PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

ASSIP-T. A THEOREM PROVING MACHINE

Werner Dilger and Hans-Albert Schneider

Computer Science Department
University of Kaiserslautern, Postfach 3049
D-6750 Kaiserslautern, FR Germany

ABSTRACT

An associative processor for theorem proving in first order logic is described. It is designed on the basis of the deduction plan method, introduced by Cox and Pietrzykowski. The main features of this method are the separation of unification from deduction and the incorporation of a method for intelligent backtracking. This kind of backtracking is based on a special unification procedure. An improved version of this unification procedure is given in this paper, which outputs a unification graph with constraints. In the case of a unification conflict, sufficient information for a directed backtracking step can be gained from the unification graph. According to the deduction plan method, the ASSIP-T processor has two associative memories, one for the deduction plan and the other for the unification graph. It can perform deduction and unification in parallel. The data structures for the representation of the deduction plan and the unification graph are given here explicitely, whereas the algorithms operating on them are only sketched. A proposal for the realization of the ASSIP-T memories is presented.

1 INTRODUCTION

The progress of microelectronics allows the realizations of more and more powerful processors for special purposes. One such type of processors is the associative processor. Its associative memory allows content oriented parallel access to the data stored in it. This makes the associative processors well suited for pattern handling processes. In artificial intelligence e.g., most processes are pattern directed deductions. One of it is theorem proving. In this paper a model of an associative processor is described which is able to prove theorems of first order logic. It is designed on the basis of the deduction plan method, i.e. it incorporates a method for intelligent backtracking.

After some basic definitions in the second section, the deduction plan method is described. The special unification procedure used within this method follows. The output of this procedure is a unification graph with constraints. In the case of a unification conflict, the unification graph gives sufficient information for a directed backtracking step. This is described in section 5. Then the structure of the ASSIP-T processor which is aimed to perform the deduction plan method is described. Section 7 contains the data structures which are to be mapped on the ASSIP-T memory and the algorithms which run on the processor. Finally, the representation of the data structures in the ASSIP-T memory is sketched.

2 BASIC DEFINITIONS

A *labelled graph* is a triple $G = (V(G),I(G),E(G))$ where $V(G)$, $I(G)$, and $E(G)$ are the sets of nodes, labels, and edges respectively. A *path* in G is a sequence $w = v_1,e_1,v_2,e_2,\ldots,e_n,v_{n+1}$ $(n\geq 0)$ with $v_i \in V(G)$ and $e_j \in E(G)$. If $v_1 = v_{n+1}$, the path is called *closed*. A closed path which contains each inner node at most once is called a *cycle*.

Assume there are given disjoint alphabets of *variables, function symbols* and *predicate symbols*. Each function and predicate symbol has an arity. A *constant* is a 0-ary function symbol. An *expression* is a variable or a term. A *term* is a constant or a string of the form $f(q_1,\ldots,q_n)$, where f is an n-ary function symbol $(n\geq 1)$ and $q_1,\ldots,q_n$ are expressions. An *atom* is a string of the form $P(q_1,\ldots,q_n)$, where P is an n-ary predicate symbol $(n\geq 0)$ and $q_1,\ldots,q_n$ are expressions. If a is an atom, then A and -A are

literals. A *clause* is a finite set of literals. The empty clause is denoted by $\square$.

A *constraint* is a set consisting of two expressions. A set of constraints is called a *constraint set*. If p and q are expressions (terms), then p is a *subexpression (subterm)* of q if p=q or $q = f(q_1,\ldots,q_n)$ and p is a subexpression (subterm) of one of the q_i. An expression (term) p is a subexpression (subterm) of a constraint set C, if there is a constraint $\{q_1,q_2\}$ in C such that p is a subexpression (subterm) of q_1 or of q_2. The set of all subexpressions of C is denoted by SEXPR(C).

A *substitution* is a finite set of pairs (v,q), denoted by v/q, where v is a variable and q an expression and $v \neq q$. *Application* of a substitution $\sigma = \{v_1/q_1,\ldots,v_n/q_n\}$ on an expression or a literal p is the replacement of each occurrence of v_i in p by q_i, for all $i = 1,\ldots,n$. σ is called a *renaming* if $q_1,\ldots,q_n$ are pairwise different variables and $\{v_1,\ldots,v_n\} \cap \{q_1,\ldots,q_n\} = \emptyset$. A clause cl_1 is called a *variant* of a clause cl_2 if cl_1 and cl_2 have no variables in common and there is a renaming σ such that $cl_1 = \sigma cl_2$. If $E = \{p_1,\ldots,p_m\}$ is a set of expressions then a substitution σ is called a *unifier* of E, if $\sigma p_1 = \ldots = \sigma p_m$. E is then called *unifiable*. σ is called a *most general unifier* of E if for each unifier τ there is a unifier ρ such that $\tau = \sigma\cdot\rho$.

