Supporting and Optimizing Full Unification in a Forward Chaining Rule System

Howard E. Shrobe
Massachusetts Institute of Technology
NE43-839
Cambridge, MA 02139
hes@zermatt.lcs.mit.edu

Abstract

The Rete and Treat algorithms are considered the most efficient implementation techniques for Forward Chaining rule systems. These algorithms support a language of limited expressive power. Assertions are not allowed to contain variables, making universal quantification impossible to express except as a rule. In this paper we show how to support full unification in these algorithms. We also show that: Supporting full unification is costly; Full unification is not used frequently; A combination of compile time and run time checks can determine when full unification is not needed. We present data to show that the cost of supporting full unification can be reduced in proportion to the degree that it isn't employed and that for many practical systems this cost is negligible.

1 Introduction

Relatively efficient mechanisms have been developed for the implementation of forward chaining rules [1; 4]. However, these mechanisms have mainly been used in the implementation of the OPS family of production system languages. Languages in this family have limited expressive power: they are pure forward chaining languages in which assertions are restricted to ground terms.[1]

In this paper we explore the use of these mechanisms in more expressive languages in the tradition of [7; 2; 3; 6]. Such languages work by *pattern directed procedure invocation*. They center around a database of assertions accessed by forward and backward chaining rules as well as by normal procedural code using a *Tell* and *Ask* interface. The bodies of rules in such languages may be full procedures. Such languages naturally fall into the full unification case: Assertions containing variables take on the force of universally quantified statements and these may match the patterns of either forward or backward chaining rules.

However, the designers of the OPS family of languages did not choose the limitation to the semi-unification case naively. Full unification significantly complicates the Rete mechanisms and leads to two forms of inefficiency:

- The resulting code executes more slowly.
- The resulting code is significantly larger.

[1] In the remainder of the paper, we will refer to a language which only allows ground terms in assertions as a "semi-unification" language. If assertions may contain variables we will refer to the language as a "full-unification" language.

Moreover, in many domains the semi-unification case closely approximates the needed expressive power; full unification is rarely required.

In this paper, we show that one can have both the expressive power of full unification and the efficiency of the techniques developed for the more limited semi-unification case. The mechanisms described here have been implemented and extensively used in the Joshua system[5].

The outline for the remainder of the paper is as follows: In section 2 we begin by reviewing the conventional Rete network, describing it as a mechanism for incrementally computing unifications between a set of patterns and a set of assertions (an unconventional viewpoint). We then show in section 2.1 that conventional Rete networks are an optimization of our viewpoint for the semi-unification case. In section 3, we describe our extensions to support full unification; section 4 presents data to show the degree of inefficiency introduced by our extensions. The next two sections present techniques for addressing the inefficiencies. Section 5 presents a set of run time optimizations that dynamically identify when semi unification alone is adequate; we show that this reduces the first form of inefficiency to negligible levels. Section 6 shows how the rule compiler can statically segregate out portions of the rule system which can be compiled under the semi unification assumption; we show that this reduces the second form of inefficiency to acceptable levels.

2 Rete Networks

We assume that the reader is familiar with Rete networks and related techniques (see, for example, [1]). In describing Rete networks, our terminology will be somewhat nonstandard; we will refer to *unification* rather than matching in an attempt to show how our extensions fit within the pattern of the original algorithm.

Rete networks incrementally maintain the partial triggering states of rules as new assertions are added and deleted. A rule is fully triggered when its set of patterns unify with a set of statements asserted in the database. Partial triggering states contain two kinds of information: 1) The unifications between individual patterns and individual assertions and 2) Extended unifications between subsets of a rule's patterns and sets of assertions. Figure 1 shows a pair of rules and the corresponding Rete network.

The network contains two sections: Match and Join. Each of these incrementally updates its internal state each time a new token is added to (or deleted from) its input nodes.

The match section is a discrimination network whose

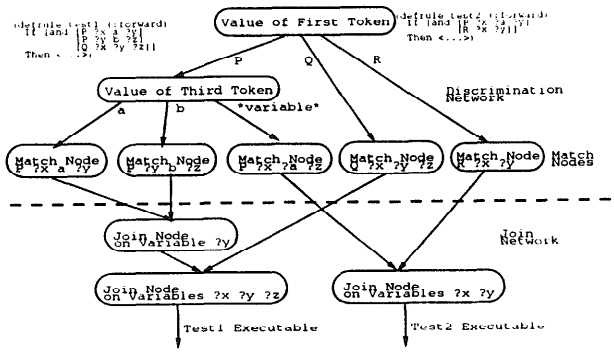

Figure 1: A Typical Rete Network

terminal nodes (the match nodes) compute unifications between rule patterns and assertions in the database. State is stored only at the match nodes (these are the alpha memories of [1]). Patterns from different rules which are *variants* (i.e. identical up to variable renaming) share match nodes; two patterns which share leading terms share a path through the network up to the point of divergence. The tests made by the nodes above the match nodes filter out assertions which cannot possibly unify with the pattern of the match nodes below them.

