Introduction to the OBDD Algorithm for the ATP Community

J Strother Moore

Technical Report 84 October, 1992

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the National Computer Security Center (Contract MDA904-92-C-5167). 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 National Computer Security Center, or the U.S. Government.

Abstract

We describe in terms familiar to the automated reasoning community the graph-based algorithm for deciding propositional equivalence published by R.E. Bryant in 1986. Such algorithm, based on *ordered binary decision diagrams* or *OBDD*s, are among the fastest known ways to decide whether two propositional expressions are equivalent and are generally hundreds or thousands of times faster on such problems than most automatic theorem proving systems. An OBDD is a normalized **`IF`** (‘‘if-then-else’’) expression in which the tests down any branch are ascending in some previously chosen fixed order. Such **`IF`** expressions represent a canonical form for propositional expressions. Three coding tricks make it extremely efficient to manipulate canonical **`IF`** expressions. The first is that two canonicalized expressions can be rapidly combined to form the canonicalized form of their disjunction (conjunction, exclusive-or, etc) by exploiting the fact that the tests are ordered. The second is that every distinct canonical **`IF`** expression should be assigned a unique integer index to enable fast recognition of identical forms. The third trick is that the operation in which one combines canonicalized subterms term should be ‘‘memo-ized’’ or cached so that if the same operation is required in the future its result can be looked up rather than recomputed.

1. Preface

This paper explains briefly the algorithm published by Bryant in [4]. Since 1986, so-called ‘‘OBDD’’ algorithms have been remarkably successfull at handling propositional equivalence problems arising in the context of digital hardware design. But despite the fact that they are often thousands of times faster than traditional tautology checkers in automated reasoning systems, the OBDD literature seems to have been largely ignored by our community. This is at first hard to understand. The IFIP benchmark files containing challenging propositional equivalence problems and used world-wide are ignored by the automated reasoning community because many of our systems ‘‘go to lunch’’ when presented with them. We ignore these challenges and their known solutions at our peril: we lose the opportunity to improve our own systems and we make ourselves irrelevant to the hardware verification community, a community with which we ought to have many links.

My personal reason for ignoring the OBDD literature was that its terminology was unfamiliar. Having finally understood some of it, I would like to take this opportunity to explain it to my friends. Before proceeding I would like to make it clear that I am not here advocating OBDDs as the end-all of propositional tautology checking. For example, recent improvements of the Davis-Putnam procedure by Stickel [10] and by Zhang [11] show promise. In addition, propositional decision procedures are notoriously sensitive to apparently minor restatements of the input problem, so it is unclear that OBDDs will be of use in our more general settings. Even if they are, much work must be done to integrate any decision procedure into our systems and, as illustrated in [2] sometimes less efficient procedures can produce better overall performance because it is cheaper to invoke them from the general setting. But, having said all this, it should be added that OBDDs can be very effective at deciding propositional equivalence problems. Furthermore, they are very easy to implement in a formula-manipulation system. This paper should therefore be taken as a tutorial introduction to them and as a suggestion that, after suitable experimentation, our systems might benefit from using OBDD techniques when appropriate.

Research into OBDDs is very active and there has been much progress since Bryant’s seminal paper in 1986. But I limit this tutorial to the algorithm in that paper because I believe once the basic idea is grasped the improvements are easily guessed or understood in our community. More importantly to me, I would like to encourage the generalization of the technique so that general-purpose theorem provers can be improved.

In closing this preface I would like to urge readers to read the original Bryant paper, [4], as well as [3] in which Brace, Rudell and Bryant present, quite clearly, the if-then-else perspective and the ‘‘coding tricks’’ described here. In addition, Bryant has recently written an excellent survey of current OBDD techniques and applications, [5].

