PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

KL1 IN CONDITION GRAPHS ON A CONNECTION MACHINE

Jonas Barklund, Nils Hagner and Malik Wafin

Uppsala Programming Methodology and Artificial Intelligence Laboratory (UPMAIL)
Computing Science Dept., Uppsala University
Box 520, S-751 20 Uppsala, Sweden
JONAS@AIDA.UU.SE, NILSH@AIDA.UU.SE, MALIKW@AIDA.UU.SE

Abstract. We present a mapping of programs in the committed-choice parallel logic programming language KL1 to condition graphs. We define an abstract machine which executes condition graphs with commit. We present a realization of the abstract machine on the Connection Machine, a massively parallel SIMD computer with general communication facilities. Each term and control element is represented by one processor. Finally we outline an extension of the machine for programs in UPLOG, which has true don't-know non-determinism. The paper reports on work in progress, and the results so far are tentative.

1. INTRODUCTION

A formalism for Computer Science has to fulfill three criteria: natural problem representation (i.e., possibility to write applications), efficient execution, and a methodology for reasoning about problems and programs (Tärnlund 1986). Logic programming succeeds in all three respects and must therefore be considered a strong candidate for future programming systems.

For parallel computation, only high level languages can fulfill the problem representation and methodology criteria above. Declarative programming languages have the advantage over imperative languages that parallelism does not have to be explicitly specified by the programmer in the problem representation. The parallelism is inherent in the structure of the language and the control of parallel programs is often concerned with restricting the parallelism, in order to avoid meaningless computation.

We have chosen the parallel logic programming language KL1 (Kimura and Chikayama 1987), which is based on Flat GHC (Ueda 1985). KL1 is a committed choice language, i.e., it offers "*don't-care*" nondeterminism. This design decision makes it possible to use a simple execution mechanism.

Our goal is to achieve as much parallelism as possible. We do believe that this can be obtained by using *data parallelism*. Therefore, we have adopted a metaprogramming approach, that is, to represent programs as data. This allows us to extract parallelism at several levels, e.g., parallel unification (term level), AND-parallelism (atom level), and OR-parallelism (clause level).

Using this approach it is rather natural to use graphs to structure the data where program execution corresponds to operations on graphs (Hagner and Wafin 1988). Graph-based algorithms are, in general, not optimal on sequential machines, since most graph algorithms have an inherently parallel component. This implies that parallel machines are wanted for their execution. Moreover, they have to be fine-grained to enable execution of interesting (large) programs.

We have investigated various existing and proposed parallel architectures for KL1. An interesting new concept among these is the Connection Machine† (Hillis 1985). It is a fine-grained SIMD machine with dynamic communication facilities. So far, few high-level languages have been presented for the Connection Machine; CM-Lisp (Steele and Hillis 1986) is one step towards such a language, but we are not convinced that it meets the natural representation and methodology criteria.

Connection graphs (Kowalski 1975) and their corresponding proof procedures have mainly been used in theorem proving. We have introduced *condition graphs* (Barklund et al. 1988), a new extension to connection graphs, which are suitable for implementation of parallel logic programming languages. We proceed to define the Condition graph Inference Machine (CIM) which is designed to execute the Condition Graph proof procedure. Furthermore, we represent CIM on the Connection Machine, and translate the execution cycle to algorithms for the Connection Machine.

Finally, we look at UPLOG, a Horn clause language featuring "*don't-know*" non-determinism. Such a language has theoretical advantages over committed choice languages, but there are some practical difficulties with their implementation, e.g., in the interface to the real world. CIM can easily be modified to execute

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

programs in this language, preserving efficient execution when the computation is really deterministic, but still being able to handle truly non-deterministic computation.

2. INTRODUCTION TO KL1

A KL1 program is a finite set of guarded Horn clauses on the form

$$H \leftarrow G_1, \ldots, G_m \mid B_1, \ldots, B_n$$

where H (head) and B_1 to B_n (body atoms) are arbitrary goal atoms, while G_1 to G_m (guard atoms) are restricted to a set of predefined tests.

Given a goal atom K, one clause whose head H matches K and whose guard atoms $G_1, \ldots, G_m$ succeed is selected and K is reduced to the conjunction $B_1, \ldots, B_n$. A clause head matches a goal atom if they can be unified without instantiating any variables in the goal atom, or unifying two goal variables.

Initially, the machine is given a goal. Execution proceeds by reducing one or more goal atoms simultaneously and terminates when an empty goal is reached or when no clauses can be selected. In the latter case *deadlock* has occurred and the computation is indefinitely suspended.

A sequential abstract machine for KL1 has been developed at ICOT (Kimura and Chikayama 1987). It is intended to be used on Multi-PSI and PIM which are MIMD parallel computers, currently based on the PSI-II computer (Nakashima and Nakajima 1987).

