Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern

SEKI - REPORT

Compartimentalized Connection Graphs for Concurrent Logic Porgramming II: Parralelism, Indexing and Unification

David M. W. Powers

SEKI Report SR-90-17

COMPARTMENTALIZED CONNECTION GRAPHS FOR CONCURRENT LOGIC PROGRAMMING

II: Parallelism, Indexing and Unification

David M. W. Powers[1]
Universität Kaiserslautern
D-6750 KAISERSLAUTERN
WEST GERMANY

SEKI Report SR-90-17

Abstract

This report continues to document the development of a logic programming paradigm with implicit control, based in a compartmentalized connection graph theorem prover. Whilst the research has as it main goal the development of a language in which programs can be written with much less explicit control than PROLOG and its existing successors, a secondary goal is to exploit the immense parallelism inherent in the connection graph.

The focus of this paper is the documentation of the extent of the parallelism inherent in the proof procedure. We characterize six different forms of parallelism These various forms of parallelism can be further classified into two classes: those associated with the performance of resolution steps, and those which are more concerned with unification.

Unification is thus also a major topic of this report. In the first report of this series unification was identified as a major source of the cost of executing a logic program, or of proving a theorem. It turns out that deferring unification is the one of the best ways of dealing with it: hashing to perform it, and indexing to avoid it.

Indexing and hashing, therefore, is the third topic covered in this report.

[1] The work reported here was in the main undertaken while the author was at Macquarie University NSW 2009 AUSTRALIA, and was supported in part by IMPACT Ltd, PETERSHAM NSW 2049 AUSTRALIA, the Australian Telecommunications and Electronics Research Board, and the Australian Research Council (Grant No. A48615954). The author is currently supported under ESPRIT BRA 3012: COMPULOG.

Introduction

Our previous reports [Powe88,90] discussed the advantages and disadvantages of PROLOG, focussing particularly on logic and control – the advantages of a clean declarative semantics and the disadvantages of a rigid predetermined control. It then presented some of the advantages and possibilities of logic programming in an automated theorem prover without cuts, annotations, builtins and other embellishments of clausal logic: in particular advantages relating to efficient treatment of recursion and examples of programming around major classes of builtin. This goals were achieved either directly in the compartmentalized connection graph theorem prover or in combination with preprocessing transformations.

The most common generalizations of PROLOG have held fast to its general control regime, whilst allowing relaxation as well. In particular, the parallel or concurrent PROLOG systems generally fall into this category. Most generalizations also tend to adhere largely to the Horn clause paradigm, while perhaps allowing more general forms with some specific model.

This work has approached logic programming from the opposite extreme, a completely general clausal theorem prover without control, and seeks to understand the behaviour of logic programs expressed in such an environment, including how efficiently to implement the environment and whether it is possible to use general search heuristics rather than an explicit control paradigm. The CONG system [Powe88] for CONcurrent logic programming is based on a CONnection Graph theorem prover and can accept pure PROLOG programs (cutless Horn clauses without builtin predicates) as well as general clause form logic program (again pure without builtins).

This paper moves on to address some of the efficiency issues which remain. Clearly the heuristics will be major determiners of efficiency, and even efficacy, and [Powe90] showed that given appropriate heuristics CONG can achieve reduction in proof lengths to logarithmic or less compared with PROLOG. But [Powe90] also characterized the complexity of the proof as being of order by $c*n*d$ operations, where the number of literals in the program is n, the maximum depth of any term is d, and the maximum number of copies of any clause is c. This reflects the subdivision of labour between (recursive) copying, (unit) resolution of (ground) Horn clauses, and instantiation (unification testing) of the proof. In the following sections we address the cost of these three components in parallel and/or sequential implementations.

The techniques in [Powe90] showed that in a theorem prover is was possible to approach this ideal in a way PROLOG does not, and that the recursive resolution copying component could be achieved by pseudo-resolution resulting in clause doubling to the appropriate number of copies in *log c* steps. But the number of copies of the terms of a clause is still in general c, so the work in unification can remain of order $c*n*d$ operations.

Whilst PROLOG traditionally measures speed of PROLOG implementations in LIPS (Logical Inferences Per Second), or resolutions per second, this hides the cost of unification inherent in the problem, and the benchmarks with trivial unification will demonstrate far greater speeds than those with complex unification characteristics. Indeed, the unification can be optimized away by a compiler in the trivial cases, but never in the complex problems where complex unification is inherent in the nature and purpose of the program.

Similarly with CONG, trivial unifications will not slow the resolution and pseudoresolution processes. But mostly one unification operation linear in c will be required at the point where a pseudoresolvent is resolved non-recursively. But in a parallel context, the reasonably trivial examples should be unifiable in unit time on $o(c\ )$ processors. In a typical complex logic program this may grow to logarithmic time, whilst in the most artfully constructed automated reasoning problems it may remain linear (in $c*n*d$).

We have therefore identified unification as the major remaining bottleneck. This paper explores the complexity characterization of unification and provides examples with the various behaviours just outlined. It further examines the feasibility of avoiding and delaying unification where the unifiability is not yet known or unifier is not yet required. In particular, we note the separability of unifiability testing and unifier determination.

The above calculations were also based on a deterministic discovery of the proof – whether guided by oracle, heuristics or control. In an Artificial Intelligence application with blind search characteristics, most unifications will fail, or at least the proof strand of which they form a part will fail. In this case the time expended on accurate unifiability testing and unifier determination is totally wasted. We therefore explore the possibility of trial unification in which unification work is undertaken in a way designed to expedite the detection of failure, and allowing the proof to proceed with only a certain likelihood of the eventual success of the unification. This allows also the concept of a trial proof in which unifiability and unifiers are only finally accurately determined when it appears that a proof has been found.

We have already given a indication of the complexity and potential of parallel unification. The importance of unification also emerges when we characterize the various sorts of parallelism which are achievable in CONG.

Proof Algorithm

Compartmentalized Connection Graph

We now recapitulate the algorithm used by CONG as presented in [Powe90], with references back to the differences from the original algorithm of [Kowa79].

Unification

In the following, unless specifically stated otherwise, *unifiable* and *unifier*, refer respectively to *weak unification* and the resulting *most general unifier*. In PROLOG terminology, these are unification in distinct environments and the unifying substitution. Weak unification can also be described in terms of unifiability after renaming.