Let $C = \{c_1,\ldots,c_n\}$ be a constraint set. The set BE(C) of *Boolean expressions* over C is defined by

1. $0,1,c_1,\ldots,c_n \in BE(C)$.
2. If $B_1,B_2 \in BE(C)$, then $(B_1 \vee B_2)$, $(B_1 \wedge B_2) \in BE(C)$.
3. BE(C) contains no other elements.

3 DEDUCTION PLANS

The deduction plan method is a resolution based method, i.e. a refutation method. It starts with a set of clauses and tries to construct a "closed" and "correct" deduction plan. If it succeeds, the clause set is proved to be unsatisfiable. The central idea of the method is to separate deduction from unification. This allows the application of a special unification algorithm which, in the case of a unification conflict, not simply stops with failure, rather it yields information about the causes of unification conflicts, namely certain deduction steps, which can then be reset. In section 5 this way of processing is called "intelligent backtracking".

The nodes of the deduction plan are the input clauses and eventually variants of them. Two clauses can be connected by an edge if they contain literals with the same precidate symbol but different signs (negated or not negated). Therefore a (labelled) edge between two clauses cl_1 and cl_2 is a triple $(cl_1,(t,u,v),cl_2)$, where u and v are literals in cl_1 and cl_2 respectively, satisfying the condition on their predicate symbols and negation signs. t is the type of the edge. There are two types of edges: SUB and RED. All edges are of type SUB except those refering backward to a clause which is already in use. If each literal in each clause occurs in an edge, the deduction plan is closed. If the set of pairs of terms arising from the pairing of literals by edges is unifiable, the deduction plan is correct. Cf. for this section (Cox and Pietrzykowski 1979) and (Cox and Pietrzykowski 1981).

Definition

Let S be a set of input clauses and $L = \bigcup_{cl \in S} cl$. A *deduction graph* on S is a graph $G = (V(G),I(G),E(G))$ which has the variants of S as node set V(G), $I(G) \subseteq \{SUB,RED\}\times L\times L$ with: if $e = (cl_1,b,cl_2) \in E(G)$ then $b = (t,u,v)$, $u \in cl_1$, $v \in cl_2$. t is called the *type* of the edge e, u the *starting literal* and v the *target literal*. A literal u of a clause cl is called *key literal* iff there is an incoming edge with type SUB and target literal u. Each literal u of a clause cl is called a *subproblem* iff it is not a key literal. A subproblem $u \in cl$ is *open* iff there is no outcoming edge with starting literal u. A subproblem u is called *closed* iff it is not open. os(G) is the set of open subproblems of a deduction graph G. G is called *closed*, iff $os(G) = \emptyset$.

A node cl_1 is called *predecessor* of a node cl_2 iff there is a path from cl_1 to cl_2 which contains only edges of type SUB (*SUB-path*). If u is the starting literal of the first edge of a SUB-path from cl_1 to cl_2, then u is called *preceding literal* of cl_2 and cl_2 is called *successor* of cl_1.

We omit the definition of the deduction plan here. It is a deduction graph which is constructed by a number of deduction steps, i.e. edge drawing steps, starting from a basic plan which consists of one node only.

A subgraph H of a deduction plan G is called *subplan* iff for each node $cl \in V(H)$ (i) and (ii):

(i) all predecessors of cl in G are in V(H),

(ii) if e is an edge in E(G) of type SUB with end node cl, then e is in E(H).

Example

S = {{P(x), Q(y), R(f(x,y))},
{-P(g(x)), V(x)},
{-P(g(x)), -V(x)},
{-Q(x), S(x), -T(x)},
{-S(a)},
{-S(b)},
{T(b)},
{-R(x)}}

is a set of eight input clauses. Figure 1 shows a closed deduction plan for S. The edges are drawn in such a way that they begin beyond the starting literal and point to the target literal. Therefore they are only labelled by their type and, beyond it, by the numbers of the steps in the plan construction within which the edges were drawn. The literals $-P(g(x_2))$, $-V(x_3)$, $-Q(x_4)$, $S(a)$, $T(b)$, and $-R(x_5)$ are key literals, the other literals are subproblems. The first clause in S is the basic node, it is a predecessor of all other nodes.

Definition

Let G be a deduction plan and e an edge of G with label (t,u,v), where (omitting the sign) $u = P(u_1,\ldots,u_n)$, $v = P(v_1,\ldots,v_n)$ ($n \geq 0$). To e a constraint set C(e) is assigned by

$$C(e) = \begin{cases} \{\{u_1,v_1\},\ldots,\{u_n,v_n\}\} & \text{if } n \geq 1 \\ \emptyset & \text{if } n=0 \end{cases}$$

A constraint set C(G) is assigned to G by

$$C(G) = \bigcup_{e \in E(G)} C(e)$$

G is called *correct* iff C(G) is unifiable. A correct subplan H of G is called *maximal correct* iff there is no correct subplan H' of G such that $V(H) \subset V(H')$ and $E(H) \subset E(H')$.

