DATA-DRIVEN BACKWARD CHAINING

Paul Haley
The Haley Enterprise, Inc.
413 Orchard Street
Sewickley, PA 14153
USA
(412) 741-6420

Abstract: CLIPS cannot effectively perform sound and complete logical inference in most real-world contexts. The problem facing CLIPS is its lack of goal generation. Without automatic goal generation and maintenance, Forward chaining can only deduce all instances of a relationship. Backward chaining, which requires goal generation, allows deduction of only that subset of what is logically true which is also relevant to ongoing problem solving.

Goal generation can be mimicked in simple cases using forward chaining. However, such mimicry requires manual coding of additional rules which can assert an inadequate goal representation for every condition in every rule that can have corresponding facts derived by backward chaining. In general, for N rules with an average of M conditions per rule the number of goal generation rules required is on the order of N*M. This is clearly intractable from a program maintenance perspective.

We describe the support in Eclipse for backward chaining which automatically asserts as it checks rule conditions. Important characteristics of this extension are that it does not assert goals which cannot match any rule conditions, that 2 equivalent goals are never asserted, and that goals persist as long as, but no longer than, they remain relevant.

Introduction

Suppose we were developing an application concerning genetically transmitted traits. Our application might need several rules that guided its reasoning. One such rule might be, "if a person has a trait and a cousin of that person has the same trait, then consider the possibility that the trait is inherited." Such a rule might be coded as follows:

```
(defrule cousins-may-inherit-trait
     (has ?grandchild-1 ?trait)
     (parent ?grandchild-1 ?parent-1)
     (parent ?parent-1 ?grandparent)
     (parent ?parent-2 ?grandparent)
     (parent ?grandchild-2 ?parent-2)
     (has ?grandchild-2 ?trait)
     =>
     (assert (inherited (status possible) (trait ?trait)))
     )
```

This is a fine rule when viewed in isolation. However, there are probably lots of rules in this application that examine conditions across siblings. All of these rules will share conditions similar to:

```
(parent ?parent-1 ?grandparent)
(parent ?parent-2 ?grandparent)
```

This amounts to a low-level encoding of the notion of a sibling. The following conditions amount to a low-level encoding of the notion of a cousin:

```
(parent ?grandchild-1 ?parent-1)
(parent ?parent-1 ?grandparent)
(parent ?parent-2 ?grandparent)
(parent ?grandchild-2 ?parent-2)
```

In an application of hundreds of rules that consider blood relationships in many different ways and combinations, having notions of "sibling" and "cousin" available as simple relationships rather than as more complex pattern matching operations not only makes the rules more perspicuous, it makes them more reliable and easier to maintain. As an example, the above rule could be recoded as:

```
(defrule cousins-may-inherit-trait
     (has ?x ?trait)
     (cousin ?x ?y)
     (has ?y ?trait)
     =>
     (assert (inherited (status possible) (trait ?trait)))
     )

(defrule cousin
     (parent ?x ?parent-1)
     (sibling ?parent-1 ?parent-2)
     (parent ?y ?parent-2)
     =>
     (assert (cousin ?x ?y))
     )

(defrule sibling
     (parent ?x ?parent)
     (parent ?y&~?x ?parent)
     =>
     (assert (sibling ?x ?y))
     )
```

Deduction using Forward Chaining

The cousin and sibling rules above make the high-level semantics of cousin and sibling explicit while in the original rule they were implicit. With these relations made explicit, coding of all rules that consider these relations can use a single pattern rather than its corresponding, implicit, constituent patterns.

Reducing the number of patterns per rule clearly improves the reliability of those rules. Also, maintaining rules that use the more abstract patterns is simplified since only the rules that maintain the relevant relation need to be modified. Furthermore, if a relation can be deduced by any of several methods (i.e., disjunction occurs) then the number of rules is reduced with resulting improvements in performance and reliability. Finally, using relations rather than unshared joins over several patterns can dramatically improve performance and reduce space requirements.

The problem with the above sibling and cousin rules is that they will assert every cousin and sibling relationship that exists given a set of parent relationships. The number of these deduced relationships can become very large, especially for the cousin relationship.

