Optimizing Clause Matching Automata in Committed-Choice Languages

Bart Massey and Evan Tick

CIS-TR-94-20
August 1994

Abstract

We introduce a formalism for clause matching in concurrent committed-choice languages based on the construction of *clause matching automata*, a heuristic for the compilation of clause matching, and a technique for more efficient implementation of matches. The formalism is notable for its generality and simplicity, the heuristic for combining important advantages of several existing heuristics. These include good typical-case time and space performance, a minimal number of suspensions, which can lead to tremendous efficiency benefits, and incremental restart after suspensions, which eliminates repeated tests.

This paper presented at the *First International Symposium on Parallel Symbolic Computation,* Linz, Austria, September 1994.

Department of Computer and Information Science
University of Oregon

Contents

1 Introduction

Efficient clause matching is necessary to achieve high performance implementations of logic languages, including committed-choice languages (CCLs). Especially in CCLs, with ask and tell unification (and even more so in languages with tell assignment only), most useful computation is performed during the matching step. Furthermore, in CCLs and logic languages with delayed constraints, suspensions can interact with matching, causing rework if the implementation is not careful. Finally, suspensions are typically extremely expensive. Thus without efficient matching, high-performance CCL implementation is impossible. In spite of this, many CCL implementations use far from optimal clause matching. In our opinion, reasons for this include: 1) lack of appreciation for the importance of efficient matching; 2) difficulties in adapting pattern-matching techniques from other language families; 3) lack of extensive study of matching algorithms, and 4) lack of clear and detailed descriptions of efficient matching algorithms.

This paper addresses these problems in two stages. First, we provide a general formalism for describing clause matching algorithms based on the notion of *clause matching automata* (CMAs). A CMA is a finite automaton whose states represent progress made during matching, and whose transitions denote atomic matching tests. Second, we describe a heuristic for the construction of high-performance CMAs satisfying two key criteria: minimal suspension and execution incrementality. A minimum suspension CMA will always try all possible tests before suspending. A fully incremental CMA will never execute a test whose outcome can be predicted. Thus CMAs with these properties would have optimum performance in practical use.

The foundational work of Kliger and Shapiro [7, 8] on decision graphs may be directly incorporated into our CMA formalism, as may ideas presented by Debray *et al.* [2] on exploiting probabilistic information in clause matching. The work of Podelski and Van Roy [10] on incremental clause matching, and the work on term pattern matching in non-strict functional languages [9] is clearly relevant, although it will be more difficult for us to incorporate. All this work is discussed in Section 3 of this paper. Although inspired by this earlier work, we believe that our formalism and heuristic together constitute a novel advance in clause matching for CCLs.

The paper is organized as follows. First, some background material is presented in Section 2. Next, state-of-the-art clause matching algorithms for CCLs are outlined in Section 3. In Section 4 the concept of clause matching automata is introduced and formalized. A heuristic to guide the construction of high-quality CCL matching automata is described in Section 5. A further optimization, disjuncts of conjuncts, is given in Section 6. Conclusions are summarized in Section 7.

2 Background

Committed-choice languages [13] are characterized by the need to determine at runtime which among a choice of several clauses of a procedure should be "committed" to. In these languages, a *procedure* consists of a number of *clauses*, each of which consists of a term *head*

```
select( nil, Result )  :-  Result = nil.
select( pair( keep( V0, V ), Rest ), Result )  :-
        V0 ≥ 0  |
        select( Rest, Result0 ),
        Result := pair( V, Result0 ).
select( pair( keep( V0, V ), Rest ), Result )  :-
        V0 < 0  |
        select( Rest, Result ).
```

Figure 1: Sample CCL Program

pattern, a (possibly empty) conjunction of boolean *guard* expressions, and a set of *body* calls.

We now briefly describe the semantics of our CCL (see Shapiro [13] for details). Our results are applicable to most CCLs. To simplify the exposition in this paper, we will work in a CCL subset containing only atoms, functors, integers, and unbound variables, with a few simple guard tests. Figure 1 shows a sample program written in this CCL, to which we will refer throughout this paper. CCL semantics are relatively simple: when a clause is selected for execution, its body calls are spawned for concurrent evaluation. Each call must then match some clause of the program. The variables of CCLs have the *Dynamic Single Assignment* (DSA) property [1]: they can be assigned a value only once during the execution of a program, but not, in general, at a statically determinable program point. In addition, variable binding also achieves synchronization: a clause match will *suspend* until variables corresponding to required parts of some clause have been bound. Matching is nondeterministic; clause matches are not considered in top-to-bottom order. Rather, as soon as enough information is available to commit to some set of matching clauses, one of them, chosen arbitrarily, is committed to. Thus, simple guarded-term pattern-matching compilers, such as that of Peyton-Jones [9], are not sufficient for these languages.

