Internal Note #104
Computational Logic, Inc.
December 9, 1988 ** Draft **

An Experimental Implementation of Equivalence Reasoning in the Boyer-Moore Theorem Prover

Bishop Brock*

Contents

*This work was supported in part at Computational Logic by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

1 Introduction

This note describes an experimental extension of the Boyer-Moore theorem prover that includes reasoning about general equivalence relations. The continuing goal of this work is to provide as many of the implicit reasoning methods and heuristics for equivalence relations as are currently provided for equality. The targeted features extend from simple reasoning about reflexivity and symmetry up to and including the use of equality hypotheses and directed rewriting. To avoid confusion, we will always refer to the current, standard version of the Boyer-Moore theorem prover as NQTHM, and we will always refer to our modification as the *equivalence prover*.

Our modifications do not extend the Boyer-Moore *logic*, but instead extend the *heuristics* of NQTHM. The concept is very simple, and rests on the notions of equivalence relations and congruence relations. An equivalence relation, of course, is any reflexive, symmetric, and transitive relation. NQTHM currently "recognizes" two equivalence relations, `EQUAL` and `IFF`, although only `EQUAL` benefits from the full range of equality heuristics. In the equivalence prover, an equivalence relation is any boolean function of two or more arguments which is provably reflexive, symmetric, and transitive. For simplicity, we impose the syntactic restriction that the reflexive, symmetric, and transitive arguments always appear as the first two arguments of the function. The other arguments represent the moduli of equivalence. This function, for example:

Definition.

```
(MOD= A B K)
    =
(EQUAL (REMAINDER A K) (REMAINDER B K))
```

is the equivalence relation $a \equiv b \pmod{k}$ for the natural numbers.

We also employ a very general notion of congruence relations. Assume that `r1` and `r2` are (possibly identical) equivalence relations, with moduli `m1`,...,`mL` and `p1`,...,`pN`, respectively. Further, let `f` be an arbitrary function in the Boyer-Moore logic. Then `r1` is a congruence relation for the indicated argument of `f` if

```
(IMPLIES
 (r1 x y m1 ... mL)
 (r2 (f t1 ... x ... tM) (f t1 ... y ... tM) p1 ... pN))
```

is a theorem. This lemma, for example, establishes `MOD=` (mod `K`) as a congruence relation for the first argument of `TIMES`:

Congruence Frame.

```
(IMPLIES
 (MOD= A B K)
 (MOD= (TIMES A C) (TIMES B C) K)).
```

In the equivalence prover, lemmas of this form are called *congruence frames*.

The most important types of congruence frames are those for which the equivalence symbol in the conclusion is `EQUAL`. This congruence frame, for example, is the way that the equivalence prover notes the transitivity of `MOD=` :

Congruence Frame.

```
(IMPLIES
 (MOD= A B K)
 (EQUAL (MOD= A C K) (MOD= B C K))).
```

This is an important lemma because it shows that if the first argument of `MOD=` is replaced by any other `MOD=` (mod `K`) term, then the value of the resulting term is `EQUAL` to the original. In other words, `MOD=` (mod `K`) preserves equality for the first argument of `MOD=`. Congruence frames can also be linked together. Since `MOD=` (mod `K`) preserves congruence (mod `K`) for the arguments of `TIMES`, we may infer that `MOD=` (mod `K`) preserves equality for arguments of `TIMES` terms occurring at locations where `MOD=` (mod `K`) preserves equality. That is, the two congruence frames above show that we may replace `A` in

```
(MOD= (TIMES A B) C K)
```

by any `MOD=` (mod `K`) term without altering the value of the entire expression.

The preceding introductory remarks were purposefully brief. The philosophy and operation of the equivalence prover are completely elaborated in the coming pages. We begin with a detailed example using a modulus free equivalence relation; a proof that a simple *quicksort* routine for lists returns a permutation of the original list. The next section documents the two new lemma types available in the equivalence prover, followed by a detailed comparison between NQTHM and the equivalence prover. We conclude with a list of deficiencies in the equivalence prover. A listing of the events for the example appears as an Appendix.

2 Example: A Simple Sorting Algorithm

We show the utility of equivalence reasoning with an example, a partial proof of correctness of a simple sorting routine. This example demonstrates at an intuitive level how the equivalence prover reasons about equivalence relations, and also why the equivalence prover provides an advantage over NQTHM. The majority of the technical details are left for later sections.

2.1 PERMUTATION

A sorting routine for lists has two specifications: The output list should be a permutation of the input, and the output should be ordered in some way. Here we are only concerned with the former property. This definition of **PERMUTATION** is taken from a standard library of Boyer-Moore events created by Bill Bevier [Bev88][1]:

Definition.

```
(MEMBER X L)
  =
(IF (NLISTP L)
    F
  (IF (EQUAL X (CAR L))
      T
    (MEMBER X (CDR L))))
```

Definition.

```
(DELETE X L)
  =
(IF (LISTP L)
    (IF (EQUAL X (CAR L))
        (CDR L)
      (CONS (CAR L) (DELETE X (CDR L))))
  L)
```

Definition.

```
(PERMUTATION A B)
  =
(IF (LISTP A)
    (AND (MEMBER (CAR A) B)
         (PERMUTATION (CDR A)
                      (DELETE (CAR A) B)))
  (NOT (LISTP B)))
```

The function **PERMUTATION** is also an equivalence relation, as this lemma shows:

[1] The function **MEMBER** is a built–in function. We have included its definition for completeness.