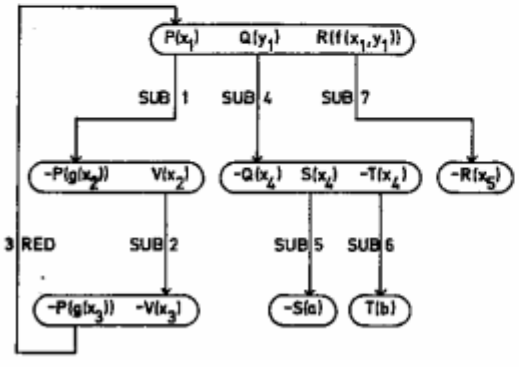

Figure 1: A closed deduction plan

θ(G) denotes the most general unifier of C(G). θ(G)os(G) is the clause *derived from* G. If G is closed, i.e. os(G) = ∅, the clause derived from G is the empty clause.

Soundness and completeness of the deduction plan method are shown in the references given above.

4 UNIFICATION GRAPHS WITH CONSTRAINTS

Unification by means of unification graphs with constraints is closely related to the unification method of Cox, (Cox 1981). It simplifies this method but is still sound and complete. Cf. for this section (Dilger and Janson 1983) and (Dilger and Janson 1984).

The unification process starts with a constraint set C. By two steps, the *transformation step* and the *sorting step*, it yields a *unification graph with constraints, UwC* for short, for C. An UwC consists of

- the node set V(UwC) = SEXPR(C)
- the label set I(UwC) = 2^C
- the edge set E(UwC) = EU(UwC) ∪ ED(UwC)
  where $EU(UwC) \subseteq V(UwC) \times (2^C - \{\emptyset\}) \times V(UwC)$
  and $ED(UwC) \subseteq V(UwC) \times \{\emptyset\} \times V(UwC)$

EU(UwC) is a set of undirected edges, ED(UwC) a set of directed edges. Construction of UwC starts with the initial graph UwC_I which consists only of the nodes. EU(UwC) is determined in the transformation step, ED(UwC) in the sorting step.

Definition

A path in UwC which contains only edges from EU(UwC) is called a *connection*. A connection $v = p_1,e_1,\ldots,e_n,p_{n+1}$ is called *simple* iff $p_i \neq p_j$ for all i,j such that $1 \leq i < j \leq n+1$, i.e. all nodes are pairwise different. A connection of length 0 is called *trivial*. A closed path in UwC which contains at least one edge from ED(UwC) is called a *loop*. A loop is called *simple* iff $p_i \neq p_j$ for all i,j such that $1 < i < j < n+1$. If e = (p,a,q) is an edge in E(UwC), then a is called the value of e, denoted val(e) = a. Let $w = p_1,e_1,\ldots,e_n,p_{n+1}$ be a path in UwC. Then the value of w is

$$val(w) = \begin{cases} \bigcup_{i=1}^{n} val(e_i), & \text{if } n \geq 1 \\ \emptyset, & \text{if } n=0 \end{cases}$$

The transformation step

The algorithm of the transformation step can be found in (Dilger and Janson 1984). It draws undirected edges between the nodes in the following way: If $c_i = \{p,q\}$ is a constraint, the

the nodes p and q are connected by the edge e = (p,{c_i},q). Then the constraint set C(ē) (cf. section 3) is added to the input constraint set, and the constraints of C(e) are treated later on in the same way.

Example

Let C = {c_1,c_2} be a constraint set with

c_1 = {G(s,z), G(u,F(y,y))}
c_2 = {u, F(y,G(s,z))}

The initial UwC consists only of the nodes SEXPR(C) and is shown in figure 2. The first constraint c_1 is removed, an undirected edge is added to the UwC and the new constraints {s,u} and {z,F(y,y)} are added to the constraint set. This results in the UwC of figure 3.

Now the second constraint is removed from the constraint set. Because u is a variable, there cannot be formed any new constraints, only an edge is added to the UwC. Thus one gets the UwC of figure 4. The remaining two constraints are treated as the second one. Because they had their origin in the first constraint, the edges in the UwC are labelled by {c_1}. At the end of the transformation step the UwC has the form represented in figure 5.

