Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern

SEKI - REPORT

Compartmentalized Connection Graphs for Logic Programming I: Compartmentalization, Transformation and Examples

David M. W. Powers

SEKI Report SR-90-16

COMPARTMENTALIZED CONNECTION GRAPHS FOR CONCURRENT LOGIC PROGRAMMING

I: Compartmentalization, Transformation and Examples

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

SEKI Report SR-90-16

Abstract

The research reported in this paper and its sequels represents a revolt against the explicit and restricitve control of PROLOG and the present generation of Concurrent and Parallel Logic Programming Languages. It returns to the original Connection Graph paradigm of Kowalski and provides a methodology for logic programming in this framework.

An elementary analysis of where the expenses in executing a logic program occur shows how processing of each of the linear components in a proof (or execution trace) can be executed in (non-deterministic) logarithmic time within the CONG system. Our implementation demonstrates that lemmatization can result in even more dramatic improvement.

This paper deals primarily with recursion both in relation to connection graphs and in relation to Horn logic programs. In the first case a modified "compartmentalized" connection graph framework emerges, which allows proofs which are in general logarithmic in the size of a conventional connection graph proof. Furthermore, in the latter case we exhibit a technique allowing arbitrary recursive predicates in a logic program to be reduced to a canonical form involve only one single recursive predicate.

The method is demonstrated on standard PROLOG examples.

---

[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

PROLOG has proven an exceptionally interesting language. It allows programming in an entirely new style, or in a fairly conventional style. The new declarative logic programming style has a number of theoretical advantages, but the explicit pragmatic specification of control tends to pervade any major application. The SLD control of a Horn clause theorem prover made the use of automated reasoning technology practical – using a technique of known efficiency and recognized limitation. It also admitted the possibility of a conceptually simple 'cut' mechanism through which the programmer could maintain or regain control.

The most common generalizations of PROLOG have held fast to its general control regime, possibly allowing some relaxation, but mostly constraining the programmer to use even more explicit control. 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.

This work starts 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 to implement the environment efficiently 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 [Kowa79] 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 presents an elementary analysis of where the expenses in executing a logic program occur and shows how processing of each of the linear components in a proof (or execution trace) can be executed in (non-deterministic) logarithmic time within the CONG system. A lemmatization side-effect can result in even more dramatic improvement in specific cases.

Our techniques both provide an effective proof procedure for theorem proving with full clauses in the connection graph framework (strong completeness is examined in Part III) and show how standard PROLOG logic programs can be transformed to execute efficiently and more completely in this framework (characterizing proof length requirements in this case).

The final two sections examine the handling of logic programming control and metapredicates in the context of a theorem prover which does not depart from pure logic in any way.

Model

Restrictions

Non-deterministic proof

At this point we will consider only a non-deterministic model. We assume that through some oracle, heuristics, or committed-choice style control, we find a shortest proof if one exists. This assumption is primarily useful once we leave the realm of propositional Horn clauses. The proof is the trace of a non-deterministic execution, the execution itself is assumed to have found the proof at no cost (by oracle, etc.) and the cost is therefore the cost of checking it, including unification testing of the grounding of the clauses used in the proof.

Horn clauses

We furthermore pay particular attention to the special case where we do have only Horn clauses. This restriction leads to a convenient linear bound on the length of the shortest proof when used in combination with a restriction to ground forms, and admits useful properties even when extended to full predicate forms. We note the possibility of broadening our restriction from Horn clause systems to arbitrary clause systems with linearly refutable grounded predicate forms. Indeed it is hypothesized that what we might more conventionally regard as 'programs', as opposed to 'automated reasoning applications', do admit such linear grounded forms (cf. SATCHMO).

Unification

The process of unifying terms is used in the generation of our proof. A proof need not be completely ground – some variables may not be restricted in the course of a particular proof. Unification of terms, with identification of the substitutions that will make the terms identical, can be achieved in linear (parallel or sequential) time. Unification is one of the main expenses of a logic programming system, and is a primary contributor to LIPS rates, the reciprocal of the overhead being effectively what is measured as Logical Inferences Per Second.

It is therefore a major topic of Part II of this series of reports. It turns out that deferring unification is the one of the best ways of dealing with it. Better still is avoiding it completely by using indexing. In this paper we will continue to use resolution steps as the basic metric.

Costs

Let us assume that the number of literals is n, that the maximum depth of any term is d, that the maximum number of copies used of any clause is c. Because of our assumption that we are using a linearly resolvable system of Horn clauses, we know

that it is possible to express a resolution proof using each ground instance of a clause at most once, and we therefore assume that c is an indicator of the amount of recursive expansion and that we can ignore 'subroutine' style usage of a predicate – which could be compiled into lemmata, macros, theories or the like, and are not primary contributors to the complexity of the proof.

We can now roughly characterize the complexity of the proof as bounded by $c*n*d$ operations. 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.

Algorithms

Standard Connection Graph

Links

A set of clauses to be proven inconsistent are linked into a graph by connecting clauses with a *link* whenever they have unifiable complementary literals. The links may at times most conveniently be regarded as connecting clauses. 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.

Under this definition a link is formally defined only between distinct clauses, although loosely used it may, when the context permits, include pseudolinks. In certain syntactic contexts all forms of links, including, e.g. factor links, may be intended. When we want be absolutely clear we can refer to these links between complementary literals of distinct clauses as resolution links.