3. CONDITION GRAPH PROOF PROCEDURE

Graph-based resolution procedures for execution of logic programs have achieved little popularity due to practical and combinatorial problems. Furthermore, graph algorithms are in general inherently parallel, which can make them difficult to implement on sequential machines. Today's massively parallel architectures make it possible to efficiently utilize the parallelism in such algorithms.

3.1. Connection Graphs

Let us present some background for connection graphs, condition graphs, and their proof procedures. For further reference, see e.g., Kowalski (1979) or Eisinger (1986).

Kowalski's Connection Graph proof procedure reflects the property of resolvability for a set of clauses by explicitly relating resolvable clauses to each other, using a graph structure. Matching literals of opposite polarity are connected by a link. A resolution step consists of the selection of an arbitrary link, resolving the clauses connected by the link, and connecting the resolvent to the graph.

3.2. Condition Graphs

The case is often that a formed resolvent can be deleted within one or a few cycles due to the *purity principle* (Robinson 1965). If there had been some primitive test of the resolvability for the parent clauses, the redundant resolvents would never have been created.

The Condition Graph proof procedure (Barklund et al. 1988, Hagner and Wafin 1988) is a refinement of the original Connection Graph proof procedure, where branches in the search tree that can clearly be determined not to lead to a solution (refutation) are deleted. Moreover, if it is unclear whether or not a clause will succeed, an attempt to reduce unnecessary computation is done by suspending the clause until it can be decided.

A *Condition Graph* (CG) is a pair $\langle N, L_C \rangle$, where N is a set of clauses, constituting the nodes of the graph, and L_C is a set of *conditional links*, connecting resolvable literals in the clauses. With each link there is associated a *unifying substitution*, θ, and a conjunction of *conditions*, ξ. For each (general) clause C_i,

$$A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n$$

ξ is obtained by forming the conjunction

$$\neg A_{t_i} \wedge \ldots \wedge \neg A_{t_k} \wedge B_{t_{k+1}} \wedge \ldots \wedge B_{t_l}$$

where A_{t_i} and B_{t_j} are primitive tests of clause C_i. All A_{t_i} and B_{t_j} are then removed from the clause C_i, yielding the node N_{C_i}. The conjunction ξ of primitive tests is finally associated with every link connected to an A_j in N_{C_i}. A primitive test is, informally, a single-step computation yielding a truth value. The set of primitive tests is application-dependent, and should include e.g., equality of terms and arithmetic comparison.

Resolution in Condition Graphs

Similar to Connection Graph resolution, a resolution cycle consists of:

1. evaluation of conditions with sufficient data on every link,
2. simultaneous selection of a number of links whose conditions are satisfied,
3. the creation of a resolvent for each selected link,
4. the removal of redundant nodes.

In the initial phase of the cycle all conditions associated with the links connecting a goal node are evaluated. If a condition evaluates to *false*, then the link can be removed from the graph. The links whose conditions evaluate to *true* are selected, while the selection of the remaining links is *suspended*.

The creation of the resolvents can be decomposed into three subphases. First, a resolvent node is created for each selected link. Second, each resolvent node is connected to the graph by copying the links to their parent nodes. Finally, the substitution θ associated with each selected link is applied to each link to the corresponding resolvent node.

There are a number of reasons why certain nodes will not participate in a refutation (Hagner and Wafin 1988), and therefore may be removed from a condition graph. Examples of this are pure clauses (clauses with an unlinked literal) and tautologies (Robinson 1965).

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. If all links are suspended, then the proof procedure terminates.

The Condition Graph proof procedure has several interesting proof-theoretical properties which are presented elsewhere (Hagner and Wafin 1988).

4. THE INFERENCE MACHINE

The Condition Graph proof procedure is the base of the *Condition graph Inference Machine* (CIM). The inference machine is designed to execute Horn clause programs. We have concentrated on top-down execution of KL1 programs, but it would be possible to adapt another strategy.

CIM consists of a compiler and a run-time system. The CIM compiler transforms a KL1 program into an *initial condition graph*. It is also possible to obtain an optimized version of the program by applying preprocessing techniques (Furukawa et al. 1987). The compilation requires a *template goal* to be included in the program, that is, a declaration of the *entry points* for the program. The template goal is substituted at run-time with the actual goal, by an instance of the goal being *injected* into the CG. We will not discuss compilation techniques in this paper.

The reduction mechanism will be able to work on all conjuncts of the goal simultaneously, tests acting as 'triggers' for unifications, as soon as the truth value of the tests can be determined.

Run-time System

The run-time system is based on a *three-phase cycle*, which repeatedly performs the following three steps:

1. *Select.* Define an *active* link to be a link with all tests ground and true. For every goal atom with one or more active links, *select* one active link and delete all the other links to the goal atom.
2. *Resolve.* Resolvents for selected links are produced simultaneously and independently added to the CG.
3. *Purge.* Pure clauses and links attached to them are removed.