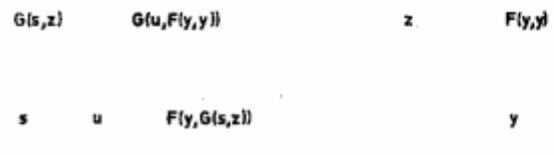

Figure 2: The initial UwC for the constraint set C

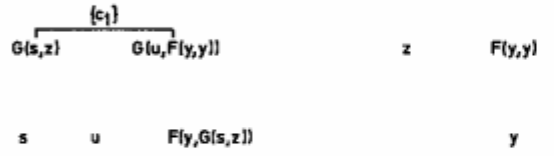

Figure 3: The UwC after the first substep

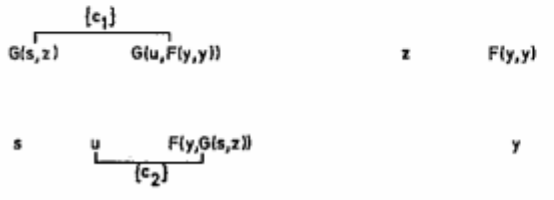

Figure 4: The UwC after the second substep

The sorting step

The transformation step classifies the nodes of UwC in such a way that two nodes belong to the same class iff there is a connection betweeen them. In the example above we have four classes. In the sorting step, first a graph U is constructed which consists of these classes as nodes and which has a directed edge labelled by f from class X to class Y iff there is a term $f(p_1,\ldots,p_n)$ in X and an expression p_i ($i \in \{1,\ldots,n\}$) in Y. This graph is shown for the example in figure 6. Now the edges of U are carried over to the UwC, but not all, rather only those which are contained in a cycle are added to the UwC as edges between the appropriate nodes and labelled by ∅. So we get the complete UwC of figure 7.

Soundness and completeness of the unification algorithm are proved in (Dilger and Janson 1984). The main theorem is: A constraint set C is unifiable iff all terms in UwC which are connected by a simple connection begin with the same function symbol and UwC contains no simple loops.

Thus, e.g., our example constraint set is not unifiable because the UwC of figure 7 contains a simple loop.

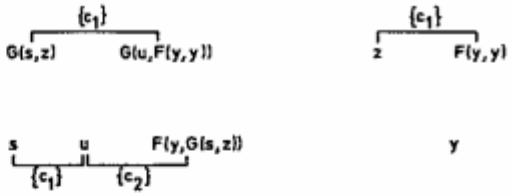

Figure 5: The UwC at the end of the transformation step

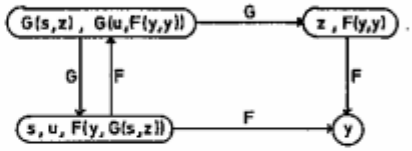

Figure 6: The graph U for the UwC

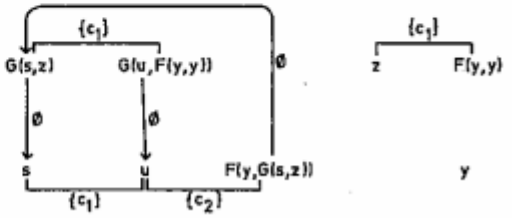

Figure 7: The complete UwC at the end of the sorting step

5 INTELLIGENT BACKTRACKING

If during the unification process a unification conflict has been detected i.e. a clash (unification of terms with different function symbols) or a cycle, the actual deduction plan is not correct. One or several steps in the construction have to be reset in order to get a correct subplan. By means of the information kept by the UwC these steps can be determined immediately. The numbers of the deduction steps are contained in the labels of the undirected edges of UwC. Therefore, we have to examine the values of certain paths through UwC. First, the relevant values are gathered in the sets ATTACH and LOOP.

ATTACH := {a $\subseteq$ C | a is the value of a simple connection in UwC between terms p and q with different function symbols}

LOOP := {a $\subseteq$ C | a is the value of a simple loop in UwC}

Let B_w be a function on 2^C which yields Boolean expressions defined by

$$B_w(a) = \sum_{c \in a} c$$

Now on the basis of ATTACH and LOOP the Boolean expressions B_{ATTACH}, B_{LOOP}, and B_{UNIF} are defined as

$$B_{ATTACH} := \begin{cases} 1 & \text{, if ATTACH} = \emptyset \\ \prod_{a \in \text{ATTACH}} B_w(a) & \text{, otherwise} \end{cases}$$

$$B_{LOOP} := \begin{cases} 1 & \text{, if LOOP} = \emptyset \\ \prod_{a \in \text{LOOP}} B_w(a) & \text{, otherwise} \end{cases}$$

$$B_{UNIF} := B_{ATTACH} \wedge B_{LOOP}$$

It is easy to verify that B_{UNIF} is a conjunction of disjunctive terms. The minimal disjunctive normal form of this term has the form

$$B'_{UNIF} = B_1 \vee \ldots \vee B_k$$

for some $k \geq 1$, where each B_i is a conjunctive term. From B'_{UNIF} the minimal conflict sets are determined by

$$mcs_i := \bigcup_{\substack{c \text{ occurs} \\ \text{in } B_i}} \{c\} \quad (i = 1,\ldots,k)$$

For details cf. (Dilger and Janson 1984).

