UPMAIL Technical Report No. 48
June 10, 1988

Condition Graphs

Jonas Barklund, Nils Hagner and Malik Wafin

Uppsala Programming Methodology and Artificial Intelligence Laboratory
Computing Science Dept., Uppsala University
Box 520, S-751 20 Uppsala, Sweden
+46–18–18 25 00

Electronic mail: JONAS@AIDA.UU.SE, NILSH@AIDA.UU.SE, MALIKW@AIDA.UU.SE

Abstract

A connection graph represents resolvability in a set of clauses explicitly. The corresponding connection graph proof procedure has the deficiency that the graphs may grow, becoming impractical to handle. We present a novel proof procedure for *condition graphs*, which is a refinement of connection graphs. Our idea is to control the growth of the graph by recognizing tests in clauses and translating them to restrictions on the resolvability of links. A link is not selected for resolution unless all its associated tests have been determined to be true.

The condition graph proof procedure exploits the inherent parallelism in logic programs. We present a parallel logic programming language with appealing theoretical properties, which is executed efficiently using the condition graph proof procedure. An abstract parallel machine for the language is defined. A realization of the machine on the Connection Machine, a massively parallel computer, is outlined.

The research reported herein was supported by the National Swedish Board for Technical Development (STU).

1. INTRODUCTION

Our goal is to efficiently execute programs in an expressive logic programming language. One way is to make use of the inherent parallelism in logic programs. For this purpose we introduce condition graphs and an associated proof procedure. Theorems for basic theoretical properties of the proof procedure are presented. Only informal proofs are given, to keep this a short paper.

We suggest a language, Uplog, to be the source for the condition graphs, although other languages could be considered as well, with minor modification of the proof procedure.

Throughout the work, a primary consideration has been the possibility to implement the system on a massively parallel computer, such as the Connection Machine.*

Section 2 gives a brief introduction to connection graphs. Section 3 defines condition graphs and the condition graph proof procedure. The Uplog language is introduced in section 4. The Condition Graph Inference Machine is defined in section 5. Finally section 6 contains a realization of the Condition Graph Inference Machine on the Connection Machine.

2. THE CONNECTION GRAPH PROOF PROCEDURE

The *connection graph* proof procedure [4,14,15] reflects the property of resolvability for a set of clauses, by explicitly relating resolvable clauses to each other. This is accomplished by using a graph structure for representing the clauses. Matching atoms of opposite polarity are connected by a link, labelled by the most general unifying substitution (mgu) θ.

A resolution cycle consists of the selection of an arbitrary link, resolving the clauses connected by the link (thereby applying θ) and connecting the resolvent to the graph. The links to the resolvent are *completely* determined by the links to the parent clauses and θ of the selected link (*link inheritance*).

Clauses in the graph, containing an unlinked atom, can be deleted from the graph according to the *purity principle* [16]. Thus, redundant clauses are easily detected and deleted. For completeness, factoring must also be applied.

The resolution cycle is repeated until one of the following conditions is met. If a resolvent is the *empty clause*, then a refutation has been found. If the *empty graph* is reached, then the proof procedure terminates with failure.

The connection graph proof procedure does not assume any particular selection strategy. Any link can, at every instant, be selected for resolution. To benefit from the parallelism in logic programs, it is natural to look into selection and resolution upon several links in a connection graph simultaneously. Doing so, it is difficult to ensure that the refutation completeness and confluence properties will hold [4]. We can do that, provided that proper restrictions are made for the graphs and the selection of links.

A Horn clause is a clause with exactly one atom in its left side and a goal clause is a clause with an empty left side. If clauses are restricted to Horn and goal clauses and connection graph derivations are restricted to be top-down (i.e., only links between goal clauses and Horn clauses are resolved upon), the proof procedure will be complete and confluent.

Even given refutation completeness and confluence, the link selection is still essential. If the "wrong" links are resolved upon, the proof procedure does not terminate and may in the worst case generate larger and larger graphs. It is of course not possible to detect all non-terminating programs, but to enable programmers to control the growth of the graph, we define a new class of connection graphs.

3. THE CONDITION GRAPH PROOF PROCEDURE

Connection graphs have been considered less appropriate for logic programming, particularly since the graphs can be extremely space-consuming. It is often the case that a formed

* Connection Machine is a registered trademark of Thinking Machines Corp.

resolvent is removed within one or a few cycles, since some atom constituting a simple test becomes pure. We note that if such a test had been regarded as a restriction on the resolvability (i.e., the link), the resolvent would never have been formed. This would have saved time as well as space.

