MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Working Paper No. 296 May 1987

# The Interaction Between Truth Maintenance, Equality, and Pattern-Directed Invocation: Issues of Completeness and Efficiency

by

Yishai A. Feldman and Charles Rich

**Abstract**

We have implemented a reasoning system, called BREAD, which includes truth maintenance, equality, and pattern-directed invocation. This paper reports on the solution of two technical problems arising out of the interaction between these mechanisms. The first result is an algorithm which ensures the completeness of pattern-directed invocation with respect to equality. The second result is an algorithm which reduces a class of redundant proofs.

Copyright © Massachusetts Institute of Technology, 1987

A.I. Laboratory Working Papers are for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be considered papers to which reference can be made in the literature.

# 1 Introduction

We have implemented a reasoning system, called BREAD,[1] which includes truth maintenance, equality, and pattern-directed invocation. The interaction between these mechanisms introduces two major technical problems. The first problem is the incompleteness of pattern-directed invocation with respect to equality. The second problem is the creation of redundant proof paths due to congruence closure. This paper describes the solution to these two problems, as well as some related technical issues.

The previous system which combined these three mechanisms was McAllester's Reasoning Utility Package (RUP) [4]. BREAD extends and improves RUP in a number of ways.

The BREAD system comprises the bottom three layers of CAKE [2,5], which is the knowledge representation and reasoning system supporting the Programmer's Apprentice project [6,7]. We have re-packaged these three layers as a separate system in recognition of the fact that these facilities are of general use.

Truth maintenance supports explanation, incremental retraction of beliefs and decisions, and the use of dependency-directed backtracking to search spaces of alternatives efficiently. In the programming context, these facilities are necessary to support an evolutionary approach to program design.

Reasoning with equalities is a fundamental and ubiquitous feature of formal reasoning systems. Systems need to understand that if two individuals are identical, then all their properties are the same, and vice versa. In the Programmer's Apprentice project, equality is used to model data flow between procedures, data abstraction, and side effects.

Pattern-directed invocation is (for better or for worse) the most commonly used method for incrementally extending the power of reasoning systems. In CAKE this facility is used, among other things, to add some quantificational reasoning to the underlying propositional framework.

### Pattern-Directed Invocation

In pattern-directed invocation, a procedure is associated with a pattern (the combination is typically called a *demon*). The argument to the procedure is either a term in the reasoning data base matching the pattern, or else the set of bindings to the pattern variables resulting from matching such a term (in BREAD the former convention is used). The reasoning system maintains the following simple invariant:

[1] for Basic REAsoning Device.

*Every demon is invoked on every matching term in the data base.*

This invariant needs to be maintained across two types of events. First, when a new term is added to the reasoning data base, it must be matched against the patterns of all demons. Second, when a new demon is installed, its pattern must be matched against all terms in the data base.

A typical use of pattern-directed invocation is to implement reasoning involving quantified knowledge. For example, suppose that for some function, $f$, we had the axiom

$$\forall x \; f(0, x) \geq 0.$$

One way of using this knowledge would be to install a demon with the pattern $f(0, ?x)$ (variables in patterns are denoted here by the prefix "?"). When this demon is invoked, for example, on the term $f(0, 2)$ it would assert

$$f(0, 2) \geq 0.$$

### Equality Reasoning

Reasoning about equality involves two main tasks. The first is to maintain the equivalence classes of the equality operator. This is the task responsible, for example, for concluding $a = c$ from $a = b$ and $b = c$.

The second task is congruence closure, i.e., propagating equalities by substitution. This is the task responsible, for example, for concluding $f(a) = f(b)$ from $a = b$.

Furthermore, in equality reasoning with truth maintenance, both of these kinds of deductions are retractable. The conclusion $a = c$ depends on $a = b$ and $b = c$. If either of these supporting facts is retracted, $a = c$ will automatically be retracted (unless it has alternate support). Similarly, $f(a) = f(b)$ is supported by $a = b$.

## 2 The Incompleteness Problem

To illustrate the interaction between pattern-directed invocation and equality reasoning, suppose that, in the presence of the demon described above, we assert $t = f(0, 3)$. The creation of the term $f(0, 3)$ will invoke the demon, which will assert

$$f(0, 3) \geq 0.$$

The system can then deduce that $t \geq 0$ by substitution of equals. Both BREAD and RUP succeed in making this deduction, including installing the appropriate dependencies.