Links

A set of clauses to be proven inconsistent are linked into a graph by connecting distinct clauses with a *link* whenever they have unifiable complementary literals. A link may at times most conveniently be regarded as two connecting clauses, at other times as connecting the two literals which gave rise to it. However, they actually indicate potential resolutions, or equally well potential resolvents. The substitution giving rise to the most general unifier of the linked literals is associated with the link.

Resolution links

Under this above definition a link is formally defined only between distinct clauses, although loosely used it may, when the context permits, include internal links. In certain syntactic contexts all forms of links, including even links between non-complementary unifiable literals may be intended. When we want be absolutely clear we can refer to the links between complementary literals of distinct clauses as *resolution links*.

Pseudolinks

Additional connections between unifiable literals of opposite sign within a single clause are added and are termed *pseudolinks*. These internal links represent the potential for copies of the clause to resolve, but are themselves never actually resolved on. They may thus also be referred to as *self-resolving links*. The substitution giving rise to the most general unifier of the linked literals is associated with the pseudolink.

Initial graph

The graph resulting from the addition of resolution links and pseudolinks to a set of clauses is called the *initial graph*. Figure 1 shows the initial graph and substitutions for a simple append example.

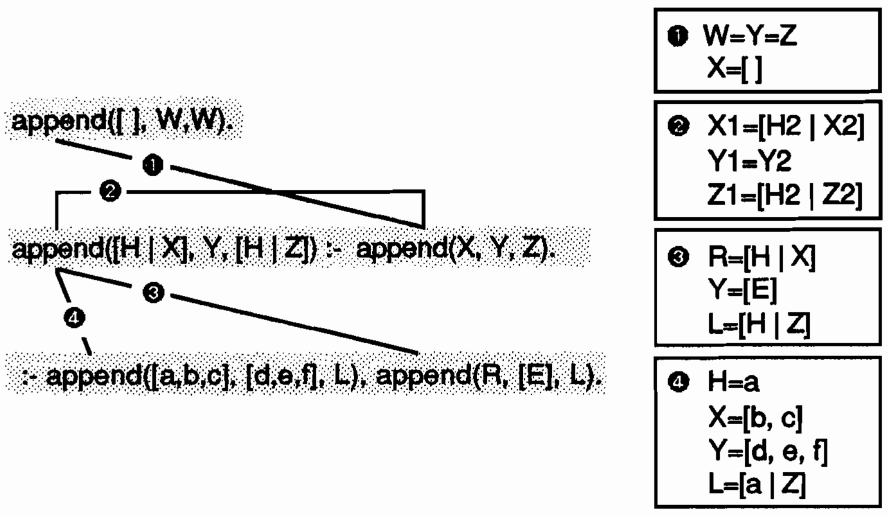

append([ ], W,W).
❶
❷
append([H | X], Y, [H | Z]) :- append(X, Y, Z).
❸
❹
:- append([a,b,c], [d,e,f], L), append(R, [E], L).
❶ W=Y=Z X=[ ]
❷ X1=[H2 | X2] Y1=Y2 Z1=[H2 | Z2]
❸ R=[H | X] Y=[E] L=[H | Z]
❹ H=a X=[b, c] Y=[d, e, f] L=[a | Z]

Fig. 1. *Initial graph for* append.

Resolution rule

A link may be resolved upon by resolving upon the linked clauses (the *parents)* and *replacing* the link by a new clause (the *resolvent*) obtained from the union of terms of the two clauses excluding the linked terms with the application of the associated substitution.

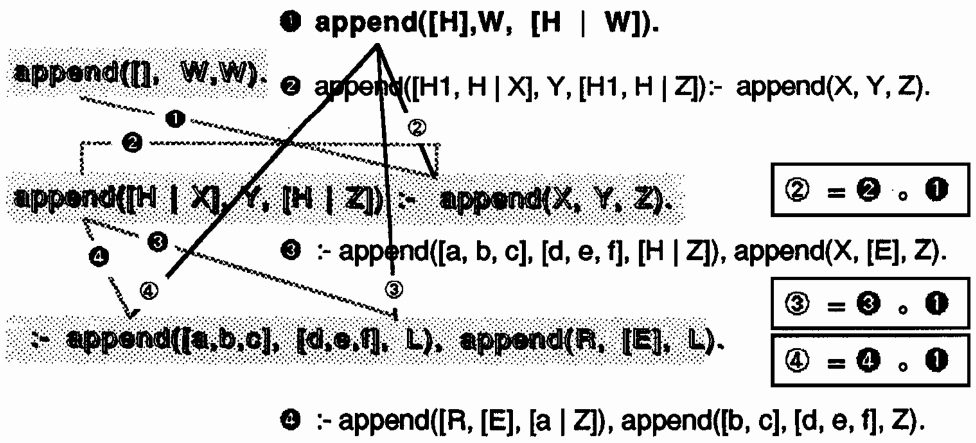
In the compartmentalized connection graph a constraint is added, that neither of the parent clauses may contain a pseudolink. But a complementary constraint is lifted, in that pseudolinks may themselves be resolved upon as if it linked two separate copies of its parent clause. The terms of the resolvent are said to have *inherited* from its parents.

Inheritance rule

Upon resolution, links and pseudolinks impinging on the inheriting terms of the parent also inherit to give new links and pseudolinks attached to the inherited terms which have the composition of the resolving link's substitution and that of the original links, and will be a resolution link or a pseudolink according to whether the link connects distinct clauses or is internal to one clause, respectively.

We define the composition operator as performing unification of the substituted variables occuring in two link substitutions. That is, when the substitutions both instantiate the same variable, the bound structures are unified.

Figure 2 shows the results of resolving on each of the four links of Fig. 1 and in the case of the clause ❶ (resulting from link ❶) the inherited links are also shown.