Pseudolinks

The connections between unifiable literals of opposite sign within a single clause are termed *pseudolinks*. These links represent the potential for copies of the clause to resolve, but are themselves never actually resolved on. They are restricted to inheritance and can inherit to standard resolution links or to new pseudolinks. The substitution giving rise to the most general unifier of the linked literals is associated with the pseudolink.

In this case unification is interpreted as unification in distinct environments. That is, unifiable after renaming of the variables of one or both clauses, or equivalently, weakly unifiable. The sense of this is that clauses are internally universally quantified and the same variable name represents distinct variables when it occurs in distinct clauses. This definition of pseudo link arises as we wish to capture the fact that distinct copies of the clause would have a resolution link between the corresponding terms.

Note that the strong unifier, representing the potential to unify in the same environment without renaming, represent the substitution under which the linked terms make the clause tautologous. Some authors, rather unhelpfully, use the term tautology link to denote any pseudolink, even those that admit no tautology or have interpretations (substitutions) which are not tautologous.

Resolution rule

Upon resolving on a link, the *resolvent* is the clause containing copies of all literals other than those linked by the selected link substituted by the substitution associated with the resolved link. The linked clauses (resp. terms) are termed the *positive* and *negative parents* according to the sign of the respective linked term. Once it has been resolved upon, a link is deleted, as there is no need ever to repeat this resolution step. There is a sense in which a link already represents the resolvent and in an implementation the same structure may actually be used with just the complementing of a bit to indicate the difference: the resolvent literally replacing the link.

Inheritance rule

Following resolution on a link, the copies of the literals need to be connected in to the rest of the graph. Rather than trying unification with all possible complements, the potential of the connection graph comes from the inheritance of new links from those which have not yet been resolved on (or otherwise deleted) and remain extant as links. The links represent work remaining to be done.

Upon copying a literal, all links associated with that literal are also copied, but the type will change to that appropriate to the new link. Thus links (and pseudolinks) between other terms of the parents will inherit as pseudolinks, and pseudolinks impinging on one of the parent terms will inherit to links from the parent term to the copy of the other term.

Factoring rule

Applied to general clauses (unrestricted to Horn), resolution is incomplete without factoring (or merging). This means that terms of the same sign in the same clause can be unified and all but one discarded as redundant. In the case of merging, no information is lost, and there is no advantage in delay. But in the case of multiple factorable literals in the clause, a whole family of factors would be generated, along with all inherited successors. The obvious, but explosive, factoring rule is to insert all possible factors as soon as factoring is possible – viz. initially, and when a resolvent inherits unifiable terms of the same sign.

As we defer discussion of factoring to later parts of this series, we can assume this naive approach to factoring in relation to any discussions without the Horn restriction.

Proofs

The clauses in a connection graph are inconsistent if and only if there exists a sequence of resolutions on links (inheriting and factoring as required) such that the empty clause (containing no literals) is derived.

Purity deletion

An important pruning of the graph may be achieved by observing that clauses containing a literal without links cannot contribute to a derivation of the empty clause. Such a term and the containing clause are said to be *pure*. Pure clauses and associated links may thus be deleted.

Tautology deletion

A clause which is true in all interpretations can never be essential to any proof. Thus tautologous clauses may in principle also be deleted. However, it is possible that in the connection graph paradigm such a clause is part of a bridge of remaining links essential to a proof, and the associated links cannot indiscriminately be discarded. Moreover, tying links together to inherit new links may lead to cyclic behaviour.

Precise conditions have been defined under which tautology deletion is safe, but in general it is unnecessary, and tautologies are moreover rare in logic programs. Thus we define the connection graph without tautology deletion.

Subsumption deletion

A clause which is a special case of another clause is said to be subsumed by it and may also be deleted. However similar problems arise and similar special conditions must be adhered to to make use of it. Again we prefer to do without for the present purposes.

Logic Programming

The execution of straightforward cutless PROLOG programs without library predicates is straightforward in a connection graph theorem prover. However note that without explicit control and in the absence of adequate heuristics we must for now appeal to an oracle to choose an appropriate sequence of link resolutions.

At this point, however, we can already observe some of the advantages of programming in a control free (Horn) theorem prover. In particular, the macro effect means that in some sequences of resolutions performed toward the solution of one goal may actually have produced a *lemmas* as resolvent which is useful in the solution of other goals.

This is not pure theory, but emerged in the very first implementation of CONG as the following example shows. Here we see a very clear advantage of CONG over PROLOG.

Example of lemmatization

Note that in the listings irrelvant lines of the trace (including all E, R and F lines of the PROLOG trace) are removed to give a fairer indication of what is achievable with cuts and tail recursion optimization and a better basis for comparison, not to mention keeping it to a reasonable length. Moreover since comparison of the general look of the proofs, and in particular their shape and relative lengths, is the main purpose of showing the listings, lines have been truncated to avoid wrapping them.

Figure 1 illustrates how once CONG has expanded the unit clause up to a certain size, so that in effect it can handle lists of that size in one hit, it need never repeat the work. For larger lists it just keeps on expanding from where it was up to.