Lemma. PERMUTATION-EQUIVALENCE

```
(AND (PERMUTATION A A)
     (EQUAL (PERMUTATION A B) (PERMUTATION B A))
     (IMPLIES
      (AND (PERMUTATION A B)
           (PERMUTATION B C))
      (PERMUTATION A C))).
```

2.2 QSORT

The sorting routine we have chosen is a recursive quicksort for lists of numbers. The sorting program is implemented by three functions. The functions `LESSP-LIST` and `GEQ-LIST` partition the input list, and `QSORT` recursively sorts and assembles the result:

Definition.

```
(LESSP-LIST VAL LIST)
  =
(IF (NLISTP LIST)
    NIL
  (IF (LESSP (CAR LIST) VAL)
      (CONS (CAR LIST) (LESSP-LIST VAL (CDR LIST)))
    (LESSP-LIST VAL (CDR LIST))))
```

Definition.

```
(GEQ-LIST VAL LIST)
  =
(IF (NLISTP LIST)
    NIL
  (IF (LEQ VAL (CAR LIST))
      (CONS (CAR LIST) (GEQ-LIST VAL (CDR LIST)))
    (GEQ-LIST VAL (CDR LIST))))
```

Definition.

```
(QSORT LIST)
  =
(IF (NLISTP LIST)
    NIL
  (APPEND (QSORT (LESSP-LIST (CAR LIST) (CDR LIST)))
          (CONS (CAR LIST)
                (QSORT (GEQ-LIST (CAR LIST) (CDR LIST))))))
```

2.3 Example Proof

We would like to prove the conjecture:

```
(PERMUTATION (QSORT LIST) LIST).
```

The proof is by induction, using the induction scheme suggested by the definition of `QSORT`. The base case,

```
(IMPLIES
 (NLISTP LIST)
 (PERMUTATION (QSORT LIST) LIST),
```

follows easily from the definitions of `PERMUTATION` and `QSORT`. In the inductive case we are required to show

```
(IMPLIES
 (AND (LISTP LIST)
      (PERMUTATION (QSORT (LESSP-LIST (CAR LIST) (CDR LIST)))
                   (LESSP-LIST (CAR LIST) (CDR LIST)))
      (PERMUTATION (QSORT (GEQ-LIST (CAR LIST) (CDR LIST)))
                   (GEQ-LIST (CAR LIST) (CDR LIST))))
 (PERMUTATION (APPEND (QSORT (LESSP-LIST (CAR LIST) (CDR LIST)))
                      (CONS (CAR LIST)
                            (QSORT (GEQ-LIST (CAR LIST) (CDR LIST)))))
             LIST)).
```

Here is the key to the proof: If the inductive hypotheses were `EQUAL` terms instead of `PERMUTATION` terms, we could reduce the above to

```
(IMPLIES
 (LISTP LIST)
 (PERMUTATION (APPEND (LESSP-LIST (CAR LIST) (CDR LIST))
                      (CONS (CAR LIST)
                            (GEQ-LIST (CAR LIST) (CDR LIST))))
             LIST)),
```

which is provable from the definitions of `PERMUTATION`, `LESSP-LIST`, and `GEQ-LIST`. This is exactly what the equivalence prover will do. With the aid of selected congruence frames, the equivalence prover will be able to establish that `PERMUTATION` preserves equality at the proper points in the term, and to use the inductive hypotheses as if they were equalities.

The Necessary Congruence Frames

When the equivalence prover accepted `PERMUTATION` as an equivalence relation, two facts were noted. These facts follow from the symmetry and transitivity of `PERMUTATION`:

Congruence Frame. `PERMUTATION-LEFT-FRAME`

```
(IMPLIES
 (PERMUTATION A B)
 (EQUAL (PERMUTATION A C) (PERMUTATION B C)))
```

Congruence Frame. `PERMUTATION-RIGHT-FRAME`

```
(IMPLIES
 (PERMUTATION A B)
 (EQUAL (PERMUTATION C A) (PERMUTATION C B))).
```

These lemmas show that `PERMUTATION` is a congruence relation for both arguments of `PERMUTATION`. Before the equivalence prover can complete the desired reduction, however, it will also need these three congruence frames for `CONS` and `APPEND`:

Congruence Frame. `PERMUTATION-CONS-RIGHT-FRAME`

```
(IMPLIES
 (PERMUTATION Y Z)
 (PERMUTATION (CONS X Y) (CONS X Z)))
```

Congruence Frame. `PERMUTATION-APPEND-LEFT-FRAME`

```
(IMPLIES
 (PERMUTATION X Y)
 (PERMUTATION (APPEND X Z) (APPEND Y Z)))
```

Congruence Frame. `PERMUTATION-APPEND-RIGHT-FRAME`

```
(IMPLIES
 (PERMUTATION X Y)
 (PERMUTATION (APPEND Z X) (APPEND Z Y))).
```