Naive implementations of nondeterministic clause matching semantics use a technique incorporated in several systems [15] such as Gregory's SPM [4]. In this technique, the match is performed left-to-right. However, if the match could suspend because of an unbound DSA variable, that variable is added to a *suspension set*, and matching of that clause is abandoned. If the suspension set is nonempty and no clause can be matched, the match is suspended in such a way that binding of any DSA variable in the suspension set will cause the match to be completely retried. If the suspension set is empty and no clause can be matched, the match has "failed."[1]

In CCL implementations, clause matching speed is typically the limiting factor in program execution speed, for several reasons. First, the other execution step, expanding body calls, is typically simple and efficient. Second, the interaction between suspension and matching (to avoid rework) is typically difficult to implement efficiently. Finally, although

[1]In flat CCLs with eventual tell unification, match failure implies program failure. Other CCL language variants handle failure locally, see [13, 15]. This distinction is not relevant to this paper.

much research has been done on efficient clause matching for CCLs (Section 3) most CCL implementations still use naive matching techniques. While these naive techniques are correct and simple to implement, they can lead to serious inefficiencies in execution for three reasons: 1) more suspensions may be performed than strictly necessary to select a clause; 2) a suspended match will be entirely repeated on resumption, possibly performing redundant tests, and 3) redundant tests may be performed because of the left-to-right match order.

3 Related Work

Inefficiencies in naive clause matching algorithms were recognized some time ago. Hickey and Mudambi [5] addressed this "indexing problem" for Prolog. They introduced the concept that one should first select a head argument based on some better criteria than mere left-to-right ordering: this was called the "caring" heuristic (defined below in context of CCLs). This influenced Kliger and Shapiro's seminal work on *decision graphs* and *decision trees* for clause matching in CCLs [7, 8]. More recent work on Prolog indexing, using a technique involving string-matching automata, has been reported in [11], although this work is not directly relevant due to the different Prolog semantics (in particular top-to-bottom clause matching and the absence of suspensions).

In a nutshell, Kliger's decision tree algorithm is based on the idea that each argument test in the sequence of decisions made during clause matching can be thought of as splitting the clauses into two groups or *residuals*. The first group contains clauses which are still matching candidates if the argument has *some* given value from among a set of values. The second group contains clauses which are no longer matching candidates if the argument has *any* of these values. One can thus think of each decision as a node in a search tree. The naive search tree is effectively linear, which is bad because the number of tests which must be performed before the match is complete (a clause is committed) is equal to the height of the tree.

Kliger's *indexing heuristic* effectively encodes two basic criteria for matching order. The first criterion is based on the idea that a clause "cares" about a head argument if that argument is a constant in the head or is used by a guard test. The primary criterion used to determine matching order is to match first the arguments cared about by the *most* clauses. The secondary criterion is based on the idea that the code generated for the decision trees will be smaller overall if the nodes near the root have the smallest branching factors (because these tests will then be coded once instead of multiple times). This idea motivates the "minimum variability" criterion: among arguments with equal caring, choose the one whose tests have the smallest number of interesting results.

The decision tree algorithm is efficient, but it has an important drawback which Kliger addresses: the worst-case size of the tree (and thus the size of the code generated by a compiler for the match) is exponential in the size of the target program, and this complexity unfortunately occasionally (albeit infrequently) arises in practice [6]. This led to an ingenious modification of the technique to produce decision *graphs*, which sacrifice a small amount of worst-case speed in order to achieve a major reduction in worst-case size. This is accomplished by constructing, instead of a decision tree, a DAG of decisions such that several tests may jump into the same code (effectively a continuation) to handle failure cases. The worst-case size and speed of this algorithm are both polynomial in the size of the match, although Kliger shows that the speed is comparable with that of decision trees in practice [6].

Kliger's algorithms have been influential in the CCL implementation community. Examples include the decision graphs used in the Monaco [14] FGHC compiler and a variant of decision trees developed for determinacy testing in Andorra-like language compilation [16].

Debray *et al.* [2] proposed an alternate indexing heuristic, intended to allow optimal native code generation from a decision tree. This heuristic is based on the idea that a more realistic measure of which test should be performed first is to maximize the information-theoretic *entropy* dispelled by the test, i.e., maximize the amount of information the test gives per unit test cost. They combine this technique with an algorithm for keeping track of the implications of each test (by constructing equivalence classes of tests) to generate decision trees having near-optimum performance on certain classes of problems. For example, lexical analyzers are prime candidates because the probability of each outcome of a test can be estimated.

