ICOT Technical Report: TM-0046

TM-0046

Several Aspects on Unification

by
ICOT Working Group WG.5

February, 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Several Aspects on Unification

1. Introduction

Toshiaki Kurokawa

The principal purpose of the WG5 of the Institute for New Generation Computer Technology is to investigate the theoretical background on the Fifth Generation Computer Systems (FGCS in short).

Since last year, we have surveyed the various areas relating to the FGCS such as computational complexities, foundation of mathematics, VLSI algorithms, logic programming languages, etc.

This year, we have selected a few major areas and have focused our discussions to these areas. They are higher order, unification, and parallel computation.

We had the successive meetings from May to August in 1983 to investigate the various aspects of 'unification'. This report is based on the discussions, but it contains the extended results since then.

It is well known that 'unification' is a key mechanism of logic programming. And it is fruitful to pursue the every aspect of 'unification', to survey the current state of the art, to present our new research results, and to list the open problems for the future attack.

It might be unnecessary to the reader, but I will introduct the concept of 'unification' very briefly.

'Unification' is, in a sense, an extended idea of the variable binding. A term is defined as a constant, a variable, or a functor denoted as f(x1,...,xn) where f is a function designator and the argument, xi is (recursively defined) term.

Two term is said to be unifiable or not depending the following algorithm:

i) If both term does not contain variables, then they are unifiable only if they are the same.

ii) If one is a variable, they are unifiable with the condition that the variable is 'unified' with the other terms. The substitution of the variable is called the 'unifier'. Note that this case covers when the both are the variables.

iii) In the other cases, to find the set of 'unifiers' to make the substituted terms are same in the sense of i) and ii).

Please note that I omitted the definition of the function and the algorithm to find the set of 'unifiers'. They are the very theme of the following sections.

Now I would like to introduce the contents of this report.

In section 2, Mukai and Kasai survey the well known algorithms for unification of the first order terms, where only the function symbol (constant) is permitted for the functor. They analyze the computational complexities of these algorithms. As Hirose commented, we need more elaborated discussions about the computational complexities, and more specialized algorithms for unification in the future.

So far, the algorithms are built for the sequential machines. However, the new VLSI technology provides the highly parallel algorithms for unification. In section 3, Yasuura presents a parallel algorithm for unification and analyze the computational complexity of the algorithm. His main result is that the general unification problem is log-space complete for PTIME, that is the most difficult problem. In his conclusion, however, he suggest that there is a

possibility in building the efficient unification procedure for the practical cases.

To implement the efficient unification algorithm, it is necessary to consider the hardware supports. The hashing mechanism is one of the candidates, which is explained in section 4 by Ida.

Moreover, 'unification' has relevances with other fields in the foundation of computer science. In section 5, Sakai and Matsuda explain the notion of 'universal unification' where most of the problems of theoretical interest can be discussed under the term 'unification'. And they listed various cases for the unifications with the existence of the unification algorithms including the availability of the 'most general unifier'.

'Unification' casts another aspect. That is, 'unification' can be interpreted from another point of view. For example, in section 6, Ueda and Kurokawa argue that the unification process can be seen as the constraint process. Ueda claims that the unification as constraint will play the important role also in the knowledge representation.

As for the semantics of unification, in section 7, Hagiya discusses the relation of lazy unification and bind-hook mechanism, which is introduced for the SIM hardware to support KL0 language of FGCS.

In section 8, Nagata presents the note concerning unification and formula manipulation. His point is that the problem solving and the knowledge representation are the key ideas. He states the problems and the methods about this important area.

Adachi presents another view on the unification in section 9, namely from the categorical point of view.

We do not claim that we have covered the all area of unification. For example, we omit the unification in the design and the execution in the programming language issue. The interesting attempt is made in the compilation of PARLOG language: it compiles the general unification to the one-way unification, i.e. the matching. [Gregory 83] It means that there is a way, as Yasuura suggested, to make a very efficient unification program for the practical application.

REFERENCE

[Gregory 83] Gregory, S. "Getting started with PARLOG" memo ICOT, (Oct. 1983)

2. Complexity of Unification Algorithms

T. Kasai and K. Mukai

There are two classes of unification algorithms for first order terms. One is for finite terms, and the other is for infinite trees. Computational complexities of unification algorithms for finite terms are compared in [M&M 82]. The next subsection is a summary of the comparison.

A rough discussion such as to determine whether or not a given problem can be solved in polynomial time is independent of the machine model and the way of defining the size of problems, since any of the commonly used machine models can be simulated by any other with a polynomial loss in running time and by no matter what criteria the size is defined, they differ from each other by polynomial order. However, in precise discussion, for example, in the discussion whether the computation of a problem requires O(n) time or O(n log n) time, the complexity heavily depends on machine models, representation methods of problems and the definition of size. The following discussion, however, uses the standard model of current computers (RAM models) and uses the same complexity measure, and hence it would give a good criteria of the efficiency of the algorithms.

2.1 Unification of Finite Terms

The algorithms to be compared are from [M&M 82], [P&W 78] and [Hu 76].

Complexity result:

[M&M 82]	m + n*log n; where m is the sum of sizes of two terms given, and n is the number of distinct variables of the input system.
[P&W 78]	linear
[Hu 76]	almost linear

When occur-checks are performed?:

[M&M 82]	step by step incrementally
[P&W 78]	step by step incrementally
[Hu 76]	delayed till last steps

Hight of terms:

[M&M 82]	any
[P&W 78]	1
[Hu 76]	1

Remark: the conventional unification algorithm used in Dec-10 Prolog has an exponential order of computational complexity and don't care occur-check. It has not guaranteed termination.

Comparison of efficiency under some situations: let Pm, Pc, and Pt be the probabilities of stopping with success, failure for the detection of a cycle, and failure for the detection of a clash.

(1) Pm>>Pc, Pt (very high probability of stopping with success)
[P&W 78] > [M&M 82] > [Hu 76]
(more efficient > less efficient)

(2) Pc>>Pt>>Pm (very high probability of detecting a cycle)
[P&W 78] > [M&M 82] > [Hu 76]

(3) Pt>>Pc>>Pm (very high probability of detecting a clash)
[Hu 76] > [M&M 82] > [P&W 78]

2.2 Unification of Infinite Terms

It seems that there are at least two motivations behind introducing infinite trees into Prolog [Col 80]:

(1) the time expensive operation occur-check is unnecessary for a unification algorithm.

(2) data structure like a directed graph can be nicely represented by the infinite trees and manipulated efficiently.

A unification algorithm for infinite trees don't care occur-check, but in stead of the check it must have the guaranteed termination property. A theoretical algorithm is in [Col 80] and [Col 82]. Efficient algorithms are proposed in [Mo 78], [Fi 78], [Fa 83], and [Mu 83]. Unfortunately, complexities of these algorithms are not described explicitly. The followings are only rough estimations of efficiency.

> [Mo 78] n**3 ; because it maintains a list of pairs of nodes and take linear searches over the list repeatedly.
>
> [Fi 82], [Fa 83] n log n ; because they seem to be the same as [Hu 76] minus occur-check.
>
> [Mu 83] n * log n ; conjecture

There are two implementation types for Prolog. One is "copying" type. The other is "structure sharing" one. [Mu 83] is the algorithm which was designed to work efficiently for the structure sharing implementation (of PSI machine of ICOT, for instance). The unification algorithm in [M & M 82] uses "FRONTIER" and "COMMON-PART" operations and also uses "counters" and "multi-terms" as data types. The algorithm in [Mu 83] uses only "FRONTIER" operation out of them. Note that "COMMON-PART" operation is useful only for "copying" implementatin.

Most recently, J. Jaffar proposed a new algorithm of unification over infinite terms in [Ja 84], which has almost linear time complexty. The algorithm uses "COMMON-PART", "FRONTIER" and "counters" mentioned above. He also gave time efficiencies of the following algorithms by implementations in the programming language C and a VAX 11/780 running UNIX 4.1 BSD.

J. Jaffar
A. Colmerauer-1
A. Colmerauer-2
K. Mukai
J. Corbin and Bidoit-1
J. Corbin and Bidoit-2

He made a summary his results as follows: all the algorithms have about equal speed when the unification is straightforward. His algorithm progressively grows relatively faster with progressively more complicated unification.

REFERENCE

[Co 80] Colmerauer,A.: Prolog and Infinite Trees, in Logic Programming, Academic Press, 1982.

[Co 82] Colmerauer,A.: Prolog II, manual de reference et modele theorique, Rapport interne, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II, Mars 1982.

[Fa 83] Fages, F.: Note sur l'Unification des Terms de Premier Order Finis et Infinis, Laboratoire Informatique Therique et Programmation, 83-29, Mai 1983.

[Fi 82] Filgueiras, M.: A Prolog Interpreter Working With Infinite Trees, FCT/UNL-20/82, Universidade Nova de Lisboa, Nov. 1982.

[Hu 76] Huet,G.: Resolution d'equations dans des languages d'ordre 1,2 ... omega. These D'Etat, Universite de Paris VII, 1976.

[Ja 84] Jaffar,J.: Efficient Unification over Infinite Terms, (private communication and forthcoming), Jan. 1984.

[Mo 78] Morris,F.L.: On List Structures and Their Use in the Programming of Unification, School of Computer and Information Science, Syracuse Univ. Aug 1978.

[Mu 83] Mukai,K.: A Unification Algorithm for Infinite Trees, Proceedings of the Eighth IJCAI, Aug. 1983.

[M&M 82] Martelli,A. and Montanari, U.: An Efficient Unification Algorithm. ACM TOPLAS., Vol. 4,No. 2, Apr 1982.

[P&W 78] Paterson,M.S. and Wegman,M.N.: Linear Unification, J. of Computer and System Sciences 16, 1978.

On the Parallel Computational Complexity of Unification

Hiroto YASUURA

Introduction

The unification problem is defined as follows: Given two terms consisting of function symbols and variables, find, if it exists, a simple substitution which makes two terms equal.

Two linear unification algorithms were proposed, in sequential computation, independently by Paterson-Wegman[1] and Martelli-Montanari[2]. For more efficient implementation of unification, one may try to design a parallel unification algorithm. However, no such efficient algorithm has been known and implemented.

