  (if (consp expr)
      (let ((r 1) (l (mapcar 'meritf-count-op (cdr expr))))
        (while l
          (setq r (+ r (car l))
                l (cdr l)))
        r)
    0))

(defun meritf-count-plus-4mul (expr)
  (if (consp expr)
      (let ((r 1) (l (mapcar 'meritf-count-plus-4mul (cdr expr))))
        (if (equal (car expr) '*)
            (setq r 4))
        (while l
          (setq r (+ r (car l))
                l (cdr l)))
        r)
    0))

(defun meritf-count-4plus-mul (expr)
  (if (consp expr)
      (let ((r 4) (l (mapcar 'meritf-count-4plus-mul (cdr expr))))
        (if (equal (car expr) '*)
            (setq r 1))
        (while l
          (setq r (+ r (car l))
                l (cdr l)))
        r)
    0))
```

8 The Rules

We continue considering only addition and multiplication. We have to define a set of rules for the `PIDS` algorithm. Note that these rules consist of a left side which gives the condition when to apply the rule and a right side which consists of a rewrite function and a direction to move the deepest window. Especially this direction is important for our rules. If we never move it up, we never get a result, if we always move it up, we might overlook some important simplifications. Also

providing an empty rule, i.e. one that just moves the deepest window up, seems to be important. Another representation, especially a hierarchical one, like a decision tree, might improve the whole algorithm considerably. Note that a good evaluation of the rules also depends on the used merit function. Introducing a function for the evaluation of the rules might be a good idea.

```
;; rules.el - rules for example
;;
;; Copyright (C) 1994-1995 Frank C Langbein
;;                         <langbein@mathematik.uni-stuttgart.de>
;;
;; This program is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published by the Free
;; Software Foundation; either version 2 of the License, or (at your option)
;; any later version.
;;
;; This program is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
;; or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
;; for more details.
;;
;; You should have received a copy of the GNU General Public License along
;; with this program; if not, write to the Free Software Foundation, Inc.,
;; 675 Mass Ave, Cambridge, MA 02139, USA.

(setq Rules (RNEmpty))

; empty-rule
(setq Rules (RNInsert Rules 'test-empty
                      '((simp-empty 0))))
(defun test-empty (expr)
  (consp expr))
(defun simp-empty (expr)
  (list expr 1))

; ((a + b) + c) = (a + (b + c))
(setq Rules (RNInsert Rules 'test-plus-ass1
                      '((simp-plus-ass1 2))))
(defun test-plus-ass1 (expr)
  (and (equal (car expr) '+)
       (consp (car (cdr expr)))
       (equal (car (car (cdr expr))) '+)))
(defun simp-plus-ass1 (expr)
  (list (list '+ (nth 1 (nth 1 expr))
        (list '+ (nth 2 (nth 1 expr)) (nth 2 expr)))  -1))

; (a + (b + c)) = ((a + b) + c)
(setq Rules (RNInsert Rules 'test-plus-ass2
                      '((simp-plus-ass2 2))))
(defun test-plus-ass2 (expr)
  (and (equal (car expr) '+)
       (consp (nth 2 expr))
       (equal (car (nth 2 expr)) '+)))
(defun simp-plus-ass2 (expr)
```

```
  (list (list '+ (list '+ (nth 1 expr) (nth 1 (nth 2 expr)))
              (nth 2 (nth 2 expr))) -1))

; a + b = b + a
(setq Rules (RNInsert Rules 'test-plus-com
                      '((simp-plus-com 1))))
(defun test-plus-com (expr)
  (equal (car expr) '+))
(defun simp-plus-com (expr)
  (list (list '+ (nth 2 expr) (nth 1 expr)) 1))


; a * (b *c) = (a * b) * c
(setq Rules (RNInsert Rules 'test-mul-ass1
                      '((simp-mul-ass1 2))))
(defun test-mul-ass1 (expr)
  (and (equal (car expr) '*)
       (consp (nth 2 expr))
       (equal (car (nth 2 expr)) '*)))
(defun simp-mul-ass1 (expr)
  (list (list '* (list '* (nth 1 expr) (nth 1 (nth 2 expr)))
              (nth 2 (nth 2 expr))) -1))

; (a * b) * c = a * (b * c)
(setq Rules (RNInsert Rules 'test-mul-ass2
                      '((simp-mul-ass2 2))))
(defun test-mul-ass2 (expr)
  (and (equal (car expr) '*)
       (consp (nth 1 expr))
       (equal (car (nth 1 expr)) '*)))
(defun simp-mul-ass2 (expr)
  (list (list '* (nth 1 (nth 1 expr))
              (list '* (nth 2 (nth 1 expr)) (nth 2 expr))) -1))

; a * b = b * a
(setq Rules (RNInsert Rules 'test-mul-com
                      '((simp-mul-com 1))))
(defun test-mul-com (expr)
  (and (equal (car expr) '*)))
(defun simp-mul-com (expr)
  (list (list '* (nth 2 expr) (nth 1 expr)) 0))

; a * (b + c) = a * b + a * c
(setq Rules (RNInsert Rules 'test-mulplus-dist1
                      '((simp-mulplus-dist1 4))))
(defun test-mulplus-dist1 (expr)
  (and (equal (car expr) '*)
       (consp (nth 2 expr))
       (equal (car (nth 2 expr)) '+)))
(defun simp-mulplus-dist1 (expr)
  (list (list '+ (list '* (nth 1 expr) (nth 1 (nth 2 expr)))
              (list '* (nth 1 expr) (nth 2 (nth 2 expr)))) -1))