Podelski and Van Roy [10] proposed a general algorithm for incremental *entailment* (and *disentailment*) treated as term constraint satisfaction. The incrementality addresses the problem with the naive algorithm noted in Section 2 and shared by Kliger's algorithms, namely that clause matching must be completely restarted after every resumption. Podelski and Van Roy's algorithm achieves full incrementality, never unnecessarily repeating tests on resumption, but at the expense of implementation difficulty. The algorithm proceeds by attempting to find a sequence of tests which will distinguish the objects. If suspension is necessary, a structure they call the "beast" is constructed, which maintains the information necessary for resumption. The algorithm as given appears to not consider the case of arithmetic inequalities. Nonetheless, this algorithm represents an important contribution to this area, which needs to be further explored. In particular, we believe their algorithm to be unique in correctly handling incremental, dynamic passive unification: this portion of their algorithm should be usable in conjunction with our technique.

Peyton-Jones [9] summarizes the state-of-the-art in term pattern matching for non-strict functional languages; the pattern matchers produced by his technique are optimal for this sort of application. Because his work concentrates mainly on a language with deterministic (left-to-right top-to-bottom) matching semantics, it is not directly relevant. However, the complication of non-strict matching semantics does require dealing with many issues analogous to those of dealing with DSA variables. The overall design of our algorithm was strongly influenced by Peyton-Jones' compiler. Our algorithm also seems to have notable similarities to AI pattern-matching algorithms such as Forgy's Rete [3] and its successors, although we have not yet had time to explore these similarities.

Argument Matrix M

nil	$\times$	$\times$	$\times$	$\times$	$v_{\langle 2\rangle}$	$\emptyset$
pair/2	keep/2	$v_{\langle 1,1,1\rangle}$	$v_{\langle 1,1,2\rangle}$	$v_{\langle 1,2\rangle}$	$v_{\langle 2\rangle}$	$\{v_{\langle 1,1,1\rangle} \geq 0\}$
pair/2	keep/2	$v_{\langle 1,1,1\rangle}$	$v_{\langle 1,1,2\rangle}$	$v_{\langle 1,2\rangle}$	$v_{\langle 2\rangle}$	$\{v_{\langle 1,1,1\rangle} < 0\}$

Argument Index G

$\langle 1\rangle$	$\langle 1,1\rangle$	$\langle 1,1,1\rangle$	$\langle 1,1,2\rangle$	$\langle 1,2\rangle$	$\langle 2\rangle$	guards

Suspension Columns $k = \emptyset$

Figure 2: Example Program Initial Match State

nil	$\times$	$\times$	$\times$	$\times$	$\emptyset$
pair/2	keep/2	$v_{\langle 1,1,1\rangle}$	$v_{\langle 1,1,2\rangle}$	$v_{\langle 1,2\rangle}$	$\{v_{\langle 1,1,1\rangle} \geq 0\}$
pair/2	keep/2	$v_{\langle 1,1,1\rangle}$	$v_{\langle 1,1,2\rangle}$	$v_{\langle 1,2\rangle}$	$\{v_{\langle 1,1,1\rangle} < 0\}$

(a) Reduce $\langle 2\rangle$

$\times$	$\times$	$\times$	$\times$	$v_{\langle 2\rangle}$	$\emptyset$

(b) Test $\langle 1\rangle = \text{nil}$

Figure 3: Example Program Next States

4 Clause Matching Automata

In constructing the algorithms described in Section 3, we first define a common terminology. An *argument* denotes, not just the top-level arguments of a clause, but subterms of the top-level arguments as well. Thus, the second clause of our example program of Figure 1 has not two arguments, but six. We will often label an argument with the sequence of indices (or *path*) which leads to it, e.g., the argument V of the second clause is indicated by the sequence $\langle 1,1,2\rangle$. A *clause match* (or simply *match*) is the action of choosing a particular clause for a procedure. A *test* is the act of examining some particular argument. The fundamental semantic values to be tested for are: $k \in \mathbf{Z}$ (integer constants, such as 3 and -5), $f/n \in \mathbf{A} \times \mathbf{Z}$ (functors of a given arity, such as $p/3$. Atoms are merely functors of arity 0), and $v \in \mathbf{V}$ (unbound variables). Repeated occurrences of a variable in a clause head are transformed to passive unification guards during an initial "flattening" pass.

A match is compiled into a sequence of abstract instructions, e.g., those of the Monaco system [14]. For simplicity of the exposition, we limit ourselves to strict two-argument guards, specifically arithmetic comparisons and term equality and inequality. There is, however, one important modification to the match instruction set necessary to achieve incrementality in our system. A suspension reports, along with each variable on which it is suspending, a match code address at which to resume execution if this variable is bound. This address is effectively a continuation, which places the CMA in a correct match state to resume, as described below.