There are three possible terminating states to a proof in CIM, namely the following:

1. If any of the resolvents produced is an *empty goal*, then the original goal has been solved.
2. If the resulting CG contains no goal, then the attempt to prove the original goal has failed.
3. If no test is sufficiently instantiated to be executed and every link is waiting for at least one test to succeed, then the CG is infinitely suspended (in a state of *deadlock*).

One advantage of CIM is the simplicity of detecting deadlock. Other proposed abstract machines for concurrent logic programming languages have more complicated deadlock detection mechanisms. This is important if a *meta-call* construction is later to be implemented (Foster 1987). There is a close correspondence between the tests in CIM and the guard atoms of KL1. Thus it would be easy to realize an "otherwise" construction, by using the disjunction of negations of other tests. Finally, garbage collection is easily introduced into the three-phase cycle. At the purge step, we may reclaim all resources used by the clauses that are removed from the graph. Garbage collection thus occurs continuously.

5. CIM ON THE CONNECTION MACHINE

The Connection Machine is a novel massively parallel computer architecture, grown out of the observation that a fundamental bottleneck in parallel 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 processing element (PE) executes the same instruction in each cycle, but can choose, depending on the context, to ignore incoming instructions. The instructions may operate either locally in 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 intelligent memory or a co-processor. One important feature of the Connection Machine is the possibility to divide a physical processor into many virtual processors in a completely transparent manner.

This allows "customization" of the machine for the size of the problem.

In this section we give a representation of CIM on the Connection Machine along with algorithms for the run-time system. Some algorithms can be realized as *parallel prefix computations* (Hillis and Steele 1986), where a reduction operation is applied to a large number of similar objects simultaneously. Suitable implementation languages for our purpose are *Lisp (Thinking Machines Corp. 1987) or Nova Prolog (Barklund and Millroth 1988).

5.1. Fine-grained Representation

Using virtual processors, we can increase the number of available PEs up to several millions. This means that the virtual processors have a relatively small amount of memory. The use of virtual processors, however, enables us to have a very fine-grained representation of the data: each virtual processor represents one term or a single control element, such as a test. The main reason for such a fine-grained representation is to obtain maximal parallelism; to execute an operation on several objects the same type (e.g., unification of terms), represent each of them by a separate PE and perform all the operations simultaneously. Thus, the algorithms will be formulated to operate on many instances of the same type of term or control element simultaneously. Logarithmic unification of terms has been successfully programmed by in this style by Barklund and Millroth (forthcoming). Some parts of the algorithms used in our system have also been implemented.

5.2. Representation of the Inference Machine

In our approach, we use a uniform representation for terms and control components. In general, the surface level of each such construct is represented by one single PE. Uniformity has a great advantage since many similar operations (e.g., copying) are applied on several types of objects. It is also the uniformity which is the key to the relative simplicity of the implementation scheme.

Each PE contains a number of fields, three of which are common for all types of constructs. The first field, ID, identifies the type of construct represented by the PE. The second field is a status word, which is used for several purposes. One important bit in the status word is the FREE flag, used to indicate that a PE is not in use. In order to obtain high performance when copying terms and control elements, the COPY field contains an identifier common to all PEs which are copied simultaneously (e.g., the PEs which together represent a clause).

Below we give a representation of the different constructs. For the sake of readability, some details have been omitted in the representation given. For convenience we give short names for the different kinds of control elements.

Tests (TPE). Each test is represented by a TPE. The ID field of the test identifies the type of test represented. The set of available tests depends on the actual implementation. Except for the status word and the copy field, each TPE contains a connection to (address of) a LPE and n connections to the n terms which the test operates on.

Unifications (UPE). A unification goal is represented by a UPE. Except the three standard fields, each UPE contains a connection to a LPE and two connections to the terms which are to be unified.

Links (LPE). Each link is represented by one LPE. It contains a connection to an APE and a field SELECT which is true for selected links.

Atoms (APE). An APE contains a connection to a GPE. It also contains a field REDUCE which is used for selection of links.

Goals (GPE). A goal is represented by a GPE. It contains a connection to names of the original goal variables. This is for printing the answer substitution on success.

Terms. There are three different types of terms: variables, lists and constants. A variable is represented by a PE containing fields for information about binding status etc. All occurrences of a variable in a clause are represented by the same PE, i.e., there is a unique PE for each variable. A list is represented by a PE containing connections to its head and tail, but also temporary fields for logarithmic unification etc. Constants are not represented by PEs, but as unique identifiers, e.g., addresses in the host machine memory space. For further details about this representation of terms, see Barklund and Millroth (forthcoming).