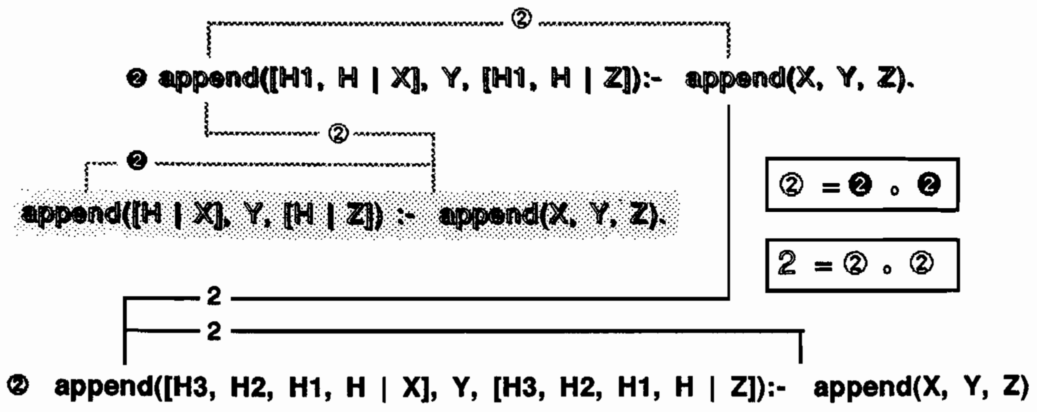


❶ append([H],W, [H | W]).
append([], W,W).
❷ append([H1, H | X], Y, [H1, H | Z]):- append(X, Y, Z).
②
append([H | X], Y, [H | Z]) :- append(X, Y, Z).
② = ❷ ∘ ❶
❸ :- append([a, b, c], [d, e, f], [H | Z]), append(X, [E], Z).
③ = ❸ ∘ ❶
④ = ❹ ∘ ❶
:- append([a,b,c], [d,e,f], L), append(R, [E], L).
❹ :- append([R, [E], [a | Z]), append([b, c], [d, e, f], Z).

Fig. 2. *Inheriting and Composing Links*

Positive-to-parent rule

In the compartmentalized connection graph, pseudolinks will not be inherited from during a link resolution as a consequences of the restriction against parents containing pseudolinks. However they may be inherited from during a pseudoresolution. In this case the pseudolink currently being resolved could give rise to a new pseudolink and two new links connecting the parent clause and the resolvent – as the parent clause is playing a dual role and resolving with a copy of itself producing a redundant link [Eisi88]. This is resolved arbitrarily by the *positive to parent rule* [Powe90] which says that, apart from the pseudolink, only the resolution link connecting the *positive* literal of the resolvent *to* (the negative literal of) the *parent* clause is inherited from the pseudolink undergoing pseudoresolution (which is in the process of being processed and removed).



②
❷ append([H1, H | X], Y, [H1, H | Z]):- append(X, Y, Z).
②
❷
append([H | X], Y, [H | Z]) :- append(X, Y, Z).
② = ❷ ∘ ❷
2 = ② ∘ ②
2
2
❷ append([H3, H2, H1, H | X], Y, [H3, H2, H1, H | Z]):- append(X, Y, Z)

Fig. 3. *The Doubling Effect and Pseudoinheritance.*

The pseudoresolution operation leads to particularly efficient handling of recursion through the doubling effect, reaching a point in the logarithm of the number of steps required by resolution, as illustrated in Fig. 3.

Factoring, purity, tautology and subsumption rules

Applied to general clauses (unrestricted to Horn), resolution is incomplete without factoring (or merging). Also there are various optimizations whereby certain clauses which are inessential to the proof may be deleted. These considerations are irrelevant to this paper and the rules are therefore not presented. See [Eisi88; Powe90].

Ordering and restriction filters

Certain heuristics may be used to further specify the selection of links, which are left considerable latitude under both the original and the compartmentalized connection graph procedures. In fact, strong completeness cannot be assured without such filters, and even with filters no proofs of strong completeness have yet survived scrutiny [Eisi88].

This is again not an issue for the Horn case, where strong completeness is assured by a variety of strategies. Filters recommend include unit resolution, hyperresolution and orphaning [Eisi88; Powe90]. The standard and compartmentalized connection graph procedures presented may also be seen as the results of applying particular filters to the generalized procedure of [Eisi88], in which no restrictions are applied to when or whether one can resolve on particular types of links (viz. resolution links or pseudolinks).

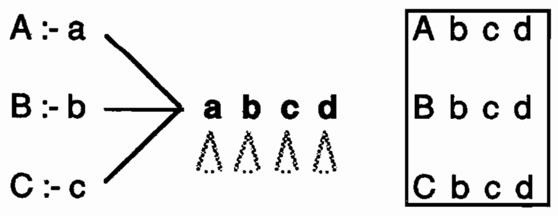
The compartmentalized filter was specifically designed to take advantage of the power of treating recursion separately and in view of apparent strong completeness properties in combination with the *parent-before-child* (orphaning) and other filters – a draft proof exists must has not yet been subject to sufficiently rigourous examination.

Latent Parallelism

While the primary motivation for choosing to consider Logic Programming in the context of a Connection Graph Theorem Prover was to allow escape from the strictness of PROLOG control and the possibility of concurrent solving of independent parts of the problem (e.g. working around unknowns in Machine Learning and Natural Language applications), a secondary motivation was that not only this application level parallelism, but the fine grained parallelism of a graph in which all links could in principle be resolved in parallel.

Therefore, ignoring application level parallelism, we now look at half a dozen possible ways of exploiting parallelism within the connection graph formalism itself. We start by looking at the system from the traditional Parallel Logic Programming perspective.

OR-Parallelism



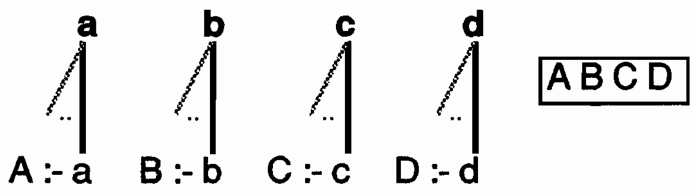
√: **Links totally independent**

Fig. 4. *Or-Parallelism*

The concept of OR-parallelism in relation to PROLOG encompasses the idea that given a (sub)goal, all matching clauses can be pursued in parallel – and indeed a set of (partial) solutions is returned. In the PROLOG context, this set of parallel solutions can be carried forward independently, effectively replacing backtracking.

OR-parallelism is trivially modelled in the connection graph by resolving all links on a term in parallel, producing independent resolvents and inheriting links straightforwardly (Fig. 4).

AND-Parallelism



X: Can't remove links – links not used up
?: Compose combination of substitutions

Fig. 5. *And-Parallelism*

AND-Parallelism in PROLOG involves solving the present set of subgoals in parallel. This incurs the difficulty that substitutions must later be composed, and that those which might have been far more tightly specified (e.g. deterministically instantiated) with the PROLOG goal ordering may now find many irrelevant bindings.