2. Logical Basis

I base my description on the Nqthm (i.e., Boyer-Moore) logic because it is familiar to me. For readers unfamiliar to it, I offer the following remarks. Consider propositional calculus with function symbols and equality. Let there be two distinct individuals, denoted by the constant symbols **T** and **F**. Thus,

Axiom. `T` $\neq$ `F`

Let **IF** be a three-place function symbol satisfying the following two axioms:

Axiom. `X` $\neq$ `F` $\rightarrow$ `(IF X Y Z)` $=$ `Y`

Axiom. `X` $=$ `F` $\rightarrow$ `(IF X Y Z)` $=$ `Z`

No harm arises by restricting one’s attention to the situation in which **T** and **F** are the only individuals and the **IF** axioms are:

Axiom. `(IF T Y Z) = Y`

Axiom. `(IF F Y Z) = Z`

Equivalently, one can imagine that every formula shown below has some implicit hypotheses restricting each variable to be *Boolean*, i.e., either **T** or **F**. We call the first argument to **IF** the *test* and the other two arguments the *true* and *false branches,* respectively.

There are four important theorem (schemas) about **IF**. The first is that **IF** distributes over all function symbols:

Theorem. **IF** Distribution

```
(fn A_1 ... (IF X Y Z) ... A_n)
=
(IF X
    (fn A_1 ... Y ... A_n)
    (fn A_1 ... Z ... A_n))
```

The second is known as the ‘‘reduction’’ schema and just says that once **X** has been tested along a branch, its value is known.

Meta-Theorem. Reduction
Consider **(IF** *x* α β**)**, where *x* is a Boolean-valued term. All occurrences of *x* in α may be replaced by **T** and all occurrences in β may be replaced by **F**.

The third result is just known as the ‘‘**IF-X-Y-Y** theorem:’’

Theorem IF-X-Y-Y
`(IF X Y Y) = Y.`

The fourth result is ‘‘**IF-X-T-F**.’’

Theorem IF-X-T-F
`(IF X T F) = X.`

All four of these results are proved by considering the cases: is the test of the **IF** equal to **T** or **F**?

With **IF** we can define the usual propositional connectives as function symbols:

Definitions.

```
(NOT P) = (IF P F T)

(AND P Q) = (IF P (IF Q T F) F)

(OR P Q) = (IF P T (IF Q T F))

(XOR P Q) = (IF P (IF Q F T) (IF Q T F))

(IFF P Q) = (IF P (IF Q T F) (IF Q F T))
```

The challenge is to write an algorithm to decide every question of the form **(IFF** *x y***)**, where *x* and *y* are Boolean expressions. A *Boolean expression* for the present purposes can be defined as a Boolean valued variable symbol, the constants **T** and **F**, or the application of one of the propositional functions above to the appropriate number of Boolean expressions.

3. IF-Normal Form

The way such challenges are attacked in Nqthm is both illustrative and takes us most of the way to OBDD solution: Expand the **`IFF`** term into ‘‘**`IF`**-normal form’’ and then see if it is **`T`**.

We define **`IF`**-*normal form* as follows. **`T`** and **`F`** are in **`IF`**-normal form. The only other terms in **`IF`**-normal form are expressions of the form **`(IF`** *x y z* **`)`** where

- *x* contains no **`IF`**s, and is neither **`T`** nor **`F`**;
- *x* does not occur in *y* or *z*,
- *y* and *z* are not identical, and
- *y* and *z* are in **`IF`**-normal form.

Thus, the Boolean-valued variable symbol **`X`** is not in **`IF`**-normal form, but the equivalent **`(IF X T F)`** is in **`IF`**-normal form. **`(IF (IF A B C) X Y)`** is not in **`IF`**-normal form but the equivalent **`(IF A (IF B X Y) (IF C X Y))`** is.

Any Boolean valued term can be *normalized*, i.e., reduced to an equivalent **`IF`**-normal form, by distributing **`IF`**s so as to remove all **`IF`**s from tests, replacing each terminal variable symbol, *x*, by the equivalent **`(IF`** *x* **`T F)`** (a transformation justified by **`IF-X-T-F`**), and exhaustively applying reduction, **`IF-X-Y-Y`**, and the axioms defining **`(IF T ...)`** and **`(IF F ...)`**. Normalization may increase the size of an expression exponentially since **`IF`** distribution duplicates expressions.