Fig. 5.1 is an example of the representation of a goal and two simple predicates. The term PE's have been omitted in the figure.

```
:- p(X,Y), q(Y,Z).

p(A.B,C) :- true | A=B, C=A.nil.
p(nil,C) :- true | true.
q(a.P,Q) :- true | P=S, Q=c.
```

The representation of a goal and clauses, defining the *append* relation, is shown in fig. 5.2.

```
:- append(1.2.nil,3.nil,R).

append(A.X,Y,R) :- true | R=A.Z,
  append(X,Y,Z).
append(nil,Y,R) :- true | Y=R.
```

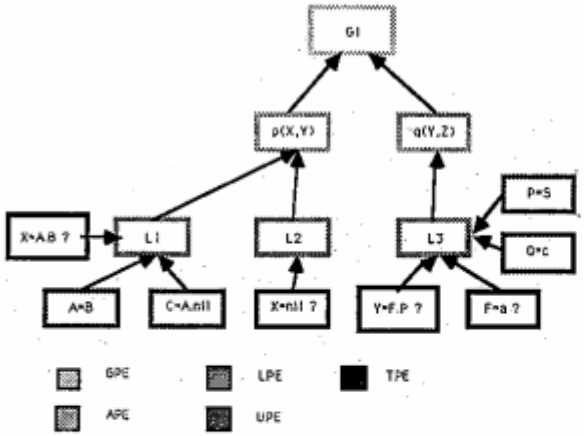

Figure 5.1. Example of goal, literal, link, test and unification PEs.

For the sake of intelligibility, this and the following examples contain only the terms, tests and unifications. The upper half represents the goal and the head of the clause, while the lower half represents the recursive call of append.

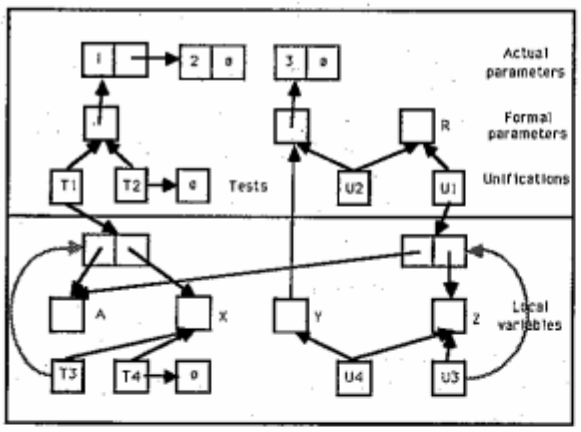

Figure 5.2. Initial graph for goal and the definition of *append*.

5.3. Execution

We will present the run-time system at three different levels. First, we give an informal introduction, presenting the main ideas behind the algorithms, second we give a somewhat more detailed presentation of the main loop. Finally we give the algorithms. These are presented at a high level, but all instructions have a close correspondence to Connection Machine operations.

In the run-time system, the three-phase cycle of CIM has been integrated into a single main loop. Informally, the main loop contains the following 6 steps, which are repeated until one of the three termination conditions of CIM is satisfied:

1. All tests associated with the goal are run,
2. a 'test-and-set' technique is used for selection of a number of links to be resolved upon,
3. the three termination conditions (success, deadlock and fail) of CIM are checked,
4. garbage collection (of PEs) is done,
5. all unifications are done,
6. each solved goal atom is replaced by a copy of the body atoms of the matching clause.

The first step actually consists of a finite number of iterations, given by the number of available tests, which are implementation dependent. E.g., first all '<' tests are run, next all '>' tests, etc. This is due to the SIMD characteristics of the Connection Machine. It is obviously desirable to keep the number of primitive tests as small as possible. The tests notify the corresponding goal atom about the result by sending a message via the link. If any test of a link fails, then the link fails and both the link and its associated tests are marked for garbage collection. If no tests have failed, but some tests have suspended for a link, then the link is suspended.

Commit is easily implemented using a test-and-set technique: all links to a goal atom whose tests have all succeeded will attempt to write their own address into the SELECT field of the goal atom. It can be assured that the PE with the highest address will be the only one that succeeds in this. After writing, each of these links reads the SELECT field and compares it with their own address. If it is not the same, then the link is discarded and becomes marked for garbage collection. Thus, only one link to a goal atom is selected, which makes the computation fully deterministic.

The three termination conditions are easily checked out. If the goal has no links connected to it, then the algorithm terminates with failure. If there is a goal, but none of its links have been selected (due to suspension), then a deadlock has occurred and the algorithm terminates. If the goal contains no atoms then the algorithm terminates with success. All three cases are very easy to detect. For example, deadlock is detected by computing a global OR of a bit in the STATUS field of all LPEs: this is supported by the CM hardware. In practice it may be more convenient to check for the empty goal case immediately after step 6 in the loop above.

Garbage collection, i.e., reclaiming unused PEs, can be made fast on the CM. All PEs who has a connection to a PE with the FREE flag set will set their own FREE flag etc., until no more PEs can be marked. Since PEs not in use are identified by their FREE flag being set, there is no 'collect' phase. Thus, incremental garbage collection is provided at a low cost.