The PROLOG proof procedure traced in figure 2, by contrast, always starts from scratch for each new goal and clearly redoes a great deal of work even in this small example<– hence the serrated effect.

```
> cat append.cong.1
   append([a,b,c,d,e],[f,g,h],L),
   append([1,2,3,4,5,6,7,8],M,[1,2,3,4,5,6,7,8,9,10,11,12]),
   append(N,[f,g,h],[1,2,3,4,5,6,7,8,a,b,c,d,e,f,g,h])/?
append([],W,W).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).

> cong append.cong.1
CONG MQU/UKL$Revision: 1.3.1.3 $$State: Exp $
Copyright (C) DMWP,MQU 1983-84,87-89,90 Version$Date: 88/12/15 15:27:17 $
Cong> units, trace, go!
   append([H'1],Y'3,[H'1|Y'3]).
   append([H'1,H'5],Y'3,[H'1,H'5|Y'3]).
   append([H'1,H'5,H'9],Y'3,[H'1,H'5,H'9|Y'3]).
   append([H'1,H'5,H'9,H'13],Y'3,[H'1,H'5,H'9,H'13|Y'3]).
   append([H'1,H'5,H'9,H'13,H'17],Y'3,[H'1,H'5,H'9,H'13,H'17|Y'3]).
:- append([1,2,3,4,5,6,7,8],M'2,[1,2,3,4,5,6,7,8,9,10,11,12]),
   append(N'3,[f,g,h],[1,2,3,4,5,6,7,8,a,b,c,d,e,f,g,h])?
   append([H'1,H'5,H'9,H'13,H'17,H'21],Y'3,[H'1,H'5,H'9,H'13,H'17,H'21|Y'3]).
   append([H'1,H'5,H'9,H'13,H'17,H'21,H'25],Y'3,[H'1,H'5,H'9,H'13,H'17,H'21,H'25|Y'3]).

append([H'1,H'5,H'9,H'13,H'17,H'21,H'25,H'29],Y'3,[H'1,H'5,H'9,H'13,H'17,H'21,H'25,H'29|Y
:- append(N'3,[f,g,h],[1,2,3,4,5,6,7,8,a,b,c,d,e,f,g,h])?

append([H'1,H'5,H'9,H'13,H'17,H'21,H'25,H'29,H'33],Y'3,[H'1,H'5,H'9,H'13,H'17,H'21,H'25,H

append([H'1,H'5,H'9,H'13,H'17,H'21,H'25,H'29,H'33,H'37],Y'3,[H'1,H'5,H'9,H'13,H'17,H'21,H

append([H'1,H'5,H'9,H'13,H'17,H'21,H'25,H'29,H'33,H'37,H'41],Y'3,[H'1,H'5,H'9H'13,H'17,H'

append([H'1,H'5,H'9,H'13,H'17,H'21,H'25,H'29,H'33,H'37,H'41,H'44],Y'3,[H'1,H'5,H'9H'13,H'

append([H'1,H'5,H'9,H'13,H'17,H'21,H'25,H'29,H'33,H'37,H'41,H'44,H'47],Y'3,[H'1,H'5,H'9H'

** yes **

L = [a,b,c,d,e,f,g,h]
M = [9,10,11,12]
N = [1,2,3,4,5,6,7,8,a,b,c,d,e]
```

Figure 1. *Triple append executed under CONG*

```
> cat append.prolog.1
append([],W,W).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
trace append!
?- append([a,b,c,d,e],[f,g,h],L),
   append([1,2,3,4,5,6,7,8],M,[1,2,3,4,5,6,7,8,9,10,11,12]),
   append(N,[f,g,h],[1,2,3,4,5,6,7,8,a,b,c,d,e,f,g,h]).

> prolog append.prolog.1
UNSW PROLOG MQV4.3 (C) 1983,5,7
C|>append([a, b, c, d, e], [f, g, h], [a, ..._5])
C||>append([b, c, d, e], [f, g, h], [b, ..._9])
C|||>append([c, d, e], [f, g, h], [c, ..._13])
C||||>append([d, e], [f, g, h], [d, ..._17])
C|||||>append([e], [f, g, h], [e, ..._21])
C||||||>append([], [f, g, h], [f, g, h])
C|>append([1, 2, 3, 4, 5, 6, 7, 8], _1, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
C||>append([2, 3, 4, 5, 6, 7, 8], _1, [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
C|||>append([3, 4, 5, 6, 7, 8], _1, [3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
C||||>append([4, 5, 6, 7, 8], _1, [4, 5, 6, 7, 8, 9, 10, 11, 12])
C|||||>append([5, 6, 7, 8], _1, [5, 6, 7, 8, 9, 10, 11, 12])
C||||||>append([6, 7, 8], _1, [6, 7, 8, 9, 10, 11, 12])
C|||||||>append([7, 8], _1, [7, 8, 9, 10, 11, 12])
C||||||||>append([8], _1, [8, 9, 10, 11, 12])
C|||||||||>append([], [9, 10, 11, 12], [9, 10, 11, 12])
C|>append([1, ..._57], [f, g, h], [1, 2, 3, 4, 5, 6, 7, 8, a, b, c, d, e, f, g, h])
C||>append([2, ..._61], [f, g, h], [2, 3, 4, 5, 6, 7, 8, a, b, c, d, e, f, g, h])
C|||>append([3, ..._65], [f, g, h], [3, 4, 5, 6, 7, 8, a, b, c, d, e, f, g, h])
C||||>append([4, ..._69], [f, g, h], [4, 5, 6, 7, 8, a, b, c, d, e, f, g, h])
C|||||>append([5, ..._73], [f, g, h], [5, 6, 7, 8, a, b, c, d, e, f, g, h])
C||||||>append([6, ..._77], [f, g, h], [6, 7, 8, a, b, c, d, e, f, g, h])
C|||||||>append([7, ..._81], [f, g, h], [7, 8, a, b, c, d, e, f, g, h])
C||||||||>append([8, ..._85], [f, g, h], [8, a, b, c, d, e, f, g, h])
C|||||||||>append([a, ..._89], [f, g, h], [a, b, c, d, e, f, g, h])
C||||||||||>append([b, ..._93], [f, g, h], [b, c, d, e, f, g, h])
C|||||||||||>append([c, ..._97], [f, g, h], [c, d, e, f, g, h])
C||||||||||||>append([d, ..._101], [f, g, h], [d, e, f, g, h])
C|||||||||||||>append([e, ..._105], [f, g, h], [e, f, g, h])
C||||||||||||||>append([], [f, g, h], [f, g, h])
N = [1, 2, 3, 4, 5, 6, 7, 8, a, b, c, d, e]
M = [9, 10, 11, 12]
L = [a, b, c, d, e, f, g, h]
```

