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

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 currently 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`**-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 [3]. Since 1986, so-called ''OBDD'' algorithms have swept aside all other ways of deciding propositional equivalence. But despite the fact that they are often thousands of times faster than tautology checkers in automated reasoning systems the OBDD literature has 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 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.

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, [3], as well as [2] 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, [4].

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

Theorem. **IF** Distribution

```
(fn A1 ... (IF X Y Z) ... An)
=
(IF X
    (fn A1 ... Y ... An)
    (fn A1 ... Z ... An))
```

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.

All three of these results are proved by considering the cases: is the test of the **IF** equal to **F** or not?

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 NIL)** (a transformation justified by **IF-X-Y-Y** and reduction), 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 B=(IF B 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)**.

Below we show the **IF**-canonical form of this expression and the corresponding OBDD.

```
(IF A                          A
    (IF B                     / \
        (IF C T F)           /0 1\
        (IF C F T))         B    B
    (IF B                   |\  /|
        (IF C F T)         0|1\/1|0
        (IF C T F)))        | /\ |
                            |/  \|
                            C    C
                            |\  /|
                           0|1\/1|0
                            | /\ |
                            |/  \|
                            0    1
```

The left-most **C** node 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 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 eachother, 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 argument four 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 we canonicalize *op* applied to *fx* and *y* and then we combine them in an **IF** that tests *tx*. 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.

All other operations being constant, the canonicalization algorithm just scans linearly down the two **IF** trees. Note also that it does not matter what operation we are performing, except on the ‘‘base cases.’’

This nice state of affairs is thwarted somewhat by the requirement that we apply **IF-X-Y-Y**. That is, when the canonicalizer has recursively created the two branches of its resultant **IF**, say *tb* and *fb* and is about to create **(IF** *vx tb fb***)** it must first check whether *tb* is identical to *fb* and if so, just return either. But if the **IF** trees are large, checking their identity requires time proportional to their size. We can speed up this identity check—which after all is the fundamental operation on canonical forms—by the ‘‘hash cons’’ idea.

Therefore, 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 for <*x*,*y*,*z*> or determine that none has been assigned. In the latter case, we assign one by incrementing a global counter, and add it to 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). This avoids the exploration of large expressions.

The final coding trick is to ‘‘memo-ize’’ the operation of merging canonical **IF**s. Even though the ‘‘merge sort’’ and ‘‘hash cons’’ trick 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 must 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 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. 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.

The OBDD algorithm as described in [3] is canonicalization implemented with ‘‘merge sort,’’ ‘‘hash cons’’ and ‘‘memo-izing.’’

5. A Few Experiments

I have coded this implementation of the OBDD algorithm 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. 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 its performance 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.

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.

These times are still 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.

References

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

2. 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.

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

4. 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.

Table of Contents