This is a fundamental problem. For most domains, there are at least an infinite number of irrelevant truths that can be deduced. The challenge in building a rational problem solving system is to actually deduce truths that are (or have a good chance of being) relevant to the problem at hand.

Deduction using Backward Chaining

Focusing deduction such that it furthers problem solving, rather than merely deducing irrelevant truths, is often done by generating subgoals during problem solving. Goals are generated as the conditions of rules are checked. These goals then trigger the checking of rules that might deduce facts that would further the matching of the rule which generated the goal.

To be more concrete, the cousin and sibling rules from above could be recoded as:

```
(defrule cousin
     (goal (cousin ?x ?y))
     (parent ?x ?p1)
     (parent ?y ?p2)
     (sibling ?p1 ?p2)
     =>
     (assert (cousin ?x ?y))
     )

(defrule sibling
     (goal (sibling ?x ?y))
     (parent ?x ?parent)
     (parent ?y ?parent)
     =>
     (assert (sibling ?x ?y))
     )
```

In these rules the goal condition is triggered when a goal to establish a sibling relationship is generated. The actions of these rules assert facts which satisfy the goals, thereby deducing only facts which might further the matching of the rules which led to the goals' generation.

We call the above rules data-driven backward chaining rules. Of course, for these rules to be driven some goal data is required. Either other rules or the inference engine architecture itself must assert these goals. In either case, goals must be generated as if by the following rules:

```
(defrule cousins-may-inherit-trait-goal-generation-1
     (has ?x ?trait)
=>
     (assert (goal (cousin ?x ?y)))
     )

(defrule cousin-goal-generation-1
     (goal (cousin ?x ?y))
     (parent ?x ?p1)
     (parent ?y&~?x ?p2)
     =>
     (assert (goal (sibling ?p1 ?p2)))
     )
```

Manual Goal Generation

The above goal generation rules, if they could be implemented, would correctly generate the goals required to implement the explicit cousin and sibling relations using the previously mentioned data-driven backward chaing rules. However, beyond the need for an adequate representation for goals, the manual coding of goal generation rules would remain problematic.

In general, for a rule of N conditions, N+1 rules will be needed to implement those rules such that they can support sound and complete reasoning. The original rule which matches in the standard, data-driven, forward chaining manner is, of course, required. An additional rule per condition is needed to assert the goals that will allow backward chained inference to deduce facts that will further the matching of the original rule.

Clearly, multiplying the number of rules required by one plus the average number of goal generating conditions per rule is unacceptable. Even if the effort is made, it is extremely error prone. Even automating the maintenance of goal generation rules would increase space and time requirements significantly, just to encode the actions and names of the rules and to activate the rules and interpret their actions.

Representing Goals

Even though manual coding of goal generation is impractical, CLIPS, OPS5, and many other production system languages are unable to implement the above rules for several even more fun-

damental reasons. The most obvious reason is that they provide no capability for distinguishing facts from goals. Moreover, these systems provide no means of representing unspecified values (for unbound variables) that occur in the conditions for which they might otherwise assert goals. For example, the following goal generation rule, in which the variable ?y is unbound, cannot even be simulated without explicit support for goals which include universally quantified values:

```
(defrule cousins-may-inherit-trait-goal-generation-1
     (has ?x ?trait)
=>
     (assert (goal (cousin ?x ?y)))
     )
```

Even supporting universally quantified values within goals is not enough to support backward chaining, however. If the variable ?y in the first condition of the following rule matches a literal value, CLIPS or OPS5 extended to support goal generation could function properly. If, however, ?y matches a universally quantified value, then neither CLIPS or OPS5 could join that unbound variable with any parent fact corresponding to the third condition, as would be logically required.

```
(defrule cousin-goal-generation-1
     (goal (cousin ?x ?y))
     (parent ?x ?p1)
     (parent ?y&~?x ?p2)
     =>
     (assert (goal (sibling ?p1 ?p2)))
     )
```

Clearly these systems are unable, not only to generate goals in the first place, but also to join those goals with facts.

