# Binary Decision Diagrams as a HOL Derived Rule

John Harrison

University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge
CB2 3QG
England
jrh@cl.cam.ac.uk

**Abstract.** Exhaustive testing of boolean terms has long been held to be impractical for nontrivial problems, since the problem of tautology-checking is NP-complete. Nevertheless research on Ordered Binary Decision Diagrams, which was given a great impetus by Bryant's pioneering work, shows that for a wide variety of realistic circuit problems, exhaustive analysis is tractable. In this paper we seek to explore how these datastructures can be used in making an efficient HOL derived rule, and illustrate our work with some examples both from hardware verification and pure logic.

## 1 Binary Decision Diagrams

There are many ways to test whether a Boolean expression is a tautology (i.e. true for all truth assignments of the variables it involves), such as the use of truth tables or transformation to conjunctive normal form. The problem of tautology checking is NP-complete, and therefore all known algorithms have exponential worst-case performance; in fact the complementary operation of testing Boolean satisfiability was the original NP-complete problem given by Cook [6]. Nevertheless recent work has shown that in many practical situations, particularly in the verification of combinational logic circuits, binary decision diagrams do provide a practical means of tautology checking for expressions involving a large number of variables.

The basic idea of binary decision diagrams is to build up a 'decision tree' with the variables at the nodes and either 1 (true) or 0 (false) at the leaves. At each node the tree branches into two subtrees (often referred to as the 'then' and 'else' branches, regarding each node as a conditional) which represent the subfunctions formed by assuming that variable to be true or false, respectively. In that simple form, binary decision diagrams would merely be a way of organizing a truth table, and therefore have little interest. But the following two refinements, especially the latter, have led to an enormous increase in their efficiency.

– Rather than using trees, use directed acyclic graphs, sharing any common subexpressions which arise, as proposed by Lee [11] and Akers [2].

– Choose a canonical ordering of the variables at the outset, and arrange the graph such that the variables occur in that order down any path. This idea of Bryant's [5], in conjunction with maximal structure sharing (i.e. never duplicating nodes in the graph) gives a completely canonical representation of a boolean function, so boolean equivalence can be tested simply by comparing graphs. Such structures are often referred to as 'reduced ordered binary decision diagrams' (ROBDDs), but we'll just use the shorter title of BDDs.

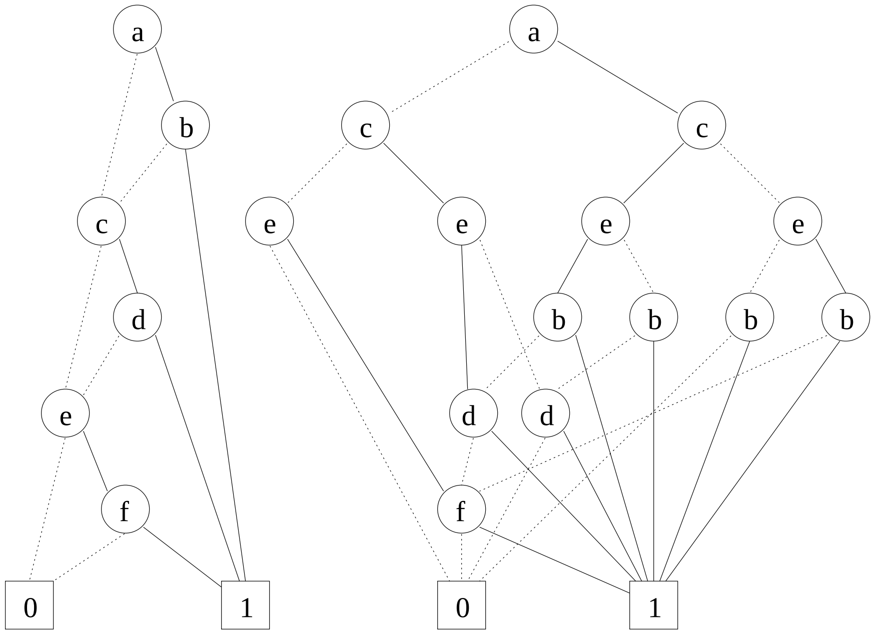


**Fig. 1.** The same function with different variable orderings

### Efficiency and variable orderings

Figure 1 shows two BDDs for the same function, but with different variable orderings. One question which naturally arises when presented with a Boolean function is how to choose the canonical variable ordering required for BDDs. As the above picture illustrates, the choice of variable ordering can make a difference to the size of the BDDs produced. On large examples the difference can be between practicality and impracticality. Considerable work has been done on variable ordering heuristics [7, 12], and although nothing especially conclusive has emerged, there are a number of techniques which usually give satisfactory results in practice. For example, in verifying an adder it seems best to take the following order: first the carry-in, then the two low-order bits of the inputs, then the next input bits and so on up to the two high-order input bits. Other circuits can be handled with a similar ordering based on the network topology.