A procedure's clauses are rewritten into an abstract form which is amenable to further processing. First, whenever possible, explicit term matching in the guard is incorporated back into the head. Second, all variables of the procedure are renamed to be of the form v_s

where s is the path leading to v. Our canonical form, which we will call a "match state" α, will consist of

- A matrix M_α of values to be matched, where the rows of the matrix correspond to clauses, and the columns of the matrix to arguments. An additional column vector contains the set of explicit guards associated with each clause.
- A row vector G_α which keeps track of the argument corresponding to each column of M. This is similar to the ψ function in Kliger [6] and Tick and Korsloot [16].
- A set of column index/match state pairs k_α, with $k_\alpha \neq \emptyset$ iff α is a "successor state." The use of k_α is explained in detail below. This corresponds to the suspension stack in the standard schemes.

The elements of M are either a value to be matched, or a '$\times$' corresponding to an argument not present in a clause. As an example, Figure 1, and its corresponding initial match state, shown in Figure 2. A partial order Q_α is imposed on the columns of M_α. This is necessary because the subterms of a term cannot be matched before the term itself is matched. The partial order can always be inferred from G_α, and thus need not be maintained explicitly.

The key idea of the CMA is to construct a DAG of match states such that successive states are *reduced*, where a state α is reduced if M_α is reduced or if $|k_\alpha|$ increases. A matrix M_α is reduced by removing a column and possibly one or more rows, in one of three ways. First, a test may be performed on an argument column, and that column and any rows incompatible with the test result are removed. Second, a guard test may be performed and removed from a guard set, again removing rows incompatible with the test result. Third, a column consisting only of variables and don't-cares $\times$ can be eliminated. To increase $|k_\alpha|$, a column index is added to k_α and that column is eliminated from M_α.

Thus, a deterministic finite automaton is constructed whose states are match states α_i and whose out-transitions are tests of a column in the current state. As an example of this, consider again the initial match state α of Figure 2. Figure 3 shows two possible tests to perform first on α, and the resulting match states α'. In Figure 3b, the next state shown is that in which the test is true (other possible results are for the test to be false, or for the argument being tested to be unbound). Note that the tests which may be performed are exactly those which test the topmost set of columns in the partial order Q_α. Once a value has been found in a column, that column is removed from Q_α — if the value was a functor, this exposes the arguments of the functor for testing purposes.

As arguments become available, guard tests may be performed. When a guard test has been performed, it is discarded from the match state: thus, eventually all guard tests will be performed. Passive unification and disunification are typically expensive, arithmetic tests frequent. It is thus desirable to keep track of what tests have been performed so far when reaching a given match state, in order to avoid guard and argument tests whose outcomes can be predicted from previous computations, as is done in Van Roy [12] and Tick and Korsloot [16].

In a state with matrix M_α containing a single row, the corresponding clause may commit. The tests implied by the remaining columns must still be performed, but if they fail, then the entire match should fail. The initial match state α_0 of a CMA is determined by the static procedure. The set of (accepting) final match states $\{\bar{\alpha}_1, ..., \bar{\alpha}_j\}$ will in general contain one state for each clause. However, some clauses may not be represented, either because they are logically unreachable (e.g., their guard tests are not satisfiable) or because they are entirely subsumed by other clauses. In either case, it is useful for the CMA construction algorithm to issue a warning, because this behavior is probably not what the programmer intended. Note that there must always be at least one transition out of any match state, although it may be a failure transition. When a matrix M_α contains no more columns, but more than one row still remains, then the result of the computation depends on k_α. If $k_\alpha = \emptyset$, then the match fails. Otherwise, the match suspends on k_α.

In some cases, the CMA may attempt to test a procedure argument before it is bound. The CMA will discover this problem as the result of a boundness test. In general, some clauses may not require the desired argument position to be bound. Thus, the CMA can transition to a state which tests other arguments without reducing M. When it does this, the argument being bypassed and the match state in which this occurred are recorded in the "successor set" k_α. If, however, no further match progress can be made, then suspension will be necessary. At this point, k_α will contain a set of variables on which to suspend, and a match state in which matching should resume for each variable which may later be bound. Note that it is always correct to resume a match at the initial match state. However, the retention of later match states in k_α enables incremental matching, by restarting the match at the lowest (closest to the leaves) safe state in the CMA.