Definition. A *condition graph* (CG) is a connection graph where a (possibly empty) conjunction of primitive tests (e.g., equality, non-equality, or arithmetic comparison) is associated with each link.

Given a connection graph, the corresponding condition graph is obtained by the following procedure: for each clause

$$C = (A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n),$$

form a conjunction

$$\varphi = (\neg A_{t_1} \wedge \ldots \wedge \neg A_{t_k} \wedge B_{t_{k+1}} \wedge \ldots \wedge B_{t_l})$$

where A_{t_i} and B_{t_j} are primitive tests of clause C. Remove all A_{t_i} and B_{t_j} from the clause C, yielding the clause D. Finally, associate φ with every link connected to an A_j. See fig. 3.1 for an example. ∎

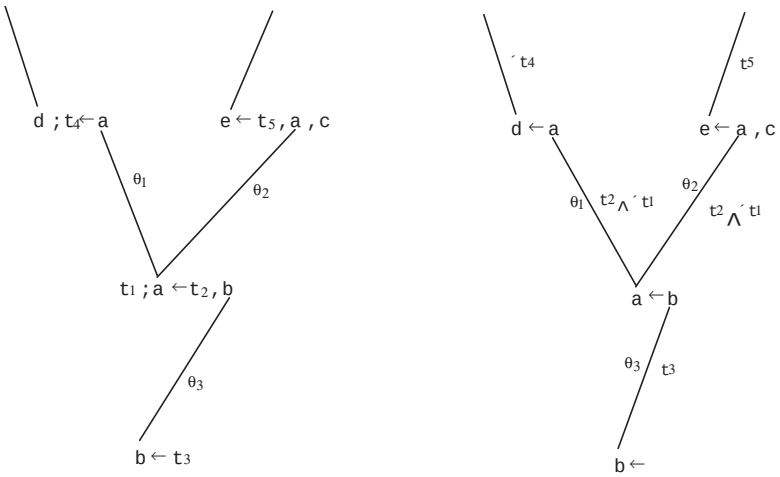

Fig. 3.1 Example of a connection graph and the corresponding condition graph.

Definition. A test is *determined* when it is instantiated enough to deduce its truth value. ∎

Definition. The *condition graph proof procedure* is defined as follows: select all or a subset of the links whose tests are determined to be true and remove all links whose tests are determined to be false. Then proceed as the connection graph proof procedure. When creating links to newly formed resolvents, the tests are taken from the links inherited from the parent clauses, in analogy with how its mgu is determined. ∎

Theorem. *The condition graph proof procedure is sound.*

Proof: By the definition of condition graphs, and the procedure to construct them, it follows that $C \leftrightarrow \varphi \wedge D$. It is clear that anything provable from D, when φ is determined as true, must also be provable from the original clause C. Soundness of the condition graph proof procedure thus follows from the soundness of connection graphs. ∎

Theorem. *The condition graph proof procedure is complete, provided that all tests in the graph are ground.*

Proof: Because all tests are determined, in each cycle every link must be either removed or selected. If a proof fails because the tests φ of some link fail, then it is because some test in φ is false. But this test is also an atom in the original clause C and a proof using C would therefore also fail. Thus, the result follows from completeness of the connection graph proof procedure [4,17]. ∎

It is trivial to provide a counter-example to completeness of the condition graph proof procedure when all tests are not determined, 'deadlock' may occur when there is insufficient data. The essential property is thus expressed in the following theorem.

Theorem. *The condition graph proof procedure is complete, provided that all tests in the graph are determined when they are used in the derivation.*

Proof: If a proof fails because the tests φ of some link is never determined, then it is a contradiction to the assumption that all tests are determined when they are used. ∎

This sounds trivial, but the non-trivial part is instead the burden laid upon the programmer.

4. THE UPLOG LANGUAGE

The condition graph proof procedure could be used with several different languages. Let us, for example, study a particular language: Uplog, a Horn clause language. It is similar to Flat PARLOG [5] or Flat Guarded Horn Clauses (FGHC) [22] but there is no commit operator. An Uplog clause has the form

$$H \leftarrow T_1 \wedge \ldots \wedge T_m \,\&\, B_1 \wedge \ldots \wedge B_n$$

where H and B_i are atoms, T_j are primitive tests, and & is a *fence*. Variables with multiple occurrences in H and non-variable terms in H are seen as tests, as in FGHC. The purpose of the fence is to syntactically separate the tests from the atoms. The declarative interpretation of the fence is equal to that of a conjunction. The actual set of primitive tests is not considered at this point.