In CONG the difficulty manifests itself in that the indexing function of the links cannot be maintained, as links are only partially used.. The substitutions on the set of selected links must be composed to produce the resolvent and inherited links. The additional overhead required to index the partial use of links negates the point of the graphical marking of work to be done, unless it is part of a complete elimination of the clause

CLAUSE Parallelism

But the most effective use of AND-Parallelism in CONG comes about when one eliminates the whole clause in parallel. This eliminates the problem of indexing which combination of links has been used – or uses sub-indexing of highly linked clauses in a progressive way which precludes redundancy. And it combines OR-Parallelism into the solution as progress is being made through resolution with every matching clause for every term of the target clause, as illustrated in Fig. 6.

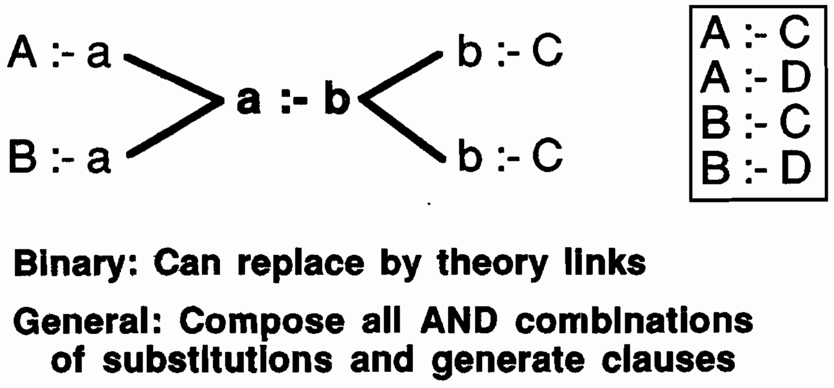

A :- a
B :- a
a :- b
b :- C
b :- C
A :- C
A :- D
B :- C
B :- D
Binary: Can replace by theory links
General: Compose all AND combinations of substitutions and generate clauses

Fig. 6. *Clause Parallelism*

Note that, where clauses are multiply linked, that is in the one pair of clauses there is more than one distinct pair of linked terms, new links would normally be inherited back (reflecting the implicit recursion). This problem is avoided by the combination of orphaning and compartmentalization [Powe90].

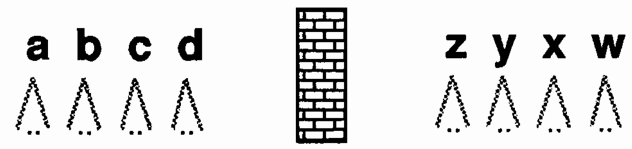
The composition operation logically takes place before further work on deepening the search, and many of the resulting combinations tend to be incompatible – that is the unification of substitutions during composition fails. This acts to prune the search.

As a further special case, binary clauses can be eliminated in a totally different way by compilation into theory links [Ohlb90], extending the definition of unification with a theory represented as a substitution tree. In combination with unit resolution and reformulation into at most ternary clauses, whole Horn programs can be compiled into unification. This approach is presently being investigated in combination with other optimizations of representation.

LINK Parallelism

The application level parallelism, that evident through having completely independent problem parts, or at least parts linked through a very well defined and relatively small interface, is available by the straightforward observation that if two links have no clause in common, there is no possibility of interference with the creation of the resolvent, the inheritance of the new links, or the removal of old links.

Note that this parallelism involving independent links can be combined with clause parallelism with the independence condition being modified so as to allow parallel clause elimination only where the sets of clauses linked with each pair of clauses are disjoint. Again this makes best sense in combination with compartmentalization.



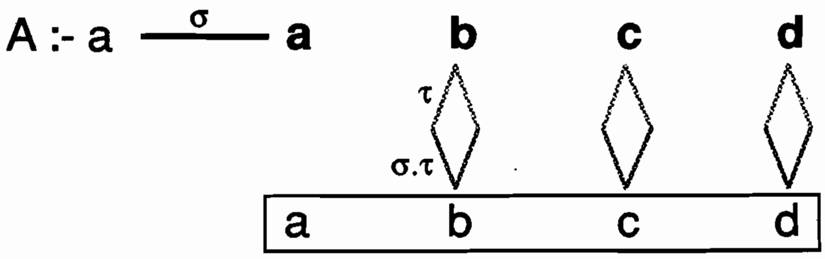
General: Any set of links with no clauses in common may be resolved in parallel without any interference with link removal or inheritance processes

Corollary: Gross application level parallelism may be exploited

Fig. 7. *Link Parallelism*

Inheritance Parallelism

The introduction of a graph, however, introduces a form of indexing overhead which is itself parallelizable. As illustrated in Fig. 8, the inheritance, and the composition and unification involved, can be performed in parallel. However, we can do better still by looking at the decomposition of this work of inheritance.



√: Inheritance of new links can always be done independently in parallel

Fig. 8. *Inheritance Parallelism*

Composition Parallelism

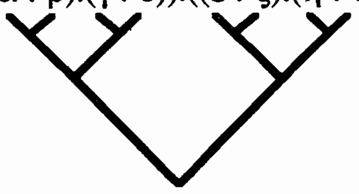
The actual work involved in resolution on links is primarily the inheriting of new links, which in tern involves composition of substitutions using unification. Thus the parallelism represented in the AND- and OR- parallelism can be combined into Clause Parallelism and can be multiplied by taking advantage of Link Parallelism, which results in a multiplied plethora of Inheritance Parallelism. This boils down to us doing a lot of unification in parallel.

$\alpha . \beta . \gamma . \delta . \varepsilon . \zeta . \eta . \theta$

$((((((\alpha.\beta).\gamma).\delta).\varepsilon).\zeta).\eta).\theta)$

Substitutions are by default composed sequentially and are thus left associated

$((\alpha . \beta).(\gamma . \delta)).((\varepsilon . \zeta).(\eta . \theta))$



If associated pairwise substitutions can be composed with only logarithmic delay

Subcompositions are reusable

Fig. 9. *Composition Parallelism*

But there are dependencies within the composition and unification work which cut across the other forms of parallelism. In PROLOG and its concurrent and parallel variants, composition would tend to be done sequentially, but as shown in Fig. 9, it can best be done in a pairwise parallel fashion. Clause Parallelism already allows this, because the AND-Parallel composition admits precisely this form of parallelization. Link Parallelism excludes direct dependency. But a sequence of such parallel steps would still tend to be combining (composed) substitutions in a linear way.

Trial Proofs

Delayed Unification

By introducing a lazy unification scheme, we can allow composition to be delayed in ways which allows advantage to be taken of composition parallelism. One extreme is that we could produce a propositional-style proof based just on the initial graph (viz. without further composition of substitutions), and then check this *trial-proof* once found, by using efficient composition parallelism.