An example normalization is shown below.

```
(IF (IF A C B) (IF A B T) T)
=                               [by IF distribution]
(IF A
    (IF C (IF A B T) T)
    (IF B (IF A B T) T))
=                               [by reduction on A]
(IF A
    (IF C (IF T B T) T)
    (IF B (IF F B T) T))
=                               [by IF axioms]
(IF A (IF C B T) (IF B T T))
=                               [by IF-X-Y-Y]
(IF A (IF C B T) T)
=                               [by IF-X-T-F]
(IF A (IF C (IF B T F) T) T)
```

Normalization can be used as a tautology checker. To determine if a Boolean expression as defined above is a propositional tautology merely expand all the propositional function symbols into **`IF`** terms and normalize the result. If the normal form is **`T`** the expression is a tautology; otherwise it is not. When the normal form is not **`T`**, a counterexample can be read off any branch concluding with **`F`**. This obvious theorem relating tautologies and normalization was proved by Nqthm in 1976 [1].

`IF`-normal forms are not canonical: an expression may have multiple non-identical but equivalent normal forms. For example, **`(IF A (IF B F T) (IF B T F))`** and **`(IF B (IF A F T) (IF A T F))`** are both in **`IF`**-normal form and are equivalent. Both of these terms are equivalent to **`(XOR A B)`** and **`(XOR B A)`**.

To define a canonical form we must adopt some order in which the variable symbols are to be tested. For the moment we will fix the order to be simply alphabetic, though in practice one chooses the ordering to suit the problem.

A term is in **`IF`**-*canonical* form precisely if it is in **`IF`**-normal form and the sequence of variables tested down each branch is ascending in the order. Thus **`(IF A (IF B F T) (IF B T F))`** is in **`IF`**-canonical form. **`(IF B (IF A F T) (IF A T F))`** is not, because **`B`** is tested before **`A`**.

By replacing *x* by the equivalent **`(IF`** *var x x***`)`** and reducing on *var* in the two occurrences of *x* we can ‘‘lift’’ any variable, *var*, to the top of a term and eliminate all other occurrences of it. By lifting the ‘‘smallest’’ variable in *x* and then recursively doing that to the two branches, we obtain an **`IF`**-expression whose ‘‘natural’’ normalization is canonical. This inefficient algorithm is noted only to prove that there exists a canonical form for every *x*.

That this form is indeed canonical is analogous to the theorem that there is only one ordered permutation of a given list of numbers.

In the terminology of the hardware verification community, a canonical **`IF`** expression is an ‘‘ordered binary decision diagram’’ or ‘‘OBDD.’’

A common illustration in the OBDD literature is to consider the ‘‘binary decision diagram’’ corresponding to a nest of **`XOR`**s. Let us consider **`(XOR (XOR A C) B)`**.

```
(IF A
    (IF B
        (IF C T F)
        (IF C F T))
    (IF B
        (IF C F T)
        (IF C T F)))
```

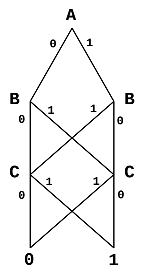

Figure 1: A Canonical **`IF`** and Its OBDD

In Figure 1 we show the **`IF`**-canonical form of this expression and the corresponding OBDD. The left-most **`C`** node in the OBDD means ‘‘if **`C`** is 1 (i.e., **`T`**) return 1; otherwise return 0 (i.e., **`F`**).’’ We write this **`(IF C T F)`**. The right-most **`C`** node is **`(IF C F T)`**. The left-most **`B`** node is thus **`(IF B (IF C F T) (IF C T F))`**, etc.

`(IFF` be_1 be_2**`)`** is a tautology iff the canonical form of be_1 is identical to that of be_2. This is all there is, logically speaking, to the OBDD algorithm. The trick is how to compute the two canonical forms efficiently.

4. Efficiency Considerations

We are interested in canonicalizing Boolean expressions as defined above. For example, we want a program to map **`(XOR (XOR A C) B)`** into the **`IF`**-tree shown above. The basic canonicalization algorithm simply descends recursively through the expression, canonicalizing the arguments to an operation and then merging the results to form the answer. Three simple programming tricks make it extremely efficient. Roughly speaking they are the ideas in ‘‘merge sort,’’ ‘‘hash cons,’’ and ‘‘memoizing’’ or ‘‘caching.’’