The Join network begins at the Match nodes. State is stored at all join nodes (these are the beta memories in the terminology of [1]). Each node of the join section merges the partial unifications represented by its two parent nodes, checking that the shared variables are unifiable. Each terminal node of the join network corresponds to a complete set of rule patterns. If two rules share leading patterns, they share join nodes up to the point of divergence.

The rete network compiler emits code for each node in the network to perform the above functions.[2]The code for match and join nodes perform the indicated unifications; they also package up the results into *state tokens* stored at the node.[3]

2.1 Optimizations for the Semi Unification Case

In classic Rete networks all assertions contain only ground terms and therefore no variable in a rule's pattern may ever be bound to another variable. Under these conditions the Rete network can be viewed as computing relational selects (at the match nodes) and relational joins (at the join nodes) (as pointed out in [4]).

Operationally matching reduces to: 1) checking that constants in the assertion are equal to corresponding constants in the pattern and 2) checking that terms of an assertion which match different occurrences of the same variable are equal.

When assertions contain only ground terms the discrimination nodes perform part of the unification by testing for equality between constants in the pattern and constants in the assertion. Therefore, the code at a match node may omit these tests.

Similarly, the tests at the join nodes can be reduced to checking that variables shared between the parents of the join are bound to equal values. Hashing (or other forms of indexing) may be used to speed up the join computation. Each parent of a join node maintains a hash-table of tokens; the key for this table is a list of the values of the shared variables.

In the semi-unification case, a hash probe will find the precise set of unifiable tokens; therefore, no other code is needed at the join nodes.

Finally, in the semi-unification case, the state tokens need not contain a variable binding environment; each variable can be identified with a particular term from one of the matching assertions.

These optimizations are not fully available in the full-unification case.

3 Extending the Rete Network to Support Full Unification

We have described the Rete algorithm in a very general context, that of computing and extending unifications. The semi-unification case allows a variety of optimizations to be made by replacing unification with equality tests. To extend the traditional Rete algorithm to support the full-unification case we must undo these optimizations, replacing equality checks by unifications.

The following questions must be addressed:

1. How are logic variables represented?
2. What code is compiled to conduct the unification matching?
3. How are state-tokens represented and computed?
4. How do auxiliary indices (e.g. hash-tables at join nodes) handle logic variables?

Having made these choices we will then need to see what impact they have on the components of the Rete network.

3.1 Data Structures and Basic Operations

3.1.1 Representation of Logic Variables

We adopt a representation for logic variables based on the Prolog oriented techniques of the Warren Abstract Machine [8].

Logic variables are represented as pointers to their values; an unbound logic variable points to itself. An unbound logic variable is unified with a value by making it to point to the value; this value might be another unbound logic-variable which might later be bound to a value, leading to a chain of logic-variable pointers as shown in figure 3.

To find the value of a logic variable one must follow the chain of pointers until encountering either a value which is not a logic variable or a logic variable which points to

[2]In the Joshua system, the methods for generating this code are customizable by the user; this is part of the Protocol of Inference see [5].

[3]Joshua allows user-supplied procedural condition elements; these require a special node type. Procedural nodes are attached to a single parent node; they contain the original procedure surrounded by supporting code which generates a new token each time the procedure "succeeds". For brevity, we will not further discuss these condition elements.

```
(lambda (assertion)
 (with-unification
  (with-unbound-logic-variables (?x)
   (unify 'P (dereference (pop assertion)))
   (unify ?x (dereference (pop assertion)))
   (unify 'a (dereference (pop assertion)))
   (unify (dereference ?x) (dereference (pop assertion)))
   (unify 'b (dereference (pop assertion)))
   ... code to be executed upon success ...)))
```

Figure 2: Full Unification Code Corresponding to [P ?x a ?x b]

itself. This operation is referred to as *dereferencing.* A logic variable must be dereferenced before its use.