A blind implementation of this extreme scheme, however, eliminates the possibility of pruning on the way through. Whilst for some programs this may not be a problem, in others an explosion could occur in cases which should be deterministic. In such programs, most unification attempts actually fail. We want to have the failure information as soon as possible, but to delay the full unification expense as long as possible, and minimize the total expense as much as possible.

This leads to the concept of *trial unification* as an extension of clause indexing [Powe88]. If we can eliminate 99% of the non-unifiable cases with 1% of the expense, we will make a huge saving in those cases where the failure information is important, both in terms of our *trial proof* and our eventual unification bill. If we can further use the information gained in the *trial unification* to gain a head start with the actual unification, so much the better. And ideally as we find ourselves searching more and more explosive paths, it would be nice to be able to become increasingly certain that we are not predicating our search on a *false drop*.

And if this *trial unification* can be done in a way which both extends to *full unification* (and appropriately represented substitutions) and is consistent with our pairwise *composition parallelism* ideal, then we can achieve several orders of magnitude improvement, quite apart from the speedup directly due to parallelism.

Trial unification requires the obtaining of some sort of approximate indication of unifiability based on incomplete samples or inexact associations. Fast unification requires some way of quickly identifying which atomic subterms need to be compared. We show below how this can be achieved using indexing and hashing techniques.

Intersected Composition

As we have defined it, our composition operator is not only associative but commutative.

Composition parallelism exploits associativity by taking the original links, the first level of (*weight* 1) substitutions, and combining them pairwise to form the *double weight* substitutions of the next level, repeating until the single substitution representing the composition of the multiple composition task has been obtained. In general, not all of these paired substitutions correspond to actual links in the graph, although in general one multiple composition (e.g. *trial proof* or *AND-combination*) will have significant overlap with others.

Thus it would be advantageous also to choose pairs to compose which are subsets of other required compositions. In the case of *AND-combinations*, this is automatic. In the case of *trial proofs*, it may be advantageous to exploit the commutativity of composition to allow the precise choice of which pairs to unify can be selected to be those which are useful in more compositions. However, again given the sequential component and tree structuring in the growth of partial proofs, some advantage may be had from intermediate substitutions found simply by the heuristic of combining substitutions of similar *weight* as they become available.

A composition failure or substitution produced in one context can thus be made available for re-use when, via some other path through the graph, the same initial links are again brought into a multiple composition – which can itself be restructured to take advantage of existing compositions. Moreover, the failed compositions may be used to identify and prune incompatible links. Any attempt to make use of this preexisting unification and composition information will only be reasonable if links and compositions are efficiently indexed and/or sorted.

The result of using information from intersecting or subset compositions would be that information obtained in the checking of one trial proof could prune away part of the remaining resolution search space, or act as a non-link substitution at the beginning of subsequent parallel compositions.

If all sub-compositions of a multiple composition were available, this would clearly have more potential for effective pruning than if only *power-of-two weight* compositions are performed through the use of pairwise composition to obtain composition parallelism. Whilst additional processors could be allocated to check other combinations, or in particular, other initial compositions, it is most productive to have the allocation guided by other parts of the connection graph procedure, and processors are therefore better allocated to discovering other, preferably related, trial proofs.

Trial Unification

Superimposed Code Words

The key to trial unification is indexing in a way which allows a selection of bits cutting across the terms structure to be used to assess the potential for a match. This type of approach was originally introduced for database search with superimposed codes.

Here we use a special form of hashing, the *Superimposed Code Word (SCW)*, to overlay hash type codes in a word, and use a subset operation to determine if the code for the key we are seeking has been ordered in. This is fine for databases, but we need to handle both variables and term position information. Hence we have defined a special variant for Logic Programming applications [Wise84; Powe88].

Interestingly, indexing with SCWs can itself be construed as exploiting the parallelism inherent in even a single sequential processor. A processor with 32-bit data paths and logical operations gains advantage from superimposed coding in part because it allows 32 logical operations to precede in parallel.

Field Encoded Words

In the *Field Encoded Words (FEWs)*, we retain the subset operation, but allocate fields of the word to subterms down to a certain depth, and use a ternary tree for each functor to index into terms. This is illustrated in Fig. 10, where the saving by using this technique on a particular set of benchmarks (about 20 subterms down to depth 5).

```
a(b,c):              a                 b          c
active:   01011001110101100  10010110   11010001     A1
passive:  01011001110101100  10010110   11010001     P1

 a(X,c):             a                 X          c
active:   01011001110101100  10010110   11010001     A2
passive:  01011001110101100  10010110   11010001     P2

Quick Unification Check:     (A1 ⊆ B2)  &       (A2 ⊆ B1)
Quick 3-Tree Indexing: left: (A1 < B2)  right:  (A2 < B1)  mid: rest
```

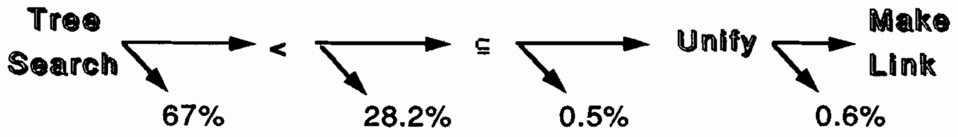


Fig. 10. *Field Encoded Words & Tree Indexing*

We now need to consider how to extend it usefully to deeper structures, and in particular linear structures. We should also note that by keeping the fields as powers of two, it is possible to do variable substitution totally with the codes.

Unification Algorithm

We now go on to report on some preliminary investigations on the application of hashing and indexing techniques to full unification. In particular, we sketch a unification algorithm (not yet fully implemented or specified) which has the potential for expected unit time unification on an expected linear number of processors. It is trivial to specify if we allow cubic order processors. We need very careful hashing if we hope to achieve near unit unification on linear processors or unit unification on near linear processors.

In trying to develop an algorithm for unit-time unification on a number of processors linear in the size of the problem, we are dealing with a task which is known to be impossible in the general case [Dwor84] – which even in the cases which are not pathological still requires careful treatment.

In the following we spend some time exploring the possibilities for hashing and show that given perfect hashing we can find some simple characterizations of the conditions under which parallel unification exhibits particular worst case orders. We further show how to develop a hashing function which we would expect to provide unit-time association on regular logic programming examples, leading to an "expected" unit-time unification algorithm (in the sense that pathological examples are by definition "unexpected", but without defining how we would characterize the probability of a particular term structure).