Using Congruence Frames

We can now show how the equivalence prover determines that `PERMUTATION` preserves equality at the desired points in the term. Returning to the term under scrutiny,

```
(IMPLIES
 (AND (LISTP LIST)
      (PERMUTATION (QSORT (LESSP-LIST (CAR LIST) (CDR LIST)))
                   (LESSP-LIST (CAR LIST) (CDR LIST)))
      (PERMUTATION (QSORT (GEQ-LIST (CAR LIST) (CDR LIST)))
```

```
                  (GEQ-LIST (CAR LIST) (CDR LIST)))
 (PERMUTATION (APPEND (QSORT (LESSP-LIST (CAR LIST) (CDR LIST)))
                     (CONS (CAR LIST)
                           (QSORT (GEQ-LIST (CAR LIST) (CDR LIST)))))
              LIST)),
```

we follow the rewriter as it recursively rewrites the conclusion. Since the truth value of this clause[2] only depends on the propositional values of its arguments, the rewriter may replace the conclusion not only by a term `EQUAL` to it, but by any term propositionally equal (`IFF`) to it. To generalize, at any point during rewriting there is a set $\mathcal{A}$ of function symbols of equivalence relations such that if `r` is in $\mathcal{A}$, the current term is `t1`, and `(r t1 t2)` is true, then replacing `t1` with `t2` maintains the truth value of the entire clause. Thus, prior to rewriting the above conclusion, $\mathcal{A} = \{\texttt{EQUAL}, \texttt{IFF}\}$.

The equivalence prover maintains the set $\mathcal{A}$ as each clause is recursively rewritten. When rewriting the first argument of `PERMUTATION`,

```
(APPEND (QSORT (LESSP-LIST (CAR LIST) (CDR LIST)))
        (CONS (CAR LIST)
              (QSORT (GEQ-LIST (CAR LIST) (CDR LIST))))).
```

propositional equivalence no longer preserves equality, and the symbol `IFF` is removed from $\mathcal{A}$. Note, however, the information implicit in the congruence frame `PERMUTATION-LEFT-FRAME`. This lemma shows that if the left-hand side of a `PERMUTATION` term is replaced by a `PERMUTATION`, then the value of the new term is `EQUAL` to the value of the original. Thus, entering the first argument of `PERMUTATION`, $\mathcal{A} = \{\texttt{EQUAL}, \texttt{PERMUTATION}\}$.[3]

Now the rewriter begins rewriting the first argument of `APPEND`,

```
(QSORT (LESSP-LIST (CAR LIST) (CDR LIST))).
```

In this case, the congruence frame `PERMUTATION-APPEND-LEFT-FRAME` shows that replacing the first argument of `APPEND` by a `PERMUTATION` produces a term which is a `PERMUTATION` of the original. Since this occurrence of `APPEND` appears in an environment where `PERMUTATION` preserves equality, `PERMUTATION` remains in $\mathcal{A}$, and again $\mathcal{A} = \{\texttt{EQUAL}, \texttt{PERMUTATION}\}$.

The rewriter will now attempt to recursively rewrite

```
(QSORT (LESSP-LIST (CAR LIST) (CDR LIST))),
```

and find that no rules apply to its subterms. Since `PERMUTATION` preserves equality at the current point in the term, however, the rewriter will use the inductive hypothesis

[2] Recall that `(IMPLIES (AND p (NOT q)) r)` is the printed representation of the clause `(OR (NOT p) q r)`

[3] Technically, `PERMUTATION` *replaces* `EQUAL` in $\mathcal{A}$ as we move into the argument during rewriting. But since `EQUAL` always preserves equality, `EQUAL` is added back into $\mathcal{A}$.

```
(PERMUTATION (QSORT (LESSP-LIST (CAR LIST) (CDR LIST)))
             (LESSP-LIST (CAR LIST) (CDR LIST))),
```

and replace

```
(QSORT (LESSP-LIST (CAR LIST) (CDR LIST)))
```

with

```
(LESSP-LIST (CAR LIST) (CDR LIST)).
```

A similar sequence of events takes place for the other argument of `APPEND`. The equivalence prover uses the congruence frames `PERMUTATION-APPEND-RIGHT-FRAME` and `PERMUTATION-CONS-RIGHT-FRAME` to establish that $\mathcal{A} = \{\texttt{EQUAL}, \texttt{PERMUTATION}\}$ when rewriting

```
(QSORT (GEQ-LIST (CAR LIST) (CDR LIST))),
```

and reduces this term to

```
(GEQ-LIST (CAR LIST) (CDR LIST))
```

with the inductive hypothesis. The end result is that

```
(PERMUTATION (APPEND (QSORT (LESSP-LIST (CAR LIST) (CDR LIST)))
                     (CONS (CAR LIST)
                           (QSORT (GEQ-LIST (CAR LIST) (CDR LIST)))))
             LIST)
```

has been heuristically rewritten to

```
(PERMUTATION (APPEND (LESSP-LIST (CAR LIST) (CDR LIST))
                     (CONS (CAR LIST)
                           (GEQ-LIST (CAR LIST) (CDR LIST))))
             LIST),
```

which can be further reduced to `T` by means of the appropriate lemma.