Here, we are concerned with the complexity of unification in parallel computation. As the result, we show that it is very hard to design a much more efficient unification algorithm using parallel computation than sequential one[3]. In other words, unification contains essentially sequential computation which might not accelarate by parallel compuatation scheme.

For the model of parallel computation, we use combinational logic circuits constructed of fan-in restricted logic elements. This model seems to be one of the most stable models of parallel computation from computational complexity view point. Moreover, combinational logic circuits are the most natural model of implementation of parallel algorithms by hardware. The computation time is measured by the depth of circuits.

We adopt a graph representation of terms for unification. In order to discuss the relation between the complexity of logic circuit and unification, we introduce a new problem, called the reachability problem on directed hypergraphs[3]. A directed hypergraph is a generalization of a directed graph and the reachability problem on the directed hypergraph is defined as an extension of one on the directed graph. We show a parallel algorithm for unification in which the unification problem is

reduced to the reachability problem of a directed hypergraph. We discuss about the lower bound of the computation time of unification in parallel and show the unification problem is log-depth complete for a class of problems that can be solved by polynomial size circuits[4][5]. Namely, if we can design circuits with depth $O(\log^k n)$ for unification, all problems in this class can be computed by circuits with depth $O(\log^k n)$ for any positive integer k. Therefore, we can say that unificatin is the hardest problem in this class which includes all problems having polynomial algorithms in sequential computation. It is also shown that unification is log-space complete for PTIME, by the simillar discussion on the above parallel computation.

Definitions

Let A_i (i=1,2,...) be a set of i-adic function symbols and A_0 be a set of constant symbols. $\bigcup_{i=0,1,...} A_i$ is denoted by A. Let X be a set of variables. We assume that $A \cap X = \phi$.

A term t can be represented by an acyclic directed graph G=(V,E), called a term graph, as the following manner:

(1) Each vertex v in V has a label p in $A \cup X$. No two vertices have a same label in X, and the outdegree of them are 0. A vertex having a label in A_i ($i \geq 0$) has i outgoing edges each of which is labeled by a distinct positive integer j in {1, 2, ..., i}.

(2) A vertex with label p in $A_0 \cup X$ represents term p (p is a constant or a variable).

(3) A vertex v with label f in A_i ($i \geq 1$) represents term $f(t_1, t_2, ..., t_i)$ where t_j is a term represented by the vertex pointed by the j-th outgoing edge of v.

We sometimes call vertices with labels in X variable vertices, and ones with labels in A function vertices.

A term graph G=(V,E) is encoded in the following set of tuples.

$\{(v,p,v_1,v_2,...,v_i) \mid$ v is a vertex in V, p is the label of v, and (v,v_j) $(1 \leq j \leq i)$ in E has the label j$\}$

We consider a binary coding of the tuple sequence $\int G$. Let n be the number of vertices in V and m be the number of edges in E. Since each vertex v and its label p can be coded by $\lceil \log_2 n \rceil$ bits, respectively, the length of the binary coding is O(m log n + n log n) bits.

[Definition 1] The unification problem (UP) is defined as follows: For a given coding $\int G$ of a term graph G and two vertices v_1 and v_2 in G, find, if it exists, a most general unifier s for terms t_1 and t_2 which are represented by v_1 and v_2 respectively.

[Definition 2] The unifiability decision problem (UDP) is defined as follows: For a given coding $\int G$ of a term graph G and two vertices v_1 and v_2 in G, decide whether or not t_1 and t_2 are unifiable.

We adopt combinational logic circuits as a model of parallel computation. Combinational logic circuits are the most fundamental components of digital systems and many researches have been carried out on the complexity theory of logic functions realized by combinational circuits[4]. There is two measures to eveluate the complexity of circuits, size and depth, which correspond to the number of resources and the computation time spent in the computation respectively. In the following section we will mainly consider the depth of a circuit that computes UDP and UP.

[Definition 3] A directed hypergraph H is denoted (V,E), where V is a set of vertices and E is a set of directed hyperedges. A hyperedge e is an ordered pair of a set of vertices V_e in V and a vertex v_e $V-V_e$, denoted (V_e,v_e). The cardinarity of V_e is called the rank of hyperedge e. The rank of the directed hypergraph H is defined by the maximum rank of all hyperedges in E.

[Definition 4] In a directed hypergraph H=(V,E), v in V is said to be reachable from a subset S of V if and only if v is the member of S or there exists a hyperedge (V_e, v) such that all vertices in V_e is reachable from S.

[Definition 5] The reachability problem of directed hypergraphs (DHGAP) is defined as follows: For a given incidence matrix $H\int$ of a directed hypergraph H and a subset of vertices S, obtain all vertices in H which are reachable from S.

A Parallel Unification Algorithm

First we show an algorithm for UDP.

[Algorithm 1] UNIFY

Input A binary coding $\int G$ of a term graph G=(V,E) on $X \cup A$, and vertices v_1 and v_2 in V which represent terms t_1 and t_2, respectively.

Output If t_1 and t_2 are unifiable, the output is "1". Otherwise it is "0".

Method

Step 1. Generate a directed hypergraph H=(V',E') from G as follows:

(1) For every pair of vertices v_i and v_j in V, there is a vertex v_{ij} in V' where $v_{ij}=v_{ji}$.

(2) If v_i and v_j have the same label in A and the h-th outgoing edges of them points to v_q and v_r respectively, an edge (v_{ij}, v_{qr}) is in E'.

(3) If the label of v_q is a variable in X, a hyperedge $(\{v_{iq},v_{jq}\}, v_{ij})$ is in E'.

Step 2. Compute the reachability problem of H from $\{v_{12}\}$ in parallel.

Step 3. If there exists a vertex v_{ij} in V' such that it is reachable from v_{12} and v_1 and v_2 have different labels in A, output '0' and stop.

Step 4. For all v_{ij}'s which are reachable from v_{12} and v_i or v_j has a label in X, add edges into G by the following way. We call resulting graph G'.

(1) If the label of v_i (v_j) is in X and the label of v_j (v_i) is in A, then add edge (v_i,v_j) ((v_j,v_i)) into G.

(2) If both v_i and v_j have distinct labels in X, add edge (v_i,v_j) into G, where v_i is assumed to be assigned the smaller integer than v_j in the coding.

Step 5. Examine whether the graph G' is acyclic in parallel. If G' is asyclic output '1', otherwise '0'.

[Theorem 1] Algorithm UNIFY computes UDP correctly.

It is easy to obtain a most general unifier from graph G' which is constructed in algorithm UNIFY. Thus we have an algorithm for UP by trivial modification of UNIFY.

[Algorithm 2] UNIFY-2

Input A binary coding $\int G$ of a term graph G=(V,E) on $X \cup A$, and vertices v_1 and v_2 in V which represent terms t_1 and t_2, respectively.

Output A most general unifier for t_1 and t_2. The mgu is represented by a term graph and a set of pairs $\{(v_i, x_i)\}$ where v_i is a vertex in the graph and x_i is a variable in X.

Method

Step 1. By algorithm UNIFY, generate a directed graph G' in Step 4 of UNIFY.

Step 2. For all pairs (v_i, v_j)'s in G', if there is a path from v_i to v_j in which all vertices except v_i and v_j have labels in X, connect v_i with v_j directly.

Step 3. For every vertex v with a variable label x, if there is an outgoing edge from v, make a pair (v', x) as follows:

(1) If all outgoing edges from v are pointing only variable vertices, make pairs (v', x)'s for every v' pointed by one of these edges.

(2) If there are function vertices pointed by outgoing edges, select a function vertex v' in them and generate a pair (v', x).

Step 4. Delete all edges from variable vertices.

Step 5. Delete all vertices that are not reachable from vertices in the pairs obtained in Step 3.

Depth Complexity of Algorithm UNIFY

Let n be the number of vertices in a given term graph G, n' be the number of vertices with labels of variables, and m be the number of edges in G.

[Lemma 1] The reachability problem of a hypergraph H in Step 2 can be computed by a combinational circuit with depth $O(\log^2 n + n' \log n')$.

[Lemma 2] We can construct a combinational logic circuit with depth $O(\log^2 n)$ which decides whether a graph G' in Step 5 is acyclic.

From Lemma 1 and Lemma 2, we directly obtain the following theorem.

[Theorem 2] Algorithm UNIFY is implemented by a combinational logic circuit with depth $O(\log^2 n + n' \log n')$.

Since Step 2-5 in UNIFY-2 requires a circuit with depth $O(\log n')$, we have the following corollary.

[Corollary] Algorithm UNIFY-2 is implemented by a combinational logic circuit with depth $O(\log^2 n + n' \log n')$.

The Lower Bound

Before discussing the lower bound of the depth complexity of UDP, we introduce several concepts and notations. First, we define complexity classes of decision problems. Let P be a problem on alphabet $\{0,1\}$, P_n denotes a subproblem of P with length n, i.e.,
$P_n = P \cap \{0,1\}^n$.
Thus P_n can be cosidered as an n-variable logic function. We define complexity classes related with size and depth of logic circuits.

$PSIZE = \{P \mid$ For each P, there exists a polynomial p(n) such that $C(P_n) < p(n)\}$

$LOG^k DEPTH = \{P \mid$ For each P, $D(P_n) = O(\log^k n)\}$

Since the size of a circuit for UDP constructed in the previous section according to algorithm UNIFY is bounded by a polynomial of the input length, UDP is in PSIZE.

[Definition 6] A problem P is said to be log-depth complete for PSIZE if and only if P satisfies the following two properties:

(1) P is in PSIZE.

(2) For every problem Q in PSIZE, there exists a circuit with depth O(log n) which transduce Q_n to $\{P_1, P_2, \ldots, P_m\}$ where m is bounded by a polynomial of n.

It is directly concluded that if a problem P is log-depth complete for PSIZE and P is in LOG^kDEPTH for a positive integer k then PSIZE is included in LOG^kDEPTH. However, we have known no evidence to suggest that there is some k such that PSIZE is in LOG^kDEPTH. In other words, P is the hardest problem in PSIZE concerned with the delay complexity.

Here we claim that UDP is one of such problems.

[Theorem 3] UDP is log-depth complete for PSIZE.