All unifications associated with the goal are done using the parallel unification algorithm of Barklund and Millroth which is capable of unifying e.g., the top level of lists in logarithmic time. (The algorithm is capable of handling multiple pointers to terms and has been implemented on a Connection Machine). If any of the unifications fail, then the whole algorithm terminates with failure. This should, however, not happen too often, since such unifications should be programmed as tests and such a program can therefore be considered incorrect.

Finally, the goal is reduced by replacing goal atoms by their resolvents. In practice, the body atoms with their terms, links etc. are copied and are given the reduced goal atoms' connections to the GPE. This is possible due to the determinism of KL1 programs, but for a non-deterministic language such as e.g., UPLOG (section 6) this is not completely straightforward.

5.4. Algorithms

We will now present three of the algorithms used in the run-time system. The unification algorithm is that of Barklund and Millroth. Algorithm R is only presented informally while the others are given in full.

Algorithm P. (*Prove goal*). This algorithm represents the main loop of the runtime system. It attempts to prove a goal using a condition graph. If the proof succeeds, **yes** together with the answer substitution in the GPE is printed. If the algorithm terminates with deadlock, **maybe** is printed together with the current goal. If the algorithm terminates with failure, **no** is printed.

P1. [Run tests.] Run the tests using algorithm T.

P2. [Select links.] Select links to be resolved upon using algorithm S.

P3. [Deadlock?] Count the non-free LPEs with the SELECT field set. If there is none, terminate the algorithm with deadlock.

P4. [Unify.] Every UPE connected to a LPE with the SELECT field set initiates unification of the two terms it is connected to. This is done by connecting one of the terms to the other and applying algorithm U. Unification will report failure by setting the FAIL field of some term.

P5. [Termination with failure?] Apply algorithm F to detect failure. If the FAIL field is set in any PE, terminate the algorithm with failure.

P6. [Reduce.] Apply algorithm R to replace the goal by the result of the reduction.

P7. [Successful termination?] Count all APEs connected to the GPE. If there are none, the algorithm terminates successfully.

P8. [Purge.] Set the FREE flag of every non-free TPE which is connected to a free LPE, then return to **P1.**

Algorithm T. (*Run tests*). An LPE contains a COUNT field, which is the number of tests connected to it. A successful tests decrements the COUNT field of the LPE it is connected to by one. A failing test sets the FREE flag of the LPE. A test which cannot be run because some of its data is uninstantiated does nothing, i.e., remains suspended. Finally all non-suspended tests set their FREE flag. This is repeated for each type of test.

The following steps are executed simultaneously in every TPE which is not free.

T1. [Loop.] Repeat steps T2 to T5 for all types of tests.

T2. [Current test?] If ID is not the current test, ignore steps T3 to T5.

T3. [Remain suspended?] If the connected terms are not enough instantiated, ignore steps T4 and T5.

T4. [Run test.] Execute the current test. If the test succeeded, then send -1 (with ADD-combination of messages) to the COUNT field of connected LPE. Otherwise, send 1 to the FREE flag of the connected LPE.

T5. [Free test.] Set the FREE flag.

Algorithm S. (*Selection of links*). All LPEs with COUNT zero send their address to the connected APE. One will succeed and is selected; the others are freed.

The following steps are executed simultaneously in every LPE which is not free.

S1. [Candidate?] If COUNT is non-zero, ignore steps S2 and S3.

S2. [Set.] Send own address (with MAX-combination of messages) to the REDUCE field of the connected APE.

S3. [Test.] Read the contents of the REDUCE field of the connected APE. If this is equal to own address, set SELECT field. Otherwise, set FREE field.

Algorithm F. (*Termination with failure?*). The FAIL flag is set in every non-free APE with only free LPEs connected to it.

The following step is executed simultaneously in every APE.

F1. [Set.] Set the FAIL flag to the complement of the FREE flag.

The following step is executed simultaneously in every LPE which is not free.

F2. [Reset.] Send 0 to the FAIL field (with any combination of messages) of the connected APE.

Algorithm R. (*Reduction*). Every APE which has a selected LPE should be reduced to the goals connected to that LPE. This is accomplished by copying the graph defining the subgoals and connecting them to the GPE. Finally, the APE and the LPE can be freed.

The Appendix shows an example of an execution of *append*.

6. THE UPLOG LANGUAGE

Committed choice languages feature 'don't-care' non-determinism for practical purposes, but for ease of programming, as well as for theoretical properties, 'don't-know' non-determinism is more appealing. The condition graph inference machine can be modified to handle 'don't-know' non-determinism in an OR-parallel or OR-sequential fashion. We believe that execution can still be made efficient for deterministic programs.

UPLOG is a Horn clause language with the same syntactical structure as a KL1 program, except for a *fence* operator, which is used instead of a commit operator. Apart from this, it uses the same execution mechanism as KL1. Fig. 6.1 shows an example of an UPLOG quicksort program.

The declarative semantics of the fence operator is a conjunction, that is, $A \,\&\, B$ has the same declarative semantics as $A \wedge B$. The commit operator of e.g., KL1 has exactly the same declarative semantics. The difference lies in the operational semantics.