When a logic variable is bound, an entry consisting of the logic variable is made on a stack called the *trail.* Before a pattern matching operation is begun, the level of the trail is saved. To return to the binding state which obtained at the beginning of the operation (e.g. when the unification fails) each logic variable above the marked point on the trail is reset to point to itself and the trail level is reset to the marked point. This operation is usually called *unwinding the trail,* or *untrailing.*

3.1.2 Implementation of Unification

The match compiler is responsible for emitting the unification code corresponding to a pattern. When given an assertion to match, the code must *fail* if the pattern and the assertion are not unifiable; otherwise it must *succeed* and bind the logic-variables of the pattern to the values implied by the unification.

Figure 2 shows the code emitted for the pattern [P ?x a ?x b].

In this code, *With-unification* establishes a unification context (i.e. it notes the level of the trail on entry and unwinds the trail to that level upon exit. Also it establishes a catch tag which is thrown to in the event of *failure. With-unbound-logic-variables* creates a set of new logic-variables (typically these are stack allocated). Notice that each term of the assertion must be dereferenced before the call to *Unify* since the term might be a logic variable.

Unify is called with atomic elements (including logic variables) as the first argument; when the pattern contains compound terms, the match compiler must recurse into the substructure of these terms. For simplicity of presentation we omit the details, see [8]. The behavior of Unify is as follows:

- If neither argument is a logic-variable, then UNIFY succeeds if the arguments are EQUAL and otherwise fails.
- If exactly one argument is a logic-variable, UNIFY succeeds, the logic-variable is bound to the other argument and a trail entry is made.[4]
- If both arguments are logic-variables then one is bound to the other a trail entry is made and UNIFY succeeds. (If both logic-variables are stored on the stack, then the one pushed more recently must point to the one more deeply nested).

[4]The unification is only allowed if the variable does not occur within the structure of the other arguments. Prolog implementations typically skip this "free-for" check for efficiency as do we in our implementation.

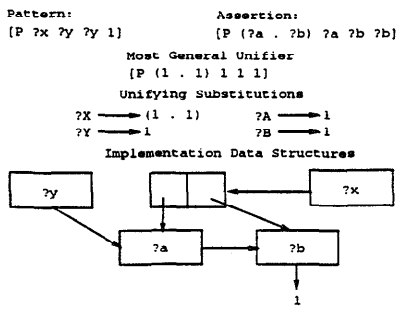

Figure 3: A Unification and its Implementation Level View

Failing is accomplished by throwing the value NIL to a catch-tag for *FAIL.* This is normally established by *with-unification*; this causes the trail to be unwound.[5]

3.1.3 Saving the Binding State in Tokens

The code emitted by the Rete network compiler for a match node tests whether the triggering assertion can be unified with the rule pattern; if so it produces a state-token containing the bindings of the pattern's logic variables.

Consider the unification shown in figure 3. The variable ?x of the rule's pattern is unified with the list (?a . ?b) of the assertion; this list contains variables which are bound to ground terms (e.g. 1). The value of ?x is valid, therefore, only as long as ?a and ?b continue to be bound to 1. However, ?a and ?b are contained in a database assertion whose intent is to state a universal quantification. Therefore, the binding of ?a and ?b must be untrailed and the values of their current bindings must be preserved elsewhere.[6] Notice that this is quite a bit more expensive than the semi-unification case where the assertion can serve as an adequate representation of the binding state as explained in 2.1.

To preserve the volatile binding state over a longer duration, state-tokens maintain an *environment* of logic-variable values with a slot for each variable in the pattern. Each slot is filled with the *unified-value* of its corresponding logic-variable. The unified value of a logic-variable is computed as follows:

- The logic-variable is dereferenced.
- If the variable is unbound, its unified value is a new logic-variable. All occurrence of a particular unbound logic-variable have the same new logic-variable as their unified-value.[7]

[5]In the implementations of Joshua on Symbolics equipment, Dereference is a microcoded instruction. In implementations on more conventional machines it would be implemented either as subroutines or an inline code fragment; either approach is both slower and consumes more instructions. Our measurements are made on Symbolics equipment, yielding more favorable results for the full-unification case than would result on more conventional machines.

[6]I.e. our implementation uses a shallow binding scheme for logic variables but needs to preserve their values beyond the dynamic extent.

[7]Unbound logic-variables are replaced by new variables to

- If the variable is bound and the bound value is atomic, then the unified-value is the bound-value.
- If the bound value is a compound data-structure, then the sub-structure is traversed replacing each term by its unified value.