Hashed Unification

The answer to the problem of deep and linear structures is to use another form of hashing: hashing of the terms into field positions of the FEW. This extends the capability to an extent. To go further would clearly require more bits, as we simply don't have enough information. When we add the capability of extension, of adding these extra bits, we have the possibility of extending to exact unification. Or looking at in from the point of view of parallel unification, if we can hash our terms into processor space so that subterms in corresponding positions hash into the same processor, we have the capability of performing a large class of unifications in unit time, and another large class in logarithmic time. Or looking at it yet another way, we want to associate corresponding heap addresses for the two terms (or sequences of lefts and rights).

If we can do this association in unit time, then in unit time we can do simple matching of ground structures or terms without repeated variables. The problems related to association or hashing are illustrated in Fig. 11 and the inherent linear and logarithmic nature of some unification problems are illustrated Fig. 12.

Problem: Term skeletons of order N may take many forms and several may match the same tree of order exp(N).

|7| **a(X, b(Y, c(Z, d)))** ***and a(b(c(d, Z), Y), X)***

both match

|15| **a(b(c(d,d), c(d,d)), b(c(d, d), c(d,d)))**

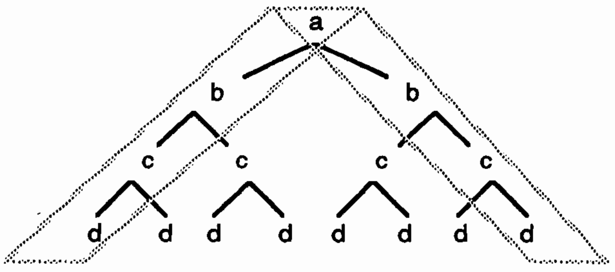


Fig. 11. *Worst Case Example for Unification Hashing*

In Fig. 11, we see the problems of attempting to hash terms simply by depth. If the actual bit determining whether a left or right position is being occupied occurs either in the addressing or the content of a hash entry, unambiguous matching of degenerate (viz list) structures is possible. However, all of these degenerate structures can also match against the balanced structure of the same depth. Thus we want to match a term linear in the depth of the tree with another term exponential in the depth of the tree, and in general mapping, even superimposing, the balanced tree into a table bounded according to its depth loses too much information.

But not always! Note that if the b's, c's and d's in Fig. 11 are actually the same, folding to the same hash position is no clash. But, if even just the d's are distinct we must use some method such as a common index and replace them by pointers (instantiated variables) to an overflow area. If we do this at every level we have no gain over descending the term normally.

Another alternative is to hash into a balanced tree structure linear in the size of the term, but then terms of degenerate structures won't have addresses in the hash table which correspond to those to which occur in the same position in the balanced tree of the same depth. We we both schemes the worst cases involve a mismatch of the order of the table and the tree we are trying to hash into it.

A separate problem is that the attempt to *associate* keys based on the sequences of lefts and rights (or indexes into a virtual heap) must recognize that the length of these keys (or virtual) addresses is linear in the case of the degenerate term, and that we can no longer hide behind the charitable assumption that we can manipulate addresses in unit time – which we do when they are logarithmic in the size of our data structures, as is the case with the balanced tree. An associative memory looks for one key amongst n in linear time, and involves n active memory units in doing so. To match n such keys simultaneously requires n of these, and by the time we take into account that the match involves n bits, we are up to a n^3 processing units!

The linear length of our keys is of course still a problem even if we were to use special hardware and/or language features (*à la* Linda).

Even during hashing, we must be careful that we don't try to use an n bit quantity.

In view of all of these pitfalls, we must be very careful to choose a hash function which is perfect for the sort of structures which are likely to occur. Moreover, we would like to ensure that dependencies between addresses in a structure are minimized by ensuring that a clash at one point in the tree doesn't imply it continues for all descendants of a node. But unless we take into account every bit of the address key, the bit which is omitted could be the only one which is different between two keys, and all subtrees subtended by them.

Note, moreover, that in many practical cases the functors (and arities) of the interior nodes are identical, and it is thus sufficient to ensure that the leaves can be matched effectively. Indeed, any structure can be mapped into a binary tree, thus ensuring satisfaction of the condition for such leaf matching.

And then, not only must different size structures be able to match specified parts of the different sized hash tables, but the unifying substitutions and/or the unified terms must be made available in hash table form for subsequent use by the algorithm. In the first case, this limits the extent to which we can pervert the actual values for the depth of a term, if we aim at hashing into a table linear in depth. In the second case, there is the problem of the redundant representation upon instantiation of multiple variables (as we have given up the advantages of a DAG). We look at this later on.

As a final comment on the question of hashing and association, we note that it is possible to superimpose the entries destined for a hash address in the event of a clash, and to have additional structures which store the two variants for checking if the imprecise subset-matching unification succeeds – viz. applying ideas from FEWs again. If we introduced tables of double the size after each clash, we would end up with a number of passes logarithmic in the maximum number of clashes at any one hash address, and an squared order worst case memory overhead.

Sliced Unification

But before we turn to the development of a hash function, let us look at the other use we want to make of these hash tables. We not only want to perform full unification in unit time (when the processor resources are available and the problem complexity allows), but to do *trial unification* which gives a first approximation of unifiability and aims to detect *unification failure* as soon as possible.

By actually using only one random bit of each row of the tree in Fig. 11, a *bit slice*, we achieve matching of two degenerate ground structures (like the |7| examples). with a probability of not detecting nonunifiability which reduces exponentially in the depth of the tree. This probability itself reduces exponentially with each subsequent bit slice we take. Thus with depth d and taking b bits the probability of detecting an inequality is $(1{-}2^{bd})$. When b actually is the full number of bits required to represent the symbols, the probability is that of them not actually being equal, and failure to detect an inequality therefore is a guarantee of unifiability (remembering that in this ground case life is not complicated by multiple variables and occur checks).

Thus providing we can *slice* through the terms, we can approach full matching as closely as desired. With enough processors, linear in the total number of bits in the terms, we can establish the match in unit time. Where there are variables, by adopting the same technique as for variables in the FEWs, we can again have a match in unit time. Since we can also establish the proposed bindings for variables, we can then continue with a pairwise unification of repeated variables as illustrated in Fig. 12.