Unfortunately, the usual method of pattern-directed invocation does not achieve complete use of the knowledge embodied in demons. For example, suppose that the term $f(a,b)$ is in the data base and $a = 0$ is true. It follows logically from the knowledge in the demon that

$$f(a,b) \geq 0.$$

This fact will not, however, be known, because no term in the data base matches the pattern of the demon. (RUP suffers from this problem.)

## The Solution

A simple solution to this problem is to have the reasoning system create all possible variants of terms in the data base (i.e., to close the data base under substitution of equals). A *variant* of the (non-atomic) term, $t$, is any term derived from $t$ by substituting equals for one or more subterms of $t$. A variant of $t$ is therefore equal to $t$. For example, given $a = 0$, $f(0,b)$ is a variant of $f(a,b)$ obtained by substituting 0 for $a$ in the second subterm (in BREAD the subterms include the operator).

If the term $f(0,b)$ *is* in the data base, then it would trigger the demon, which would assert

$$f(0,b) \geq 0,$$

which by substitution of equals would imply $f(a,b) \geq 0$, as desired.

The approach of creating all variants is not only extremely expensive at best, but is potentially infinitely explosive. For example, consider what happens if you attempt to create all possible variants in a data base with the term $f(x)$ and the equality $x = f(x)$.

In order to improve upon the brute force solution, we must assume some restriction on the operation of demons. Basically, we assume that demons make visible in their patterns all of the information upon which their behavior logically depends. (This restriction will be defined more formally later, in the notion of a *transparent* demon.) Given this restriction, when a demon doesn't match a term directly, the reasoning system can determine from the pattern whether the knowledge in the demon could be applied to the term by creating one of its variants.

## Closest Matching Variant

To guarantee complete use of the knowledge in (transparent) demons in the presence of equalities, BREAD maintains the following reformulated invariant:

*Every demon is invoked on the* closest matching variant *of every term in the data base.*

For a term and a pattern (and a given set of equalities), the *closest matching variant* is defined to be either the term itself (if it matches), or else a variant of the term which matches, such that all variants closer to the term do not match. The distance between terms is defined to be the number of subterms in which they differ.[2] Note that the closest matching variant of a term and a pattern may not exist, and if it does exist, may not be unique.

Satisfying this invariant sometimes requires the creation of new terms. Consider the case when a term in the data base does not match a demon's pattern, but a variant of the term does. This is the case with $f(a,b)$, $a = 0$, and the example demon with pattern $f(0,?x)$. One might be tempted in this situation to say that, since a variant of $f(a,b)$ matches the pattern, we should invoke the demon on $f(a,b)$ itself, and not bother actually creating the variant term $f(0,b)$. This approach will indeed give the correct immediate conclusion. It does not, however, install the correct dependencies. In particular, the conclusion should depend on $a = 0$, but it will not.

The algorithm BREAD uses to compute the closest matching variant for single-level patterns (i.e., patterns in which all the variables appear at top level) is given in Figure 1. Multi-level patterns are handled in BREAD by cascading single-level demons for the sub-patterns, as described below.

For example, matching the single-level pattern $f(0,?x)$ against the term $g(a,b)$ in the presence of the equalities $f = g$, $a = 0$, and $b = c$, returns the variant $f(0,b)$. It would not be correct to return the variant $f(0,c)$, which also matches the pattern, because this variant differs in three subterms from $g(a,b)$, whereas $f(0,b)$ differs in only two.

Matching the pattern $f(?x,?x)$ against the term $f(a,b)$ in the presence of the equality $a = b$, returns the variant $f(a,a)$. It would be equally correct to return the variant $f(b,b)$, since it also matches the pattern and is just as close. Which variant the algorithm in Figure 1 picks depends on the order of binding of the variables.

## Maintaining the Demon Invariant

There are three types of events across which the new demon invariant needs to be maintained (closest matching variant is the type of matching used in all cases below):

[2]The Gray code distance.

```
(Defun Match (Pattern Subterms)
  (Do ((P Pattern (Cdr P))
       (S Subterms (Cdr S))
       (Variant)
       (Bindings))
      ((Null S)                    ;no more subterms:
       (If P                       ;is there more pattern left?
           Nil                     ;if so, match fails.
           (Reverse Variant)))     ;otherwise, return closest variant.
    (Let ((Pn (Car P))
          (Sn (Car S)))
      (Cond ((Null Pn) (Return Nil))  ;no more pattern, match fails.
            ((Variable? Pn)
             (Let ((Value (Cdr (Assoc Pn Bindings))))
               (Cond (Value
                      (If (Equal-Term Sn Value)  ;variable has value:
                          (Push Value Variant)   ;if matches, include in variant.
                          (Return Nil)))         ;otherwise, match fails.
                     (T
                      (Push (Cons Pn Sn) Bindings)  ;variable gets new value:
                      (Push Sn Variant)))))         ;include value in variant.
            ((Equal-Term Pn Sn)          ;pattern element is constant:
             (Push Pn Variant))          ;if matches, include in variant.
            (T (Return Nil))))))         ;otherwise, match fails.
```