In the example of figure 3 the logic-variable ?x is bound to the pair (?a . ?b). But ?a is bound to ?b which is in turn bound to 1; so the unified value of ?x is (1 . 1), a value which persists even after unwinding the trail.

3.2 Extending the Algorithm

3.2.1 The Discrimination Network

We begin with the discrimination nodes of the Match network. Each discrimination node dispatches on the value of a term in the assertion, see figure1. For large branching factors, a hash table is an appropriate implementation. If the term being dispatched on is a constant then it serves as the hash-key. The value retrieved is the next discrimination node to visit. If there is a branch for the key *variable* (indicating a rule pattern with a variable at this position), this must also be followed.

Notice that the term being discriminated on may itself be a logic-variable. In this case all outgoing branches must be followed, since a variable can match anything.[8]

3.2.2 The Match Nodes

The discrimination network search discards most match nodes that don't unify with the assertion; however, some non-unifiable match nodes may still be reached. For example:

```
Rule Pattern:     (P a  ?c  b ?c)
Assertion:        (P ?x  c ?x  c)
```

The discrimination network treats each occurrence of ?x in the assertion as independent, allowing this assertion to reach the match node although it isn't unifiable with the pattern. Notice that the inconsistency between the assertion and the pattern occurs at constant terms in the pattern. As mentioned in section 2.1, this can never happen in the semi unification case and the match code need only check the positions corresponding to variables.

In contrast, the full unification match code must perform the entire unification as explained in section 3.1.2 (i.e. tests must be generated for both constant and variable positions). It must also save the results in a binding vector by copying out the unified values, as explained in section3.1.3. The match compiler, therefore, emits code containing two sections: The first conducts the unifications, the second creates the binding vector and fill it with the unified values of the logic variables.

3.2.3 The Join Nodes

Join nodes are extended in a similar manner. The rule compiler generates a map for each join node specifying which variables from the two parent nodes are to be unified. The compiler emits code to perform these unifications and to copy the unified values of all the variables into a new state-token. Figure 4 shows a join to be performed and the corresponding code generated by the rule compiler.

```
(P ?x ?y ?z) environment ?x → 1 ?y → 2 ?z → 3
(Q ?y ?z ?w) environment ?y → 1 ?z → 2 ?w → 3

(lambda (token-1 token-2)
 (with-unification
  ;; unify ?y from 1 with ?y from 2
  (unify (token-slot token-1 2) (token-slot token-2 1))
  ;; unify ?z from 1 with ?z from 2
  (unify (token-slot token-1 3) (token-slot token-2 2))
  (let ((new-token (make-new-token :n-variables 3)))
   ;; copy ?x
   (setf (token-slot new-token 1) (copy-unified-value (token-slot token-1 1)))
   ;; copy ?y
   (setf (token-slot new-token 2) (copy-unified-value (token-slot token-1 2)))
   ;; copy ?z
   (setf (token-slot new-token 3) (copy-unified-value (token-slot token-1 3)))
   ;; copy ?w
   (setf (token-slot new-token 4) (copy-unified-value (token-slot token-2 3)))
   new-token)))
```

Figure 4: Join Code for The Full Unification Case

Many Rete network implementations use hashing (or other indexing) to speed up the join computation, as explained in section 2.1. In the full unification case, any of the shared variables in either token might be an unbound logic-variable. Unlike ground terms, two distinct logic variables might match; a list of the values of the shared variables is, therefore, not an adequate retrieval key. A simple extension which solves this problem is as follows:

- When storing a new token in a node:
  - If any of the shared variables are unbound, then hash the token under a special key: **unbound-variable**.
  - Otherwise use the list of the shared variable values as the key.
- When looking for stored tokens to join with a new token:
  - If any of the new token's shared variables are unbound then look at every token stored in the other parent node.
  - Otherwise form a key which is the list of shared variables and attempt to join with every stored token hashed in the other parent node under this key. Also attempt to join with every token hashed under the key **unbound-variable**.

3.2.4 Compiling Rule Bodies

Forward rule bodies may contain normal procedural code which references the logic variables of the patterns. In the full unification case, variables referenced in the body of a rule may be left unbound by the matching process; they therefore must be treated as logic variables and be dereferenced before being used.

The values of the logic-variables are stored in the environment of the triggering state token. The rule compiler, therefore, first emits a prologue which fetches the variable values from the environment into local variables. The rest of the rule body code is transformed so that every reference to a logic-variable is wrapped within a call to *dereference*.

prevent sharing of logic-variables held in state tokens with those in assertions (or other state-tokens). Were this not done, the unifications performed at join nodes would unintentionally bind the variables in the assertions. Resolution systems rename variables in the resolvent for the same reason.