The CMA must be acyclic; otherwise it could infinite loop. The major worry about cycles is that we might move cyclically through a series of states with identical M_α. To avoid this, it is sufficient to ensure that, when moving from α to α' with $M_\alpha = M_{\alpha'}$, $|k_\alpha|$ always increases. Since $|k_\alpha|$ is bounded by the number of arguments, this will ensure that no cycles are formed. To move to a new match state, we select an argument to test using an ordering heuristic, or *CMA construction heuristic.* Several such heuristics are discussed in Section 5. The argument selected must also be consistent with the partial order Q_α. There will in general be several transitions out of a state α: one for each of the possible values of the argument i being tested, and one corresponding to the case where the argument being tested is unbound. This last transition α leads to a state α' in which M_α is not reduced, but in which

$$k_{\alpha'} = k_\alpha \cup \{\langle i, \alpha\rangle\}$$

which reduces α by increasing $|k_\alpha|$. Thus, if $\langle j, \beta\rangle \in k_\alpha$, this implies that argument j was examined in match state β and found to be unbound.

To make all of this concrete, Figure 5 shows one possible CMA construction for our sample program. To save space, only the matrix portion M of the match states is shown, and the ellipses in the figure indicate sequences of states and transitions not shown. In its basic outline, this construction is produced by all three of the heuristics discussed in Section 5. The argument to the suspend() states is a list of columns of the initial matrix M_{α_0} on which to suspend. The transitions leading from suspension to resumption are not shown in the figure. If these transitions all lead back to the initial state of the CMA, then the CMA as shown is a traditional restarting matcher rather than an incremental one. A partly incremental CMA could be constructed, for example by allowing the suspend(3) to resume in the state immediately above it in the diagram. The construction of partly incremental CMAs allows a space/efficiency tradeoff, as a fully incremental CMA would need to be somewhat larger than the one shown. From the CMA representation of a clause match, it should be apparent how to generate the matching code for an abstract instruction set. The states of the CMA are encoded in the labels of the matching program, and the transitions from state to state are performed by matching instructions.

5 A Heuristic For Efficient CMA Construction

There is a large amount of freedom in the construction of correct CMAs for a given procedure. Indeed naive algorithms can be implemented via CMAs. We would like to do better than this, however, along several dimensions. In order of importance:

1. We want our CMA to perform a minimal number of suspensions during any clause match. Because suspension significantly dominates the cost of matching, such a CMA improves performance even for programs that suspend with moderate frequency.

2. We want an incremental CMA. As described previously, the CMA technique allows incremental matching, and there are applications with large procedures (e.g., our Monaco assembler has a procedure with over 100 clauses), where it is of tremendous performance benefit.

3. We want a CMA requiring minimal time (linear tree height).

4. We want a CMA of minimal size. In particular, it would be nice if the CMA was of polynomial size.

Achieving the first two goals with a CMA is straightforward. The final two goals, however, are provably difficult; in fact of intractable worst-case complexity [5]. Thus, we need a heuristic CMA construction technique which appears to approximate the third and fourth goals well in most cases likely to be encountered in practice, and which does not violate the first two goals.

A CMA construction heuristic we call *maxmin* aims to satisfy these goals. The basic idea behind maxmin is similar to that of Kliger's decision graph heuristics, namely, testing "important" arguments first. Indeed, in many common cases the CMA produced by adapting Kliger's heuristics will be the same as those produced by maxmin. However, we believe that we can improve upon Kliger's heuristics, and thus produce reasonable code size in almost all practical cases, using our maxmin heuristic.

The role of the heuristic in CMA construction is to choose, for each CMA match state, what tests should be performed. These tests produce successor states, and thus control the shape of the graph. We say a heuristic is "admissible" if it always reduces a match state, in the sense of the previous section. Thus, the real purpose of the heuristic is to select the *order* in which tests are performed. Consider the left-to-right heuristic which is certainly admissible because it will always either reduce the current state by testing or delay by placing a column in k_α. However, there are cases in which a naive left-to-right match is far from optimal.

1	1	1
×	2	2
×	×	3

(a) Left-To-Right

a	c	§1
a	d	§2
a	e	§3
×	f	§4
b	g	§5
b	h	§6
×	i	§7
b	×	§8

(b) Kliger's

Figure 4: Pathological Cases for Heuristics

Consider the initial match state of Figure 4a. In this state, a left-to-right match may discover that the first argument is bound to 1, and the second to 1, but that the third is unbound. Upon resuming after the third argument is bound, the match may discover that this argument is bound to the value 3. But in this case, the matches against the first and second arguments were redundant. Worse yet, a naive left-to-right match, upon finding no arguments bound, has to suspend on *all three* arguments. When it resumes to find the first argument bound to 1 but the second and third still unbound, it will be forced to immediately suspend again to obtain these values. All of this illustrates that an inferior match heuristic may have both unnecessary suspensions and poor incrementality.