Figure 2. *Triple append executed under PROLOG.*

Extended Connection Graph

Links and Pseudolinks

The extended connection graph is a graph with the same clauses, links and pseudolinks as the standard form. It may however have additional types of links, or additional subclassification of the standard clauses, terms and links. In this section we introduce the new types of links and the concept of *filters* which governs the use of subclassifications of links.

The links we introduce now are analogous to the original resolution links and pseudo links, but connect literals of the same sign rather than literals of opposite sign.

Factor-links

Two unifiable literals in the same clause may be linked by a *factor link* which has a substitution associated with it. In this case unifiability without renaming is required. The substitution must produce the most general unifier of the terms within the environment of a single clause as it defines the condition under which the terms are identical and can be *merged.* Only identical terms may be merged. Unifiable terms indicate that under the corresponding substitution the clause has a simpler special case, called a *factor.*

Factor links are used so that factor possibilities can be quickly identified, but also so that factoring can be delayed until one of the factor-linked terms is resolved on when the factorizing requirement can be instantly detected. They can also participate in restrictions (see below) – e.g. that terms without factor-links should be resolved before terms with factor-links.

Factoring is probably not necessary in logic programming applications. Factoring must be handled very carefully if the purity and tautology deletion rules are both being used. It is safe, but explosive, if performed as soon as the possibility exists.

Subsumption links

Special links can be kept between unifiable literals of the same sign from different clauses, which will help when subsumption tests are performed, and will inherit as factor links if both terms end up in the same descendant clause. These we call *subsumption links* for want of a better name. As we have mentioned, subsumption is dangerous in the connection graph paradigm. Corresponding links may have been used for a term from one clause but not the other – taking the union can cause looping and taking the intersection can cause incompleteness (losing a potential proof) [Ei86,88].

Orderings & Restrictions

Within the scope of the extended connection graph procedure there is the possibility of imposing additional conditions to filter out the less helpful choices of next link to resolve (or factor) on. The procedure has a property called *confluence* which says that whichever non-deterministic path through proof-space you take there is always a common successor graph. Soundness says the procedure is valid, completeness says there is a way of finding a proof if there is one, but there is nothing said about getting you there.

Strong completeness adds the condition that the procedure guarantees to get you there. Even strong completeness however may not even be strong enough. Ideally we want our oracle or heuristics to tell us the fastest way to get our proof. Strong completeness is discussed in detail in [Eisi88] and in Part III of this series.

For now we note that there are two types of filters which can be distinguished. Those which say *never* are said to be *restrictions*, and those which say *first* are called *orderings*. The first are quite dangerous – they do not preserve confluence and may actually exclude the part of the search space which contains the proof, affecting completeness. The second are more advisory and preserve the properties of soundness, completeness and confluence. Whether they guarantee termination, finding the proof, is another question.

Ordering restrictions aim to force towards the proof. An ordering which systematically avoids any possibility of the proof escaping is said to be *exhaustive*. An exhaustive ordering leads automatically to strong completeness. Unfortunately none are known for either the standard or the extended connection graph. There may be none!

An example of a restriction is the *unit restriction*. Unit clauses are clauses with exactly one literal. The unit restriction says all resolution steps undertaken must involve.

An example of an ordering strategy is predicate elimination: you resolve only on terms with a particular functor, resolving away occurence by occurence, until none are left. That it can't help is obvious if you consider that any program or problem could be rewritten with a single unary predicate `p(..)` wrapped around the original literals. In this case the ordering gives no help.

Compartmentalized Connection Graph

We turn now to a modification of the connection graph algorithm which reverses some of the basic principles and changes some of its properties dramatically.

Pseudolinked clauses

Recall that in the standard and extended definitions we are permitted only to resolve on links and pseudolinks may only be inherited from. Here we change this around and specify that in the case of clauses with pseudolinks the only permitted operation is resolution on the pseudolink and links may only be inherited from.

Self-resolving clauses with pseudolinks are restricted to self-resolution on these pseudolinks, and links between clauses containing unresolved pseudolinks are declared *illegal*. This ensures that pseudolinks cannot inherit and allows the distinction between two types of operation, resolution on normal links and resolution on pseudolinks, to be extended to a differentiation of distinct phases of the algorithm – a *static* phase in which legal normal links are resolved, and a *dynamic* phase in which legal pseudolinks are resolved.

Static purity

A clause is of no further use in the current static phase once all legal links from a single term have been deleted. The clause is then said to be *statically pure*.

Corridors – links connecting compartments

Clauses with and without pseudolinks are said to be respectively in the dynamic and static compartments. The links between clauses in different compartments are illegal and are said to define a corridor between the two compartments.