[8] We do not attempt to carry along the variable bindings while traversing the discrimination network.

example	Time in Rete Network		Total Run Time	
	Semi	Full	Semi	Full
Natural Deduction	14,114	84,725	167,816	360,740
Troubleshooting	79,353	346,726	645,634	943,602
Cryptarithmetic	851,288	2,872,276	3,489,086	7,309,309
Planning	209,308	739,605	891,742	1,413,495

Table 1: Run Times of Full and Semi Unification Code

4 Full Unification Support is Costly

These extensions lead to semantically correct behavior in the full unification case. However, as can also be seen, each extension removes a constraint of the semi-unification case which was used in optimizing the original algorithm.

Table 1 shows the relative performance of the matching and merging portions of a a number of demonstration systems. Table 2 shows the relative sizes of the generated code for a variety of systems.

The following are among the causes of this difference:

- Equality tests in the match and join code are replaced by unification.
- The discrimination network in the match network acts only as a partial filter in the full unification case (due to the possibility of logic-variables in the assertions, see section 3.2.1). The match code generator cannot assume that every constant has already been checked and must generate checks for the constants as well as the variables.
- Hashing alone is a sufficient join test in the semi-unification case. In the full-unification case this is not true. A rather bulky merge procedure must be still be generated and called, see section 3.2.3.
- The code generated for rule bodies must replace variable references by calls to *dereference*. This incurs both increased code size and slower performance.
- In the semi-unification case, state tokens need not contain environments. In the full unification case environments must be built and values copied between them.
- Calls to *copy-unified-value* must be used to copy logic-variable values to new state tokens. If variables are bound to compound data-structures this incurs the cost of traversing and incrementally rebuilding those sub-structures containing logic-variables.
- In the semi-unification case, match procedures typically check only the terms corresponding to variables in the pattern; constants in the pattern are ignored since the discrimination nodes check them. This means that two match nodes which have the same pattern of variable occurrences but distinct constants may nevertheless share match procedures. This is not the case for the full unification case, only variant patterns may share match procedures.

5 Dynamic Optimizations

The programming style of typical knowledge based systems infrequently employs the full unification case. Unfortunately, the expressiveness of the full unification case leads to code with much worse run-time performance.

This section addresses one approach to this problem: optimizations performed at run-time.

We have extended the rete network compiler to generate two sets of procedures for each of the match and join nodes. The first of these is the less efficient but more general code capable of handling the full unification case; the second procedure handles only the semi-unification case, but is considerably more efficient. The rete network interpreter is responsible for dispatching to the semi-unification procedure if allowable, otherwise it must call the full unification procedure.

To make this decision, our system checks each newly created assertion for the presence of logic-variables and stores the result in the data-structure representing the assertion. (The check must be made in any event, since all logic-variables in a database assertion must be copied so as to be unique to that assertion). At a match node, the rete interpreter uses this information to determine which procedure to call.

When a new state-token is created, a we check whether any element of the environment is an unbound logic variable; the token is marked with the result of this check. At a join node the semi-unification code is called if both input tokens are marked as logic-variable free. Notice that only the full unification procedures need to check for the presence of unbound logic-variables in the output token since in the semi unification case the output will necessarily contain only ground terms.

The crucial question for a dynamic optimization is whether the cost of detecting the opportunity swamps out the resulting benefit. In this case, the detection cost is that incurred in checking assertions for non-ground terms and (if we're running a full-unification procedure) the cost of a similar check on any newly generated token. Metering indicates that this consumes about 1.5% of total run time.

To test the efficacy of dynamic optimization, we ran a rule-based natural deduction system on three versions of the same problem. The first version uses only ground terms, the second has a mix of ground terms and terms with variables and the third version is completely quantified. In the first case, all matches and joins used the semi-unification code and ran the problem 6.9 times faster than would the full-unification code. In the second case, 86% of the matches and 59% of the joins used the semi-unification code with a resulting speedup of 2.4. In the last case, of course, all matches and joins used the full-unification code. These results show that the system is highly effective in dynamically identifying when the semi unification code can be utilized.

6 Static Optimizations

The dynamic optimizations incur the additional cost of generating two procedures at each node. A traditional production system would generate only the semi unification code; but our system also generates code to support the rarer case of full unification. As table 2 indicates this code is considerably larger. Furthermore, we generate only a single version of the code for rule bodies which is required to be the slower and bulkier code capable of handling full unification.