2.4 Equivalence Rewrite Rules

Normally, a rule like

Lemma. `LESSP-GEQ-QSORT-PARTITION`

```
(IMPLIES
 (LISTP LIST)
 (PERMUTATION (APPEND (LESSP-LIST (CAR LIST) (CDR LIST))
                      (CONS (CAR LIST)
                            (GEQ-LIST (CAR LIST) (CDR LIST))))
              LIST))
```

is used a a matching rule; that is, this rule reduces instances of its conclusion to `T`. However, if this lemma were proved *after* `PERMUTATION` was admitted as an equivalence relation, then this lemma will be stored as a directed rewrite rule, to be applied whenever `PERMUTATION` $\in \mathcal{A}$. If such were the case in the preceding example, then the conclusion

```
(PERMUTATION (APPEND (LESSP-LIST (CAR LIST) (CDR LIST))
                     (CONS (CAR LIST)
                           (GEQ-LIST (CAR LIST) (CDR LIST))))
             LIST)
```

would have first been reduced to

```
(PERMUTATION LIST LIST)
```

by directed rewriting. The above term would then have been reduced to `T` by the reflexivity of `PERMUTATION`. Of course, this simple example does not really demonstrate the utility of directed rewriting by equivalence rules. In more complex examples, however, it appears to be a very useful feature.

2.5 Justification

Did the equivalence prover provide any real advantage in the preceding proof? Imagine trying to prove `PERMUTATION-QSORT` in NQTHM, given the same definitions and lemmas available to the equivalence prover. The congruence frames used above are almost worthless as directed rewrite rules, because they all contain a free variable in the hypothesis. Possible approaches to getting this proof in NQTHM would be to `USE` the proper instances of the congruence frame lemmas, or to combine the congruence frames into these two highly specialized lemmas:

Lemma. `CROCK-1`

```
(IMPLIES
 (PERMUTATION X Y)
 (EQUAL (PERMUTATION (APPEND X Z) L)
        (PERMUTATION (APPEND Y Z) L)))
```

Lemma. `CROCK-2`

```
(IMPLIES
 (PERMUTATION Y Z)
 (EQUAL (PERMUTATION (APPEND W (CONS X Y)) L)
        (PERMUTATION (APPEND W (CONS X Z)) L)))
```

Notice that even here there are free variables in the hypothesis. These crock lemmas will "work" in the proof only because NQTHM will happen to find an exact match for the hypotheses in the clause representing the inductive step.

3 User Interface

Externally, the equivalence prover differs from NQTHM only by the addition of two new lemma classes for `PROVE-LEMMA`: `ESTABLISH-EQUIVALENCE` and `CONGRUENCE-FRAME`. After introducing these two lemma classes, we describe in detail how the modified rewriter uses congruence frames.

3.1 The `ESTABLISH-EQUIVALENCE` Lemma Class

If `r` is an equivalence relation, then this fact is recorded by the event:

```
(PROVE-LEMMA name ((ESTABLISH-EQUIVALENCE r) [other-classes])
  (AND (r x x m1 ... mN)
       (EQUAL (r x y m1 ... mN) (r y x m1 ... mN))
       (IMPLIES
        (AND (r x y m1 ... mN)
             (r y z m1 ... mN))
        (r x z m1 ... mN)))
  [hints]),
```

subject to the constraints that:

- `r` is boolean. That is, the equivalence prover must have noted that

  ```
  (OR (FALSEP (r x y m1 ... mN)) (TRUEP (r x y m1 ... mN))
  ```

  is a theorem when `r` was admitted.
- Within each conjunct, the terms `x`, `y`, `z`, and `m1`,...,`mN` must all be distinct variables.
- Any of the symmetric variants of the transitivity conjunct are allowed.
- The three conjuncts asserting the reflexivity, symmetry, and transitivity of `r` may appear in any order.

Once a function `r` is established as an equivalence relation by the event `name`, all implicit use of `r` as an equivalence relation (see Section 4) is controlled by `name`. That is, if `name` is disabled, then implicit reasoning about `r` will cease. Disabling `name` will also effectively disable any congruence frame for which `r` is the equivalence relation appearing in the hypothesis of the frame. There is no way to disable equivalence reasoning for `EQUAL` or `IFF`.

3.2 The `CONGRUENCE-FRAME` Lemma Class

Congruence frames are created by events of the form:

```
(PROVE-LEMMA name ((CONGRUENCE-FRAME (r1 x y m1 ... mL))
                   [other-classes])
  (IMPLIES
   hyps
   (r2 (f t1 ... x ... tM) (f t1 ... y ... tM) p1 ... pN))
  [hints]),
```

We will refer to the designated hypothesis, `(r1 x y m1 ... mL)`, as the *frame hypothesis.* The body of the congruence frame is subject to the following restrictions:

- Both `r1` and `r2` are equivalence relations. The equivalence relations are `EQUAL`, `IFF`, or any other symbol for which an `ESTABLISH-EQUIVALENCE` lemma has been proved in the current history.
- The terms `x` and `y` are distinct variables.
- The term `hyps` is either the frame hypothesis, or a conjunction containing the frame hypothesis. Furthermore, the variable `y` may not appear in any hypothesis other than the frame hypothesis.
- The terms `t1`,...,`tN` are arbitrary except that they may not contain either of the variables `x` or `y`.
- The moduli `p1`,...,`pN` are arbitrary except that none of them may contain the variable `y`.
- The moduli `m1`,...,`mL` are arbitrary except that any variable occurring in `m1`,...,`mL` must either be `x`, occur in `t1`,...,`tM`, occur in `p1`,...,`pN`, or occur in another hypothesis.