Before proving Theorem 3, we prepare three lemmas.

[Lemma 3] For any combinational logic circuit C over a basis {2-AND, 2-NAND} with a single output vertex and for any input vector $(x_1, x_2, \ldots, x_n)$ for C, there is a hypergraph H with rank 2, a subset S of vertices in H and a vertex v_1 such that v_1 is reachable from S if and only if C outputs 1 for the input vector.

Note that the number of vertices and hyperedges (including edges) are bounded by 2size(C) and 4size(C), respectively, in the above construction.

A hypergraph H with rank 2 is said to be synchronous if and only if (1) vertices are partitioned into d levels; (2) for all edges (u,v) and all hyperedges ({u,w},v) if v is in level i then u and w should be in level i-1; (3) each vertex in odd levels has only rank 1 outgoing edges and each vertex in even levels is a source of at most one hyperedge with rank 2; (4) indegree of each

vertex in even levels is just one; and (5) indegree of each vertex in odd levels except the first one is positive. We call vertices in odd levels "AND-nodes", and ones in even levels "OR-nodes".

[Lemma 4] For any combinational logic circuit C in Lemma 3, we can construct a synchronous hypergraph H satisfying the condition of Lemma 3. Moreover the number of vertices and hyperedges (including edges) in H is $O(size(C)^2)$ and all vertices in S are in level 1.

[Lemma 5] For a synchronous hypergraph H, a subset S of vertices in level 1, and an AND-node v_1, we can find a term graph G such that UDP for G is false if and only if v_1 is reachable from S on H.

Note that the number of vertices and edges in G are linearly proportional to the number of vertices and hyperedges in H, respectively.

Now, we return to the proof of Theorem 3.

(Proof of Theorem 3) For any problem P in PSIZE and every positive integer n, there are a polynomial p(n) and a circuit C_n such that C_n computes a subproblem P_n of P and $size(C_n)$ is not greater than p(n). From C_n, we can construct a simple hypergraph H by Lemma 4. From Lemma 5, for a given input vector for C_n, we obtain term graph G such that UDP for G is false if and only if C_n outputs 1 for the input vector. According to the construction of H and G in the above discussion, it is easy to make a circuit transducing P_n to a UDP with constant depth. Indeed it is enough the circuit only generates edges of G in the first level from the input to P_n. As shown in the proof of Lemma 4 and 5, these adding edges in G clearly corresponds to the input for P_n. Thus the depth of the circuit is a constant independent of n. Moreover, it has been already shown in the above consideration that the length of UDP also bounded a polynomial of n. Thus we conclude that UDP is log-depth complete for PSIZE.

Q.E.D.

By the simillal discussion of this section, we can also show that unification is log-space complete for PTIME [6]. Namely, if we have an algorithm on a Turing machine for unification which uses O(log n) cells on tapes, we conclude that all problems that have polynomial time algorithms should be computable in O(log n) space complexity.

[Theorem 4] UDP is log-space complete for PTIME.

The unification problem is related with the reachability problem on directed hypergraph. In algorithm UNIFY, the unification problem is reduced to the reachability problem on a hypergraph. Contrary, some kind of the reachability problem can be reduced to the unification problem. From the consideration on the relation between PSIZE and $LOG^{k}DEPTH$, it seems hard to design efficient algorithms for these two problems.

However, we may be able to implement a hardware which can compute UDP or UP effectively, because many practical terms for unification inherently include parallelism that may compute efficiently. It is important to examine the properties of terms which appear in practical situation for designing a good parallel unification algorithm.

References

[1] Paterson,M.S. and Wegman,M.N., 'Linear Unification', JCSS vol.16, 1978.

[2] Martelli,A. and Montanari,U., 'An efficient unification algorithm', ACM Trans. on Prog. Lang. and Systems, vol.4, no.2, 1982.

[3] Yasuura, H., 'Parallel computational complexity of unification', Tech. Rep. of IECEJ, AL83-30, 1983.

[4] Yasuura, H., 'Studies on complexity theory of logic circuits and hardware algorithms for VLSI systems', Doctorial Dissertation of Kyoto Univ. 1982.

[5] Borodin,A., 'On relating time and space to size and depth', SIAM J.Computing, vol.6, no.4, 1977.

[6] Jones, N.D. and Laaser, W.T., 'Complete problems for deteministic polynomial time', Theoretical Computer Science, vol.3, no.1, 1976.

4. Hashing techniques and unification

This note provides initial reading materials for hashing techniques which would be useful in speeding up unification algorithms.

(1) Building dag using hash cons {1}

Hash cons (abbreviated as hcons) can be used to build a unique (hence sharable) list cell.

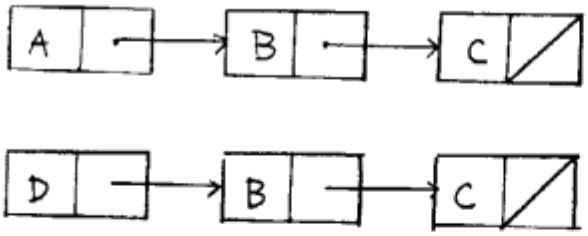
'*consdag*' defined below is a constructor for dag:

$$consdag(l) = \textit{if } atom(l) \textit{ then } l \textit{ else}$$
$$\textit{if } null(cdr(l)) \textit{ then } hcons(car(l), '()) \textit{ else}$$
$$hcons(consdag(car(l)), consdag(cdr(l)))$$

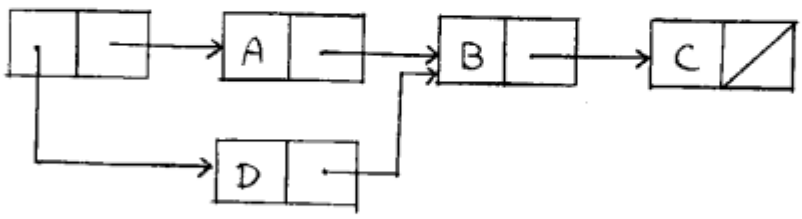
Example:

The result of consdag ('(A (B C)), '(D (B C))) is shown in the following:

Before



After



Note sharing of '(B C) is realized automatically by virtue of hcons.

(2) Equality checking of dags

Dags constructed solely by hcons can be compared by mere equality checking of the values of pointers. When the structures to be constructed are complex structures such as composite of lists and vectors, we need more sophisticated methods.

One candidate hashing method is v-key hashing discussed in {4}.

Annotated References

{1} Goto, E., Ida, T. and T. Gunji, Parallel hash algorithms, Information Processing Letters, (Feb. 1977), 8-13
Hcons is briefly discussed.

{2} Ida, T. and Goto, E. Performance of a parallel hash hardware with key deletion, Proc. IFIP 77 Congress, (Aug. 1977), 643-647
An implementaion scheme for hardware hashing is described.

{3} Ida, T. and Goto, E. Analysis of Parallel Hashing Algorithm with key deletion, JIP, Vol. 1, No.1, (April 1978), 25-32

{4} Ida, T. and Goto, E. Parallel hash algorithms for virtual key index tables, JIP, Vol. 1, No. 3, (Oct. 1978), 130-137
"V-key" hash method and its performance analysis are described.

{5} Ida, T. Hashing hardware and its application to symbol manipulation, Proc. International workshop on high level language computer architecture, Fort Lauderdale, 1980
Pilot model construction of hashing hardware is described.
It also summarises various hashing techniques.

{6} Ida, T. and Itano K. Associative descriptor scheme-for the exploitation of address arithmetic in Lisp, JIP, Vol. 4, No. 3, 1981
Speed-up of list references using cdr-coding and support of cdr-coding by hashing hardware is described.

5. UNIVERSAL UNIFICATION

K. Sakai and T. Matsuda

5.1 HISTORY:

The universal unification problem for a formal language is the problem of determining whether or not for any two terms t1 and t2 of the language there exists a substitution which makes the terms equal. The study of unification was begun by J.R.Guard [Guard 64] and J.A.Robinson [Robinson 65]. In languages without equality, two terms are equal if and only if they are symbolically identical and the unification problem for first order language without equality is decidable. The latter paper includes an algorithm that finds a unique substitution (called the most general unifier) for two formulas of a first order language without equality, together with a complete inference rule (called resolution principle) for mechanizing first order logic.

Existence of the most general unifier is of critical impotance for the proof procedure currently used in automatic theorem proving. Basically it permits us to restrict the rule of substitution to the most general unifier in order to apply the cut rule (i.e. modus ponens). Unfortunately, for theories with equality and for higher order theories there can be pairs of terms with two or more unifiers which are not more general than each other. In this case we must characterize unification problem by the set of maximal general unifiers. Unfortunately again, in some theory there are two terms which do not have the set of maximal general unifiers. So there is an extended problem of universal unification deciding for any two terms of the language the set of maximal general unifiers:

1) does not exist or
2) exists and is infinite or
3) exists and is finite or
4) exists and is a singleton set (i.e. there exists the most general unifier) or
5) exists and is empty (i.e. not unifiable).

For the mechanical theorem proving, an algorithm which generates all maximal general unifiers is quite useful even if there are infinitely many maximal general unifiers. Another problem of universal unification is the problem deciding whether the set of maximal general unifier is recursively enumerable or not.

There are few studies approaching to universal unification universally, but there are many studies about special equational theories which are important to practical use (e.g. for dealing with sets, multisets, strings etc.). The following table contains main results of such studies.