Suppose you are canonicalizing **(***op x y***)**, where *op* is some Boolean function such as **`AND`** or **`XOR`**, and you have recursively canonicalized *x* and *y*. Thus, *x* and *y* are both canonical **`IF`** trees. If either is a constant, the answer is **`T`**, **`F`**, the other tree, or the negation of the other tree, depending on the particular *op*. For example, if *x* is **`T`** then, if *op* is **`OR`** the result is **`T`**, if *op* is **`AND`** the result is *y*, and if *op* is **`XOR`** the result is the negation of *y*, i.e., globally swapping **`T`** for **`F`** and vice versa in *y*. On the other hand, if neither is a constant then both *x* and *y* are **`IF`**-terms. This is where the ‘‘merge sort’’ trick is used. Let *x* be **`(IF`** *vx tx fx***`)`** and let *y* be **`(IF`** *vy ty fy***`)`**. We are trying to form the canonical form of **(***op x y***)**. There are only three cases to consider: *vx* and *vy* are the same variable symbol, *vx* occurs before *vy* in the ordering, or *vy* occurs before *vx* in the ordering. If *vx* is *vy* then

```
(op x y) = (IF vx (op tx ty) (op fx fy))
```

by **`IF`** distribution and reduction. Thus, we recursively perform *op* on the respective branches of the two **`IF`**s and, provided the results are not identical to each other, make them the branches of an **`IF`** that tests *tx*. We know this is in canonical form: it is clearly in normal form and *vx* is not tested in either result because it is not tested in any of the four argument branches.

The more interesting case occurs when *vx* and *vy* are distinct. Say *vx* is earlier in the ordering than *vy*. Then

```
(op x y) = (IF vx (op tx y) (op fx y)).
```

That is, we canonicalize *op* applied to *tx* and *y* and we canonicalize *op* applied to *fx* and *y* and then we combine them in an **`IF`** that tests *vx*. This is valid by **`IF`**-distribution. At first sight though it may not appear to be canonical. How do we know that *vx* does not occur in *y* and hence require a reduction when we lift *vx* out? The reason is the ordering: *vx* is ‘‘smaller’’ than *vy* and *vy* is smaller than any other variable in *y*. Thus, we do not have to search *y* for *vx*. The symmetric case is, well, symmetric.

The example below illustrates this ‘‘merging.’’ Let [t_1] be **`(XOR (XOR A C) B)`** and let us adopt the alphabetic ordering on variables. Replacing the arguments, **`(XOR A C)`** and **`B`**, by their canonicalizations transforms [t_1] to [t_2], as shown below. We write the **`XOR`** in lower case to signify that we do not actually create the term [t_2] but use it as a logical stepping stone that is equivalent to our final answer.

```
(xor (IF A (IF C F T) (IF C T F))       [t2]
     (IF B T F)).
```

Since neither argument is a constant or variable, we compare the two variable symbols in their tests. That is, we compare **`A`** to **`B`** and see that **`A`** comes first in the alphabetic ordering. Therefore, we lift **`A`** and transform [t_2] to [t_3]:

```
(IF A                                   [t3]
    (xor (IF C F T) (IF B T F))
    (xor (IF C T F) (IF B T F))).
```

Because of the ordering and the canonicalization of the arguments we know that **`A`** occurs nowhere else and hence do not have to be alert for the possibility of reducing on **`A`** even though it now governs all remaining subterms. We continue to ripple the **`xor`** down the two canonicalized arguments. Since **`B`** comes before **`C`** in the ordering the next logical stepping stone is

```
(IF A                                   [t3]
    (IF B (xor (IF C F T) T) (xor (IF C F T) F))
    (xor (IF C T F) (IF B T F))).
```

Observe that both of the first two **xor**s above have a constant argument. Furthermore, **(XOR p T)** is the negation of **p** and **(xor p F)** is **p**. Thus [t_3] is

```
(IF A                                   [t4]
    (IF B (IF C T F) (IF C F T))
    (xor (IF C T F) (IF B T F))).
```

Similar transformations canonicalize the remaining **xor** and we are left with

```
(IF A                                   [t4]
    (IF B (IF C T F) (IF C F T))
    (IF B (IF C F T) (IF C T F))).
```

All other operations being constant, the canonicalization algorithm just scans linearly down the two **IF** trees obtained by recursively canonicalizing the arguments. Note also that it does not matter what operation we are performing except on the ‘‘base cases.’’ (That is, the **xor** above could have been **and** until we got down to the constant arguments.)