Characterization of Unification

These hashed unification and sliced unification algorithms leads to unit, logarithmic and linear parallel unification complexity for the algorithm, depending on various features of the terms which we now proceed to characterize.

```
p(X, X, X, X)          &      p(a(...), a(...), a(...), a(...)).      Log Case
p            X            X           X           X
p            a...         a...        a...        a...

p(X, X)                &      p(f(Y, Y), f(g(Z, Z), g(a,b)).     Linear Case
p            X                        X
p            f      Y     Y           f     g Z Z g a b
```

Fig. 12. *Log and Linear Cases with Perfect Hash*

In general, the algorithm will be worst case unit if there are no repeated variables, logarithmic in the maximum number of repetitions of a variable and linear in the number of distinct repeated variables, as illustrated in the example of Fig. 12. The logarithmic results arises from the possibility of dealing with the multiple instances of a variable pairwise as separate unification problems. The linear limit is a sequential result, and the example of Fig. 12 show that this algorithm does exhibit linear behaviour on this case. The unit result in the absence of repeated variables is a trivial consequence of perfect hashing – but note that some schemes introduce additional pointers when the hashing is not perfect.

The hashing scheme explained below (see Fig. 13) has been designed so as to be perfect for most common, and in particular regular, sparse (towards degenerate) or dense (towards balanced) subterm allocations.

Development of a Hash Function

The first observation to be drawn from our discussion above is that a hashing function which is optimized for sparse trees will be worst case for balanced trees, and vice-versa. But our superimposing concept allows us still to derive some information in either of these mismatch situations. Furthermore it is straightforward to add an additional bit to warn of a clash, or even a field to track them number of clashes or maintain an overflow list, as in traditional hash tables.

But failure to detect a clash means that matching of the hash tables is completely reliable. Thus if we has into two tables, one optimized for the more balanced tree, and one for the more degenerate, we know that if *either* fails to note a clash, the unification has been reliable.

But note too, that if a structure is close to one or other extreme, the ratio of the actual size of the tree to the size of either the degenerate or balanced tree must be close to 1. If we allocate space for a table generously whenever the ratio is bounded by some constant, then (given an appropriate hash function) the most common and regular structures should avoid clashes in that table. Furthermore, if overflow structures are developed in the form of addition of new variables, local balance or linearity can be taken into account at the cost of an additional time step.

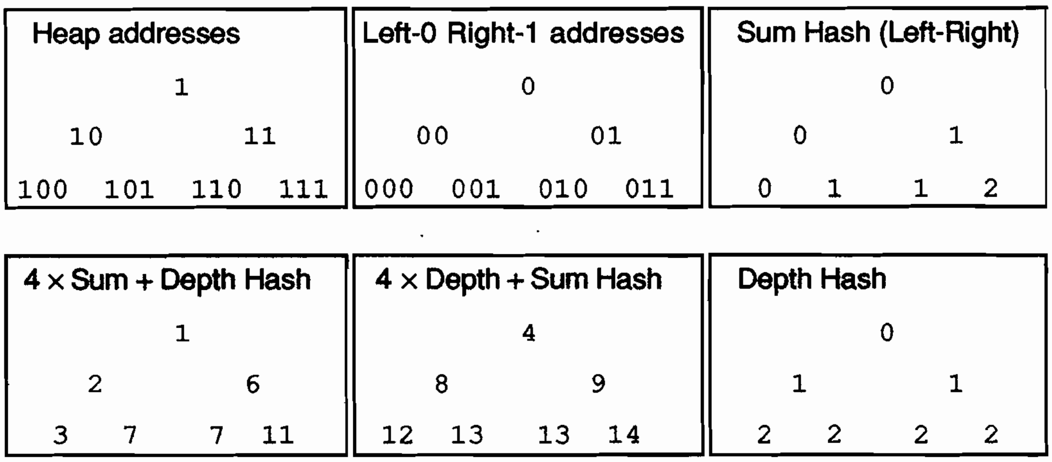


Fig. 13. *Examples of hash function for unification.*

This optimization is illustrated in Fig. 13, working towards a list optimal hash function. Here we note that allocation of a key either as an index into a heap or as a left-right string is equivalent – within a bit. In the first case the first bit is always a 1, in the second a 0. The first gives all positions distinct for a balanced tree, the second all leaves distinct. The sum of the bits is the same ± 1 (taking into account the first bit). The number of clashes in the leaves is distributed normally (the number of clashes is given by the binomial coefficients), and there are also additional clashes taking interior nodes into account. The depth hash has all leaves clashing, indeed all leaves at the same depth but none between different depths.

The example continues with the supposition that the structure actually contains about four times as many elements as the depth, and shows two ways of combining the basic hashes so as to combine the favourable features. We cannot expect to eliminate clashes amongst the elements of a level in this way, as the number of clashes in a level is still basically exponential in its depth and hence the size of the table. However, note that (within the size of the table represented) all lists can be represented without clash (for either interior or exterior nodes), and that the most likely cases of bifurcating degeneracy can also be represented (viz. both second level positions can subtend lists of the same type).

To handle the need for compatibility among different sized structures, as well as rapid copying and expansion upon substitution, it is helpful to increase the allocated size to a power of two.as we did with the FEW. This introduces at most a factor of two additional overhead. If we allocate copies at all smaller powers of two, with clashes superimposed, then we have for the cost of another power of two a family of faster approximations, and compatibility with all smaller structures. Or we can simply define matching routines which virtually reduce the table to the appropriate size for a given match. We can similarly arrange for slices of these tables to be obtained virtually for trial unification.

Finally, we point out that the number of processors required for the detection of the occasional clash and its resolution, along with the logarithmic and linear overheads resulting from examples with multiple variable, usually do not require.the full order of processors for their completion, and that this can be performed in parallel with other work not dependent on full knowledge of the resulting substitution.

Hashed Composition

Now the question is whether we can handle composition and ensure that results of the unification are presented in a form appropriate to subsequent unification. The case where the unified term is no larger than its parents is straight forward, as processors have already been allocated.to these terms. When it grows, we need to ensure that the appropriate copying takes place. This, like the original set up of the terms, requires that the information about where subterms will end up reaches the appropriate processors for copying. Because we deal with multiple unifications pairwise, we can be sure that each term will not need to be multiply copied because of repeated variables. And the propagation through multiple variables is already linear. Thus the copying can be achieved without worsening the order of the algorithm.

Asking Uncle

The parallel traversing and *in situ* placement of the elements of an arbitrary tree structure can be performed by linear order processors in logarithmic time. This is by application of techniques well known in parallelism for dealing with linked lists.