The operational semantics of the fence operator is a *sequential non-commutative conjunction*, that is, $A \,\&\, B$ is true if and only if, first A is true, and then B is true under the same interpretation. All tests, to the left of the fence, are computed first. If every T_i succeeds, then we pass the fence and compute all B_j.

```
qsort(nil,O) :- true & O = Z-Z.
qsort(A.X,O) :- true &
  O = W-Z, part(A,X,S,L),
  qsort(S,W-A.Y), qsort(L,Y-Z).
part(A,nil,S,L) :- true &
  S = nil, L = nil.
part(A,B.X,S,L) :- A ≥ B &
  S = B.T, part(A,X,T,L).
part(A,B.X,S,L) :- A < B &
  L = B.M, part(A,X,S,M).
```

Figure 6.1. Example of UPLOG quicksort program.

It would be possible to define a syntactic sugar for the common cases of empty tests and/or body subgoals. Mode declarations could be used as a complement to, or substitution for explicit output unifications.

Modified CIM

When CIM reduces a goal there remains only one active link to a clause, due to the semantics of the commit operator. To handle 'don't-know' non-determinism, we can use the property that

$$A \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \Leftrightarrow (A \wedge B_1) \vee (A \wedge (B_2 \vee \ldots \vee B_n))$$

to *split* a goal which has more than one active link into as many goals as there are active links. See fig. 6.2 for an example.

This can be implemented as follows. A link connected to a goal atom B is *unique* if it is the only link connected to B. For every goal C containing a goal atom B with multiple links of which at least one is active, do *goal splitting*, i.e., select one active link L and copy C to obtain C'. Remove L from C and remove every link of B except L from C'. Finally, select all unique active links.

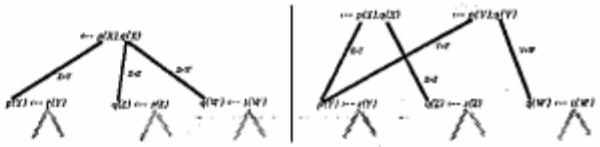

Figure 6.2. Splitting a non-deterministic goal to get two deterministic goals.

Program methodology

There seems to be a fundamental tradeoff between theoretic properties and implementation issues. It is well known that the semantics of the commit operator is a problem in program manipulations, such as folding and unfolding of goals (Furukawa et al. 1987). Since these operations form the basis of abstract interpretation (Abramsky and Hankin 1987), partial computation (Futamura 1982), etc., it is difficult to transform KL1 programs formally and/or mechanically.

It is easy to show that the fence operator does not cause these problems, so that UPLOG programs should not be more difficult to manipulate than Prolog programs (Barklund et al. 1988, Hagner and Wafin 1988). The value of these properties should not be underestimated, since these research fields are making good progress.

Non-determinism

'Don't-know' non-determinism increases the expressive power of the language but creates implementation problems, e.g., the output from a program is generally required to be deterministic. In languages such as KL1 the problem never arises due to the commit operator,

which guarantees that environments are always deterministic, and that output is only done when execution is deterministic.

We aim to execute deterministic programs efficiently, and to handle non-deterministic computation, although less efficiently, but still at a reasonable cost. At this stage, the problem of handling non-determinism in output is left open in CIM, but some possibilitites are:

- Allow output from several goals to be mixed.
- Disallow output when there are several goals (detection can be eager or lazy).
- Be clever (e.g., have separate windows for each goal, or implement backtracking output devices [Kahn 1985]).
- Suspend output until execution is deterministic.

The first approach is clearly easiest to implement, while the others require more effort. The third approach is very device-specific but may simplify programming of, e.g., graphic terminals. We think the fourth approach is well suited for CIM.

7. CONCLUSIONS

We are aware that Connection Machines are not yet widely available, but we believe that similar computers will be one of the major architectures in a longer perspective. The following are some of the new ideas which have been put forward in this paper:

- A new abstract machine, CIM, for the condition graph proof procedure, is defined for the target language KL1. We are aware that the machine is described at a high algorithmic level, with no clearly visible instructions.
- A fine-grained mapping of CIM onto the Connection Machine is proposed.
- A modified CIM which handles UPLOG programs (a language with 'don't know' non-determinism) is outlined.

We have not encountered any serious problems, but since the algorithms are only partially implemented and tested, we still lack empirical evidence that our approach is fruitful. However, we continue to do simulations of CIM.

8. RELATED AND FUTURE WORK

Bawden has presented work on implementation of "connection graphs" on the Connection Machine (1986). It seems that his definition of connection graphs differ fundamentally from ours which origins from Kowalski (1979), but his work indicates that graph-based execution is an interesting approach for high-level language execution on the Connection Machine.

P-prolog (Yang and Aiso 1986) is a language which is somewhat similar to UPLOG because it allows don't-care non-determinism.