Clearly, if the corridor is empty, the algorithm will terminate at the end of the current resp. next static phase, and there is no need to continue work in any current dynamic phase. This is the way termination is achieved in satisfiable clause sets, although it cannot be guaranteed in general, but is usual for logic programs.

Self-resolution

As was implicit in our definition of pseudolink originally, a *self-resolvent* is the result of a resolution of two distinct copies of the clause with variables renamed apart. A consequence of this is that *self-resolution* seems to introduce additional variables. In fact severy resolvent always has a brand new (but possibly empty) set of variables. However we tend to reuse the names. In the case of self-resolution we may be forced to think up new ones! In the CONG and PROLOG examples it will be noted that the systems always distinguish variables uniquely with numbers.

Positive-to-parent inheritance

The definition of self-resolution allowed in the extended connection graph has the property that the pseudolink inherits to only one normal link after its resolution. There is a sense in which it is only half there to inherit because it is currently selected for resolution and deletion, and if we allowed both descendants of the parent literals in the resolvent to inherit links to the parent literals, resolving on these links would produce identical clauses (mutually subsumable with links to same literals).

We call the original recursive clause the order 1 form, the first resolvent is the order 2 form. Resolving on either of these normal links inherited from the pseudolink would produce an order 3 form, while resolving on the pseudolink inherited from the original pseudolink would produce an order 4 form. The order is in fact the count of the number of copies of other literals of the original clause which are found in a descendant in the absence of factoring.

In fact, we have defined self-resolution in terms of copies. [Eisi88,p128] shows that the self-resolution operation can be added to the connection graph and is then equivalent to making a copy and inheriting links, performing a resolution on one of the two descendants of the pseudolink, and then subsuming away the copy (although it could equally well have been the original). The arbitrary choice of which descendant to resolve on (or equally which copy to resolve away) affects whether the pseudolink

inherits from the positive term of the resolvent to the negative term of the original parent, or vice-versa.

Our positive-to-parent inheritance rule arbitrarily disambiguates this in a consistent fashion by requiring that the link to inherit goes from the positive term of the resolvent to the negative term of the parent. This has, in a sense, the side effect of biasing the links in the corridor toward goal direction.

Interestingly [Eisi88] only introduces self-resolution to simplify proofs of formal properties, and elsewhere [Eisi89] shows that it is an unnecessary operation.

Delayed composition

For various reasons pertinent to both efficiency and strong completeness, we note that the composing of new substitutions need not be performed immediately. The link may simply be inherited and retain information about the substitutions which compose to give the associated substitution.

In the case of links which may not actually inherit with a valid substitution this can mean *false* links are present in the graph. False links may delay recognition of purity of a clause. On the other hand, false pseudolinks may accelerate termination of the current static phase and thus tighten the ordering conditions relating to pseudolinks.

In this context the orderings are regarded as more important than purity, which is in any case largely superceeded by the concept of static purity.

We specify that composition for links is done only in the static phase, and possibly *lazily* (that is a false link may actually be resolved on), and that composition for pseudolinks is done only in the dynamic phase, and always *eagerly* (that is a false pseudolink may never actually be resolved on).

Example of Doubling

We again use append as our example to show the speed at which append can now get up to big lists – the number of resolutions being logarithmic in the length of the list. (The number of lines and the general flavour is again more important than the details and lines have thus again been truncated for clarity.)

```
>cat append
append([],W,W).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
append([a,b,c,d,e,f,g,h],[i,j,k,l],L)/?

>cong append.cong.2
CONG MQU/UKL$Revision: 1.3.1.3 $$State: Exp $
Copyright (C) DMWP,MQU 1983-84,87-89,90 Version$Date: 88/12/15 15:27:17 $
Cong> pseudores, hyper, minterm, hyperfac, unit, trace, go!
:- append([a,b,c,d,e,f,g,h],[i,j,k,l],L'1)/?
   append([H'1|X'2],Y'3,[H'1|Z'4]):-
                          append(X'2,Y'3,Z'4).
   append([],W'1,W'1).
   append([H'1,H'5|X'6],Y'3,[H'1,H'5|Z'8]):-
                          append(X'6,Y'3,Z'8).
:- append([b,c,d,e,f,g,h],[i,j,k,l],Z'5)?
   append([H'1,H'5,H'9,H'13|X'14],Y'3,[H'1,H'5,H'9,H'13|Z'16]):-
                          append(X'14,Y'3,Z'16).
:- append([d,e,f,g,h],[i,j,k,l],Z'13)?
:- append([c,d,e,f,g,h],[i,j,k,l],Z'9)?

append([H'1,H'5,H'9,H'13,H'17,H'21,H'25,H'29|X'30],Y'3,[H'1,H'5,H'9,H'13,H'17,H'21,H'25,H
                          append(X'30,Y'3,Z'32).

append([H'1,H'5,H'9,H'13,H'17,H'21,H'25,H'29,H'33,H'37,H'41,H'45,H'49,H'53,H'57,H'61|X'62
                          append(X'62,Y'3,Z'64).
:- append([],[i,j,k,l],Z'33)?
** yes **
L = [a,b,c,d,e,f,g,h,i,j,k,l]
```

Figure 3. *Self-resolving append executed in CONG.*

Example of Explication

The big advantage of compartmentatilzation over any other resolution theorem proving or logic programming effect is the doubling effect realizable on explicitly recursive clauses as just shown. Later we show how multiply recursively defined PROLOG predicates can be reduced to a canonical form to reduce the amount of explicit recursion. The canonical form has only a single singly recursive clause and doubling produces only one form of a given size.