When adapted to CMA construction, Kliger's heuristic (*max caring + min variability*) is an admissible heuristic, because it merely changes the order in which arguments are considered during pattern matching. The concept behind Kliger's heuristic is to perform tests which are most likely to be "useful" earliest, where a test is useful if it gives good information about clause selection whether it succeeds or fails. Unfortunately, Kliger's heuristic is of questionable utility in some cases. Consider the partial procedure of Figure 4b. The max caring heuristic suggests that the second argument be tested first. This will require a boundness test and a seven-way switch. Note that in the case where the second argument is unbound, a boundness test and a two-way switch will still be required on the first argument before suspension. If the arguments were tested in the other order, however, the test on the first argument would eliminate three of the clauses if the first argument was bound to a matching value (*i.e.,* a or b), and six of the clauses if it was bound to some other value. This problem seems to arise because the primary heuristic, max caring, sometimes conflicts with the secondary heuristic, min variability. These conflicts arise in practice so it is desirable to design a single heuristic which smoothly incorporates both of these ideas.

Our maxmin heuristic attempts to achieve the desired effects of Kliger's max caring and min variability heuristics in a single heuristic, effectively incorporating the philosophy of Debray's probabilistic technique. It is an open question whether our method is equivalent to Debray's method when no clause selection frequency information is known. The basic idea behind our heuristic is the desire to have each test eliminate as many clauses from consideration as possible. The difficulty is that in some clauses some argument values are not needed, either because they correspond to variables in the clause head, or because they