Automatic Goal Generation

Eclipse is a syntactically similar language to NASA's CLIPS and Inference Corporation's ART, each of which include functionality similar to that of OPS5. Unlike CLIPS and OPS5, however, Eclipse supports a goal database and automatic generation of goals. In fact, the above pseudo-CLIPS rules which reference goals in their conditions and which assert facts are legal Eclipse rules. However, Eclipse does not require the addition of goal generation rules.

Eclipse automatically asserts goals precisely as would the goal generation rules described earlier. Goal generation in Eclipse adds no scheduling or interpretive overhead. There is no space overhead per rule or condition that generates a goal. Moreover, Eclipse goals can represent and include universally quantified (or unbound) values. Eclipse also supports the unification of universally quantified values with literals that occur in facts.

Goals as Data

Procedural backward chaining languages, such as Prolog, do not represent goals as data. In Prolog, goals are equivalent to procedure calls, if an invoked goal procedure fails the goal cannot be achieved. Moreover, in Prolog, if a goal fails at the time it is initially pursued, it will not be achieved unless a new and equivalent goal is reestablished. Thus, Prolog would fail to deduce a sibling relationship if a goal were established before a relevant parent relationship were known.

In Eclipse, goals are represented as propositions in a database. By representing goals as data and allowing patterns to distinguish facts from goals, Eclipse allows goals to drive pattern matching in combination with facts in the normal, data-driven manner. By representing goals in a database several goals can exist simultaneously and each goal can persist even if - at the time it is generated - it cannot be achieved. The simultaneous existence of multiple goals allows rules to do strategic reasoning and planning which is not possible if only one goal at time can be considered. The

persistence of goals allows goals to be achieved opportunistically. Unlike Prolog, if Eclipse generates a sibling goal which cannot be established, subsequent assertion of a relevant parent relationship would result in proper deduction.

Goal Maintenance

Eclipse also supports truth maintenance. In its standard application, truth maintenance allows a fact to be given *a prior* and/or supported by a disjunction of facts or sets of data which satisfy all or part of the conditions of one or more rules. For example, the following rule would make fact C logically dependent on the fact A and the absence of fact B[1]:

```
(defrule A-and-not-B-implies-C (A) (not (B)) => (infer (C)))
```

Subsequent retraction of A or assertion of B would lead to the retraction of the match for rule A-and-not-B-implies-C which would support the inference of C. If this support is removed, C (which we assume has not been asserted without logical dependency) would no longer be logically grounded and would therefore be automatically retracted.

In effect, Eclipse goal generation behaves as the goal generation rules described earlier using these logical dependencies. That is, if the following rules lead to the generation of a (goal (D)) after the assertion of A:

```
(defrule A-and-D-implies-C (A) (D) => (infer (C)))
(defrule D-is-implied-by-A (goal (D)) (A) => (infer (D)))
```

and A is subsequently retracted, the (goal (D)) will also be automatically retracted.

Using dependencies on goals results in a number of functional advantages. The most immediately obvious advantage is that goals only persist as long as they are relevant. This is another advantage over attempting to assert and maintain a crude representation of goals using OPS5 or CLIPS which do not support such dependencies. Secondly, if the facts inferred by goals are made to depend on the continued existence of those goals, then facts which were deduced but which subsequently become irrelevant to the ongoing process of problem solving are automatically retracted from the database. The automatic maintenance of deductions versus goals frequently improves performance and certainly reduces the need for manual coding of "cleanup" rules.

Goal Canonicalization

It is common for several rules to generate equivalent goals. For example, the following rules would both generate (goal (C ?1)) given facts A and B[2]:

```
(defrule A-and-C (A) (C ?x) =>)
(defrule B-and-C (B) (C ?x) =>)
```

Just as asserting equivalent facts twice results in one fact, so does the generation of two goals from these two rules result in only one goal. This one goal will have two sources of support. Removing one source of support will not result in the automatic retraction of the goal. Eclipse automatically maintains goals only as long as they are relevant by allowing a goal to persist only as long as at least one of the reasons for its generation persists.

An Example of Opportunistic Forward and Backward Chaining in Eclipse

In what follows we give an extensive trace and explanation of the simple genetic trait rules discussed earlier.