This nice state of affairs is thwarted somewhat by the requirement that we apply **IF-X-Y-Y**. That is, whenever the canonicalizer creates an **IF** expression, say **(IF** *vx tb fb***)**, it must compare the two branches, *tb* and *fb*, and collapse the **IF** if they are identical. A naive implementation of the **IF-X-Y-Y** check would require time proportional to the size of branches. Furthermore, the check is very common; in the simple example above seven **IF** expressions are created. We can speed up this identity check—which after all is the fundamental operation on canonical forms—by the ‘‘hash cons’’ idea. Hash consing, introduced for theorem-proving purposes by Peter Deutsch in [7], is the idea of representing each distinct term by a unique concrete object, so that syntactic identity can be checked in constant time by checking for pointer identity. Hashing is used when (the representations of) new terms are ‘‘consed up.’’

Our implementation of this idea is as follows. In the representation of each **IF** expression we include an integer, called the *unique id*. The integer, say *k*, associated with an **IF**, say **(IF** *x y z***)**, is unique to the triple <*x*,*y*,*z*>. In our implementation, we represent **(IF** *x y z***)** by the Lisp s-expression **'(***k x y* **.** *z***)**. The uniqueness is obtained by hashing. That is, when we wish to represent **(IF** *x y z***)** we first compute a ‘‘hash index,’’ *i*, from *x*, *y*, and *z*. Since *x* is a variable symbol in our ordering it is easy to map it to an integer; *y* and *z* are canonicalized **IF**s and so each has a unique id. The hash index *i* is thus essentially computed from three integers. The hash index is not necessarily unique to the triple <*x*,*y*,*z*>. Instead, it merely gives the location in an array at which we find an association list that maps all previously seen triples to their unique ids. By searching this list we can either find the unique id, *k*, and the concrete object used to represent **(IF** *x y z***)**, **'(***k x y* **.** *z***)**, or we can determine that this term has not yet been created. In the latter case, we invent a unique *k* by incrementing a global counter and then assign it to the triple by storing **'(***k x y* **.** *z***)** in the association list in the hash array.

Given unique ids, it is possible to implement the **IF-X-Y-Y** test by asking whether the canonicalized **IF**s in the two branches have the same unique id (comparing with Common Lisp’s **=** function). At first sight the overhead of associated with unique ids may seem excessive but unique ids are crucial to the algorithm’s efficiency because they prevent the direct comparison of exponentially growing expressions.

The final coding trick is to ‘‘memo-ize’’ the operation of merging canonical **IF**s. Even though the ‘‘merge sort’’ and ‘‘hash cons’’ tricks make the merge operation fairly efficient, typical combinatoric problems will repeatedly merge the same two canonical **IF**s. To see why this happens, suppose we are creating the canonical form of **(***op* **(IF** *v tx fx***)** *y***)**. To distribute the **IF** we form **(***op tx y***)** and **(***op fx y***)**. But if *tx* and *fx* share some substructure, say *sx*, then we may have to canonicalize **(***op sx y***)** twice.

‘‘Memo-izing’’ a function, introduced in a general setting by Donald Michie in [8, 9], is just the idea of remembering the arguments to and results produced by past applications of the function and looking up the answer (when possible) before recomputing it. For example, suppose we are canonicalizing **(XOR (XOR A C) B)** and we have canonicalized the two arguments (i.e., we are at [t_2] in the example above. Each of the two canonicalized arguments has a unique id, say k_1 and k_2. The algorithm described so far would lift **A** out, producing [t_3]. Instead, we first ask if we have every canonicalized **XOR**, k_1, and k_2 before. If so, we return the already computed and stored answer. If not, we lift **A** and compute the answer, as above, but then store it in association with **XOR**, k_1, and k_2. This is in fact the same idea as the ‘‘hash-cons’’ idea, generalized to merging instead of just the construction of an individual **IF**. The same hash array can be used. ‘‘Memo-izing,’’ which is also just another form of caching, was proposed for theorem-proving applications by Donald Michie.