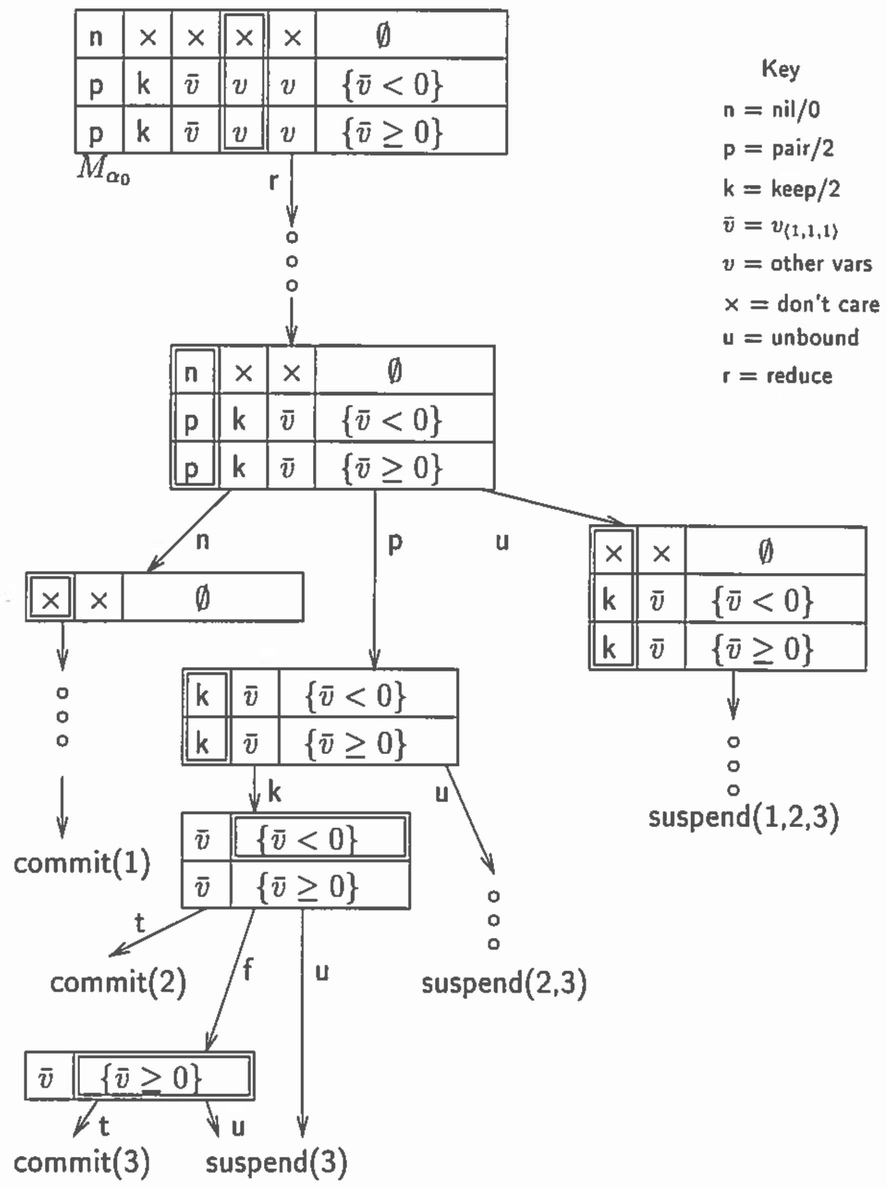


Figure 5: Example CMA for Left-to-Right Match

are not part of the clause head pattern. Thus, these clauses will *not* be eliminated by a match on this argument. We formalize this notion by the notion of a set of *segments* of a procedure argument: a segment is the set of clauses which are still match candidates if the argument takes on some value. This vaguely corresponds to Kliger's residuals [6]. Thus, each segment of a procedure argument will contain all the clauses which do not care about the argument, plus all the clauses where the argument takes on some particular value. Thus, the segment set for the first argument of Figure 4b is

$$\begin{gathered}\{\{\S1,\S2,\S3,\S4,\S7\},\\ \{\S5,\S6,\S8,\S4,\S7\}\}\end{gathered}$$

and the segment set for the second argument is

$$\begin{gathered}\{\{\S1,\S8\},\{\S2,\S8\},\{\S3,\S8\},\{\S4,\S8\},\\ \{\S5,\S8\},\{\S6,\S8\},\{\S7,\S8\}\}\end{gathered}$$

The maxmin heuristic is as follows: select for matching a procedure argument with the maximum-size minimum segment. Note that in many common cases this will achieve the effect of Kliger's minimum-variability heuristic: An argument whose values are shared by many clauses will have large segments, and thus be selected, since it has low variability. On the other hand, an argument's caring is not directly a factor in the maxmin heuristic: the discriminatory power of an argument is more directly measured by segmentation. The basic effect to be achieved is that of putting nodes with low branching factors near the top of the tree, and our heuristic accomplishes this.

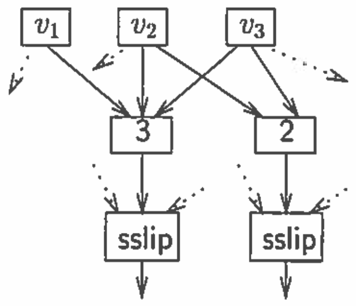
The maxmin heuristic is merely an ordering heuristic for clause tests, so like the other heuristics it is admissible. We cannot provably achieve minimum suspension, but we will reduce the number of suspensions, since we will attempt to perform important tests first. We achieve incrementality by restarting the CMA in the lowest safe match state using the techniques previously described. Because the maxmin heuristic is essentially an improvement of the Kliger heuristic, the time efficiency of maxmin CMAs should be acceptable.

The size of the CMA generated using our heuristic is problematic. Our CMAs should typically be no larger than Kliger decision *trees*. Further, there is reason to believe that our heuristic will in some cases avoid the pathologically large code sizes produced by Kliger's heuristic. We have not yet explored decision *graph* construction using CMAs.

6 Disjuncts of Conjuncts

As we have seen, the incrementality of our technique arises primarily from the inclusion of a resumption continuation in the suspension instruction. This might naturally lead one to ask whether other small modifications in the runtime support for pattern matching might lead to more efficient matches.

One particular source of inefficiency is that, regardless of the quality of the heuristic used, there may be cases in which CMA execution is resumed, and then suspended again. This results from the fact that the runtime system only allows suspension on *disjuncts* of arguments, i.e., matching will be resumed when *any* argument becomes bound. The



(a) Implementation

§1	§2	§3	§4
*	*	_	*
*	_	*	_
_	*	_	_

§1	§3	§4
*	_	*
*	*	_

(b) Abstract Matrices

Figure 6: Disjunct of Conjuncts

case where the match immediately suspends again on some subset of the arguments it was suspended on previously is relatively common, and relatively expensive. What is needed is the ability to suspend on *conjuncts* of arguments, resuming only when all arguments in the conjunction are bound. Merely suspending on conjuncts is insufficient for correct implementation: because of non-determinism, it may be impossible to tell which of several arguments will become bound and allow a match to complete.

Thus, an improvement in functionality is to allow suspension on a disjunction of conjuncts, i.e., suspend until one of several sets of arguments is entirely bound. This can be implemented in a straightforward fashion using the technique illustrated in Figure 6a: rather than being hooked directly to suspension slips, variables are hooked to a *conjunct counter* which is initialized to the number of variables in its conjunct, and which itself points to the suspension slip for a goal. When a variable is bound, its conjunct counters are decremented: only when a conjunct counter reaches zero is its goal awakened. Note that this strategy may convert a program which would otherwise suffer match failure into one which instead deadlocks; this is acceptable under most CCL semantics.