Theory T	Type of T	Unification decidable	Recursively enumerable	Referances
FREE	1	YES	YES	[Robinson 65], [Martelli 79], [Paterson 78]
A	IN	YES	YES	[Plotkin 72], [Siekmann 75], [Livesey 75]
C	FI	YES	YES	[Siekmann 76]
I	FI	YES	YES	[Raulefs 78], [Siekmann 82b], [Szabo 82]
A+C	FI	YES	YES	[Stickel 75], [Livesey 76]
A+I	?	YES	?	[Szabo 82], [Siekmann 82a]
C+I	FI	YES	YES	[Raulefs 78]
A+C+I	FI	YES	YES	[Livesey 76]
D	IN	?	YES	[Szabo 82]
D+A	IN	NO	YES	[Szabo 82]
D+C	IN	?	YES	[Szabo 82]
D+A+C	IN	NO	YES	[Szabo 82]
D+A+I	?	YES	?	[Szabo 82]
H, E	1	YES	YES	[Vogel 78]
H+A	IN	YES	YES	[Vogel 78]
H+A+C	FI	YES	YES	[Vogel 78]
E+A+C	IN	?	?	[Vogel 78]
QG	FI	YES	YES	[Hullot 80]
AG	FI	YES	YES	[Lankford 79]
H10	?	NO	?	[Matiyasevich 70], [Davis 83]
FPA	FI	YES	YES	[Lankford 80]

Abbreviations:

- FREE: without equality
- A: associativity — f(f(x,y),z) = f(x,f(y,z))
- C: commutativity — f(x,y) = f(y,z)
- D: distributivity
  - DR: f(x,g(y,z)) = g(f(x,y),f(x,z))
  - DL: f(g(x,y),z) = g(f(x,z),f(y,z))
- H: homomorphism
- E: endomorphism
- I: idempotence — f(x,x) = x
- QG: quasi-groups
- AG: Abelian-groups
- H10: Hilbert's 10th problem
- FPA: Finitely Presented Algebras

- 1: unitary (i.e. the most general unifier exists for any pair of terms)
- FI: finitary (i.e. the set of maximal general unifiers exists and is finite for any pair of terms)
- IN: infinitary (i.e. the set of maximal general unifiers exists for any pair of terms but can be infinite)

5.2 DEFINITIONS:

We will define formal framework of universal unification from an algebraic point of view, according to [Siekmann 73c].

<Algebras> Let F be a set of function symbols and Fn be the subset of F which consists of all the n-ary function symbols. An F-algebra is a set A where a function A(f): A -> A is defined for any f in Fn. Note that if f is in F0 then A(f) is a constant of A.

<Homomorphisms, Isomorphisms, Congruence> Let A and B be F-algebras.
A function H: A -> B is said to be a homomorphism if
H(A(f)(a1, ..., an)) = B(f)(H(a1), ..., H(an))
for all f in Fn and a1, ..., an in A.

A bijective homomorphism is called an isomorphism.

An equivalent relation R on A is said to be a congruence relation if
A(f)(a1, ..., an) R A(f)(b1, ..., bn)
for all a1, ..., an, b1, ..., bn in A such that a1 R b1, ..., an R bn.

The quotient algebra modulo R is denoted by A/R.

<Free algebras> Let K(F) be the set of all F-algebras. Let FR be an F-algebra and X be a subset of FR. FR is free on X if and only if for any A in K(F) and for any mapping M: X -> A, there exists a unique extention of M to a homomorphism M^: FR -> A.

<Terms> Let X be an arbitrary set. The set T(F,X) of F-terms on X is given by the following recursive definition.

(1) X is a subset T(F,X). Elements of X are called variables.

(2) If f is in Fn and t1, ..., tn are elements of T(F,X), then f(a1, ..., an) is an element of T(F,X).

Clearly T(F,X) is an F-algebra and free on X, therefore it is called the term algebra on X. The term algebra T(F,0) on the empty set is called the initial algebra (or Herbrand universe). The term algebras T(F,X) and T(F,Y) are isomorphic if X and Y have the same cardinality. Therefore let X denote a fixed denumerable set of variables hereafter.

<Substitutions> A mapping S: T(F,X) -> T(F,X) is called a substitution if there is a mapping M: X -> T(F,X) such that M is an identity mapping but for finitely many points and S is the homomorphism M^ generated by M. We can represent a substitution by finite set of pairs:
S = {t1/x1, ..., tn/xn}
where xi is an element of X and ti = M(xi) for all i = 1, ..., n and M(x) = x for all x in X other than xi's.

<Equations> If s and t are in T(F,X), then <s = t> is an equation. The equation <s = t> is valid in A, in symbols A |= s = t, if and only if M^(s) = M^(t) for any M: X -> A.

A set T of equations (called a theory) induces a congruence =T= on T(F,X) defined recursively as follows:

(1) t =T= t
(2) If s =T= t, then t =T= s

(3) If s =T= t and t =T= u, then s =T= u
(4) If S is a substitution and s =T= t, then S(s) =T= S(t)
(5) If si =T= ti for all i = 1, ..., n, then
f(s1, ..., sn) =T= f(t1, ..., tn)

<Unification> Two terms s and t in T(F,X) are said to be unifiable in A if there exists a mapping M: X -> A such that M^(s) = M^(t). The homomorphism M^ is called a unifier in A.

Let T be a theory. Two terms s and t are said to be T-unifiable if there is a substitution U such that U(s) =T= U(t). The substitution U is called a T-unifier.

Let U and V be T-unifiers of s and t. U is said to be more general than V, in symbols V <T< U, if there is a substitution W such that W(U(s)) =T= V(s).

The relation <T< is a pseudo partial order. A T-unifier U is called most general if it is a maximum element in the pseudo order, maximal general if it is a maximal element.

5.3 PERSPECTIVES

As mentioned in [Siekmann 72c], there is a wide variety of areas where universal unification can be applied. In studies of database, information retrieval and formula manipulation, pattern matching or unification technique is quite important.

Especially unification is a fundamental process for free first order theorem provers, since it is embedded as the basic operation in rules such as resolution. In general theories with equality, the absence of the most general unifier prevents us from automated theorem proving. Four approaches to cope with equational axioms have been proposed:

(1) To write the axiom into the database, and use an additional rule of inference, such as paramodulation [Wos 73].
(2) To use special "rewrite rules" [Knuth 70], [Wos 67], [Huet 80a], [Huet 80b]
(3) To design special inference rules incorporating these axioms [Slagle 72].
(4) To develop special unification algorithms incorporating these axioms [Plotkin 72]

From the practical point of view, however, unification algorithms of a specific theory is more important than universal unification algorithms. For example, strings, finite sets and finite_multisets (corresponding to the theories A, A+C+I and A+C in the above table, respectively) are quite familiar data structures in computer science and algorithms which generate the set of maximal general unifiers are of great use. So the last approach (4) appears to be most promising for practical use. However, because it is not universal at all, combination with the approach (2) will be necessary for maintenance of axiom database.

5.4 REFERENCES

[Davis 73] Davis, M.
"Hilbert's tenth problem is unsolvable,"
Amer. Math. Monthly, vol 80, (1973)

[Guard 64] Guard, J.R.
"Automated logic for semi-automated mathematics,"
Scientific report No.1, Air Force cambridge research lbs.,
64-411, AD 602 710, (1964)

[Huet 80a] Huet, G. and Oppen, D.C.
"Equations and Rewrite Rules: a survey," SRI TR CSL-111, 52, (Jan. 1980)

[Huet 80b] Huet, G.
"Confluent reductions: abstract properties and applications to term rewriting systems," J. ACM 27 (1980) 797-821.

[Hullot 79] Hullot, J.M.
"Associative commutative pattern matching,"
7th-IJCAI79, 406-412, (Aug. 1979)

[Knuth 70] Knuth, D.E., Bendix, P.B.,
"Simple word problems in universal algebras," in Computational problems in abstract algebra, J. Leech (ed), Pergamon Press, Oxford, (1970)

[Lankford 79] Lankford, D.S.
"A unification algorithm for abelian group theory,"
Rep. MTP-1, Louisiana Techn. Univ. (1979)

[Lankford 80] Lankford, D.S.
"A new complete FPA-unification algorithm,"
MIT-8, Louisiana Techn. Univ. (1980)

[Livesey 75] Livesey, M. and Siekmann, J.
"Termination and decidability results for stringunification,"
Univ. of Essex, Memo CSM-12, (1975)

[Livesey 76] Livesey, M. and Siekmann, J.
"Unification sets and multisets," Univ. Karlsruhe, Techn. Report, (1976)

[Martelli 79] Martelli, A. and Montaneri, U.
"An efficient unification algorithm,"
University of Pisa, Techn. Report, (1979)

[Matiyasevich 70] Matiyasevich, Y.
"Diophantine representation of rec. Enumerable predicates,"
Proc. of the Scand. Logic Symp., North Holland, (1978)

[Plotkin 72] Plotkin, G.D.
"Building in equational theories," Machine Intelligence, vol 7 (1972)

[Paterson 78] Paterson, M.S. and Wegman, M.N.
"Linear Unification," J. Computer and System Science 16, 158-167, (1978)

[Raulefs 78] Raulefs, P. and Siekmann, J.
"Unification of idempotent functions,"
Universitat Karlsruhe, Techn. Report (1978)

[Robinson 65] Robinson, J.A.
"A machine-oriented logic based on the resolution principle,"
J. ACM 12,(1965) 23-41.

[Robinson 69] Robinson, J.A.

"Mechanizing higher-order logic,"
Machine Intelligence 4, 151-170, (1969).

[Robinson 70] Robinson, J.A.
"A note on mechanizing higher order logic,"
Machine Intelligence 5, 123-133, (1970).

[Siekmann 75] Siekmann, J.
"Stringunification," Essex University, memo CSM-7 (1975)

[Siekmann 76] Siekmann, J.
"Unification of Commutative Terms," Universitat Karlsruhe (1976)

[Siekmann 82a] Siekmann, J. and Szabo, P.
"A Noetherian and Confluent rewrite system for idempotent semigroups,"
Semigroup Forum, vol 25, (1982)

[Siekmann 82b] Siekmann, J. and Szabo, P.
"A minimal unification algorithm for idempotent functions,"
Universitat Karlsruhe (1982)

[Siekmann 82c] Siekmann, J. and Szabo, P.
"Universal unification,"

[Slagle 72] Slagle, J. R.
"ATP with built-in theories including equality, partial ordering and sets," JACM 19, 120-135, (1972)

[Stickel 75] Stickel, M. E.
"A complete unification algorithm for associative-commutative functions," 5th-IJCAI75, 71-76, (1975)

[Szabo 82] Szabo, P.
"Theory of First Order Unification," (in German, thesis)
Universitat Karlsruhe (1982)

[Vogel 78] Vogel, E.
"Unifikation von morphismen," Diplomarbeit, universitat karlsruhe, (1978)

[Wos 67] Wos, L.T., Robinson, G.A., Carson, D.F. and Shalla, L.
"The concept of demodulation in theorem proving,"
JACM, vol 14, No.4, (1967)

[Wos 73] Wos, L.T. and Robinson, G.A.
"Maximal models and refutation completeness: Semidecision procedures in automatic theorem proving," in Word problems (Boone, W.W., Cannonito, F.B., Lyndon, R.C., eds), North Holland, (1973)

6. CONSTRAINT AND UNIFICATION

Kazunori Ueda and Toshiaki Kurokawa

DEFINITION OF CONSTRAINT

Constraint is, in a broad sense, any form of information about the state of program entities, especially the values of variables. The following examples all represent such information:

```
X = 0, X <> 0, 0 <= X <= 100,
X + Y + Z = 180,
prime(Z), X divides Y.
```

These are constraints in that they certainly constrain possible values of variables which might otherwise be unconstrained.