The OBDD algorithm as described in [4] is canonicalization implemented with ‘‘merge sort,’’ ‘‘hash cons’’ and ‘‘memo-izing.’’

5. A Few Experiments

In 1990 the IFIP WG 10.2/WG 10.5 International Workshop on Applied Formal Methods for Correct VLSI Design was held in Houthalen, Belgium [6]. At the workshop, many OBDD algorithms were compared. To facilitate this comparison, the participants agreed in advance on a set of benchmark problems. A set of files was prepared. Each file essentially presents two propositionally equivalent expressions of some combinational logic circuit in an easy to parse lisp-like notation. Often the ‘‘same’’ circuit is presented in a graduated series of word sizes. These files have become popularly known in the OBDD community as the ‘‘IFIP Boolean Equivalence benchmarks.’’ The files were distributed to all participants before the workshop and continue to be passed around by ftp for testing purposes. I know of no central site responsible for maintaining or distributing the benchmarks but have received explicit permission from the conference organizers to distribute the copy obtained by Computational Logic, Inc. Therefore, readers interested in the IFIP benchmarks should contact me (moore@cli.com).

Below we reproduce the simplest file, **add1.be**, simply to illustrate the syntactic form of these benchmarks. The file contains two Boolean expressions, labeled **BE1** and **BE2**. **BE1** involves nine input variables, **CARRYIN**, **A[1]**-**A[4]** and **B[1]**-**B[4]**. It then defines forty-two internal variables as Boolean expressions in terms of the nine inputs and previously defined internal variables. For example, **N18** is defined to be **(NOT (OR (AND N9 (NOT N3)) (AND (NOT N9) N3)))**. Next it defines five output variables, **O[1]**-**O[4]** and **COUT**. The definition of **BE2** is analogous and has the same inputs and outputs. The problem implicit in this file is to show that the corresponding outputs are equivalent.

```
@BE1
@invar
(CARRYIN A[1] A[2] A[3] A[4] B[1] B[2] B[3] B[4])
@sub
 N3 = (A[1])
 N4 = (A[3])
 N5 = (A[2])
 N6 = (A[4])
 N7 = (not CARRYIN)
 N8 = (B[3])
 N9 = (B[1])
 N10 = (B[2])
 N11 = (B[4])
 N17 = (OR (AND (NOT N3)))
 N31 = (OR (AND (NOT N4)))
 N29 = (OR (AND (NOT N5)))
 N19 = (OR (AND (NOT N7)) (AND (NOT N7)))
 N43 = (OR (AND (NOT N6)))
 N20 = (OR (AND (NOT N19)))
 N18 = (NOT (OR (AND N9 (NOT N3)) (AND (NOT N9) N3)))
 N28 = (NOT (OR (AND N10 (NOT N5)) (AND (NOT N10) N5)))
 N32 = (NOT (OR (AND N8 (NOT N4)) (AND (NOT N8) N4)))
 N16 = (OR (AND (NOT N18)))
 N24 = (OR (AND (NOT N28)))
 N22 = (OR (AND (NOT N16)))
 N42 = (NOT (OR (AND N11 (NOT N6)) (AND (NOT N11) N6)))
 N38 = (OR (AND (NOT N42)))
 N27 = (OR (AND (NOT N24)))
 N21 = (NOT (OR (AND N20 N16) (AND (NOT N20)  (NOT N16))))
 N23 = (OR (AND (NOT N16) (NOT N3)) (AND (NOT N22) (NOT N19)))
 N25 = (OR (AND (NOT N23)))
 N26 = (NOT (OR (AND N25 (NOT  N24)) (AND (NOT N25) N24)))
 N13 = (OR (AND (NOT N26)))
 N30 = (OR (AND (NOT N32)))
 N33 = (OR (AND (NOT N27) (NOT N23)) (AND (NOT N29) (NOT N24)))
 N36 = (OR (AND (NOT N30)))
 N15 = (OR (AND (NOT N21)))
 N34 = (OR (AND (NOT N33)))
 N41 = (OR (AND (NOT N38)))
 N37 = (OR (AND (NOT N30) (NOT N4)) (AND (NOT N36) (NOT N33)))
 N39 = (OR (AND (NOT N37)))
 N40 = (NOT (OR (AND N39 (NOT N38)) (AND (NOT N39) N38)))
 N12 = (OR (AND (NOT N40)))
 N35 = (NOT (OR (AND N34 N30) (AND (NOT N34) (NOT N30))))
 N14 = (OR (AND (NOT N35)))
 N44 = (OR (AND (NOT N41) (NOT N37)) (AND (NOT N43) (NOT N38)))
@out
O[1] = (N15)
O[2] = (N13)
O[3] = (N14)
O[4] = (N12)
COUT = (N44)
@end

@BE2
@invar
(CARRYIN A[1] B[1] A[2] B[2] A[3] B[3] A[4] B[4])

@sub
COUT1 =
(OR (AND CARRYIN B[1]) (AND  CARRYIN A[1]) (AND B[1] A[1]))
COUT2 =
(OR (AND COUT1 B[2]) (AND  COUT1 A[2]) (AND B[2] A[2]))
COUT3 =
(OR (AND COUT2 B[3]) (AND COUT2 A[3]) (AND B[3] A[3]))

@out
O[1] = (EXOR A[1] B[1]  CARRYIN)
O[2] = (EXOR A[2] B[2]  COUT1)
O[3] = (EXOR A[3] B[3]  COUT2)
O[4] = (EXOR A[4] B[4]  COUT3)
COUT =
 (OR (AND  COUT3 B[4]) (AND  COUT3 A[4]) (AND B[4] A[4]))
@end
```