; a * b + a * c = a * (b + c)
```

```
(setq Rules (RNInsert Rules 'test-mulplus-dist2
                      '((simp-mulplus-dist1 6))))
(defun test-mulplus-dist2 (expr)
  (and (equal (car expr) '+)
       (consp (nth 1 expr))
       (equal (car (nth 1 expr)) '*)
       (consp (nth 2 expr))
       (equal (car (nth 2 expr)) '*)
       (equal (nth 1 (nth 1 expr)) (nth 1 (nth 2 expr)))))
(defun simp-mulplus-dist2 (expr)
  (list (list '* (nth 1 (nth 1 expr))
              (list '+ (nth 2 (nth 1 expr)) (nth 2 (nth 2 expr)))) -1))



(setq BRules (RNEmpty))

; 0 + a = a
(setq BRules (RNInsert BRules 'test-plus-zero
                       '((simp-plus-zero 5))))
(defun test-plus-zero (expr)
  (and (equal (car expr) '+)
       (equal (nth 1 expr) 0)))
(defun simp-plus-zero (expr)
  (list (nth 2 expr) 1))

; n + m = (n+m)
(setq BRules (RNInsert BRules 'test-plus-add
                      '((simp-plus-add 4))))
(defun test-plus-add (expr)
  (and (equal (car expr) '+)
       (numberp (nth 1 expr))
       (numberp (nth 2 expr))))
(defun simp-plus-add (expr)
  (list (+ (nth 1 expr) (nth 2 expr)) 1))

; a + a = 2 * a
(setq BRules (RNInsert BRules 'test-plus-add2
                      '((simp-plus-add2 3))))
(defun test-plus-add2 (expr)
  (and (equal (car expr) '+)
       (equal (nth 1 expr) (nth 2 expr))))
(defun simp-plus-add2 (expr)
  (list (list '* 2 (nth 1 expr)) 0))


; 0 * a = a
(setq BRules (RNInsert BRules 'test-mul-zero
                       '((simp-mul-zero 5))))
(defun test-mul-zero (expr)
  (and (equal (car expr) '*)
       (equal (nth 1 expr) 0)))
(defun simp-mul-zero (expr)
  (list 0 1))
```

```
; n * m = (n*m)
(setq BRules (RNInsert BRules 'test-mul-mul
                       '((simp-mul-mul 3))))
(defun test-mul-mul (expr)
  (and (equal (car expr) '*)
       (numberp (nth 1 expr))
       (numberp (nth 2 expr))))
(defun simp-mul-mul (expr)
  (list (* (nth 1 expr) (nth 2 expr)) 1))
```

8.1 Testing the rules

The follow small program contains a few examples for applying the above rules.

```
;; main.el - main program for example
;;
;; Copyright (C) 1994-1995 Frank C Langbein
;;                         <langbein@mathematik.uni-stuttgart.de>
;;
;; This program is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published by the Free
;; Software Foundation; either version 2 of the License, or (at your option)
;; any later version.
;;
;; This program is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
;; or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
;; for more details.
;;
;; You should have received a copy of the GNU General Public License along
;; with this program; if not, write to the Free Software Foundation, Inc.,
;; 675 Mass Ave, Cambridge, MA 02139, USA.

(load "expr.el")
(load "rulexset.el")
(load "slist.el")
(load "pids.el")
(load "meritf.el")
(load "rules.el")

; (setq return-all-simp nil)
; (setq return-all-simp t)

(PIDS '(+ 2 3) Rules BRules 'meritf-count-4plus-mul)
(PIDS '(+ 1 (+ 2 3)) Rules BRules 'meritf-count-4plus-mul)
(PIDS '(+ (+ 0 a) a) Rules BRules 'meritf-count-4plus-mul)
(PIDS '(+ (* 0 a) a) Rules BRules 'meritf-count-4plus-mul)
(PIDS '(+ a a) Rules BRules 'meritf-count-4plus-mul)
(PIDS '(+ (+ a a) (+ a a)) Rules BRules 'meritf-count-4plus-mul)
(PIDS '(+ (+ b a) a) Rules BRules 'meritf-count-4plus-mul)
(PIDS '(+ a (+ a b)) Rules BRules 'meritf-count-4plus-mul)
(PIDS '(+ (+ a (+ a b)) (+ a a)) Rules BRules 'meritf-count-4plus-mul)
(PIDS '(* 5 (* a 4)) Rules BRules 'meritf-count-4plus-mul)
```

```
(PIDS '(* 2 (+ a a)) Rules BRules 'meritf-count-4plus-mul)

; ....
```

If we compute all the simplifications, as it is the default of the above implementation of `PIDS`, the above examples take quite some time. If we set `return-all-simp` to `nil`, speed increases and we still find something that is really simple. However to provide an efficient way, we should add some kind of sorting for plus, allow more than two arguments. Especially the sorting of the arguments might increase speed since we do not check for all permutations anymore. Some rules could be eliminated, too. The sorting could be done explicitly by `ECreate` or by the BRULES set. A combination of both might be best. There are lots of possibilities for the rules, the merit function and changing the algorithm, which might make the simplification a lot more efficient. However, time does not allow to investigate this further.