Nilsson and Tanaka (1988) have also investigated the implementation of committed-choice languages on SIMD machines. Their approach is not graph-based and can also be used on vector machines.

Other implementation techniques for committed-choice languages on parallel machines are described by, e.g., Kimura and Chikayama (1987), Crammond (1986), Ichiyoshi et al. (1987), and Sato et al. (1987).

Nova Prolog is a proposed data parallel extension of Prolog (Barklund and Millroth 1988). It is a logic programming language for the Connection Machine which gives the programmer the ability to exploit parallelism in terms explicitly. CIM on the other hand is an attempt to exploit implicit AND-parallelism.

We will proceed with further simulations of CIM and its mapping on the Connection Machine. We expect to present some efficiency measures before long. This work is done as a part of the Parallel Inference Machine project at UPMAIL.

ACKNOWLEDGMENTS

The authors want to thank their colleagues at UPMAIL for providing a fine research environment, cakes, and table tennis games. Their comments on this paper have been valuable. 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. Several anonymous referees gave many valuable comments. Finally, without the continuous support from our families, this work would never have been done.

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

REFERENCES

S. Abramsky, C. L. Hankin (ed.), *Abstract Interpretation for Declarative Languages* (Chichester: Ellis Horwood, 1987).

J. Barklund, N. Hagner, M. Wafin, "Condition Graphs," *5th International Conference and Symposium on Logic Programming*, ed. R. A. Kowalski, K. A. Bowen (Cambridge, Mass.: MIT Press, August 1988), 435–45.

J. Barklund, H. Millroth, "Nova Prolog," *UPMAIL Technical Report* 52 (Uppsala: Uppsala University, Computing Science Dept., 1988).

J. Barklund, H. Millroth, "Parallel Unification," *UPMAIL Technical Report* 46 (Uppsala: Uppsala University, Computing Science Dept.).

A. Bawden, "Connection Graphs," *1986 ACM Conference on Lisp and Functional Programming*, ed. W. L. Scherlis and J. H. Williams (New York: Association for Computing Machinery, August 1986), 258–65.

J. Crammond, "An Execution Model for Committed-Choice Non-Deterministic Languages," *1986 Symposium on Logic Programming*, ed. R. Keller (Washington, D. C.: IEEE Computer Society Press, September 1986), 148–58.

N. Eisinger, "What you always wanted to know about Clause Graph Resolution," *8th Conference on Automated Deduction*, ed. J. Siekmann (Berlin: Springer-Verlag, 1986), 316–36.

I. Foster, "Logic Operating Systems: Design Issues," *4th International Conference on Logic Programming*, ed. J-L. Lassez (Cambridge, Mass.: MIT Press, May 1987), 910–26.

Furukawa, K., Okumura, A., Murakami, M., "Unfolding Rules for GHC Programs," *Workshop on Partial Evaluation and Mixed Computation*, ed. D. Bjørner (Copenhagen: 1987).

Y. Futamura, "Partial Computation of Programs," *Lecture notes in Computer Science* 147 (Berlin: Springer-Verlag, 1982), 1–35.

N. Hagner, M. Wafin, "Parallel Logic Programming using Condition Graphs," *UPMAIL Undergraduate Theses* 1 (Uppsala: Uppsala University, Computing Science Dept., May 1988).

W. D. Hillis, *The Connection Machine* (Cambridge, Mass.: MIT Press, 1985).

W. D. Hillis and G. L. Steele, Jr., "Data Parallel Algorithms," *Communications of the ACM* 29 (December 1986): 1170–83.

N. Ichiyoshi, T. Miyazaki, K. Taki, "A Distributed Implementation of Flat GHC on the Multi-PSI," *4th International Conference on Logic Programming*, ed. J-L. Lassez (Cambridge, Mass.: MIT Press, May 1987), 257–75.

K. M. Kahn, "A Grammar Kit in Prolog," *UPMAIL Technical Report* 14C (Uppsala: Uppsala University, Computing Science Dept., February 1985).

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, August 1987), 468–77.

R. Kowalski, "A Proof Procedure Using Connection Graphs," *Journal of the ACM* 22 (October 1975): 572–595.

R. Kowalski, *Logic for Problem Solving* (New York: North Holland, 1979).

H. Nakashima, K. Nakajima, "Hardware Architecture of the Sequential Inference Machine: PSI-II," *1987 Symposium on Logic Programming*, ed. S. Haridi (Washington, D. C.: IEEE Computer Society Press, August 1987), 104–13.

M. Nilsson, H. Tanaka, "A Flat GHC Implementation for Supercomputers," *5th International Conference and Symposium on Logic Programming*, ed. R. A. Kowalski, K. A. Bowen (Cambridge, Mass.: MIT Press, August 1988), 1337–50.

J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM* 12 (January 1965): 23–41.