Disabling `name` prevents the congruence frame from being used. As we mentioned earlier, `name` can also be effectively disabled by disabling the `ESTABLISH-EQUIVALENCE` lemma for `r1`.

3.3 Using Congruence Frames

Congruence frames are not stored as rewrite rules by the equivalence prover. Instead, the information contained in a congruence frame is extracted and used heuristically. We previously introduced the set $\mathcal{A}$ of function symbols of equivalence relations that preserve equality at the current point in the term. In the equivalence prover, this set is represented by a list known as the `active-equalities`. Each entry in `active-equalities` is a triple, $\langle symbol, moduli, lemmas\rangle$ where *symbol* is the function symbol of an equivalence relation, *moduli* is a list of the moduli of equivalence, and *lemmas* is a list of the names of all of the congruence frames that support the presence of *symbol* in `active-equalities`.

Rewriting normally begins at the literal level, where propositional equality preserves the truth value of the clause. Thus, at the beginning of rewriting,

$$\texttt{active-equalities} = (\langle \texttt{EQUAL}, (), () \rangle, \langle \texttt{IFF}, (), () \rangle).$$

The `active-equalities` are also set to this value when rewriting the first argument of `IF`, and when commencing to relieve a hypothesis of a rewrite rule. The `active-equalities` are updated every time the rewriter descends into the argument of a function. This is critical for soundness: If an equivalence relation appears in `active-equalities` at one level, it must either be removed as the rewriter passes into the next level, or there must be a congruence frame that justifies its continued membership in `active-equalities` at the lower level.

Assume that the current term is

```
(f x1 ... xi ... xM),
```

and the rewriter is about to rewrite `xi`. Then the following algorithm is executed:

1. Set $\texttt{temp} = (\langle \texttt{equal}, (), () \rangle)$. We will use `temp` to collect the new value of `active-equalities` for `xi`.

2. For every congruence frame associated with the $\texttt{i}^{\text{th}}$ argument of `f` do:[4]

   (a) Assume that the congruence frame was established by the lemma `name`, has the form

   ```
   (IMPLIES
    hyps
    (r2 (f t1 ... x ... tM) (f t1 ... y ... tM) p1 ... pN)),
   ```

   and frame hypothesis `(r1 x y m1 ... mL)`.

   (b) If `r2` is not in `active-equalities`, `name` is disabled, or the `ESTABLISH-EQUIVALENCE` lemma for `r1` is disabled, then return to step 2 to try the next frame.

   (c) For every triple $\langle \texttt{r2}, (\texttt{q1}, \ldots, \texttt{qN}), \textit{lemmas} \rangle$ in `active equalities`, if:

      i. `(f t1 ... x ... tM)` unifies[5] with `(f x1 ... xi ... xM)`;

[4]Referring back to the schematic congruence frame at the beginning of Section 3.2, the abstract representation of that congruence frame would have been "associated" with the argument position of `f` where the variable `x` appears.

[5]Here, and throughout the rest of this note, whenever we use the term *unify* we mean "one-way" unification, and whenever we say that `X` unifies with `Y`, it is understood that `X` contains the "variables".

ii. `p1`,...,`pN` unify with `q1`,...,`qN` respectively;

iii. All of the `hyps` except the frame hypothesis are relieved.

then add $\langle$`r1`, (`m1`$'$, ..., `mL`$'$), (`name` . *lemmas*)$\rangle$ to `temp`, where `m`i' represents `m`i after applying the accumulated substitution.

3. Set `active-equalities` = `temp`.

The restrictions on the forms of the congruence frame lemmas guarantee that every variable in `m1`,...,`mL` will be bound before these moduli are entered into `active-equalities`. The restrictions also guarantee that the variable `y` remains unbound throughout the process.

Congruence frames are unique among lemmas in that they are applied outside-in, whereas the Boyer-Moore rewriter is strictly inside-out. The congruence frame algorithm requires that the current term, `(f x1 ... xi ... xM)`, must unify with the conclusion of the congruence frame. But notice that because the rewriter is rewriting inside-out, when the rewriter enters the subterm `xi` those arguments to the left of `xi` will have already been rewritten, while those arguments to the right of `xi` will not have been rewritten. The equivalence prover uses the original, non-rewritten values of all of the terms `x1`,...,`xM` in this unification, however. The user needs to be cognizant of this fact only if the terms in the conclusion of the congruence frames are not all unique variables. The user should also realize that although the moduli from the conclusion of the congruence frame may be modified by substitution, they are currently *not* rewritten before they are entered into the `active-equalities`.

4 Features

In this section we compare and contrast equality reasoning in NQTHM with analogous processes in the equivalence prover. Most of the information on NQTHM can be found in other sources [BM79,BM88b] and is concisely summarized here. Note that these comments are based on an examination of the current code for NQTHM [BM88a] and in some cases are slightly at odds with earlier reports. The equivalence prover treats `EQUAL` exactly as NQTHM does, and all reasoning about other equivalence relations `r` is predicated on the `ESTABLISH-EQUIVALENCE` lemma for `r` being enabled. The equivalence prover treats `IFF` like any other equivalence relation.[6] We begin the discussion with the general properties of equivalence relations, and conclude with the use of the equality axioms and directed rewriting.