**Figure 1.** A Common Lisp algorithm which computes the closest matching variant for single-level patterns. `Pattern` is a list of terms or variables. `Subterms` is a list of terms. `Equal-Term` tests whether two terms are in the same equality class.

- A new term is added to the data base.
- A new demon is installed.
- A new equality becomes true.

When a new term is added to the reasoning data base, it must be matched against the patterns of all demons, as before. If the match returns a closest matching variant that is not identical[3] to the term being matched, then the reasoning system will add the variant to the data base and invoke the demon on it.

When a new demon is installed, its pattern is similarly matched against all terms in the data base, and the closest matching variant of each term added to the data base if it is not already there.

When a new equality becomes true, two previously separate equality classes are joined. This in turn makes new variants possible. Any term which has a subterm in either of the two equality classes must be matched against the patterns of all demons. Matches which failed before may now succeed in finding a matching variant. In BREAD, this re-matching process is made efficient by using indexing to find all terms and all patterns with a given subterm.

Note that nothing needs to be done to maintain the invariant when an equality is retracted (except, in BREAD, to update some indexing structures.)

### Multi-Level Patterns

Multi-level patterns are handled in BREAD by cascading demons similarly to the way that AMORD [1] handled conjunctive patterns.

When a demon is installed with a multi-level pattern, e.g., $f(f(0,?x),?y))$, BREAD automatically generates an intermediate demon whose pattern is the first sub-pattern of the original demon (in this example, $f(0,?x)$). When invoked on the term $f(0,2)$, this demon installs another intermediate demon (in this case, with an empty procedure) with the pattern $f(f(0,2),?y)$. This pattern, by virtue of the demon invariant, effectively forces substitution of $f(0,2)$ in the first argument of any term in which $f$ is the operator. For example, suppose the term $f(a,b)$ is in the data base and $a = f(c,d)$, $c = 0$, and $d = 2$. The demon invariant will force the creation of the variant $f(f(0,2),b)$, which will match the original demon.

A more complex example of the operation of multi-level patterns, illustrating the propagation of variable bindings through intermediate demons, is shown in Figure 2.

[3]When the distinction is important, we will use the word *identical* to mean two terms that are "spelled the same", and the word *equal* to mean two terms that are in the same equality class.

```
Initial Data base: g(r,s)
                   f(a,g(b,c))
                   r = f(a,b)
                   s = f(d,c)
                   a = d

Desired Conclusion: g(r,s) = f(a,g(b,c))

Distributivity Demon: g(f(?x,?y),f(?x,?z))

        => demon: f(?x,?y)
           ...
        => matches: f(a,b)

        => demon: g(f(a,b),f(a,?z))

        => demon: f(a,?z)
           ...
        => forces variant: f(a,c) of f(d,c)     *******

        => demon: g(f(a,b),f(a,c))

        => forces variant: g(f(a,b),f(a,c)) of g(r,s)

        => asserts: f(a,g(b,c)) = g(f(a,b),f(a,c))
```

**Figure 2.** A trace of of the cascading demons created by a multi-level pattern. The starting demon represents the fact that the function $f$ distributes over $g$. The key step (marked with asterisks) is to force the creation of the variant $f(a, c)$. This trace shows only the parts of the process relevant to deducing the particular desired conclusion. Note that once the distributivity demon asserts the equality at the end of the trace, the desired conclusion follows from transitivity of equality.


**Figure 3.** An example of imaging. Suppose that $a = b$ and $c = d$ are true, and that the corresponding equality trees are constructed as shown at the left. The terms $f(a, d)$ and $f(b, c)$ will both have the image $f(b, d)$, and will therefore both be made equal to $f(b, d)$.

Although this cascading process is computationally expensive, it is required in order to guarantee completeness.