But multiply recursively defined predicates also hide implicit recursion, where one clause can call the other and vice versa (in PROLOG terms). Thus not only do we have a family of forms generated from each one, but we can have an exponential number of such families. Even once the pseudolinks have been resolved on, the clauses are in general cross linked, and resolution on any one of those links inherits the others as a pseudolink.

We illustrate with a set of Horn clauses which is beyond the power of PROLOG and which has this property.

```
q(g(f(g(f(g(f(a)))))).
q(X) :- q(f(X)).
q(Y) :- q(g(Y)).
:- q(a).
```

Note that if the positive and negative terms were interchanged, so that the query was complex and the unit clause simple, it would run under PROLOG. But as it stands PROLOG will only search for unit clauses of the form `q(f(f(..f(a)..)))`, which no `g` functors.

Resolving on the pseudolinks produces the first of a family of recursive clauses containing exclusively one functor or the other. But resolving on a link between the two recursive clauses produce the zero form of one of two families with alternating functors. Confluence and completeness guarantees that CONG can succeed in finding the proof in this way. But the explosion in the number of families is exponential.

We noted that the problem was the implicit recursion which was not already expressed by pseudolinks. Such recursion can also occur in clauses which are not directly recursive, but indirectly or *implicitly*. If we make a slight modification to the above algorithm we will see such an example.

```
q(g(f(g(f(g(f(a)))))).
q(X) :- p(f(X)).
p(Y) :- q(g(Y)).
:- q(a).
```

We would like to make this recursion *explicit* so that we may deal with it efficiently. Careful *ordering* of our choice of links in the compartmentalized connection graph can *explicate* such recursion.

Consider what happens if we use straight forward goal directed search as PROLOG does. We generate a sequence of goals: `p(f(a).` `q(g(f(a)).` `p(f(g(f(a))...` This will eventually find the unit clause in this case. If we used unit resolution in a data driven way we would produce a similar but reducing set of positive unit clauses. In both cases the process is linear in the size of the complex term. If we could *explicate* the recursion and then use *doubling* it could be done in a logarithmic number of steps.

We can achieve explication very simply in the static phase of the algorithm: we introduce a rule, technically an *ordering strategy*, which prohibits resolving on a link before it parent link (that is the link it was inherited from) is resolved upon. This stops the above unit resolution series after the first step. The generated unit clause has a link only because it was inherited from somewhere – one of the cross links. This forces resolving on the parent of the new link first, on a cross link, and forces generation of an explicitly recursive clause containing a pseudolink.

Characterization of Compartmentalization

We now characterize the Compartmentalized Connection Graph procedure more precisely in terms of ordering restrictions on a weakened version of the standard algorithm.

Formally, we have the standard connection graph algorithm, weakened by removal of two restrictions, and strengthened by definition of five ordering filters. These filters are presented in terms of the *freshness* and *legality* attributes on links, which are defined in the next section.

The standard connection graph procedure is weakened so that

(i) resolution on pseudolinks is permitted but

(ii) composition of unifying substitutions is delayed while a link or pseudolink is *illegal* (so illegal links may actually also be *false*).

It is strengthened with the *compartmentalizing* ordering filter which classifies the clauses into two *compartments*, those having pseudolinks, and those not. Links may also be totally in one compartment or another. If, however, they connect clauses from different compartments they are said to be in the corridor. The two compartments are processed in alternating phases of the algorithm, termed *dynamic* in the case when pseudolinks are being processed, and *static* when pseudolinked clauses are excluded. The *compartmentalizing* ordering filter

(i) prohibits resolving on a link involving a pseudolinked clause (i.e. before all pseudolinks of that clause have been resolved upon),

(ii) prohibits resolving on a link while.the parent link is still present (i.e. before it is *orphaned*),

(iii) prohibits resolving on a link while it is *fresh* (i.e. during the dynamic phase in which it became *legal*),

(iv) prohibits resolving on a *fresh* pseudolink (i.e. during the dynamic phase in which it was created), and

(v) prohibits resolving on a pseudolink while a normal link may legally be resolved upon (i.e. during a static phase).

Definitions

We now summarize the definitions of the terms we have introduced:

Links and Pseudolinks

Unifiable terms of opposite sign in distinct clauses are initially connected by *links*. Unifiable terms of opposite sign in the same clause are connected by *pseudolinks*. The

links and pseudolinks together with the clauses form the *connection graph.* The links basically keep track of potential resolutions and the pseudolinks of potential self-resolution or explicit recusrsion. Together they keep track of what work can and must be done to complete the proof.

False links

A link which is present in the graph but for which the associated substitution has not yet been composed and checked could be spurious. Such a link which turns out not to have a valid substitution (that is which does not connect unifiable literals) is termed *false.*

Legality and Freshness

Links which do not impinge on a pseudolinked clause against are designated *legal.* Links which become legal as a result of resolving on a pseudolink are marked *fresh,* as are links and pseudolinks which are inherited in this way. Only once all remaining pseudolinks are fresh are the fresh designations removed. The negative forms of fresh and legal are the obvious *stale* and *illegal.*

Resolution is prohibited on any link which is fresh or not legal. In fact, while any fresh links exist, only pseudolinks may be resolved upon; and while any stale legal links exist, no pseudolinks may be resolved upon. These leads us to distinguish alternate *phases* of the algorithm.