Examples

Consider the UwC of section 4, figure 7. Clearly, ATTACH = $\emptyset$. There are two simple loops in the UwC, which have the same value, namely $\{c_1,c_2\}$, thus LOOP = $\{\{c_1,c_2\}\}$. Then $B_{ATTACH} = 1$, $B_{LOOP} = c_1 \vee c_2$, and $B_{UNIF} = 1 \wedge (c_1 \vee c_2) = c_1 \vee c_2$. Therefore $B'_{UNIF} = c_1 \vee c_2$. Therefore two minimal conflict sets exist, $mcs_1 = \{c_1\}$ and $mcs_2 = \{c_2\}$.

Take as another example the deduction plan of section 3, respresented in figure 1. Following the edges according to their numbers we get the constraints

1: $\{x_1, g(x_2)\}$
2: $\{x_2, x_3\}$
3: $\{g(x_3), x_1\}$
4: $\{y_1, x_4\}$
5: $\{x_4, a\}$
6: $\{x_4, b\}$
7: $\{f(x_1,y_1), x_5\}$

The UwC for these constraints is shown in figure 8. It has no directed edges, because the graph U, constructed in the sorting step, contains no cycles.

There is a clash in the UwC, namely a simple connection between a and b. Therefore, ATTACH = $\{\{5,6\}\}$. Clearly, LOOP = $\emptyset$. Thus, $B_{ATTACH} = 5 \vee 6$, $B_{LOOP} = 1$, $B_{UNIF} = 5 \vee 6 = B'_{UNIF}$ and $mcs_1 = \{5\}$, $mcs_2 = \{6\}$.

Now by means of the minimal conflict sets a maximal correct subplan H of a deduction plan G is determined in three steps: Let $mcs_i = \{i_1,\ldots,i_1\}$ be a minimal conflict set of G.

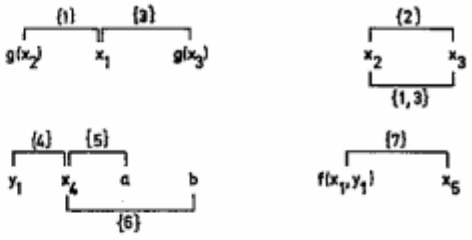

Figure 8: A complete UwC

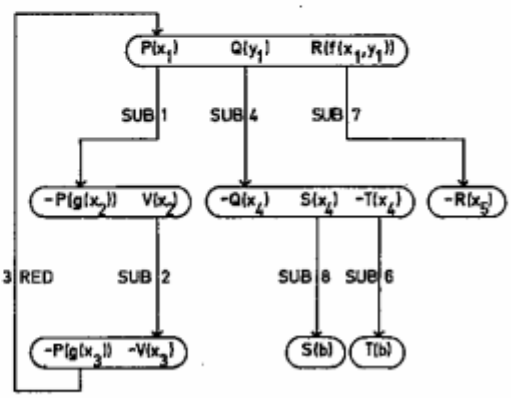

Figure 9: A closed correct deduction plan

1. Set H := G.
2. Remove the edge i_j in H for all $j = 1,\ldots,l$.
3. If i_j of type SUB, remove all nodes and edges beyond of i_j. (i.e. attainable by SUB-pathes from i_j).

If it happens during this process that some literals, say $u_1,\ldots,u_m$, become open subproblems but there are no literals at hand to close them, then for each u_j a preceding literal has to be found which can be closed by yet another literal. If for one of the u_j no such preceding literal exists, the backtracking process was not successful and another mcs_i has to be chosen.

Example

Consider the deduction plan of section 3, figure 1. As shown in the example above, the constraint set corresponding to this plan is not unifiable. Take for the backtracking step $mcs_1 = \{5\}$. Edge number 5 and node -S(a) are removed from the plan. Thereby, the literal $S(x_4)$ becomes an open subproblem. But there is another clause in the input clause set which fits to close the literal, namely {-S(b)}. This yields the closed correct deduction plan of figure 9. The reader is invited to check that backtracking with $mcs_2 = \{6\}$ does not result in a correct plan.

6 THE STRUCTURE OF ASSIP-T.

In the deduction plan method, deduction and unification are separated from each other. For deduction the data structure "deduction plan" is needed, for unification the data structure "unification graph with constraints". In ASSIP-T, both are kept in appropriate associative memories. Therefore ASSIP-T has two storage units of this type, namely AM1 for the deduction plan and AM2 for the UwC, cf. figure 10. The control unit of the processor consists of four components:

- the head control HC
- two subcontrols SC1 and SC2
- a conventional memory CM

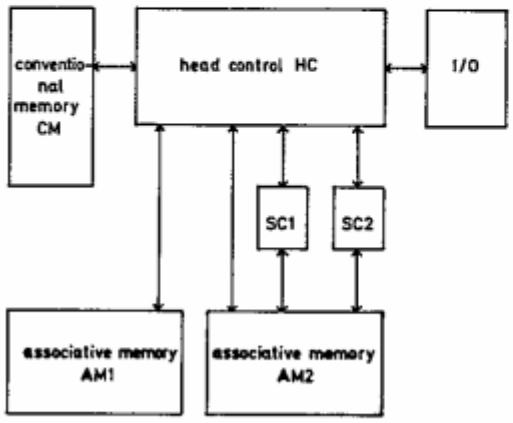

Figure 10: The structure of ASSIP-T

The subcontrols operate on the UwC. They can work independently from each other, but under control of HC, so they can work in parallel and this is useful during the initial construction of the UwC and during its reconstruction after a backtracking step. Thus, we have not only parallel access to the data in the associative memories, rather there are two further steps to parallel processing: one by the parallel treatment of deduction plan and UwC, the other by the use of SC1 and SC2 in parallel. For details cf. (Dilger and Schneider 1985). For an introduction to and a survey on the field of associative processors cf. (Fu and Ichikawa 1982), (Kohonen 1984), (Parhami 1973) and (You and Fung 1975).

7 DATA STRUCTURES AND ALGORITHMS FOR ASSIP-T

Two data structures are used for the representation of deduction plans and UwCs in the associative memories, namely CLAUSE and EXPRESSION respectively. CLAUSE represents clauses together with the edges of the deduction plans, and similarly EXPRESSION is designed to keep expressions as well as the different types of edges in the UwC. Here is the type definition for CLAUSE:

```
CLAUSE =
 RECORD
  status : RECORD
             in_use: BOOLEAN;
             next_literal,
             compl_literals: CARDINAL;
           END;
  was_basic_node: BOOLEAN;
  number_of_literals,
  number_of_variants: CARDINAL;
  variants:
    LIST OF
     RECORD
      index,
      open_subproblems: CARDINAL;
      key_literal: POINTER TO LITERAL;
     END;
  literals: LIST OF LITERALS;
END;
```

The design principle of this data type is to represent a clause together

with its variants. Thus, only those parts of a clause have to be stored several times, in which the variants differ from each other, namely an index (number of the variant), the open subproblems and the key literal. The status information and the was_basic_ node-variable are used for the construction of the deduction plan. LIST can be realized e.g. as an ARRAY of appropriate length. The data type LITERAL is defined as follows:

```
LITERAL =
 RECORD
  sign: BOOLEAN ;
  predicate_symbol: SYMBOL;
  arity: CARDINAL;
  arguments: LIST OF POINTER TO EXPRESSION;
  variants:
   LIST OF
    RECORD
     index,
     edge_number: CARDINAL;
     edge_type: (SUB,RED);
     corr_literal:
      RECORD
       cl: POINTER TO CLAUSE;
       lit: POINTER TO LITERAL;
       ind: CARDINAL;
      END;
     erased: BOOLEAN;
     potential: CARDINAL;
    END;
 END;
```

The data structure LITERAL consists of a constant part (sign,predicate symbol, arity, arguments) and a variant part which is related to the variants of the clauses by the property "index". In the variant part of LITERAL the edges of the deduction plan are stored, because in fact they are drawn between variants of clauses. The literal wherein an edge is stored is its start literal, and corr_literal is the target literal of the edge. Edges can be erased by a backtracking step, but they must be kept to avoid their redrawing, therefore they can be marked as erased by a special record entry. The potential of a literal is the number of literals which are possibly complementary to it. SYMBOL can be realized as ARRAY OF CHAR or as CARDINAL, if all symbols are numbered at the beginning. The definition of the data type EXPRESSION is

```
EXPRESSION =
 RECORD
  status:
   RECORD
    active,const,in_use,marked:BOOLEAN;
   END;
  content: SUBEXPRESSION;
  variants:
    LIST OF
     RECORD
      index,number_of_neighbours:CARDINAL;
      neighbours:
       LIST OF
        RECORD
         node: POINTER TO EXPRESSION;
         ind:  CARDINAL;
         label: SET OF LABELS;
        END;
      class: CARDINAL;
      pushed: BOOLEAN;
      cycle: ARRAY OF BOOLEAN;
     END;
 END;
```

The status information is used for construction and handling of the UwC. The variants correspond to the variants of clauses in the decution plan (by "index"). Each variant of a clause produces new expressions which are identical to former ones in the symbols they contain, but differ in the edges to some other expressions, and these are kept under the property "variants", especially under "neighbours". Expressions which are connected with each other, thus belonging to the same class, have the same number under the property "class". By the notation of neighbours, the undirected edges of the UwC are represented, whereas the directed edges are represented by means of the cycle-array. "pushed" is used for the handling of the UwC. The type LABEL keeps the labels of the undirected edges which correspond to the numbers of the edges. SUBEXPRESSION is defined as follows:

```
SUBEXPRESSION =
 RECORD
  name: SYMBOL;
  CASE type: (fnct,var) OF
     fnct: arity: CARDINAL;
           arguments: LIST OF POINTER TO
                      EXPRESSION;
 END;
```

SUBEXPRESSION contains only a symbol if the expression is a variable, otherwise it contains also the arity of the function symbol and an argument list.