Operationally, when an attempt is made to prove a goal atom H' which is unifiable with H under mgu θ; if all T_j are determined to be true, H is reduced to $\theta(B_1 \wedge \ldots \wedge B_n)$. If some T_j is determined to be false, then the proof fails. Otherwise, the proof suspends until some T_j becomes determined. The fence can thus be viewed as a sequential conjunction.

This language has the same programming power as, e.g., FGHC. Unlike FGHC, however, when the tests of several clauses are true, don't-know non-determinism is used instead of don't-care non-determinism. This means that no possible solutions are discarded. Another consequence is that we cannot assume that every attempt to communicate is made in a deterministic state. In particular, input/output is non-trivial [1]. For increasing efficiency, the tests of different clauses should be mutually exclusive when possible.

The language has good properties for program transformations, such as partial evaluation. E.g., fold and unfold operations can always be done. In the example below, proofs including clauses 1 and 2 will succeed iff proofs including clause 3 succeed. This is not true for the corresponding program in a committed choice language [6].

$$P \leftarrow T_1 \,\&\, Q \wedge R \tag{1}$$

$$Q \leftarrow T_2 \,\&\, S \tag{2}$$

$$P \leftarrow T_1 \wedge T_2 \,\&\, S \wedge R \tag{3}$$

5. THE CONDITION GRAPH INFERENCE MACHINE

The Condition Graph Inference Machine (CIM) is a specialization of the condition graph proof procedure for the Uplog language and top-down derivation.

CIM consists of a compiler and a run-time system which, in effect, is the theorem prover. The CIM compiler transforms an Uplog program into an *initial condition graph* (ICG), and preprocesses the ICG to obtain an optimized version of the program. The optimizations require *template goals* to be included in the program, that is, a declaration of the *entry points*. These template goals are substituted at run-time by the actual goals. We have considered different methods for preprocessing, e.g.; removal of *pure clauses* and *partial computation* [7].

After the CIM compiler has produced an ICG, control is transferred to the run-time system. An instance of the goal is *injected*, replacing the template goal by the actual goal. The run-time system is based on a *three-phase cycle*, which repeatedly performs the following steps:

1. *Select.* Define an *active* link to be a link whose tests have all been determined to be true. A link connected to a goal atom B is *unique* if it is the only link connected to B. For every goal G containing an atom B with multiple links of which at least one is active, select one active link L, and copy G to obtain G' (see fig.5.1). Remove L from G and remove every link of B except L from G'. Select all unique active links. (This preserves confluence and justifies our use of an irrevocable control strategy [4]. If resources are scarce, not all goals with multiple links need to be 'split'. In effect, when several alternative clauses for an atom remain after their tests are run, they will be executed in an or-parallel fashion, but the or-parallel goals will be started one by one.)
2. *Resolve.* Resolvents for selected links are produced simultaneously and independently added to the CG.
3. *Reduce.* Pure clauses are reduced after completion of the cycle.

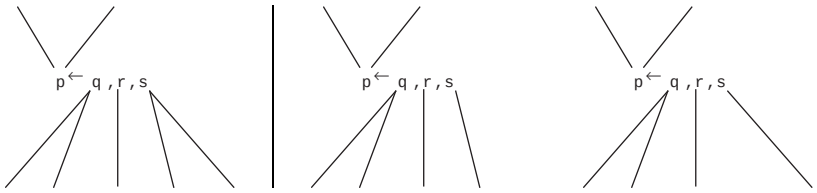

Fig. 5.1 Example of how a goal with multiple links to an atom is copied

There are three possible terminating states to a proof in CIM; (i) if any of the resolvents produced is the *empty clause* then a solution has been found, (ii) if there is no link to a goal atom then the proof has failed, otherwise (iii) the CG is *suspended*, that is, no tests are ground.

6. CIM ON THE CONNECTION MACHINE

The Connection Machine (CM) [10,11] is a novel parallel computer architecture, grown out of the observation that a fundamental bottleneck in computing is the communication between processors and memory.

In the Connection Machine each processor is comparatively simple and has direct access to only a small amount of memory. As a compensation, each processor has access to the flexible communication system, enabling it to communicate with any other processor in the machine. Every processor executes the same instructions, but can choose, depending on the context, to ignore incoming instructions. The instructions may either be local to every processor, which is very fast, or involve communication between all or some processors. The instructions are issued by a host machine, to which the Connection Machine acts like an active memory or a co-processor. A brief overview of our mapping of CIM on CM follows, more details can be found elsewhere [1].

Languages proposed for the Connection Machine include versions of Common Lisp and C [20] extended with data parallel primitives. We are unaware of any logic programming languages proposed for the Connection Machine, but DAP-Prolog [12] is a language for another SIMD machine.