Static and Dynamic

The designation *static* is applied to the phases during which the biting part of the ordering requires that no pseudolink be resolved upon (some links are legal and not fresh). While a clause has no legal links it is said to be *s-pure* (for statically pure). While a term and its descendants have no legal links it is said to have been *s-eliminated.*

The term *dynamic* phase refers to a period during which the ordering requires that no ordinary link be resolved upon (all links are fresh or not legal).

Heuristics & Control

Use of Cuts and Guards

There are many features of PROLOG and its derivatives which take them away from first order logic. Perhaps the most obvious, most abused and most unnecessary are the cuts and guards. These should not be used in good programs, but should be relegated to the dirty library predicates which are used but not seen. Here we will omit discussion of them.

Note however that the primary use of cut is to achieve negation as failure. We consider *not* below. A secondary use is to indicate that no further search is required

since we are on the right track. This is heuristic information which we wish to express more clearly and will deal with in a subsequent paper. The third use is to indicate that only one solution is required. CONG actually has two modes, one in which one solution is found, one in which all solutions are found. The latter effect can also be achieved and controlled using *bagof* or *findall*, which are also discussed below.

Preprocessing

Extra-Logical Predicates

As in this series of reports a familiarity with PROLOG is assumed, no introduction is provided to the library predicates which are or are not standard. The purpose of this section is to indicate that the four major classes of such predicates can be handled by transformation or interpretation in the CONG paradigm.

Not

Moving from Horn clauses to the general paradigm gives us classical negation. Negation as failure is often, however, more characteristic of the way we think and program, and is a special case of both default reasoning and abductive reasoning. Again this is not discussed in detail in this paper, but we show the operational equivalence of *not* and *findall.*

We first show how `not` can be implemented given `findall`.

```
not(p(X)) :- findall(X, p(X), []).
```

And now we show how `findall` can be implemented given `not`.

```
findall(X, p(X), L) :- sbagof(X, p(X), [], L).

sbagof(X, p(X), T, L) :- c(p(X),p(X1)),
                         p(X1), not(in(p(X1),T),
                         sbagof(X, p(X), [p(X1)|T], L).
sbagof(X, p(X), T, L):-  not((p(X),not(in(p(X),T))))
```

The `in` predicate is the well known membership predicate (also known as `member`) and the `c` predicate operationally acts as a copy with renaming of variables. This is non-logical, but the equivalent effect can be achieved at a meta-level in the transformational approach.

This particular technique is not advocated especially, and alternative approaches will be examined in a later part.

Bagof, Setof and Findall

In the last section we showed that `findall` is in a sense equivalent to `not`. It is well known that it can also be simulated by append (and less well known that `assert` could be simulated by reversing `findall` given higher order unification). We show here, however, how the effect of these predicates can be obtained straightforwardly by

preprocessing and interpretation of the original program, without any additional assumptions.

In this exemplification of the technique we use a set of examples which is familiar, that of `father`, `mother`, `parent` and `ancestor` predicates. For compactness the predicate names are however abbreviated to their first letter. We could assume that any recursive predicates are in canonical form, viz. that they have exactly one singly recursive clause and exactly one unit clause. This is actually stronger than the condition used here which is simply that there are no multiply recursive clauses.

The technique is based on reexpressing the databases as lists, trivial at a metalevel and a common implementation technique for predicates in any case. So here we add new predicates with the bag of clauses associated with our `father` and `mother` databases.

```
f(mary,tom).                          m(mary,jane).
...                                   ...

b_f([f(mary,tom), ...])               b_m([m(mary,jane), ...]).
```

Any `bagof` or `findall` goal is now either directly transformed into the appropriate scanning and flattening of the bagged databases, or interpreted by a scanning procedure with similar access to the predicate representation. This is straightforward when the goal concerns database facts directly or through a datalog view:

```
p(C,X) :- f(C,X).
p(C,X) :- m(C,X).

cbagof(X, f(M,X), L) :-  b_f(LL),  scan(X, f(M,X), LL, L).

cbagof(X, p(M,X), [FL,ML]) :-      cbagof(X, f(M,X), FL),
                                   cbagof(X, m(M,X), ML).

bagof(X, P, L):- cbagof(X, P, GL), flatten(GL, L).

scan(X, P, [], []).
scan(X, P, [H|T], [H|L]):-         ©((X,P), (X1,P1)), X1 = H,
                                   scan(X, P, T, L).
scan(X, P, [], []).
scan(X, P, [H|T], L):-             ©((X,P), (X1,P1)), not(X1 = H),
                                   scan(X, P, T, L).
```

Note the use of `not`. The `equals` predicate is used in both the scan clauses only for the sake of clarity (and in this case symmetry), and is trivially defined.

The last example was a simple disjunctive example. We no exhibit an example which is both conjunctive and recursive.

```
a(C,X) :- p(C,X).
a(C,X) :- p(c,Y), a(Y,X).

cbagof(X, a(M,X), L) :-  cbagof(X, p(M,X), L).
cbagof(X, a(M,X), L) :-  cbagof(X, p(M,Y), PL),
                         scanap(X, Y, a(Y;X), PL, L).

scanap(X, Y, P, [], []).
scanap(X, Y, P, [H|T], HL|L]):-    ◙((X,Y,P), (X1,Y1,P1)), Y1 = H,
                                   cbagof(X1, P1, HL),
                                   scanap(X, Y, P, T, L).
```

Input/Output

Another fundamental clause of builtin predicates concerns I/O. In this paper we wish to add nothing to the work on logical approaches to I/O, but assume *streams* by which files are treated as lists.