M. Sato, H. Shimizu, A. Matsumoto, K. Rokusawa, A. Goto, "KL1 Execution Model for PIM Cluster with Shared Memory," *4th International Conference on Logic Programming*, ed. J-L. Lassez (Cambridge, Mass.: MIT Press, May 1987), 338–55.

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.

S.-Å. Tärnlund, "Logic Programming—from a Logic Point of View," *1986 Symposium on Logic Programming*, ed. R. Keller (Washington, D. C.: IEEE Computer Society Press, September 1986), 96–103.

Connection Machine Model CM-2 Technical Summary (Cambridge, Mass.: Thinking Machines Corp., April 1987).

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

R. Yang, H. Aiso, "P-Prolog: A Parallel Logic Language Based on Exclusive Relation," *3rd International Conference on Logic Programming*, ed. E. Shapiro (Berlin: Springer-Verlag, July 1986), 255–69.

APPENDIX

Let us show an example of the execution of the code in fig. 5.2. The test Ti belongs to the same link as the unification Ui.

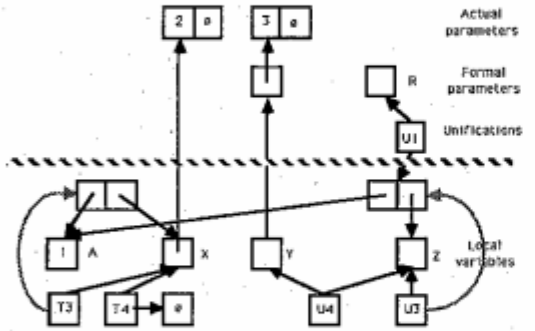

Figure A.1. After the tests T1 (succeeds) and T2 (fails), and cleaning the graph.

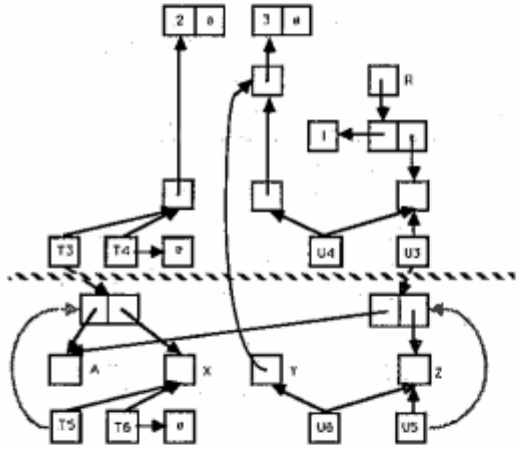

Figure A.2. After the unification U1, copying the graph for *append*, and cleaning the graph.

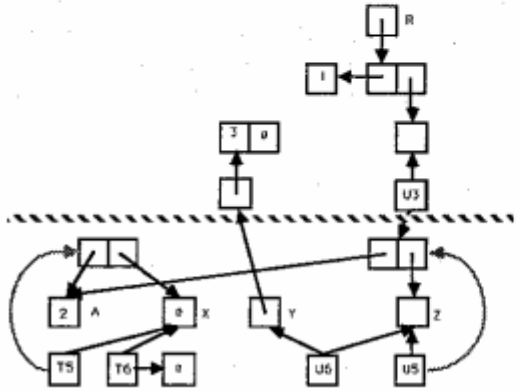

Figure A.3. After running the tests T3 (succeeds) and T4 (fails), and cleaning the graph.

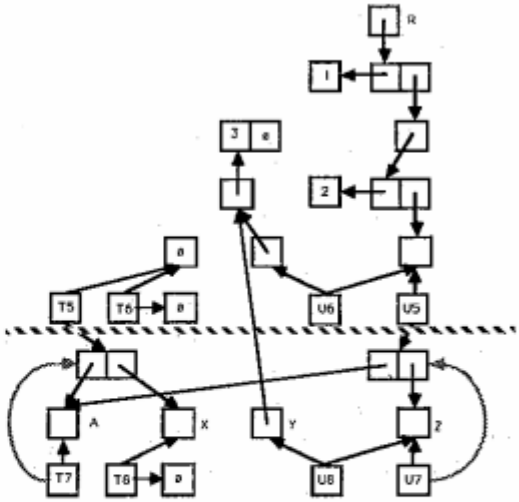

Figure A.4. After running the unification U3, copying the graph for *append*, and cleaning the graph.

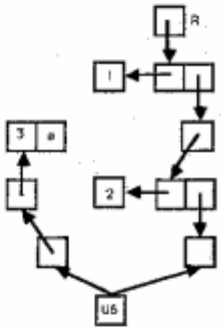

Figure A.5. After running the tests T5 (fails) and T6 (succeeds), and cleaning the graph.

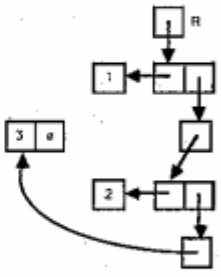

Figure A.6. After running the unification U6 and cleaning the graph.