The data type of a variable is included in the constraint because it restricts the possible value of the variable.

EXECUTION OF PROLOG PROGRAMS FROM THE VIEWPOINT OF CONSTRAINT

Prolog programs can be regarded as representing constraints in the form of predicates, and the execution of Prolog programs can be regarded as a transformation process of constraints in the form of predicates to those imposed on variables. For example, if we have the following definitions,

```
append([], X, X).
append([A | X], Y, [A | Z]) :- append(X, Y, Z).
```

the execution of the 'implicit constraint'

```
:- append([3, 5], X, Y).
```

results in the more 'explicit' form of a constraint:

```
Y = [3, 5 | X].
```

The execution of Prolog programs can also be regarded as an accumulation process of information (or knowledge) about the values of variables: the accumulation is done through successive unification operations and the obtained information is borne by resultant unifiers. Note that if the implicit constraint given by the predicate were not satisfiable, the execution would simply fail.

Colmerauer, who is the original developer of Prolog, writes as follows:

Prolog is a language which "computes" on trees "a" represented by variables "x". This computation is done by accumulating constraints that final trees must satisfy.
...
During the execution of a Prolog program, the basic operation consists of verifying whether a constraint is "satisfiable" or not (by at least one tree-assignment). ([Colmerauer 83])

UNIFICATION FROM THE VIEWPOINT OF CONSTRAINT

From the viewpoint of operations on constraints, unification can be viewed in various ways. Suppose a variable (say, X) which may have been bound (i.e. constrained) to some term is to be unified with some other term (i.e. a new constraint).

(1) Constraint checking: One can check by the unification whether the two constraints are compatible or not. They are incompatible if the unification results in failure.

(2) Constraint merging: If the two terms are unifiable, their unification yields a new unifier which adds new constraints to X and its partner terms. This can be viewed as merging of two constraints.

(3) Constraint inheritance: If the variable is unconstrained and gets information unilaterally by unification from its partner, it can be said to inherit the constraint of its partner.

A constraint to a variable can also be regarded as an approximation to its ultimate (i.e. fully specified) value. From the viewpoint of approximation, unification is a process of improving two approximations by getting information from each other, and the failure of unification is the result of overspecification.

PROBLEM WITH THE EXPRESSIVE POWER OF CONSTRAINTS ON VARIABLES

One of the problems with current Prolog systems is that they restrict the kind of constraint on variables to syntactic ones. Suppose natural numbers are represented by the constant 0 and the functor s (corresponding to the successor function). We can express the constraints "X is equal to 1" and "X is not less than 3" by

```
X = s(0) and X = s(s(s(_)))
```

respectively, but cannot express the constraint "X is not equal to 3" or "X is less than 3". To express the last case, the system can at best (1) present three ground solutions

```
0, s(0), s(s(0))
```

successively by backtracking, or (2) collect three solutions using 'set abstraction'.

EXTENDING UNIFICATION TO HANDLE POSSIBILITY SETS

The problem above can be partially solved by allowing a new form of constraint: possibility sets. For example, the constraint "X is less than 3" can be expressed by

```
X = {0, s(0), s(s(0))}.
```

A possibility set is created by exhaustive search for possible values. A constraint expressible by a single term can be regarded as the special case where the possibility set is a singleton.

The result of unification of two possibility sets {A1, A2, ...} and {B1, B2, ...} is the set of all results of successful unification of Ai's and Bj's. For instance, the unification of X above and

```
{s(Y), p(Z)}
```

results in

{s(0), s(s(0))}.

The weakness of this extension is that it allows only finite possibility sets for their generation to halt, though it may still be useful for database applications [Minker 80]. To allow infinite sets, a mechanism for binding variables with unevaluated predicates is necessary. However, there are other troublesome problems with unevaluated predicates, as will be explained below.

BIND-HOOK AND CONSTRAINT

Bind-hook facilities in KLO [Chikayama 82] and 'geler' predicate in PROLOG II [Caneghem 82] can be used for imposing non-syntactic constraints to variables. For example, the call of

bind-hook(X, X<=6)

causes the 'hook' X<=6 to be registered for future invocation; it is invoked as soon as X is instantiated to some non-variable term in some unification (or immediately after the registration if X already has a value), and prevents X from getting an inappropriate value. If X is unified with some other variable, the check is inherited. X may have additional hooks if another bind-hook is called before X is instantiated: these hooks form a conjunctive predicate call.

However, if X remains uninstantiated after all, the check is never invoked, i.e., the constraint only disappears. This is undesirable from the logical point of view. Another problem with bind-hook might be that what it represents is 'deferred' constraints. As for normal (syntactic) constraints, the unification of two incompatible constraints results in immediate failure. On the other hand, the unification of two incompatible hooks always succeeds and the doom is postponed possibly forever. Put in other words, unification with bind-hook can no longer be regarded as a simple constraint checker/merger.

CONSTRAINT IMPLEMENTED BY UNIFICATION

It is interesting that the constraint mechanism can be implemented in DEC-10 Prolog using the unification.

Kurokawa implemented the constraint as follows:

a) If the variable is to be unified with a tree, do it as usual.

b) If the variable is not to be unified but to be constrained, make a constraint list. The variable is registered as a constrained variable.

c) When the constraint variable is to be unified, the constraint list is searched. If the unification contradicts the constraint, then this unification must fail. Otherwise, the unification succeeds.

d) At the end of computation, the content of the constraint list is edited so that the information is concise and readable.

Example.

Let's take a very simple example, the problem of taking a maximum value of the two numbers. A sample Prolog program is shown below:

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X =< Y.
```

When both arguments are instantiated as integers, the program works. However, if the one argument is not instantiated, (X in this case), then a constraint will be given. Note that there are two cases, and they are displayed through backtracking.

```
?-ex(max(X,4,Y)).

----------constraint----------------------------
[integer(_1),_1 >=4|_435]
-------------value--------------------------
X=_1, Y=_1
continue?
|: yes.

----------constraint----------------------------
[integer(_1),4>= _1|_435]
-------------value--------------------------
X=_1, Y=4
continue?
|: yes.

FAIL
X = _31,
Y = _55

yes
```

In this implementation, the constraint list is realized as a D-list. The remaining problem is that the D-list will grow while the computation proceeds, and there is no way to reduce the list.

The bind-hook of KL will help the step c, however, there is the problem mentioned before. At the step b, the check must be given to see if the newly-added constraint may lead to the contradiction, i.e. fail.

EXTENDING UNIFICATION TO NON-TERM DATA TYPE--A NUMERIC TYPE EXAMPLE

For pre-defined numeric types in current Prolog systems, the values of variables should either be exactly determined or be totally undefined. However, we can extend it to represent "ranges of possible values". The unification of two ranges can then be defined as an intersection operation (if the result is an empty range, then it simply fails). For example, the unification of the two constraints

3 <= X and 2 <= X <= 5

yields the new constraint

3 <= X <= 5.

Recall that a set of all Prolog terms can be divided into isomorphism classes by taking renaming relation as congruence, that the resultant quotient set forms a lattice (using instantiation-generalization relation as partial ordering) if supplied with the element 'top', and that the unification operation is the take-least-upper-bound operation (with the notion of failure corresponding to the result 'top'). Likewise, a set of

all intervals forms a lattice by using interval inclusion as partial ordering, and the unification operation defined above is also the take-least-upper-bound operation. Thus, the extension above is natural.

EQUATION SOLVING, UNIFICATION, AND CONSTRAINT

Looking at the examples listed at the top of this chapter, the constraint seems to be the result of the equation/inequation. In some case, the solution would come through unification. For example, X = 1 + Y is a simple unification.
But as for the case of X + 1 = Y + 2, it cannot be handled by the simple unification. If the X and Y are known to be numbers, the formula will be simplified as X = Y + 1, and the unification will do. When the inequality is introduced, for example X > Z - 1, this is just this type of extending unification, i.e. the constraint.

The above discussion is also an example of higher-order unification where associativity, commutativity, etc. are known for the numerical variables. Again, we are using the constraint that the variable is type numeric.

LINKING REDUCTION AND UNIFICATION

Reduction is one-way computation, on the other hand, unification is two-way computation. However, both reduction and unification have a beautiful property that is more desirable than usual assignments.

Also, the user wants both reduction and unification for his various purposes. The problem is that how you can link the both mechanism.

As Colmerauer points out:
This (verifying the constraint) is done by "reducing" it,
...,
the purpose of "reducing" is to simplify the constraint in order to make all its solutions explicit. ([Colmerauer 83])

So, the constraint is a plausible point of linking reduction and unification. However, it will take more examination.

TOWARDS THE UNIVERSAL UNIFICATION

There are several attempts to extend unification incorporating equality. The attempts are, in a sense, categorized into the constraint works explained here.

We will leave the attempts not explained here, but referring some papers handling the subjects. They are as follow:

[Kornfeld 83] - inheritance mechanism
[Kahn 81&82] - knowledge representation

REFERENCES

[Caneghem 82]
Caneghem, M. van: PROLOG II Manuel D'Utilisation, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseille (1982).

[Chikayama 83]

Chikayama, T., Hattori, T., Yokota, M.: A Draft Proposal of Fifth Generation Kernel Language Version 0.1, Technical Memo TM-007, Institute for New Generation Computer Technology (1982).

[Colmerauer 83]
Colmerauer, A. "PROLOG IN 10 FIGURES", 8th IJCAI'83, 487-499, (Aug. 1983)

[Kahn 81]
Kahn, K. M., "Uniform -- A Language based upon Unification which unifies (much of) Lisp, Prolog , and Act 1", UPMAIL, 22, (Feb. 1981). also in 7th IJCAI'81

[Kahn 82]
Kahn, K. M.,"The implementation of Uniform - A Knowledge-Representation / Programming Language based upon Equivalence of Description", UPMAIL TR-9, (Mar. 1982)

[Kornfeld 83]
Kornfeld, W.A. "equality for Prolog", MIT AI Lab., (1983)

[Minker 80]
Minker, J.: A Set-Oriented Predicate Logic Programming Language, TR-922, Dept. of computer science, Univ. of Maryland (1980).

7. Lazy Unification and Bind Hook

Masami Hagiya

7.1 Bind Hook

Bind-hook was devised by T. Chikayama as a data-driven computation mechanism of KLO (Fifth Generation Kernel Lauguage Version 0 [1]). When bind-hook(X, H) is executed, and if X is (bound to) a variable at that time, then H is added to the chain of bind-hook handlers for X, and the execution of bind-hook(X, H) immediately succeeds. The bind-hook handlers for X are actually called, when the variable X is bound to some non-variable term. If two variables are unified, their bind-hook handlers are just merged to make a new chain for the unified variable.

From the declarative meaning of the program, bind-hook(X, H) can be interpreted just as H (or call(H)), and in fact they are logically equivalent provided that X will eventually be bound to a non-variable term. But, when the caller of bind-hook succeeds before binding the variable X, its bind-hook handlers just vanish with having never been called. This situation is similar to that of the normal order reduction in a term-rewriting system, where a term, which possibly has no normal form, vanishes in the cource of a normal order reduction. It is, in another word, the problem of how to treat a term that may have no value or may be undefined.

7.2. Lazy Unification

Lazy unification was proposed by M. Hagiya in [2] for formalizing the semantics of the concurrent Prolog that processes infinite terms (typically infinite streams). By analyzing the greatest fixed-point semantics on the universe of infinite terms, he concluded that the process of unification should be broken into pieces, each of which is processed as a separate process.

Here, we consider an implementation of lazy unification by bind-hook.