[6] The user will probably never notice the difference between NQTHM and the equivalence prover in terms of handling `IFF`.

4.1 Reflexivity

NQTHM automatically reduces `(EQUAL t1 t2)` to `T` either if the terms `t1` and `t2` are textually the same, or if `t1` and `t2` have the same type, and there is only one element [7] of that type. Similarly, NQTHM reduces `(EQUAL t1 t2)` to `F` if the terms `t1` and `t2` are of different types. NQTHM also does other kinds of type reasoning if the two arguments are of different, but intersecting types, and reduces `EQUAL` terms whose argument are `EQUAL` terms.

In the equivalence prover, the term `(r t1 t2 m1 ... mN)` is also reduced to `T` if `r` is an equivalence relation and `t1` and `t2` are the same, or are members of the same singleton type set. None of the other sorts of reasoning done for `(EQUAL t1 t2)` are justified simply by the fact that `r` is an equivalence relation.

4.2 Symmetry

Reasoning about the symmetry of `EQUAL` appears in several guises. In NQTHM, `EQUAL` is the only function whose arguments will be commuted by the unifier when attempting to unify two terms.

Whenever the term `(EQUAL t1 t2)` is assumed false, either because it appears as `(EQUAL t1 t2)` in the current clause or because the rewriter is rewriting the false branch of an `IF` controlled by `(EQUAL t1 t2)`, the prover stores both symmetric versions in an internal data structure known as the `type-alist`, along with the indication that the equality literal is false. Thus should either symmetric version appear in a term being rewritten, the `EQUAL` term is rewritten to `F` after a quick scan of the `type-alist`.

Whenever the term `(EQUAL t1 t2)` is assumed true, a slightly different process takes place. In this case, only one form of the equality is stored in the `type-alist`, and the form that is stored depends on the relative "weights" of the left and right hand sides. Quoting from [BM85],

> The "heavier" relation is a total ordering on terms. We say t_1 is *heavier* than t_2 iff either the number of variables in t_1 is greater than that in t_2, or the number of variables in the two are equal but the "size" of t_1 is greater than that of t_2, or the number of variables in and the sizes of the two are equal and t_1 comes later than t_2 in the lexicographic ordering of terms. By *size* we mean the number of open parenthesis in the unabbreviated presentation of the term.

So, if `(EQUAL t1 t2)` is assumed true, it will only be stored as `(EQUAL t1 t2)` if `t1` is "heavier" than `t2`. Otherwise, the equality is stored as `(EQUAL t2 t1)`. This procedure, coupled with techniques described later, insures ground completeness for equality.

[7] A singleton type set is created by a shell with no destructors and no bottom objects, e.g. `(TRUE)` and `(FALSE)`.

The equivalence prover implements all of the above for equivalence terms `(r t1 t2 m1 ... mN)`, except for commuting the arguments during unification.

4.3 Transitivity

Reasoning about the transitivity of `EQUAL` is subsumed by the processing of the equality axioms for functions in NQTHM. The same is true for the equivalence prover.

4.4 Using the Equality Axioms

The most common form of equality reasoning in NQTHM is the implicit use of the equality axioms, or "substituting equals for equals". This type of reasoning also appears in several forms.

Using Equality Hypotheses

The last step in rewriting a term is to check whether the term matches the left-hand side of a "true" equality, that is, an equality literal stored in the `type-alist` as true. If so, the term is replaced by the right-hand side of the equality.

The equivalence prover treats every "true" equivalence as an equality at this point. Assume that the term is `t1`, and the literal `(r t1 t2 m1 ... mN)` appears as a "true" equivalence in the `type-alist`. Then the rewriter will return `t2` provided there is a triple $\langle \mathtt{r}, (\mathtt{m1}, \ldots, \mathtt{mN}), \mathit{lemmas} \rangle$ in `active-equalities`. Notice that the moduli must be equal, since neither the moduli in the `type-alist` nor the moduli in `active-equalities` contain "variables".

Removing Trivial Equations

At the beginning of each round of simplification, NQTHM first tries to simplify the clause by removing trivial equations. If the clause contains a literal of the form `(NOT (EQUAL var term))`, where `var` is a variable and `term` does not contain that variable, then `var` is replaced by `term` throughout the clause, and the equality literal is deleted from the clause. If the clause contains a literal of the form `(NOT (EQUAL term exterm))` where `term` is not an explicit value and `exterm` is an explicit value, then `term` is replaced with `exterm` throughout the clause, and the equality literal is retained.

Removing trivial equivalences is accomplished by a "mini-rewriter" whose sole purpose is to replace the variable or non-explicit value term by the equivalent expression. This rewriter maintains the `active-equalities` as it rewrites each literal, and performs the substitutions as if they were "true" equality hypotheses (see above). For technical reasons, the only congruence frames utilized by this process are those *without hypotheses*, other than the frame hypothesis.[8] Also notice that when a variable is replaced by a term `EQUAL` to it, we are guaranteed that no information is lost by deleting the equality literal, since every instance of the variable will have been substituted. As there may not always be congruence frames which justify replacing a variable by an equivalent term, the equivalence prover only deletes the literal `(NOT (r var term m1 ...  mN))` if every occurrence of the variable is substituted.