The implementation of the OBDD algorithm described in this paper has been coded in the applicative language Acl2 (‘‘A Computational Logic for an Applicative Core Language’’), an applicative subset of Common Lisp being developed at Computational Logic, Inc. My OBDD source code is available upon request to the author (email: moore@cli.com). I have tested it on many of the IFIP Boolean Equivalence benchmark files.

Just to give the reader a feel for the contribution of the three tricks to overall performance, I here consider the performance of various algorithms on a series of three files: **`add2.be`**, **`add3.be`** and **`add4.be`**, each of which contains two alternative definitions of binary addition for successively larger bit-vectors. The first problem involves 13 Boolean variables, the second 21 and the third 29.

The best I could get with an algorithm based on normalization (as opposed to canonicalization) was 54.85 seconds to do **`add2.be`**. (Times here are all on a Sun Sparc 2, but that is actually irrelevant since we are not comparing our results to those of others.) This essentially reflects the time it takes to consider 2^{13} cases. Tackling the larger **`add3.be`** required over 5000 seconds, which illustrates the exponential growth involved in case analysis. The still larger **`add4.be`** was essentially impossible to do by normalization.

A canonicalizer coded with the ‘‘merge sort’’ trick but neither of the other two, required 6.67 seconds on the **`add2.be`** problem. When ‘‘hash-cons’’ was added, the time dropped to 0.27 seconds, but **`add3.be`** required almost a minute and **`add4.be`** required about an hour. When ‘‘memo-ization’’ was added, the time on **`add2.be`** climbed to .57 seconds but the time on the larger examples dropped considerably, to 1.68 seconds and 4.07 seconds, respectively. Note the dramatic savings due to ‘‘memo-ization.’’ This is indicative of the frequency with which the same propositional problem arises over and over again in these benchmarks.

Our implementation has been tested on other Boolean equivalence benchmark files (including the expensive **`mul08.be`** which it completes in 144.95 seconds) and the times above are indicative of its performance. It should be noted that normalization is simply impractical in the medium to large examples; the normalization algorithm eventually exhausts the available space.