```
lu(X, Y) :- var(X), var(Y), !, X = Y.
lu(X, Y) :- var(X), !, bind-hook(X, lu(X, Y)).
lu(X, Y) :- var(Y), !, bind-hook(Y, lu(X, Y)).
lu(X, Y) :- !, X =.. [F|Xs], Y =.. [F|Ys], lu_list(Xs, Ys).

lu_list([], []) :- !.
lu_list([X|Xs], [Y|Ys]) :- !, lu(X, Y), lu_list(Xs, Ys).
```

If X and Y are both variables, then they are just unified with the built-in unifier of Prolog. If X is a variable and Y is a non-variable term, then X is bind-hooked with lu(X, Y); i.e. the lazy unfication of X and Y resumes after X is bound to some non-variable term. If X and Y are both non-variable terms, their functors are first checked and their arguments are lazy-unified with each other.

When

```
?- lu(X, [0|X]), producer(X).
```

is executed, lu(X, [0|X]) works as a consumer that checks if the term

X producer(X) produces is an infinite stream of the form [0, 0, ...].

Since bind-hook is a passive mechanism, lu cannot serve as a producer. Thus

```
?- lu(X, [0|X]), lu(Y, [0|Y]), X = Y.
```

just suspends, since there is no producer.

7.3. Bind Hook Hook

To make a productive lazy unifier, we need the reverse of bind-hook, i.e. bind-hook-hook; bind-hook-hook(X, H) will enter H to the chain of the bind-hook-hook handlers of the variable X. The bind-hook-hook handlers will be called when X is bind-hooked with some handler, or X is unified with a bind-hooked variable. The productive lazy unifier, plu, will be defined by replacing lu by plu and bind-hook by bind-hook-hook in the definition of lu.

If we execute