This algorithm involves having each algorithm pass back or forward information successively to its immediate parent, (one generation), its grandparent (two generations), and all its other ancestors a power of two removed. This information includes the number of known levels of descendants/ancestors, and whether that information is complete. Thus after time logarithmic in the depth of the tree (that is, if degenerate logarithmic in the size of the structure, and if balanced logarithmic in the logarithm of that size), the number of levels of descendants/ancestors is known.

In the case of a CONG term, there is the possibility of termination in a variable as well as termination with an atom (such as *nil*). Thus there are three cases which can be stored in a trit called *UNCLE*: *UNComputed*, *Longer* (variable termination) and *Exact* (nil termination). It is recommended that this trit, along with the number of descendants currently known, be not only calculated during hashing, but stored permanently with subterms and variable bindings.

This not only allows faster determination of composed structures, but is useful itself as a quick unification check, the UNCLE check. In this check, two constraints are available for quick determination: *Exact* pairs must compare exactly; *Exact* terms must be at least as long as *Longer* terms (or *UNComputed* terms).

This check is extremely useful in the Compartmentalized Connection Graph as it is usually sufficient to prevent Pseudoresolution proceeding beyond what is necessary for the problems at hand.

Application

The Compartmentalized Connection Graph has been applied, as presently implemented, to a small number of standard PROLOG benchmarks. Furthermore some of these benchmarks have been analyzed in relation to their behaviour under the further techniques of this paper – in particular the ability to match lists in unit time and unfold recursion in unit time.

One of these standard algorithms is QuickSort. As no standard sequential sorting algorithm was known to be amenable to optimal speed up through parallelization, it was a surprise to discover that our analyses predicted that running QuickSort on CONG using the techniques described here will in fact produce linear speedup given linear processors. To verify this quickly, in the absence of a version of CONG implementing the parallel unification, the predicted behaviour was further analyzed and used to design a conventional *in situ* style parallel sorting algorithm which did indeed demonstrate this behaviour [Powe89].

Further analysis, tracing through the maze of unifications, developed an even more efficient version based on linked lists [Powe91]. We will report on the analysis of the CONG performance of QuickSort in due course.

Acknowledgements

I wish to acknowledge the participation of Graham Wrightson, Debbie Meagher, Laz Davila, David Menzies, Martin Wheeler, Graham Epps, Richard Buckland and Philip Nettleton in the MARPIA project at Macquarie University, and their varying contributions to the development of CONG. Laz Davila wrote the first version of the present incarnation of CONG. Debbie Meagher has been responsible for its further development including the addition of compartmentalization.

In addition I thank Norbert Eisinger and Hans Jürgen Ohlbach for helpful discussions during my time in Kaiserslautern.

References

[Bark90] Jonas Barklund, "Parallel Unification", doctoral dissertation, UPMAIL, Uppsala University, Sweden (October 1990).

[Dwor84] C. Dwork, P.C. Kanellakis and J. Mitchell, "On the Sequential Nature of Unification", J. Logic Programming **1**, pp35-50 (1984).

[Eisi88] Norbert Eisinger, "Completeness, Confluence and Related Properties of Clause Graph Resolution", Doctoral Dissertation, SEKI Report SR-88-07, FB Informatik, University of Kaiserslautern FRG (1988)

[Eisi89] Norbert Eisinger, "A Note on the completeness of resolution without self-resolution.", Information Processing Letters **31**, pp323-326 (1989)

[Kowa79] Robert Kowalski, "Logic for Problem Solving", North Holland (1979)

[Mill90] Håkan Millroth, "Reforming Compilation of Logic Programs", doctoral dissertation, UPMAIL, Uppsala University, Sweden (October 1990).

[Ohlb90] Hans-Jürgen Ohlbach, "Compilation of Recursive Two-Literal Clauses into Unification Algorithms", Proceedings of AIMSA 1990, Albena, Bulgaria, (Wiley 1990).

[Powe88] David M. W. Powers, Lazaro Davila and Graham Wrightson, "Implementing Connection Graphs for Logic Programming", Cybernetics and Systems '88 (R. Trappl, Ed), Kluwer (April 1988)

[Powe89] David M. W. Powers, "Parallelized QuickSort with Optimal Speedup", submitted. An expanded version is available as SEKI Report-90-09, University of Kaiserslautern FRG.

[Powe90] David M. W. Powers, "Compartmentalized Connection Graphs for Concurrent Logic Programming I: Compartmentalization, Transformation and Examples", SEKI Report SR-90-16, University of Kaiserslautern FRG (1990).

[Powe91] David M. W. Powers, "Optimal Parallel Speedup of QuickSort and RadixSort both *in situ* and in Lists", to be submitted. An earlier version is available as SEKI Report-90-09, University of Kaiserslautern FRG.

[Wise84] Michael J. Wise and David M. W. Powers, "Indexing PROLOG Clauses via Superimposed Code Words and Field Encode Words", Proc. Int'l Symp. on Logic Programming, IEE Computer Society, pp203-210 (1984).

History of CONG and Related Research

1982	Dissatisfaction with PROLOG control for Machine Learning research. Specification of CONG including: Transformation away of PROLOG builtins; Trial proofs by indexing; Implementation of FEW indexing and tree search for CONG.
1983	Implementation of *half cut* in UNSW PROLOG. Reimplementation of FEW for PROLOG (*Wise).* *Compendium of Interesting PROLOG Programs*, DCS Report 8313, UNSW.
1984	*Indexing PROLOG Clauses via SCWs and FEWs*, Proc. Int'l Symp. on LP.
1985	Disk controler hardware implementation of SCW (*Jayasooriah*).
1986	Demonstration of lemmatization speedup in CONG (*Davila*).
1987	Prototype of RAA hardware implementation of SCW indexing (*Colomb*). Demonstration of pseudoresolution speedup.
1988	Prediction of optimal speedup for QuickSort. Definition of compartmentalized CONG. *Implementing CONG for LP*, Cybernetics and Systems '88.
1989	Simulation of optimal speedup for *in situ* QuickSort. Simulation of optimal speedup for *linked list* QuickSort. Determination of CONG heuristics for QuickSort Specification of Parallel Unification by Indexing and Hashing. *Parallelized QuickSort with Optimal Speedup*, SEKI Report 90-09.
1990	Compilation of CONG to theory unification Transformation away of multiple recursion Preliminary version of strong completeness proof for CCONG *Compartmentalized CONG for Concurrent LP*, SEKI Reports 90-16 & 17.