Disjuncts of conjuncts of suspension sets can be generated as follows. Elements of the initial argument matrix are abstracted to either "*" if they are ground or guard inputs, or "_" if they are variables or don't cares ($\times$). One example is shown on the left in Figure 6b. At a leaf match state, where suspension will occur, examine k_α. Construct a new abstract matrix in two steps. First, remove every column corresponding to an argument not in k_α. Next, remove all empty rows. A row in the resulting abstract matrix represents a conjunct of suspended arguments. The rows together represent a disjunct: any of them can resume the suspended match. Returning to Figure 6b, suppose k_α contains §1, §3, and §4. The transformed matrix is shown on the right. We suspend on (§1 $\wedge$ §4) $\vee$ (§1 $\wedge$ §3).

7 Conclusions

We have shown how a unified approach to committed-choice language clause matching can be achieved with a *clause matching automata* (CMA) formalism. This formalism is beneficial for several reasons. First, it is general, allowing a number of clause matching techniques to be described. Second, it is easy to understand and use in code generation. Finally, and most importantly, it allows some extensions to produce efficient clause matchers. In particular, through a minor runtime system extension, it allows incrementality.

We have also suggested a clause matching heuristic, *maxmin*, which should allow good clause-matching performance in real programs, for two reasons. First, it leads to minimal numbers of suspensions. Second, it tests in an efficient order. This heuristic is similar to Kliger's heuristic for decision trees and graphs.

Finally, we have suggested an implementation technique by which the effectiveness of incremental suspension might be enhanced. This technique allows suspension on disjuncts of conjuncts in order to assure that a match may complete on resumption.

Future work includes implementing the scheme and measuring its utility. Formal proofs of the correctness and complexity of the technique would be useful. In addition, further examination of the work of Podelski and Van Roy [10] should be done in the context of CMAs.

Acknowledgements

B. Massey was supported by a grant from the Institute of New Generation Computer Technology (ICOT). E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Inc.

References

[1] Z. M. Ariola, B. C. Massey, M. Sami, and E. Tick. Compilation of Concurrent Declarative Languages. *New Generation Computing*, 1995. Submitted to special issue. Also available as University of Oregon Technical Report CIS-TR-94-05.

[2] S. K. Debray, S. Kannan, and M. Paithane. Weighted Decision Trees. In *Joint International Conference and Symposium on Logic Programming*, pages 654–668. Washington D.C., MIT Press, November 1992.

[3] C.L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[4] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley Ltd., Wokingham, England, 1987.

[5] T. Hickey and S. Mudambi. Global Compilation of Prolog. *Journal of Logic Programming*, 7(3):193–230, November 1989.

[6] S. Kliger. *Compiling Concurrent Logic Programming Languages.* PhD thesis, The Weizmann Institute of Science, Rehovot, October 1992.

[7] S. Kliger and E. Y. Shapiro. A Decision Tree Compilation Algorithm for FCP(|,:,?). In *International Conference and Symposium on Logic Programming*, pages 1315–1336. University of Washington, MIT Press, August 1988.

[8] S. Kliger and E. Y. Shapiro. From Decision Trees to Decision Graphs. In *North American Conference on Logic Programming*, pages 97–116. Austin, MIT Press, October 1990.

[9] S. L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International Ltd., 1987. Chapters 4 and 5.

[10] A. Podelski and P. L. Van Roy. The Beauty and the Beast Algorithm: Testing Entailment and Disentailment Incrementally. In *International Logic Programming Symposium.* MIT Press, November 1994. In press.

[11] R. Ramesh, I.V. Ramakrishnan, and D.S. Warren. Automata-Driven Indexing of Prolog Clauses. *Journal of Logic Programming*, 1994. In press.

[12] P. L. Van Roy. *Fast Logic Program Execution: Design and Implementation of the Aquarius Prolog Compiler.* Intellect Books, Oxford, 1993.

[13] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[14] E. Tick. Monaco: A High-Performance Flat Concurrent Logic Programming System. In *PARLE: Conference on Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 266–278. Springer Verlag, June 1993.

[15] E. Tick. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming*, 1995. In Press. (Also available as University of Oregon Technical Report CIS-TR-94-07).

[16] E. Tick and M. Korsloot. Determinacy Testing for Nondeterminate Logic Programming Languages. *ACM Transactions on Programming Languages and Systems*, 16(1):3–34, January 1994.