```
?- lu(X, [0|X]), plu(Y, [0|Y]), X = Y.
```

lu(X, [0|X]) and plu(Y, [0|Y]) work will as the consumer and the producer of the infinite stream [0, 0, ...], after the execution of X = Y.

Anyway the usefulness of lazy unification and bind-hook-hook is of great doubt, even if interesting.

References

[1] T. Chikayama, M. Yokota, T. Hattori, Fifth generation kernel language version 0, Proceedings of the Logic Programming Conference '83, Tokyo, 1983.

[2] M. Hagiya, On lazy unification and infinite trees, Proceedings of the Logic Programming Conference '83, Tokyo, 1983 (in Japanese).

@

8. FORMULA MANIULATION AND UNIFICATION

Morio Nagata

A FORMULA MANIPULATION SYSTEM AS THE KNOWLEDGE BASE

Algebraic formulas are useful for representing knowledge or facts in a formal manner. Moreover, much knowledge has been represented in mathematical formulas. Thus, a formula manipulation system can be considered as the knowledge base.

If both logical and algebraic formulas were manipulated by a computer, users would have a flexible system for their problem solving. On the other hand, there are common techniques in the implementation of logical and algebraic formula manipulation systems. Therefore, we inspect such techniques and some problems of the manipulating both logical and algebraic formulas. Especially, we shall examine problems on unification and formula manipulation because it is a core problem of such a system.

PROBLEMS

The problems on unification and formula manipulation are classified into problems of several levels. In the most general level, the problem corresponds to one on unification and reduction. Rules of formula manipulation are considered as reduction rules in this level.

However, if we inspect the relationship of formula manipulation and unification in detail, we can find some problems peculiar to several types of formulas. These types are divided into polynomials, rational functions, transcendental functions and so on. For example, a polynomial

$$(x+1)(x-1)$$

equals to $x^2 - 1$, therefore

$$P((x+1)(x-1))$$

must match

$$P(x^2 - 1).$$

When we take the position that the Prolog system uses knowledge on algebraic formulas, some functions of formula manipulation is useful. For instance, if the Prolog system knows that

$$x = x+1$$

is always false, then this fact can be used in "occur check". In addition to such a fact, if the system have common sense on elementary mathematics, represented by algebraic formulas, we can develop its ability of problem solving. For example, if it knows that a quadratic equation representing a motion of the ball thrown by a man

$$x^2 + x + 3 = 0$$

has no real roots, then some propositions including 'x' may be false or undefined in the situation.

Moreover, we should consider certain types of manipulation of formulas [Bundy 81]. Bundy and Welham proposed meta-level inference and object-level inference. In their system called PRESS (PRolog Equation Solving System), algebraic expressions are manipulated by a series of methods. The appropriate method is chosen by meta-level inference and itself uses meta-level reasoning to select

and aply rewrite rules to the current expression. One of rules of meta-level inference of PRESS is represented as follows:

> If an equation L=R contains precisely one occurence of X located at position P in L and if the result of isolating X in L=R is Ans, then Ans is a solution of L=R with respect to X.

An example of rules of object-level inference, isolation, consists of 'stripping off' the functions surrounding the single occurences of the variable by applying the inverse function to both sides of the equation. Many techniques of formula manipulation can be used in the object-level inference, but methods for the meta-level inference should be studied for our purpose.

METHODS

We have already research reports on methods for formula manipulation. Some of them are useful hints for the study of unification and formula manipulation. For instance, Moses examined algebraic simplification from points of views of a user and a designer of an algebraic manipulation system [Moses 71], He classified such systems into radical, conservative, liberal, new left an Catholic ones. For example, radical systems can handle a single, well-defined class of expressions. Thus

$$3x^2y^2 - 2x^2yz + 4x^3y^4 + z$$

would be represented as

$$(3y^2 - (2z^2)y)x^2 + (4)x^3 + (y^4 + z).$$

This type of the system is suited for unification.

An easy answer to the problem of unification and formula manipulation is to incorporate Prolog with a radical formula manipulation system.
We can easily represent much, mathematical common sense in such a system.
However, there is widespread disagreement with the radical approach usually concern expressions which contain powers of sums, e.g. $(x+1)^{1000}$. Moreover, we sometimes want the partial fraction decomposition, thus we leave the expression as it stands.

If we use non-radical systems, we must pay attention to strategies for combining unification with formula manipulation. Especially, controlling unification of logical expressions and manipulation of algebraic formulas is a core problem. The method for controlling two kinds of processing should be developed.

A practical answer to our problem is that our system have a radical subsystem like the MACSYMA system [Martin 71]. That is, in the subsystem all formulas are translated into a single class of expressions. Thus, unification or occur check is easily performed. In a non-radical subsystem, we must discover rules of meta-level inference on formula manipulation and unification. An example of such rules is the following:

> If two expressions to be unified have mathematical formulas as sub expressions, then try to simplify those expressions by factoring, computing GCD, and so on.

Using all facilities of existing formula manipulation systems, we can develop similar rules.

CONCLUSION

We have inspected the problems on unification and formula manipulation from point of view of problem solving. In order to represent knowledge or to solve problems by using a computer, we should construct a system which accepts logical and algebraic expressions.

Finally, some ideas of this report will be useful for non-resolution theorem provers [Bibel 83, Bledsoe 77].

REFERENCES

[Bibel 83] Bibel, W: Mating in Matrices, CACM, Vol. 26, No.11, 1983, pp. 844-852

[Bledsoe 77] Bledsoe, W.W.: Non-resolution Theorem Proving, Artificial Intelligence, Vol.9, 1977, pp. 1-35

[Bundy 81] a Bundy, A, and Welham, B.: Using Meta-level Inference for Selective Application of Multiple Rewrite Rule Sets in Algebraic Manipulation, Artificial Intelligence, Vol. 18, 1981, pp. 189-212

[Martin 71] Martin, W.A. and Fateman, R.J.: The MACSYMA System, Proc. 2nd Symposium on Symbolic Manipulation, 1971, pp. 59-75

[Moses 71] Moses, J.: Algebraic Simplification: A Guide for the Perplexed, Proc. 2nd Symposium on Symbolic Manipulation, 1971, pp. 282-304

9. Unification in Categories

Takanori Adachi

In this section we will look at unification and a computation model based on it from a category theoretical point of view.
First, in 9.1 we generalize the notion of universal unification explained in section 5 by Sakai and Matsuda. In 9.2 we introduce the notion of kites which is corresponding to that of clauses in Logic Programming, and give a categorical computation model that is to be efficient using unification algorithms, in a sense. In 9.3 we consider the category determined by a program which is called an information system.

9.1. Unification from a Categorical Point of View

The notion of unification was introduced by J. A. Robinson in the context of formulating first-order logic which is designed for use as the basic theoretical instrument of a computer theorem proving program [Rob65], and this was generalized to problems in several algebras [Plo72, SS82]. Using the terminology of algebraic theories in the sense of Lawvere [Law63], the definition of a unification problem is restated as follows:

9.1.1. Definition. Let A be an algebraic theory, and $f, g : a \to b$ be a parallel pair of arrows in A.

(i) $U_A(f, g) = \{ h \mid fh = gh \}$ (subscript may be omitted).

(ii) An element of $U_A(f, g)$ is called a <u>unifier</u> for $\langle f, g \rangle$.

(iii) A pair $\langle f, g \rangle$ is called <u>unifiable</u> when it has a unifier.

(iv) A <u>unification problem</u> for A is to decide whether or not $\langle f, g \rangle$ is unifiable.

9.1.2. Definition. Let f, g be a parallel pair of arrows.

(i) For $h, h' \in U(f, g)$, we write $h < h'$ when there is an arrow i such that $h = h'i$.

(ii) The subset $E \subset U(f, g)$ is called <u>complete</u> if for every $h \in U(f, g)$ there exists $e \in E$ such that $h < e$.

(iii) $CU_A(f, g) = \{ E \mid E \text{ is a complete subset of } U(f, g) \}$.

(iv) $MCU_A(f, g) = \{ E \mid E \in CU(f, g)$ is a minimal element with respect to set-inclusion ordering $\}$.

Note that $U(f, g)$ is itself a complete subset of $U(f, g)$.

9.1.3. Fact. For $E \in MCU(f, g)$ and $e, e' \in E$,
$e < e'$ implies $e = e'$.

Proof. Suppose that $e \neq e'$. Define E' by $E' = E - \{ e \}$. Then we can easily check that E' is complete. But this contradicts the minimality of E. Hence $e = e'$.

9.1.4. Proposition. For E and E' in $MCU(f, g)$, $\#E = \#E'$, where $\#E$ is the cardinality of E.

Proof. Since E and E' are both complete,
$\forall e \in E\ \exists e' \in E' .\ e < e'$ and $\forall e' \in E'\ \exists e \in E .\ e' < e$.
Hence there are two functions $\varphi : E \to E'$ and $\psi : E' \to E$ such that $\forall e \in E .\ e < \varphi(e)$ and $\forall e' \in E' .\ e' < \psi(e')$.
Then we have $e < (\psi.\varphi)(e)$ for every e in E. Thus, $\psi.\varphi = 1_E$.
Similarly we have $\varphi.\psi = 1_{E'}$.
Therefore φ is bijective.

This proposition asserts that we may classify algebraic theories by the cardinality of some E in $MCU(f, g)$. In fact, Siekmann and Szabo investigated this classification in [SS82].

Note that for an element E in $MCU(f, g)$, if $E = \{ e \}$, then e corresponds to a usual most general unifier (mgu), and moreover categorically e is a weak equalizer in the sense of MacLane [Mac71].

The above discussion does not depend on any structure of algebraic theories. So we can easily give a generalization of the above discussions in the case that two arrows to be unified need not have same domain.

9.1.5. Definition. Let X be a category, and f and g two arrows in X with $cod(f) = cod(g)$.

(i) $U_X(f, g) = \{ \langle i, j \rangle \mid fi = gj \}$.

(ii) A pair $\langle i, j \rangle$ in $U_X(f, g)$ is callled a unifier for $\langle f, g \rangle$.

(iii) A pair $\langle f, g \rangle$ is called unifiable when it has a unifier.

(iv) A unification problem for X is to decide whether or not $\langle f, g \rangle$ is unifiable.

9.1.6. Definition. Let f and g be arrows in X with the same codomain.
For $\langle i, j\rangle$ and $\langle i', j'\rangle \in U(f, g)$, we write $\langle i, j\rangle < \langle i', j'\rangle$ when there is an arrow h such that $i = i'h$ and $j = j'h$.

The notion of complete subsets of $U_X(f, g)$ and the sets $CU_X(f, g)$ and $MCU_X(f, g)$ are similarly defined in the case of algebraic theories. Moreover we can have the same result as Proposition 9.1.4.

9.1.7. Proposition. Let X be a category. Then $E, E' \in MCU_X(f, g)$ implies $\#E = \#E'$.

Proof. Omitted.

Note that for an element of MCU(f, g), if $E = \{ \langle i, j\rangle \}$, $\langle i, j\rangle$ is a weak pullback of $\langle f, g\rangle$ in the sense of MacLane [Mac71].

9.1.8. Remark. Let X be a category with products, $f : x \to z$ and $g : y \to z$ be arrows in X. Define $f' = fp$ and $g' = gq$, where $p : x \times y \to x$ and $q : x \times y \to y$ are projections. Then there is a bijection $\varphi : U_X(f, g) \to \{ e \mid f'e = g'e \}$.
Thus, the notion of unifiers in categories is, in fact, a generalization of that of algeraic theories.

9.1.9. Definition. A unification algorithm for a category X is an algorithm A which takes two arrows f and g with same codomain as input and generates a complete subset $A(f, g)$ of $U_X(f, g)$.

9.2. Kites over Categories

The theses of this subsection are in categories that arrows are information and that compositions are instantiations.

9.2.1. Definition. Let X be a category.
(i) For an object x in X, $[X;x] = \{ u \mid u$ is a finite subset of arrows in X such that for all f in u $dom(f) = x \}$.
(ii) $[X] = \bigcup_{x \in X} [X;x]$.
(iii) A kite of type x is a pair $\langle u, v\rangle$ where u and v are elements of $[X;x]$ for some x in X.
(iv) A Horn kite of type x is a kite $\langle u, v\rangle$ of type x where $\#v = 1$.

9.2.2. Definition. Let X be a category and x an object of X.

(i) For $u \in [X;x]$ and an arrow g with $cod(g) = x$,
$ug = \{ fg \mid f \in u \}$.

(ii) For a kite $k = \langle u, v \rangle$ of type x and an arrow g with $cod(g) = x$, $kg = \langle ug, vg \rangle$, called an instance of k. We sometimes write $k < l$ when k is an instance of l.

(iii) For kites $k = \langle u, v \rangle$ and $k' = \langle u', v' \rangle$ of type x, $k.k'$ is the kite of type x defined by $k.k' = \langle (u - v') \cup u', v \rangle$, called a resolvent of k and k'.

(iv) For kites $k = \langle u, v \rangle$ and $k' = \langle u', v' \rangle$ of same type, we write $k \subset k'$ when $u \subset u'$ and $v = v'$.

(v) For kites $k = \langle u, v \rangle$ and $k' = \langle u, v' \rangle$,
$k \cup k' = \langle u, v \cup v' \rangle$.

Note that the set of all Horn kites is closed under the operations of creating instances and resolvents.

9.2.3. Fact. Let k, l and m be three kites of type x.

(i) $k \subset l$ implies $k.l = l$ [especially $k.k = k$].

(ii) $(k.l).m \subset k.(l.m)$.

(iii) For an arrow f with $cod(f) = x$, $kf.lf \subset (k.l)f$.

Proof. Straightforward.

9.2.4. Definition. Let K and L be sets of kites.

(i) $\downarrow K = \{ k' \mid k' \text{ is an instance of some } k \text{ in } K \}$.

(ii) $K.L = \{ k.l \mid k \in K \text{ and } l \in L \text{ have same type} \}$.

9.2.5. Definition. Let K be a set of kites.

(i) K^* is the smallest set of kites satisfying:
(1) $K \subset K^*$,
(2) $\langle u, v \rangle \in K^*$ whenever $u \supset v$,
(3) $k \cup l \in K^*$
whenever $k = \langle u, v \rangle$ and $l = \langle u, v' \rangle \in K^*$,
(4) $\downarrow K^* \subset K^*$,
(5) $K^*.K^* \subset K^*$.

(ii) $K\$ = \{ u \mid \langle \emptyset, u \rangle \in K \}$.

(iii) $K\$u = \{ v \in K\$ \mid v \text{ is an instance of } u \}$.

Intuitively, a kite is a clause in the sense of Logic Programming, and K is a program of computation. Hence $K^*\$u$ is the output obtained by the execution of K with an input u.

9.2.6. Definition. Let K be a set of kites. Then K^n (n = 0, 1, 2...) is inductively defined by $K^0 = I$ and $K^{n+1} = K^n.\downarrow K$, where $I = \{ \langle u, u\rangle \mid u \in [X] \}$.

9.2.7. Lemma. For every kite k in K^*, there exists l in $\bigcup_n K^n$ such that $l \subset k$.

Proof. By Fact 9.2.3 (ii) and (iii).

9.2.8. Proposition. $K^*\$ = \bigcup_n (K^n\$)$.

Proof. By Lemma 9.2.7.

9.2.9. Definition. Let K be a set of kites and $u \in [X]$.

(i) The set Comp(n, K, u) (n = 0, 1, 2...) [we may omit K and u] is inductively defined by
$Comp(0) = \downarrow\{ \langle u, u\rangle \}$ and $Comp(n + 1) = Comp(n, K, u).\downarrow K$.

(ii) $Comp(K, u) = \bigcup_n Comp(n, K, u)$.

9.2.10. Theorem. $K^*\$u = \bigcup_n (Comp(n, K, u)\$)$.

Proof. It is immediate by Proposition 9.2.8 because for every n = 0, 1, 2..., $Comp(n, K, u)\$ = K^n\$u$.

In the rest of this section we assume that the program K is a set of Horn kites. But this restriction raises no loss of generality. Because for an arbitrary set of kites, say L, we can define the program K which is equivalent to L (in other words $K^* = L$) by
$K = \{ \langle u, \{ f \}\rangle \mid$ there is v such that $\langle u, v\rangle \in L$ and $f \in v \}$.
We also assume that the category X has a unification algorithm A.

9.2.11. Definition. Let u be an element of [X].

(i) The set Mcomp(n, K, u) (n = 0, 1, 2...) [we may omit K and u] is inductively defined by $Mcomp(0) = \{ \langle u, u\rangle \}$ and
$Mcomp(n + 1) = \{ ki.lj \mid k = \langle u, v\rangle \in Mcomp(n)$ and
$l = \langle w, \{ g \}\rangle \in K$ and
there is $f \in u$ such that $\langle i, j\rangle \in A(f, g) \}$.

(ii) $Mcomp(K, u) = \bigcup_n Mcomp(n, K, u)$.

We want to show that Mcomp is a somewhat efficient universal machine, in a sense. In order to show that, we introduce a special preordering between kites.

9.2.12. Definition. Let k and l be two kites. Then we write $k \prec l$ when there exists a kite m such that $k \supset m < l$.

9.2.13. Fact. Let k, k', l and l' be kites.

(i) $k' \supset k \prec l < l'$ implies $k' \prec l'$.

(ii) $k \prec l$ whenever there is a kite m such that $k < m \supset l$.

(iii) The relation $\prec$ is a preorder.

Proof. (i) Immediate since $<$ is transitive.

(ii) Put $k = mf$. Then $k = mf \supset lf < l$. Hence $k \prec l$.

(iii) Suppose that $k \prec l \prec m$. Then there are kites k' and l' such that $k \supset k' < l \supset l' < m$. Thus, by (ii) $k \supset k' \prec l' < m$. Hence by (i) $k \prec m$. Therefore the relation $\prec$ is transitive. The reflexivity of $\prec$ is trivial.

9.2.14. Lemma. For every $k \in \mathrm{Comp}(K, u)$ there exists $l \in \mathrm{Mcomp}(K, u)$ such that $k \prec l$.

Proof. Let k be a kite in $\mathrm{Comp}(n, K, u)$. And we prove by induction on n.

$n = 0$: $k \in \mathrm{Comp}(0) = \downarrow\{ \langle u, u \rangle \}$. Then $k < \langle u, u \rangle \in \mathrm{Mcomp}(0)$. Hence $k \prec \langle u, u \rangle$.

$n > 0$: Let $k \in \mathrm{Comp}(n) = \mathrm{Comp}(n-1).\downarrow K$. Then there exist k_1 in $\mathrm{Comp}(n-1)$ and k_2 in $\downarrow K$ such that $k = k_1.k_2$.

By hypothesis, there is $m_1 \in \mathrm{Mcomp}(K, u)$ such that $k_1 \prec m_1$.

Hence there is f_1 such that $k_1 \supset m_1 f_1$.

On the other hand, there are m_2 in K and an arrow f_2 such that $k_2 = m_2 f_2$. Let $m_1 = \langle u_1, q \rangle$ and $m_2 = \langle u_2, \{ g \} \rangle$.

Then there is $v \supset u_1 f_1$ with $k_1 = \langle v, qf_1 \rangle$.

Thus, $k = k_1.k_2 = \langle (v - \{ gf_2 \}) \cup u_2 f_2, qf_1 \rangle$.

Let $w = \{ h \in u_1 \mid hf_1 = gf_2 \}$.

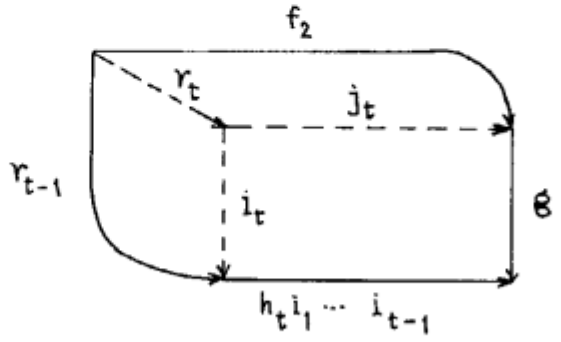
Then since u_1 is finite, so is w.

Let $w = \{ h_1, h_2, \ldots, h_s \}$ $(s \geq 0)$.

Now we define the arrows r_t $(t = 0,\ldots,s)$; i_t, j_t $(t = 1,\ldots,s)$; sets v_t $(t = 0,\ldots,s)$ and kites c_t $(t = 0,\ldots,s)$ by the following program:

```
begin
    c_0 := m_1;
    if s ≠ 0 then
        begin
            r_0 := f_1;
```

```
        v_0 := u_1;
        for t = 1 to s
            if h_t i_1 ... i_{t-1} ∈ v_{t-1} then
                begin
                    let i_t, j_t, r_t be those such that
                        the diagram
```




```
                        commutes
                        and
                        <i_t, j_t> ∈ A(h_t i_1 ... i_{t-1}, g);
                    v_t := (v_{t-1} i_t - { g j_t }) ∪ u_2 j_t;
                    c_t := c_{t-1} i_t . m_2 j_t;
                end
            else
                begin
                    i_t := i_{t-1};
                    j_t := j_{t-1};
                    r_t := r_{t-1};
                    v_t := v_{t-1};
                    c_t := c_{t-1};
                end
        end
end
```

Note that the outer square of the diagram in this program commutes because $i_1 \cdots i_{t-1} r_{t-1} = f_1$. And the existence of arrows i_t, j_t and r_t are assured since the unification algorithm A generates a complete set.

Now there is a relation between c_t and v_t such that $c_t = \langle v_t, qi_1 i_2 \cdots i_t \rangle$.

Thus, we can easily check that each c_t is in Mcomp(K, u).

Using the above data, we can show that

$v_s r_s = (u_1 f_1 - \{ gf_2 \}) \cup u_2 f_2$.
Hence $k \supset c_s r_s$. Therefore $k \prec c_s \in \mathrm{Mcomp}(K, u)$.

9.2.15. Theorem. $K^{*}\$u = \downarrow(\mathrm{Mcomp}(K, u)\$)$.

Proof. By Theorem 9.2.10 and Lemma 9.2.14.

9.3. Information Systems over Categories

Let X be a fixed category throughout this subsection. We have already defined the set $[X;x]$ in the previous subsection. This set is obviously a poset with the ordinary set-inclusion relation. Moreover this is closed under union. Hence the dual category of $[X;x]^{OP}$ has finite products.

9.3.1. Definition. Let $f : x \to y$ be an arrow in X.

(i) $[X;f]$ is a function from $[X;y]$ to $[X;x]$ defined by $[X;f](u) = uf$, for every $u \in [X;y]$.

This function is clearly monotonic, and hence a functor.

(ii) $[X;-] : X^{OP} \to \mathrm{Cat}$ is a naturally defined full functor, where Cat is the category of all small categories.

9.3.2. Definition. An <u>information system</u> over X is a pair $\langle Y, \epsilon \rangle$, where $Y : X^{OP} \to \mathrm{Cat}$ is a functor such that for every object x in X Yx is a category with finite products and $\epsilon : [X;-] \dot{\to} Y$ is a natural transformation such that for every x in X, $\epsilon_x : [X;x] \to Yx$ is a product-preserving functor that is bijective on the object class.

Now since ϵ_x is bijective on the objects, we denote the corresponding objects in Yx and in $[X;x]$ with the same signs, and also denote the product object of u and v in Yx by $u \cup v$.

9.3.3. Fact. Let $I = \langle Y, \epsilon \rangle$ be an information system over X. Then for every x in X and u, v in Yx,
$Yx(u, v) \neq \emptyset$ and $Yx(u', v') \neq \emptyset$ implies
$Yx((u - v') \cup u', v) \neq \emptyset$.

Proof. For $k : u \to v$ and $m : u' \to v'$ we define
$k.m = k.p_{v' \times u, u}.(1_{u-v'} \times m)$. Then $k.m : (u - v') \cup u' \to v$.

9.3.4. Theorem. Let K be a set of kites over X, and $f : x \to y$ an arrow in X. Define Yx as a poset whose objects are those of $[X;x]$ and ordering is defined by $u \leq v$ iff $\langle u, v \rangle \in K^{*}$, and Yf

as an arrow from Yy to Yx such that Yf(u) = uf. Then Y is a functor from X^{op} to Cat. Moreover define $\varepsilon_x : [X ; x] \to Yx$ the natural embedding since [X;x] is a subposet of Yx.
Then the pair $\langle Y, \varepsilon \rangle$ is an information system over X, denoted by I(K).

Proof. Straightforward.

In the beginning of the previous subsection we pointed out that we would consider arrows as information. According to this thesis, we can think of elements of Yx as information - property. And arrows in Yx may be considered to be proofs which indicate derivations from one property to another.

Essential meaning of the last theorem is that every program determines a categorical structure. So we conclude this section with stating the final thesis:
"Programs are Categories".

9.4. References

[Law63] Lawvere, F. W., Functorial semantics of algebraic theories, Proc. of the Nat. Acad. of Sci. of the USA 50 (1963) 869-872.

[Mac71] MacLane, S., Categories for the Working Mathematician, (Springer, 1971).

[Plo72] Plotkin, G. D., Building-in equational theories, Machine Intell. 7 (1972) 73-90.

[Rob65] Robinson, J. A., A machine-oriented logic based on the resolution principle, J. ACM 12 (1965) 23-41.

[Sco82] Scott, D. S., Domains for denotational semantics, LNCS 140 (1982) 577-613.

[SS82] Siekmann, J. and Szabo, P., Universal unification and a classification of equational theories, LNCS 138 (1982) 369-389.

LIST OF CONTRIBUTORS

Takanori	Adachi	Tokyo Institute of Technology
Masami	Hagiya	RIMS, Kyoto University
Tetsuo	Ida	Institute of Chemical and Physical Research
Takumi	Kasai	The University of Electro-Communications
Toshiaki	Kurokawa	3rd laboratory, ICOT
Toshio	Matsuda	Science University of Tokyo
Kuniaki	Mukai	2nd laboratory, ICOT
Ko	Sakai	3rd laboratory, ICOT
Kazunori	Ueda	C&C Systems Res. Lab., NEC
Hiroto	Yasuura	Kyoto University