# 3 The Redundant Proof Problem

The algorithms and data structures that BREAD uses to maintain equality classes and compute congruence closure are basically the same as in RUP. A negative property of these algorithms is that they require the creation of additional variants, beyond those required for the completeness of pattern-directed invocation. These extra variants reduce the efficiency of the reasoning system by introducing a class of redundant proofs. RUP eliminates these redundant proofs at the cost of completeness. This section describes an approach we have implemented that significantly reduces these redundant proofs while preserving completeness.

### Image Terms

An equality class in BREAD is stored as a $n$-ary tree, in which each term points to its parent in the direction of the root. The *image* of a (non-atomic) term is the variant obtained by replacing each subterm by its parent (the parent of a root is considered to be the root itself). In order to support retractable equality reasoning, BREAD automatically creates the image of every term in the data base, and makes the term equal to its image. The equality between a term and its image depends in the truth maintenance system upon the equalities between the subterms and parents. This process guarantees that if two terms can be concluded equal by substitution, then they will be in the same equality class. Figure 3 shows an example of the imaging process.

Closing the data base under images is a reasonable algorithmic tradeoff from

the standpoint of equality reasoning. However, it causes unnecessary demon invocations, which introduce redundant proof paths. Consider, for example, a demon which implements the antecedent use of the axiom

$$\forall x\ P(x) \rightarrow Q(x).$$

The pattern of this demon will be $P(?x)$ and the procedure, when invoked on the term $P(a)$, will assert the implication $P(a) \rightarrow Q(a)$.[4]

Now suppose that the equality $a = b$ becomes true. The imaging process will create the new terms $P(b)$ and $Q(b)$.[5] $P(b)$ also matches the demon's pattern, which leads it to assert the implication $P(b) \rightarrow Q(b)$. But this implication is redundant. If $P(b)$ becomes true, the system can already deduce $Q(b)$ by equality reasoning from $P(a)$, $P(a) \rightarrow Q(a)$, and $a = b$, so there is no point in invoking the demon again on $P(b)$.

Part of the problem is that, depending on what demons are in the system, some images are redundant, and some are not. For example, suppose there is a demon in the system with some specific knowledge about $b$, such as

$$\forall f\ f(b) = \textit{undefined}.$$

This demon *should* be invoked on $P(b)$, even though it is an image term.

Whether invoking a demon on an image term is redundant also depends on the state of the equalities in the system. Suppose in this example that $a = b$ is retracted. In this situation, the implication $P(b) \rightarrow Q(b)$ is needed to prove $Q(b)$ from $P(b)$.

## The Solution

The solution in RUP to the problem of redundant proofs due to images is not to invoke demons on image terms at all, unless specifically requested. Unfortunately, this gives up completeness, as illustrated above.

Our solution is BREAD is to keep track of how each term in the data base is created and used. (This entails maintaining only a few flags on each term.) Given this information and the assumption that demons are transparent (see next section),

[4]Note that for boolean-valued terms, i.e., propositions, existence in the data base is not the same as being true or believed. Propositions have a separate "truth value" of true, false, or unknown. The propositional reasoning mechanisms of BREAD (which are outside the topic of this paper) will take care of making $Q(a)$ true whenever $P(a)$ is true.

[5]This assumes that the equality tree is constructed such that $b$ is the parent of $a$. Sometimes the tree will be built in the other order. In general, however, adding new equalities does cause the creation of additional image terms.

the system can reduce the redundant invocation of demons, while still maintaining the demon invariant and its guarantee of completeness.

Consider what happens when a new equality becomes true. First the congruence-closure algorithm closes the data base under images. Terms created during the imaging process are marked as being images only, and are not immediately matched against demons. In the examples above, this corresponds to the situation with $P(a)$ in the data base, $a = b$, and only the demon with pattern $P(?x)$. $P(b)$ is not matched against demons.

The event of a new equality becoming true also triggers an action to maintain the demon invariant: Any term which has a subterm in either of the two joined equality classes must be matched against the patterns of all demons. It is possible for this action to result in demons being invoked on one of the newly created image terms (i.e., if it is the closest matching variant of some non-image term.) In the examples above, this corresponds to the situation with the demon with pattern $?f(b)$. The demon invariant forces invocation of demons on $P(b)$.

When an equality supporting an image is retracted, the system checks whether the term was used for any other purpose (e.g., whether it was typed in by the user, created by another demon, etc.). If the term is being used, and hasn't been matched against demons yet, then it is matched now. In the examples above, this corresponds to the situation in which $a = b$ is retracted and $P(b)$ has been asserted. Demons are invoked on $P(b)$, so that $Q(b)$ can be deduced.