Example

Assume there are given the clauses:

(1) {P(g(x),x), R(f(z,b),g(x))}
(2) {-P(u,f(g(u),v)), Q(f(v,u))}
(3) {-Q(f(w,b))}

These clauses contain the following expressions and subexpressions:

```
(1)  g(x)        (7)  f(g(u),v)
(2)  x           (8)  g(u)
(3)  f(z,b)      (9)  v
(4)  z           (10) f(v,u)
(5)  b           (11) f(w,b)
(6)  u           (12) w
```

In the data structure CLAUSE the clauses are respresented as follows:

```
c1: [[F,1,O],F,2,1,(),([T,P,2,(e1,e2),()],
                       [T,R,2,(e3,e1),()])]
c2: [[F,1,O],F,2,1,(),([F,P,2,(e6,e7),()],
                       [T,Q,1,(e10),()])]
c3: [[F,1,O],F,1,1,(),([F,Q,1,(e11),()])]
```

e1,e2,... are pointers to expressions. This is not yet a deduction plan because the lists of variants are empty, it is only the respresentation of the clauses in the data structure CLAUSE.

The expressions in the data structure EXPRESSION have the form:

```
e1: [[T,F,F,F],[g,1,(e2)],()]
e2: [[T,F,F,F],[x],()]
e3: [[T,F,F,F],[f,2,(e4,e5)],()]
e4: [[T,F,F,F],[z],()]
e5: [[T,T,F,F],[b,O,()],()]
e6: [[T,F,F,F],[u],()]
e7: [[T,F,F,F],[f,2,(e8,e9)],()]
e8: [[T,F,F,F],[g,1,(e6)],()]
e9: [[T,F,F,F],[v],()]
e10:[[T,F,F,F],[f,2,(e9,e6)],()]
e11:[[T,F,F,F],[f,2,(e12,e5)],()]
e12:[[T,F,F,F],[w],()]
```

Again, this is not an UwC but only the representation of the expressions in the data structure EXPRESSION.

The algorithms for the construction and handling of the deduction plan and the UwC can be found in (Dilger and Schneider 1985). Within the construction of the UwC an important task is to determine new constraints from the arguments of terms. Assume a constraint {p,q} is given and p and q belong to different classes, i.e. there is no connection between them. This can be immediately realized by the property "class". This situation happens often during the construction process, especially at the beginning. Now, all terms of the classes to which p and q belong have to be determined and from their arguments new constraints are computed. The search for terms is performed by the subcontrols of ASSIP-T and can be done in parallel because the classes of p and q are disjoint. Synchronization of the access to AM2 is simply realized by semaphores. Augmenting the number of subcontrols, further parallelism could be introduced, but then the overhead for synchronization would increase.

Example

From the clauses of the example above the following deduction plan can be constructed:

```
c1: [[T,3,O],T,2,1,([1,1,NIL]),
        ([T,P,2,(e1,e2),([1,1,SUB,[c2,11,1],
        F,O])],[T,R,2,(e3,e1),([1,O,λ,[],
        F,O])])]
c2: [[T,3,O],F,2,1,([1,O,11]),
        ([F,P,2,(e6,e7),([1,O,λ,[],
        F,O])],[T,Q,1,(e10),([1,2,SUB,
        [c3,11,1],F,O])])]
c3: [[T,2,O],F,1,1,([1,O,11]),
        ([F,Q,1,(e11),([1,O,λ,[],
        F,1])])]
```

It is illustrated in figure 11. This deduction plan is not closed, because in c1 there is an open subproblem. The edges of the deduction plan yield the constraint set

```
C = {{g(x),u}, {x,f(g(u),v)},
               {f(v,u),f(w,b)}}
```

which is for the purpose of ASSIP-T denoted by

```
S = ([[e1,1],[e6,1],{1}],
     [[e2,1],[e7,1],{1}], [[e10,1],
                           [e11,1],{2}])
```

The numbers "1" and "2" denote the deduction step from which the respective constraints has been obtained. The unification algorithm builds from the input set S an UwC of which the first four expressions represented in the data structure EXPRESSION are shown below and which is illustrated in figure 12.

```
e1: [[T,F,F,F],[g,1,(e2)],
       ([1,1,([e6,1,{1}]),1,F,<T>])]
e2: [[T,F,F,F],[x],([1,1,([e7,1,{1}]),2,F,
                     <>] )]
e3: [[T,F,F,F],[f,2,(e4,45)],
       ([1,O,(),3,F,<F,F>])]
e4: [[T,F,F,F],[z],([1,O,(),4,F,<>])]
```

Those expressions with a T in the cycle array are involved in a cycle with regard to the respective arguments.