On the other hand there are circuits for which no variable ordering will give better than exponential performance. In his original paper [5], Bryant proved that the multiplier is one such circuit, and so BDD-based techniques are not easy to apply to large multipliers. Of course it is not surprising that an algorithm for solving an NP-complete problem will perform badly in some cases. Perhaps it is *more* surprising that such cases seem to occur so rarely in practice.

### Complement Edges

Since the explosion of interest in BDDs, many extensions to the basic idea have been proposed. Though many of these are worthy of consideration, it was our intention to explore only the basic algorithm. Nevertheless the idea of *complement edges* as described by Brace et al. in [4] seemed worth incorporating in the implementation. The idea is simple: we allow each edge of the BDD graph to carry a tag, usually denoted by a small black circle in pictures. This denotes *the negation of* the subgraph it points to.

Complement edges have the great merit that complementing a BDD now takes constant time: one simply needs to flip the complement tag. In addition, greater sharing is achieved because a graph and its complement can be shared; only the edges pointing into it need differ. In particular we only need one terminal node, which we choose (arbitrarily) to be 1, with 0 represented by a complement edge into it. Notice that by a BDD we shall afterwards mean a pointer, complemented or otherwise, to a BDD node, which itself contains more such pointers.

Complement edges do create one small problem: without some extra constraints, canonicality is lost. This is illustrated in figure 2, where each of the four BDDs at the top is equivalent to the one below it. This ambiguity is resolved by ensuring that whenever we construct a BDD node, we transform between such equivalent pairs to ensure that the 'then' branch (i.e. that taken if the variable is 1) is uncomplemented. This choice is completely arbitrary.

### Constructing BDDs

Assuming we are given a HOL boolean expression, the question arises of how to construct the corresponding BDD. Any top-down strategies based on case splits over variables would immediately have exponential performance and vitiate any advantages of the eventual BDD representation. Instead, a bottom-up strategy is used, where we first construct BDDs for the arguments of a given operator (e.g. conjunction) and then merge these together. Following Bryant's terminology, we will refer to each such step as an 'APPLY' operation.

Let us represent BDD nodes using HOL-like syntax for conditional expressions; i.e. $v \to t1|t2$ means 'if $v$ then $t1$ else $t2$'. Suppose that we have two BDDs which we want to conjoin. If one node is 0 or 1 then we can easily reduce the node using a trivial propositional tautology, viz:

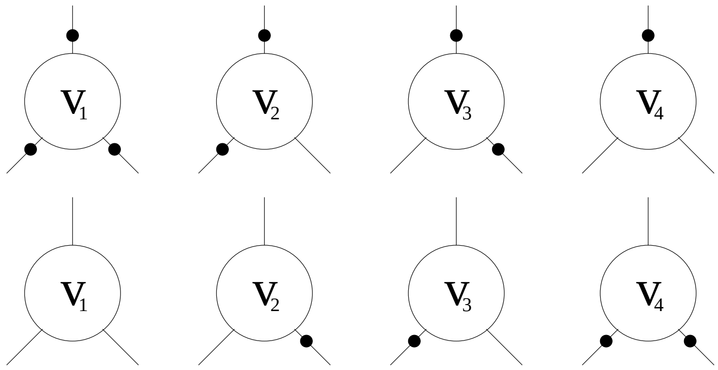


**Fig. 2.** Redundancy with complemented edges

$$\begin{aligned}
node_1 \wedge 0 &= 0 \\
node_1 \wedge 1 &= node_1 \\
0 \wedge node_2 &= 0 \\
1 \wedge node_2 &= node_2
\end{aligned}$$

If none of these special cases apply, then neither BDD is a terminal node. It may be that either or both is negated, but if so we can use the following transformation:

$$\neg(v \to t|e) = (v \to \neg t|\neg e)$$

Hence we may now assume that neither of the nodes is negated. The problem is now reduced to finding the BDD corresponding to:

$$(v_1 \to t_1|e_1) \wedge (v_2 \to t_2|e_2)$$

To do this, we just choose whichever variable comes first in the variable ordering. There are three possibilities. If $v_1$ and $v_2$ are identical, we apply the first rule below. If $v_1$ comes earlier in the canonical ordering we apply the second, and if $v_2$ comes earlier we apply the third.

$$\begin{aligned}
(v_1 \to t_1|e_1) \wedge (v_2 \to t_2|e_2) &= v_1 \to (t_1 \wedge t_2)|(e_1 \wedge e_2) \\
(v_1 \to t_1|e_1) \wedge (v_2 \to t_2|e_2) &= v_1 \to (t_1 \wedge (v_2 \to t_2|e_2))|(e_1 \wedge (v_2 \to t_2|e_2)) \\
(v_1 \to t_1|e_1) \wedge (v_2 \to t_2|e_2) &= v_2 \to ((v_1 \to t_1|e_1) \wedge t_2)|(v_1 \to t_1|e_1) \wedge e_2)
\end{aligned}$$