---

1 Under the closed-world assumption, absence of a fact is equivalent to a fact being false.

2 The ?1 denotes the first universally quantified value in a goal.

Given an empty database, first assert that John has freckles:

```
==> f-1 (has John freckles)
```

The above fact matches the first condition of the first rule. This causes a goal for the second condition of that rule, given John, to be generated as follows:

```
==> g-1 (cousin John ?1) by for 1 of cousin-may-inherit-trait f-1
```

This goal in turn matches the first condition of the cousin rule. However, since we do not know either of John's parents, the rule cannot apply... yet. Unlike Prolog and other procedural backward chaining languages, using a declarative representation for goals (rather a function-call semantics) allows the goal to persist in a database. In fact, representing goals as data allows an application to consider multiple goals that exist in the database rather than focus solely and ignorantly on only the most recently generated goal pushed onto a stack.

At this point then, a fact and a goal exist in the database. By asserting one of John's parents, problem solving can continue in the light of the established and outstanding goal.

```
==> f-2 (parent John George)
```

This parent fact matches the second pattern of the cousin rule and satisfies the mutual occurrence constraint on "John" given the outstanding goal. This leads to a match for the first two conditions:

```
==> thru 2 of cousin g-1,f-2
```

which results in a goal to establish facts which satisfy the third condition of the cousin rule given John's parent, George:

```
==> g-2 (sibling George ?1) by thru 2 of cousin g-1,f-2
```

Again, none of George's parents are known so problem solving cannot proceed. Note that at this time two facts about John having freckles and his father, George, and two goals exist in the database. By establishing one of George's parents, problem solving gets a little bit further:

```
==> f-3 (parent George Adam)
```

This fact, given the goal to determine George's siblings, matches the first two conditions of the sibling rule.

```
==> thru 2 of sibling g-2,f-3
```

Again, problem solving cannot proceed since no other offspring of Adam are known. If we also identify Sally as a child of Adam's:

```
==> f-4 (parent Sally Adam)
```

then the third and final condition of the sibling rule is satisfied for George, Sally, and their mutual parent, Adam.

```
==> thru 3 of sibling g-2,f-3,f-4
==> activation 0 sibling g-2,f-3,f-4
```

Executing this rule asserts the sibling relation between George and Sally:

```
==> f-5 (sibling George Sally)
```

which in turn satisfies the goal and matches condition 3 of the cousin rule.

```
==> thru 3 of cousin g-1,f-2,f-5
```

Once again, problem solving halts, however, since no children of Sally are known. By asserting a child for Sally,

```
==> f-6 (parent Mary Sally)
```

the fourth and last condition of the cousin rule is satisfied for John given his father George, George's sister Sally, and Sally's daughter, Mary.

```
==> thru 4 of cousin g-1,f-2,f-5,f-6
==> activation 0 cousin g-1,f-2,f-5,f-6
```

Executing this rule asserts the cousin relationship between John and Mary:

```
==> f-7 (cousin John Mary)
```

which matches the second condition of the first rule and joins with the fact that John has freckles to match the first two conditions of that rule:

```
==> thru 2 of cousins-may-inherit-trait
```

From which point, asserting that Mary has freckles activates the rule, thereby leading to the assertion the freckles may be inherited.

Conclusion

By supporting automatic generation and maintenance of goals across multiple rules with many potential causes for each goal to exist and by allowing each goal to represent literal and universally quantified values which constrain the facts that could satisfy any given goal to a subset of all possible facts which has a higher probability of furthering problem solving, Eclipse allows data-driven rule technology, which is the only practical technology for rule-based programming and the implementation of expert systems, to perform logical reasoning. Moreover, the software engineering, maintenance, extensibility, and performance characteristics of rule-based programs afforded by reduced coding and interpretation of redundant, potentially disjunctive, combinations of patterns within many rules is also considerable. Finally, the resulting programs are simply much easier to understand since they make explicit the high-level knowledge which would otherwise be encoded implicitly using more complicated, less perspicuous, less efficient, and less maintainable combinations of patterns across many rules.