In Figure 2 we compare the performance of several algorithms on the IFIP benchmarks. The column labeled ‘‘norm’’ shows the times for each benchmark file done by our **`IF`** normalization procedure. When the column contains **`***`** it means the procedure was aborted before it completed. The next four columns are taken from the indicated pages of [6]. However, the experiments reported in [6] were conducted on a Sun Microsystems 3/60 and while our tests were conducted on a Sparc2. In our experience, the Sparc2 is roughly 4 times faster than the 3/60. Therefore, the times taken from [6] were quartered for Figure 2 so they would be roughly comparable to our experiments. The right-most column, labeled ‘‘can,’’ shows the times for our applicative OBDD algorithm.

Our times are almost an order of magnitude worse than the best OBDD implementations, a situation largely explained by the generality of our setting, the fact that we are running (essentially) a 1986 version of the OBDD algorithm, the fact that our code is written in a high-level language, and the fact that our code is entirely applicative. It would be, I believe, straightforward to verify the correctness of our implementation formally and mechanically.

But we are here not trying to compete with other implementations; we are merely trying to impress upon the automated reasoning community the power of canonicalization for this sort of problem. That power is best illustrated by the slow degradation of performance as the problem size grows. In addition, we would be well-advised to consider the remarkable performance improvements obtained by ‘‘hash cons’’ and ‘‘memo-ization,’’ especially as measured on relatively large problems.

	norm	Fischer &Bryant pg 103	Minato &Ishiura &Yajima pg 111	Madre &LeProvost pg 120	Simonis pg 128	can
add1	0.33	0.07	0.10	0.05	0.04	0.15
add2	54.85	0.23	0.20	0.17	0.39	0.62
add3	5334.53	0.41	0.32	0.22	3.65	1.70
add4	***	0.73	0.52	0.42	2.65	3.92
addsub	***	0.38	0.52	0.15	1.27	1.47
mul03	0.13	0.01	0.05	0.05	0.02	0.05
mul04	2.20	0.07	0.10	0.10	0.10	0.23
mul05	40.97	0.66	0.25	0.35	0.43	1.05
mul06	***	2.28	0.70	1.52	1.70	4.78
mul07	***	8.86	2.37	6.27	6.85	25.78
mul08	***	33.61	9.02	27.60	21.92	151.18
rip02	0.01	0.02	0.07	0.02	0.01	0.02
rip04	0.12	0.04	0.07	0.05	0.01	0.07
rip06	2.73	0.06	0.07	0.07	0.03	0.13
rip08	105.98	0.08	0.12	0.07	0.04	0.25
transp	0.01	0.01	0.07	0.02	0.01	0.02
ztwaalf1	0.23	0.07	0.10	0.05	0.04	0.08
ztwaalf2	0.28	0.07	0.05	0.05	0.03	0.08

Figure 2: Performance Comparisons (see text)

References

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

2. R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study with Linear Arithmetic. In *Machine Intelligence 11*, Oxford University Press, 1988. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

3. K.S. Brace and R.L. Rudell and R.E. Bryant. Efficient Implementation of a BDD Package. 27th ACM/IEEE Design Automation Conference, 1990, pp. 40-45.

4. R.E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers C-35*, 8 (August 1986), 677--691.

5. R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. Tech. Rept. CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, July, 1992.

6. L.J.M. Claesen (Ed.) *Formal VLSI Correctness Verification, VLSI Design Methods - II.* North-Holland, 1990.

7. L.P. Deutsch. An Interactive Program Verifier. Tech. Rept. CSL-73-1, Xerox Palo Alto Research Center, May, 1973.

8. D. Michie. Memo functions: a language feature with rote learning properties. Tech. Rept. MIP-R-29, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1967.

9. D. Michie. "‘Memo’ functions and machine learning". *Nature 218* (1968), 19-22.

10. J. Slaney and M. Fujita and M. Stickel. "Automated Reasoning and Exhaustive Search: Quasigroup Existence Problems". *Computer Mathematics and Applications* (1993?). (to appear).

11. H. Zhang. "Sato: A Decision Procedure for Propositional Logic". *Association for Automated Reasoning Newsletter* , 22 (March 1993), 1-3.

Table of Contents

1. Preface 1
2. Logical Basis 1
3. IF-Normal Form 3
4. Efficiency Considerations 5
5. A Few Experiments 7

List of Figures

Figure 1: A Canonical **`IF`** and Its OBDD 4
Figure 2: Performance Comparisons (see text
) 10