This now gives us a simple recursive procedure to produce the BDD for any conjunction: we apply whichever of the above three rules applies till we reach a terminal node on one side or the other, when one of the four terminal cases given above terminates the recursion. This of course will not necessarily give us a completely reduced BDD. In Bryant's original treatment [5] the 'APPLY' operation was followed by a separate 'REDUCE' phase. This is complicated and inefficient, and in more recent implementations the two operations are combined. The above recursion is applied, but a global hash table of all nodes is maintained. Before a new node is created, the function looks in the table to see if it already exists. If so, a pointer (complemented as necessary) is used to the existing node. Maximal sharing is necessarily achieved because no node is ever duplicated.

There is one further optimization that is usually made. A second global hash table is maintained of all previous 'APPLY' operations together with their results. Before calling the 'APPLY' function recursively, we first check whether the desired result has already been computed, and if so, we use that result (which will itself be a BDD of course). Note further that when making this lookup we can, as described by Brace et al. [4] exploit special properties of the operator to be applied. In the present case of conjunction, we can exploit symmetry by looking up whichever of $b_1 \wedge b_2$ and $b_2 \wedge b_1$ satisfies some canonicality property (based on any fixed but arbitrary ordering among BDDs, such as pointer comparison). This cache of precomputed results has a major impact on efficiency, since it eliminates the need for a whole series of recursions down to a terminal node. Nevertheless at the frontiers of BDD research it sometimes proves necessary to garbage collect this cache, otherwise memory fills up completely.

## 2 Implementation

BDDs have hitherto mostly been implemented in low-level languages, mainly C. This allows efficient manipulation of pointers, and we can put in various low-level optimizations. For example, Brace et al. suggest using the bottom bit of a pointer to indicate complement status, rather than having a separate field. (This assumes that pointers to BDD nodes will be even: word-alignment can be relied on in most architectures.) Since it is our hope to implement the routine as a derived rule, we have to face two quite separate problems:

1. ML is inevitably less efficient than C, if only because tricky low-level optimizations like the above pointer hack are impossible to duplicate. Some dialects (e.g. Classic ML) lack the imperative features such as arrays which are necessary to implement fast lookup via hashing, though some structures such as AVL trees [1] might provide an acceptably efficient substitute.
2. We need to actually construct a HOL proof. This means that ad-hoc term manipulations need to be replaced by primitive inference.

We should notice first that producing a HOL derived rule does *not* necessarily mean that all the proof-finding has to be in ML. In fact there have been several experiments with linking HOL to external programs which produce proofs or checkable answers (for an unusual example see the papers by Laurent Théry and the present author [9, 8]). It would be quite possible to have an orthodox C implementation of BDDs produce a HOL proof which can be imported into ML.

We have not actively explored this possibility here, largely because of implementation complexity, but it does have some merit. In particular, it was mentioned above that choice of variable ordering can have an enormous impact of the size of the BDDs produced. An efficient external oracle might explore several ordering heuristics and only commit itself to producing a formal proof when a satisfactory ordering is found.

### Three implementations

In order to clarify exactly where the efficiency problems arise, we have written three implementations:

1. An ANSI C implementation, using all optimizations which occurred to us, including the pointer trick mentioned above to represent complemented edges.
2. An implementation in Standard ML, not performing inference in the HOL sense.
3. An implementation in Standard ML producing a proof in `hol90`.

Considerable pains were taken to make the underlying algorithms in the three implementations as near identical as possible. This was mainly in order to allow meaningful comparisons, but as we shall explain later, it is important that the two Standard ML versions produce isomorphic BDD graphs.

### More details of the algorithm

Here we fill in the details in the above sketch of the algorithm. The algorithm is based on two hash tables (implemented as arrays of lists):

1. A 'unique table' containing all the nodes in the BDD graph (represented as triples: the variable and the two branches). Before an 'APPLY' operation creates a node, it checks whether the resulting node would be trivial, i.e. has both branches identical. If so then the following transformation can be applied:

   $$(v \to t|t) = t$$

   If the branch is not trivial, it is looked up in the table, and inserted if it is not already there. However first, it is necessary to ensure canonicality in the presence of complemented edges, so as explained above, if the 'then' branch is negated, the BDD is transformed like this:

   $$(v \to \neg t|e) = \neg(v \to t|\neg e)$$

   The node (which is negated in the above expression) is then entered in the unique table.