Directed Rewriting

The most visible use of equality is in user supplied rewrite rules. A `REWRITE` lemma whose conclusion takes the form `(EQUAL t1 t2)` is used by NQTHM to replace instances of `t1` by `t2` whenever the hypotheses of the rule can be established. Perhaps less utilized are `REWRITE` lemmas whose conclusions take the form `(IFF t1 t2)`. These rules are used as directed rewrite rules by NQTHM at places in a term where only the propositional value matters.

Once a function symbol `r` has been admitted as an equivalence relation, any subsequent `REWRITE` lemmas with conclusions `(r t1 t2 m1 ...  mN)` will also be stored by the equivalence prover as directed rewrite rules. These rules will be used to rewrite `term` to `t2` if all of the following conditions are met:

1. The triple $\langle$`r`, (`p1`, …, `pN`), *lemmas*$\rangle$ appears in `active-equalities`;
2. The term `t1` unifies with `term`;
3. The moduli `m1`,…,`mN` unify with `p1`,…,`pN`;
4. The hypotheses of the rule are relieved under the substitution.

Notice that these rules are *not* stored as matching rules, as they would be in NQTHM.

A minor problem may be that a rule like

```
(IMPLIES
 (AND (NLISTP X)
      (NLISTP Y))
 (PERMUTATION X Y))
```

will be rejected if `PERMUTATION` has been established as an equivalence relation. This is because the equivalence prover will correctly note that this rule attempts to rewrite a variable. Instead, this rule can be entered in the equivalent form

[8] In technical terms, trivial equations are removed before the `type-alist` is created, and the `type-alist` is critical for relieving hypotheses. We could arrange for the `type-alist` to be available at the time trivial equations are removed, but this might be very expensive computationally.

```
(IMPLIES
 (AND (NLISTP X)
      (NLISTP Y))
 (EQUAL (PERMUTATION X Y)
        T)).
```

This trick is a general way to prohibit a rule whose conclusion is an equivalence from being used as a directed rewrite rule, while allowing it to function as a matching rule.

5 Caveats

Although the equivalence prover is currently very usable, the implementation is not complete. This section documents several features that NQTHM provides for `EQUAL` but that are currently not implemented in the equivalence prover. We also include comments on related topics. No implication about the relative importance of these features is implied by their order here.

`BREAK-LEMMA`. No break facility is provided for `CONGRUENCE-FRAME` lemmas.

`META`-Lemmas. There is no support for `META` lemmas for equivalence relations other than `EQUAL`.

Linear Arithmetic. The `active-equalities` are set to $(\langle \texttt{EQUAL}, (), () \rangle)$ prior to entering the linear arithmetic package, and the code for linear arithmetic was not modified. Thus, any congruence frame for an argument of `LESSP` will probably be ignored.

Abbreviation Expansion. NQTHM applies selected hypotheses-free `EQUAL` rewrite rules when non-recursive definitions are initially expanded. This code has not been modified in the equivalence prover.

Congruence Frame Order. The user should make no assumptions about how the order in which congruence frames were proved might affect the course of a proof. If the "wrong" congruence frame is being used during an automatic proof, it can be disabled like any other lemma.

Reporting. The equivalence prover reports the use of congruence frames as if they were rewrite rules. Also, whenever implicit reasoning occurs for an instance of an equivalence relation `r`, the equivalence prover will report the use of the `ESTABLISH-EQUIVALENCE` lemma for `r`, regardless of whether the implicit reasoning had any effect on the proof. Most instances of "bogus" reporting occur because the symmetry of `r` is used during construction of the `type-alist`.

Commutative Unification. The equivalence prover does not commute the arguments of equivalence relations other than `EQUAL` during unification.

Cross Fertilization. The "cross fertilization" heuristic has not been extended to equivalence relations other than `EQUAL`.

Performance. On one large set of example proofs that used none of the new features[9], the equivalence prover required 12% more CPU time than NQTHM.

References

[Bev88] Bill Bevier. *A Library for Hardware Verification.* Internal Note 57, Computational Logic, Inc., June 1988. Draft.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[BM85] Robert S. Boyer and J Strother Moore. *Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic.* Technical Report ICSCA-CMP-44, University of Texas at Austin, 1985.

[BM88a] R.S. Boyer and J S. Moore. *The Code for a Computational Logic.* Technical Report CLI-24, Computational Logic, Inc., 1988.

[BM88b] R.S. Boyer and J S. Moore. *A User's Manual for a Computational Logic.* Technical Report CLI-18, Computational Logic, Inc., 1988.

[BMRS88] R.S. Boyer, J S. Moore, D. Russinoff, and N. Shankar. *Basic Events for a Computational Logic.* Technical Report CLI-23, Computational Logic, Inc., 1988.