Assert/Retract

The final major class of builtins to deal with are those which modify the clause database. We again defer full treatment of this, but note that the connection graph method already defines conditions under which new clauses may be added to the database and existing clauses may be removed. These may be used achieving certain monotonic effects, but in general we have to consider non-monotonicicty, and techniques proposed for backtrackable assert and retract are also applicable here and will be dealt with in a subsequent report.

Multiple recursion

We present our techniques for handling multiple recursion with a sequence of examples based on well known predicates.

Elimination of multiple recursive clauses

The predicate `flatten` (used above) would naturally have the form:

```
flatten([],[]).
flatten([H|T],L):- flatten(H,FH), flatten(T,FT),
                   append(FH,FT,L).
```

This has two directly recursive calls as well as an independently recursive subpredicate. In fact the append makes it o(n*n). And it seems that in avoiding the explicit append, it is naturally broken up into two separate types of single recursion one for flattening the head, the other for further processing of the tail.

```
flatten([],[]).
flatten([[]|T],L):- flatten(T,L).
flatten([[HH|HT]|T],L):- flatten([HH,HT|T],L).
flatten([A|T],[A|L]):- not(flist(A)), flatten(T,L).
```

This could therefore be reduced to a single tail recursive call!

```
flatflatten([],[]).
flatflatten(LI,LO):-      flat(LI,LO,LRI,LRO),
                          flatflatten(LRI,LRO).

flat([[]|T],L,T,L).
flat([[HH|HT]|T],L,[HH,HT|T],L).
flat([A|T],[A|L],T,L):-  not(flist(A)).
```

Tree search is typical and common case where the natural form has two singly recursive clauses.

```
find(X,t(A,X,B)).
find(X,t(A,Y,B)):- lt(X,Y), find(X,A).
find(X,t(A,Y,B)):- lt(Y,X), find(X,B).
```

It is straightforwardly expressed with a single singly recursive clause.

```
rfind(X,t(A,X,B)).
rfind(X,Y):- sfind(X,Y,Z), rfind(X,Z).
sfind(X,t(A,Y,B),A):- lt(X,Y).
sfind(X,t(A,Y,B),B):- lt(Y,X).
```

It is easy to see that this method can always be applied to predicates composed of at most singly recursive horn clauses. If the original clauses were not tail recursive, it might be necessary to add an index to the first new predicate so that the right choice for the second can be made:

```
a(0,0).
a(X,Z):- c1(X), a(X,Y), d1(Y,Z).
a(X,Z):- c2(X), a(X,Y), d2(Y,Z).
```

The above general framework can be transformed into the following:

```
a(0,0).
a(X,Z):- c(N,X), a(X,Y), d(N,X,Z).

c(1,X):- c1(X).                    d(1,X):- d1(X,Y).
c(2,X):- c2(X).                    d(2,X):- d2(X,Y).
```

We now turn from multiple singly recursive clauses within a predicate to multiple recursion within a single clause.

Elimination of multiple recursive goals

A typical case where two recursive calls seem to be required in the same clause is in quicksort. The partition predicate is similar to the find predicate, acting to create a tree (non-nil partitions). The sort predicate itself is naturally expressed with double recursion and append. This is reminiscent of our original flatten. (We don't bother showing choice of pivot – it isn't necessary in CONG.)

```
qsort([],[]).
qsort([H],[H]).
qsort(LI,LO):- part(LI,LLI,LRI),
               qsort(LLI,LLO), qsort(LRI,LRO),
               append(LLO,LRO,LO).
```

The append is again expensive but easily removed - by using e.g. difference lists. But for simplicity, let us not bother now. We can distinguish four stages in the processing of the recursive clause: p (partition), l (left recurse), r (right recurse), a (append).

However using dummy predicates with these names would only hide the recursion, and it would later reappear in all its glory (making implicit recursion explicit) or simply prevent exploitation of doubling of self-recursive clauses (without the use of a technique to expose recursion).

Can we flatten out this extra recursion by means of a simulated stack?

Yes! It is then exactly like `flatten`. Instead of giving `qsort` a list to sort, we give it a list of lists to sort & append. This is just `flatten` with the difference that sublists are only of one level, and that they are partitioned until they are singletons.

```
qflatten([],[]).
qflatten([H|T],L):- part(H,HH,HT), qflatten([HH,HT|T],L).
qflatten([[A]|T],[A|L]):- qflatten(T,L).

fqsort([],[]).
fqsort(X,Y):- qflatten([X],Y).
```

Note that this form of quicksort is actually more efficient than the original – as we have got rid of the `append`. It is moreover simpler than the difference list version, and it is tail recursive. The two recursive clauses can be combined using the technique of the previous subsection.

Again the method is completely general: part & append here represent the parts of the clause that come before and after the recursive calls. If there's something between the (two or more) calls, then use the multiple recursion reduction if necessary, with the dummy call coming before the in-between predicate(s). Note that order is in any case of no significance without SLD control – we just need to make sure we preserve our semantics, keeping corresponding parts of clauses together in some sense.

This defines a linear deterministic algorithm to reduce arbitrary recursive predicates to exactly one singly-recursive 3-literal form. We also reduce unit clauses to a single clause with database access link.

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

[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)

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