In this section, we discuss how we use information avail-

example	Semi Unification							Full Unification				
	Rule	Matchers	Size	Mergers	Size	Proc-edures	Rule Body	Matchers	Size	Mergers	Size	Proc-edures	Rule Body
Circuits	10	2	37	2	44	0	212	7	364	2	260	0	214
Cryptarithmetic	59	7	188	34	979	307	2251	14	703	34	6816	628	2269
Midsummer	7	3	56	3	68	46	124	6	264	3	372	73	124
Discrete Event	7	4	81	4	94	20	237	7	361	4	544	36	237
Ht Atms	7	4	82	3	66	64	153	8	412	3	390	120	153
Ht Ltms	15	5	120	6	146	80	391	15	754	6	849	169	397
NatDed	7	6	266	4	109	0	425	8	508	4	672	0	425
All Rules	123	36	950	50	1380	587	4009	74	3797	50	9186	1102	4035

Table 2: Compiled Code Size of Full and Semi Unification Code

able at compile time to help the rule compiler determine which nodes of the network (as well as which rule bodies) will never encounter the full-unification case. This lets us generate only the more efficient semi unification code at any node or rule body so classified.

In the Joshua system, all operations of the system (including the operations of the rule compiler) are driven by the *class* of the assertion (or pattern) being processed. The classes are identified with the predicate of the assertion and are, in fact, CLOS classes; see [5] for details. The data structures used to represent rule patterns and data-base assertions are instances of these CLOS classes.

One such predicate class (which can be "mixed in" to any other assertion class) is *no-variables-in-data-mixin*. If one tries to enter an assertion of this type into the database, an error is signalled. Rule patterns which are instances of this class can therefore reliably assume that any triggering assertion will contain only ground terms. This information is used at rule compilation time to determine that a match node will be "semi unification only". A join node whose parent nodes are both "semi unification only" will also be "semi unification only". If a terminal node of the rete network is "semi unification only" then all rule bodies connected to that node are also "semi unification only".[9]

Table 2 shows the relative sizes of the full-unification and semi-unification code for 7 systems. In aggregate the full unification matchers are 4 times larger than the semi-unification matchers; the full unification mergers are 6.7 times larger. With static optimizations applied, our system generates no full unification code for most of the systems (this is optimum, these systems never use the full unification capability). The system is forced to generate both versions of the code for the one subsystem which does take advantage of the full unification. In aggregate, this save 86% of the matching code and 92% of the merging code.

7 Discussion

We have shown that the expressiveness of full unification can be supported with very limited cost in efficiency. This result depends on the statistics of assertion usage: most assertions contain only ground terms. This allows us to generate optimized procedures for the semi unification case. If the triggering assertions all contain only ground terms then the more efficient semi unification case is called; this is very frequently the case. The system's performance gracefully degrades as universally quantified assertions are entered in the database.

Also we have shown that when extra information is conveyed to the system at compile time, we can avoid generating the bulkier code for the general case. Furthermore, we have indicated that in many practical cases, this *a priori* information is obtainable.

[9] Our system also supports user supplied procedural condition elements. If such a node's parent is semi-unification only and it introduces no new logic variables, then the node itself is semi-unification only.

References

[1] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–38, 1982.

[2] C.E. Hewitt. Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot. Technical Report AI-TR-258, MIT Artificial Intelligence Laboratory, 1972.

[3] G.J. McDermott, D.V.and Sussman. The conniver reference manual. Technical Report Memo 259, MIT Artificial Intelligence Laboratory, 1972.

[4] Daniel P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Morgan Kaufmann, San Mateo, California, 1990.

[5] S. Rowley, H. Shrobe, R. Cassels, and W. Hamscher. Joshua: Uniform access to heterogeneous knowledge structures (or why joshing is better than conniving or planning). In *National Conference on Artificial Intelligence*, pages 48–52. AAAI, 1987.

[6] G.J. Sussman and D.V. McDermott. Why conniving is better than planning. Technical Report AI Memo 255A, MIT Artificial Intelligence Laboratory, Cambridge Mass., 1972.

[7] G.J. Sussman, T. Winograd, and E. Charniak. The micro-planner reference manual. Technical Report AI Memo 203, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1970.

[8] D.H.D Warren. An abstract prolog instruction set. Technical Report SRI Technical Note 309, SRI International, October 1983.