Representation of CIM

The Connection Machine is a natural target machine for implementing the CIM, because the clauses, tests, and links in a condition graph can be conveniently mapped on Connection Machine processors. There are of course several ways to do it and this is a brief outline of our current approach. A *primitive test* is represented by one processor. During each inference cycle it will attempt to do its test and may either succeed, fail, or suspend. A *link* is represented by a "coordinating" processor, a set of tests, a set of unifications, and a set of atoms. An *atom* is represented by a "coordinating" processor and a set of links. A *goal clause* is represented by a "coordinating" processor, a set of atoms, and a set of tuples $\langle \text{name}, \text{variable} \rangle$ (for reporting success).

Execution

The three phases in CIM are mapped to corresponding instruction sequences on the CM. An *incremental garbage collection* is provided by the reduction step of the cycle. Algorithms for parallel unification have been developed and are presented elsewhere [2].

7. SUMMARY AND RELATED WORK

Condition graphs are a novel extension to connection graphs, with a corresponding proof procedure. They are, due to explicitly relating resolvable clauses and preventing uncontrolled growth, well suited for use in parallel inference systems. We have presented theorems and informal proofs for basic properties of the proof procedure.

Uplog is a Horn clause language, taking advantage of the condition graph proof procedure for efficient execution. It introduces the concept of tests instead of guards as in contemporary committed-choice languages.

The Connection Machine is especially well suited to handle graph structure problems. This, in addition to the similarities in synchronization, makes CM a natural and powerful computer architecture for an implementation of CIM.

COALA is an implementation of a parallel inference machine based on the CG proof procedure [8]. P-Prolog [23] is a language which like Uplog allows don't-know non-determinism.

Taylor's work on compilation of Concurrent Prolog to decision graphs [19] provided some inspiration for the work.

A completely different approach to logic programming on Connection Machines is Nova Prolog by Barklund and Millroth [3]. Prolog is extended with a new parallel data structure and a matching control structure.

8. FUTURE WORK

Some interesting problems for our future research are: introducing some user-defined tests in Uplog, partial evaluation of Uplog, a pilot implementation of CIM on the Connection Machine, and developing an inference machine for a language with a larger subset of FOPC using condition graphs.

We are also studying compilation of KL1 (a further development of FGHC) [13,21] to Condition Graphs, as an alternative to Uplog [1].

ACKNOWLEDGEMENTS

The authors want to thank their colleagues at UPMAIL for providing a fine research environment, cakes, and table tennis games. Professors Sten-Åke Tärnlund and Åke Hansson

gave many important comments on drafts of this paper. Marianne Ahrne improved the language significantly. Syracuse University and in particular professor Alan Robinson provided computer equipment for some of our related work, those experiences were important for this paper. Finally, without the continuous support from our families, this work would never have been done.

REFERENCES

1. J. Barklund, N. Hagner, M. Wafin, "KL1 in Condition Graphs on a Connection Machine," *UPMAIL Technical Report* 51 (Uppsala: Uppsala University, Computing Science Dept., May 1988).
2. J. Barklund, H. Millroth, "Parallel Unification," *UPMAIL Technical Report* 46 (Uppsala: Uppsala University, Computing Science Dept., June 1988).
3. J. Barklund, H. Millroth, "Nova Prolog," *UPMAIL Technical Report* 52 (Uppsala: Uppsala University, Computing Science Dept., June 1988).
4. N. Eisinger, "What You Always Wanted to Know about Clause Graph Resolution," *8th Conference on Automated Deduction*, ed. J. Siekmann, Lecture Notes in Computer Science 230 (Berlin: Springer-Verlag, 1986), 316–36.
5. I. Foster, S. Taylor, "Flat Parlog: A Basis for Comparison," *Report CS87-13* (Rehovot: Weizmann Institute of Science, 1987).
6. K. Furukawa, A. Okumura, M. Murakami, "Unfolding Rules for GHC Programs," *Workshop on Partial Evaluation and Mixed Computation*, ed. D. Bjørner (1987).
7. Y. Futamura, "Partial Computation of Programs," *RIMS Symposia on Software Science and Engineering 1982*, ed. E. Goto et al., Lecture notes in Computer Science 147 (Berlin: Springer-Verlag, 1983), 1–35.
8. I. Futó, C. Percebois, I. Durand, C. Simon, B. Bonhoure, *Simulation Results of a Multiprocessor Prolog Architecture Based on a Distributed AND/OR Graph* (Toulouse: Université Paul Sabatier, Laboratoire Langages et Systèmes Informatiques, 1986).
9. N. Hagner, M. Wafin, "Parallel Logic Programming using Condition Graphs," *UPMAIL Undergraduate Theses* (Uppsala: Uppsala University, Computing Science Dept., 1988).
10. W. D. Hillis, *The Connection Machine* (Cambridge, Mass.: MIT Press, 1985).
11. W. D. Hillis, G. L. Steele Jr., "Data Parallel Algorithm," *Communications of the ACM* 29 (December 1986): 1170–1183.
12. P. Kacsuk, A. Bale, "DAP Prolog: A Set-oriented Approach to Prolog," *Computer Journal* 30 (October 1987): 393–403.
13. Y. Kimura, T. Chikayama, "An Abstract KL1 Machine and Its Instruction Set," *1987 Symposium on Logic Programming*, ed. S. Haridi (Washington, D.C.: IEEE Computer Society Press, 1987), 468–77.
14. R. Kowalski, "A Proof Procedure Using Connection Graphs," *Journal of the ACM* 22 (October 1975): 572–595.
15. R. Kowalski, *Logic for Problem Solving* (New York: North Holland, 1979).
16. J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM* 12 (January 1965): 23–41.
17. J. Siekmann, W. Stephan, "Completeness and Soundness of the Connection Graph Proof Procedure," *Bericht 7/76* (Karlsruhe: Fakultät Informatik, Universität Karlsruhe, 1976).
18. G. L. Steele Jr., W. D. Hillis, "Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing," *1986 ACM Conference on Lisp and Functional Programming*, ed. W. L. Scherlis and J. H. Williams (New York: Association for Computing Machinery, August 1986), 279–97.