The UwC contains a cycle, therefore the constraint set C is not unifiable.

8 THE ASSIP-T MEMORY

We will sketch here a possible realization of AM2 which can be adopted for AM1 as well. The UwC is stored in AM2 as a set of objects of type EXPRESSION. It seems natural to divide AM2 into equal parts each of which is provided for an EXPRESSION. But one will hesitate to call such a part a "cell", because it has to

store a considerable amount of information and it should be able to perform a lot of instructions which tend to go beyond those which usually are performed by associative memory cells. Rather it seems adequate to conceive such a part as an associative processor itself. Thus, associative access to AM2 proceeds on two levels: the first one is that of the head control and the subcontrols by means of the instructions "FOR_ALL expression..." and "FOR_ONE expression..." occurring in the algorithms, the second one is that of the subprocessors to the data stored in their registers. A subprocessor has to perform entry, change and query instructions on the components of a single EXPRESSION. The subprocessor memory can be realized as a linear array of cells each of which consists of

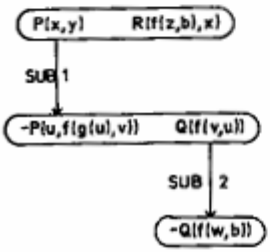

Figure 11: A deduction plan

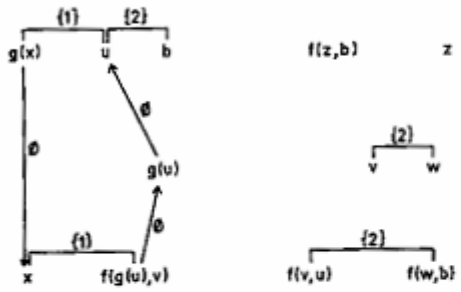

Figure 12: A UwC with a cycle

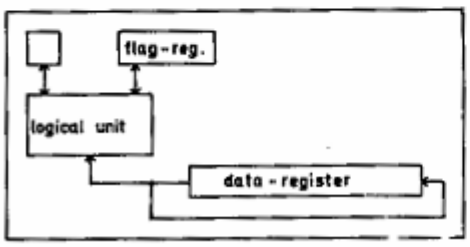

Figure 13: A subprocessor cell of ASSIP-T

- a logical unit
- a control bit
- a 4 bit flag register
- a 32 bit data register

cf. figure 13. The purpose of the flag register is to characterize the type of information which actually is stored in the data register, e.g. status information, index and class of variants, information about neighbours etc. Thus, each cell can store an arbitrary part of an EXPRESSION.

9 CONCLUSION

As far as we know there is no other approach similar to ours. The architecture of the fifth generation inference machine is data flow oriented and does not take into consideration associative access to data cf. (Motooka and Fuchi 1983). The main problem with the realization of our approach is the size of the associative memories needed for the processor. We hope to find a solution for this problem on the basis of the work of Tavangarian (Tavangarian 1982 and 1983). At present we are busy to implement simulation programs to get some experience with ASSIP-T so far described, and later on we will try to improve and to extend this processor model.

REFERENCES

Cox, P.T. On determining the causes of nonunifiability. Auckland Computer Science Report No 23, University of Auckland, 1981

Cox, P.T. and Pietrzykowski, T. Deduction plans: A basis for intelligent backtracking. University of Waterloo Res.Rep. CS-79-41, 1979

Cox, P.T. and Pietrzykowski, T. Deduction plans: A basis for intelligent backtracking. IEEE Trans. Pattern Analysis and Machine Intelligence, vol. PAMI-3, (1) 1981, 52-65

Dilger, W. and Janson, A. Unifikationsgraphen als Grundlage f0r intelligentes Backtracking. Proc. of the German Workshop on Artificial Intelligence, Informatik-Fachberichte 76, Springer-Verlag, 1983, 189-196

Dilger, W. and Janson, A. Intelligent backtracking in deduction systems by means of extended unification graphs. Interner Bericht 100/84, Fachbereich Informatik, Universität Kaiserslautern

Dilger, W. and Schneider, H.-A. A theorem proving associative processor, In preparation.

Fu, K.S. and Ichikawa, T. (eds) Special computer architectures for pattern processing. CRC Press, Boca Raton, Florida, 1982

Kohonen, T. Self-organization and associative memory. Springer, Berlin, 1984

Moto-oka, T. and Fuchi, K. The architectures in the fifth generation computers. Proc. of the IFIP 83, 1983, 589-602

Parhami, B. Associative memories and processors: An overview and selected bibliography. Proc. of the IEEE 61, 1973, 722-730

Tavangarian, D. A novel modular expandable associative memory. Proc. of EUROMICRO 82, 1982, 303-312

Tavangarian, D. A general purpose associative processor. Proc. of EUROMICRO 83, 1983, 187-197

You, S.S. and Fung, H.S. Associative processor architecture - a survey. Proc. of the Sagamore Computer Conference 1975