If an equality is retracted supporting an image that is used only as an image, then nothing is done. If an attempt is made to use the term for some other purpose later, however, the system matches it against demons at that time, just as if it were a new term. The justification for doing nothing immediately in this situation is as follows.

Consider the term $P(b)$ in the simplest case above, with $P(a)$, $a = b$, and no demons. If the *only* reason for the existence of $P(b)$ was to be the image of $P(a)$ in the congruence-closure algorithm, now that $a = b$ is retracted, there is no reason to keep $P(b)$ around at all. Said another way, if the system's reasoning was complete without $P(b)$ before $a = b$ was asserted, then it should be complete without $P(b)$ now that $a = b$ is retracted.

The algorithm above has, in our experience with BREAD, dramatically reduced the number of redundant proof paths involving image terms. Some redundant paths are still introduced, however, due to the fact that either all or none of the demons must be invoked on a given term. The system might be fine tuned to have per-demon control for each term, which would require additional indexing.

### Transparent Demons

What BREAD is trying to do in all the optimizations described above is avoid invoking the same demon on more than one element of any set of variants. The rationale behind this approach is an assumption about what kinds of things can go on inside a demon. This section formalizes this assumption via the notion of a *transparent* demon.

Let us begin with a counter-example. Consider the following demon with the pattern $f(?x, ?y)$:

```
(Lambda (Subterms)
  (Let ((Arg1 (Cadr Subterms)))
    (Cond ((Eql Arg1 0) ... )
          ((Eql Arg1 1) ... ))))
```

The problem with this demon is that it is hiding some of its pattern matching inside of the procedure, where the demon invariant can't get to it. For example, if the term $f(a, b)$ is in the data base and $a = 0$ is true, the system will not guarantee that the variant $f(0, a)$ will be created to invoke this demon.

This points out that the demon invariant guarantees completeness with respect to the *patterns* of demons, not necessarily with respect to the knowledge inside them. This makes sense, since after all, the only declarative information which the system has about a demon is its pattern.

The way to fix the problem with this demon is to rewrite it in the form of two separate demons, one with pattern $f(0, ?y)$ and one with pattern $f(1, ?y)$.

The general restriction which defines transparent demons can be stated logically as follows:

> A demon is *transparent* if and only if, when it is invoked on two variants of the same term, its result in the reasoning data base is logically equivalent in both cases (with respect to the current set of equalities).

A sufficient syntactic condition to guarantee transparency is for the demon not to test or branch (directly or indirectly) on the value of the bindings of any of its variables, or any internal or external state. BREAD does not automatically analyze demon procedures to check this condition.

# 4 Conclusion

We have described the solution to a number of technical problems concerning the completeness and efficiency of the interaction of truth maintenance, equality and

pattern-directed invocation in reasoning systems. We have implemented these solutions in a working system called BREAD.

BREAD has proved to be a useful and stable foundation upon which we have built a number of other reasoning facilities, including a frame system [3] and system for reasoning about programs [5]. BREAD is implemented in Common Lisp and currently runs on Symbolics machines. The system is available from the authors for experimental research use. We would like to reiterate our debt to McAllester's RUP for many of the basic ideas in BREAD.

# References


[1] de Kleer, J., J. Doyle, C. Rich, G. L. Steele, and G. J. Sussman, "AMORD, A Deductive Procedure System", MIT Artificial Intelligence Lab. Memo 435, August, 1977.

[2] Feldman, Y. A., and C. Rich, "Reasoning with Simplifying Assumptions: A Methodology and Example", *Proc. Nat'l Conf. Artificial Intelligence (AAAI-86)*, August, 1986.

[3] Feldman, Y. A., and C. Rich, BREAD *and* FRAPPE*: The Gourmet's Guide to Automated Deduction*, (User's Manual), in preparation.

[4] McAllester, D. A., "Reasoning Utility Package User's Manual", MIT Artificial Intelligence Lab. Memo 667, April, 1982.

[5] Rich, C., "The Layered Architecture of a System for Reasoning about Programs", *Proc. of the 9th Int. Joint Conf. on Artificial Intelligence*, Los Angeles, CA, August, 1985.

[6] Rich, C., and H. Shrobe, "Initial Report on a Lisp Programmer's Apprentice", *IEEE Trans. on Software Eng.*, Vol. 4, No. 6, November, 1978.

[7] Waters, R. C., "The Programmer's Apprentice: A Session with KBEmacs", *IEEE Trans. on Software Eng.*, Vol. 11, No. 11, November, 1985.