19. S. Taylor, E. Shapiro, "Compiling Concurrent Logic Programs into Decision Graphs," *Report CS87-12* (Rehovot: Weizmann Institute of Science, 1987).

20. "Introduction to Data Level Parallelism," *Technical Report Series* TR86-14 (Cambridge: Thinking Machines Corp., April 1986).

21. S. Uchida, "Inference Machines in FGCS Project," *ICOT Technical Report* TR-278 (Tokyo: Institute for New Generation Computer Technology [ICOT], 1987).

22. K. Ueda, "Guarded Horn Clauses," *ICOT Technical Report* TR-103 (Tokyo: Institute for New Generation Computer Technology [ICOT], 1985).

23. R. Yang, H. Aiso, "P-Prolog: A Parallel Logic Language Based on Exclusive Relation," *Third International Conference on Logic Programming*, ed. E. Shapiro, Lecture Notes in Computer Science 225 (Berlin: Springer-Verlag, 1986), 255–269.

APPENDIX

This is an example of an Uplog program for concatenation of lists (fig. 8.1), the corresponding condition graph (fig. 8.2), and the corresponding graph on CIM (fig. 8.3 [1].

$$Conc(x, y, z) \leftarrow x = \emptyset \,\&\, y = z$$
$$Conc(x, y, z) \leftarrow x = a.x' \,\&\, z = a.z' \wedge Conc(x', y, z')$$

Figure 8.1. List concatenation program.

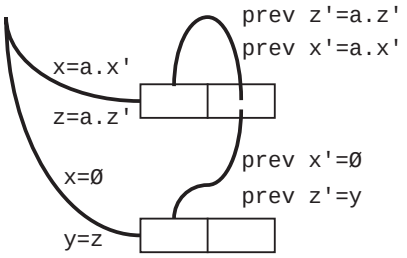

Figure 8.2. List concatenation graph.

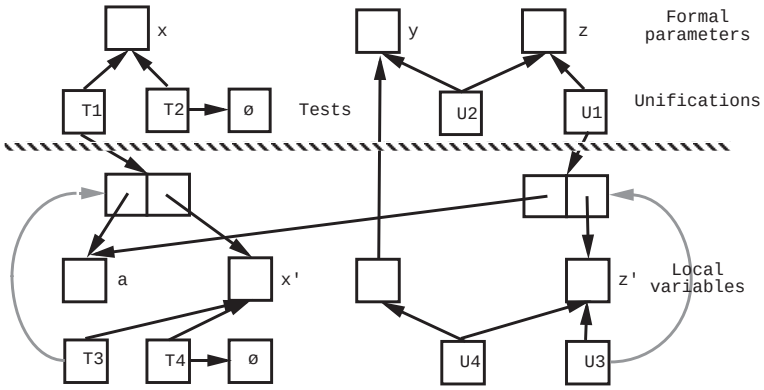

Figure 8.3. List concatenation graph on CIM.