A Events for `PERMUTATION-QSORT`

```
(proveall "qsort" '(

;;  This is Bill Bevier's SETS-AND-BAGS library, loaded and saved in the
;;  equivalence prover.

(note-lib "sets-and-bags")

;;  Turn off some possibly interfering lemmas.

(disable not-permutation)
(disable permutation-right-cons1)
(disable permutation-right-cons)
(disable commutativity-of-permutation)
(disable permutation-reflexivity)
```

[9] The first "proveall" from [BMRS88].

```
;;  The function LENGTH.

(defn length (list)
  (if (nlistp list)
      0
    (add1 (length (cdr list)))))

;;  A few facts about DELETE, APPEND, and MEMBER.

(prove-lemma delete-append (rewrite)
   (equal (delete x (append list1 list2))
          (if (member x list1)
              (append (delete x list1) list2)
            (if (member x list2)
                (append list1 (delete x list2))
              (append list1 list2)))))

(prove-lemma member-append (rewrite)
  (equal (member x (append list1 list2))
         (or (member x list1)
             (member x list2))))

;;  Establish PERMUTATION as an equivalence relation.

(prove-lemma permutation-equivalence ((establish-equivalence permutation))
  (and (permutation x x)
       (equal (permutation x y) (permutation y x))
       (implies
        (and (permutation x y)
             (permutation y z))
        (permutation x z)))
  ;;Hint
  ((use (permutation-reflexivity (l x))
        (commutativity-of-permutation (b x) (a y))
        (transitivity-of-permutation (a x) (b y) (c z)))))

;;  Congruence frames for PERMUTATION with CONS, APPEND, DELETE, and MEMBER.

(prove-lemma permutation-cons-right-frame
  ((congruence-frame (permutation y z)))
  (implies
   (permutation y z)
   (permutation (cons x y) (cons x z)))
  ;;Hint
  ((enable permutation)))

(prove-lemma permutation-append-left-frame
  ((congruence-frame (permutation x y)))
  (implies
   (permutation x y)
   (permutation (append x z) (append y z)))
  ;;Hint
  ((enable permutation)))

(prove-lemma permutation-append-right-frame
```

```
  ((congruence-frame (permutation x y)))
  (implies
   (permutation x y)
   (permutation (append z x) (append z y)))
  ;;Hint
  ((enable permutation)))

(prove-lemma permutation-delete-right-frame
  ((congruence-frame (permutation y z)))
  (implies
   (permutation y z)
   (permutation (delete x y) (delete x z)))
  ;;Hint
  ((enable permutation)))

(prove-lemma permutation-member-right-frame
  ((congruence-frame (permutation y z)))
  (implies
   (permutation y z)
   (equal (member x y) (member x z))))

;;  The functions LESSP-LIST and GEQ-LIST partition a list into two parts.
;;  LESSP-LIST contains all entries less than VAL, and GEQ list contains all
;;  entries greater than or equal to val.

(defn lessp-list (val list)
  (if (nlistp list)
      nil
    (if (lessp (car list) val)
        (cons (car list) (lessp-list val (cdr list)))
      (lessp-list val (cdr list)))))

(defn geq-list (val list)
  (if (nlistp list)
      nil
    (if (leq val (car list))
        (cons (car list) (geq-list val (cdr list)))
      (geq-list val (cdr list)))))

;;  These lemmas are necessary to prove the admissibility of QSORT.

(prove-lemma length-lessp-list ()
  (leq (length (lessp-list val list))
       (length list)))

(prove-lemma lessp-length-lessp-list (rewrite)
  (implies
   (listp list)
   (lessp (length (lessp-list val (cdr list)))
          (length list)))
  ;;Hint
  ((use (length-lessp-list (list (cdr list))))))

(prove-lemma length-geq-list ()
  (leq (length (geq-list val list))
```

```
      (length list)))

(prove-lemma lessp-length-geq-list (rewrite)
  (implies
   (listp list)
   (lessp (length (geq-list val (cdr list)))
          (length list)))
  ;;Hint
  ((use (length-geq-list (list (cdr list))))))

;;  MEMBERship lemmas for LESSP-LIST and GEQ-LIST.

(prove-lemma member-lessp-list (rewrite)
  (equal (member x (lessp-list val list))
         (and (member x list)
              (lessp x val))))

(prove-lemma member-geq-list (rewrite)
  (equal (member x (geq-list val list))
         (and (member x list)
              (leq val x))))

;;  We prove that LESSP-LIST and GEQ-LIST partition LIST.

(prove-lemma lessp-geq-partition ()
  (permutation (append (lessp-list val list)
                       (geq-list val list))
               list)
  ;;Hint
  ((enable permutation permutation-right-cons)))

;;  The form of the above lemma needed for the proof of QSORT.

(prove-lemma lessp-geq-qsort-partition (rewrite)
  (implies
   (listp list)
   (permutation (append (lessp-list (car list) (cdr list))
                        (cons (car list)
                              (geq-list (car list) (cdr list))))
                list))
  ;;Hint
  ((use (lessp-geq-partition (val (car list)) (list list)))))

;;  The recursive quick-sort, QSORT.

(defn qsort (list)
  (if (nlistp list)
      nil
    (append (qsort (lessp-list (car list) (cdr list)))
            (cons (car list)
                  (qsort (geq-list (car list) (cdr list))))))
  ;;Hint
  ((lessp (length list))))

;;  With equivalence, the proof that QSORT returns a permutation is a snap!
```

```
(prove-lemma permutation-qsort (rewrite)
  (permutation (qsort list) list))

;;  End PROVEALL

))
```