2. A ‘computed table’ containing a record of previous computations. We have only implemented ‘APPLY’ for a single operation, namely conjunction. Remembering that we have a basic negation operation, which is almost cost-free, we can easily implement the other logical operations using that, viz:

$$
\begin{aligned}
x \vee y &= \neg(\neg x \wedge \neg y) \\
x \Longrightarrow y &= \neg(x \wedge \neg y) \\
x \equiv y &= (x \Longrightarrow y) \wedge (y \Longrightarrow x) \\
x \rightarrow y|z &= (x \wedge y) \vee (x \wedge \neg z)
\end{aligned}
$$

The last two are less efficient, but seem to arise less often in practice. It’s more efficient to maintain a single ‘computed’ table for conjunctions, rather than effectively duplicate logically equivalent results for disjunction and implication. For a similar reason, the paper by Brace et al. takes the conditional (if-then-else) as basic, but using a ternary operator to implement binary ones seems to us rather wasteful. Equivalence and inequivalence can then be expressed directly via $x \rightarrow y|\neg y$, but the other connectives can be done equally efficiently in terms of conjunction and negation.

An ‘APPLY’ operation first checks for trivial cases, where one or other of the arguments is 0 or 1. These are dealt with as outlined above. Furthermore, cases where both arguments are identical, or one is the negation of the other, are also disposed of in an obvious way. Finally, based on an arbitrary ordering (pointer comparison in the C case, position in node table in the Standard ML case), the conjunction is oriented canonically, exploiting symmetry. Only when the arguments are placed in a canonical form does the ‘APPLY’ operation look to see if the result is in the computed table. If it is, then that result is used. Otherwise a recursion is performed as indicated above, and the eventual result is entered in the computed table.

### Efficient implementation of complement edges

As noted above, it is easy to implement complement edges efficiently in C (albeit strictly nonportably). Standard ML has references (pointers), but these cannot be hacked with such abandon. Representing a BDD by a composite datastructure, such as a pointer-boolean pair, seemed likely to be highly inefficient, since it means creating more `cons` cells and performing multiple dereferences. Instead we exploit Standard ML’s arrays to create our own bespoke pointers.

We wish in any case to have a global table of BDD nodes, indexed via hashing, and so we can store the nodes in a single array (in the present implementation this has a fixed size, which limits the total number of nodes, but this restriction could easily be lifted). Now the index in this array (simply an integer) can be used as a ‘pointer’ to the node, and to indicate negation we can simply numerically negate the number. (Of course we must avoid using element 0 of the array, as $-0 = 0$, but that is a small matter.) The terminal nodes are represented by numbers outside the array bounds. The slight extra cost of array indexing over pointer dereferencing seems well worthwhile, since we can manipulate BDDs in the form of integers very efficiently.

### Performing primitive inference

The implementation which performs primitive inference is essentially a conversion which takes a HOL propositional formula $E$ and produce a theorem:

$$\Gamma \vdash E = B$$

where $B$ is a representation of a BDD. Obviously, a crucial question is how a BDD is to be represented inside the logic. We cannot simply represent the BDD in HOL as a tree of conditionals, or rather we *could* but we would immediately abandon sharing inside the HOL term. (Unshared BDDs are possible, but less attractive.) There are a number of ways of achieving such sharing. Perhaps the most obvious is to use a `let` binding, or what amounts to the same thing, a beta-redex. For example

```
let v = B1 in B2[v,..,v,..,v]
```

implements a BDD `B2` which has three 'pointers' to a shared BDD `B1`. Experiments along these lines have been done, but the terms become rather complex because terms are shared among various 'levels' in the hierarchy of `let` bindings. Instead we adopted the following scheme. For all BDD nodes entered in the hash table, we invent stylized node names $n_1, n_2, n_3, \ldots$. Each such name will stand for a BDD node, and we have for each one a corresponding 'definition' term:

$$n_i = v \to b_j \mid b_k$$

where $b_j$ and $b_k$ are (possibly negated) other node variables or the special terminal node 1 which is of course represented by `T` in HOL.

The most obvious plan is then to carry all the relevant node definitions around as hypotheses in the theorem produced. Then, each 'APPLY' step simply needs to substitute the definition and do some very simple inference. However this proves unattractive because the hypotheses are stored as an unsorted list. It soon becomes hopelessly impractical to perform the implicit union of lists which occurs each time an 'APPLY' operation brings together subproofs which used different sets of hypotheses.

There is a simple way in which to avoid this inefficiency. Rather than having a list of assumptions, we can collect all the definitions into a single conjunctive term: all theorems carry just this one hypothesis. We can once and for all split up the conjuncts and put the 'definitional' theorems in an array for easy access; this only takes time $O(n)$ where $n$ is the number of nodes in the BDD. Now all the algorithm can be implemented efficiently exactly as in the version without inference. Inference is necessary at every 'APPLY' operation, but only simple propositional reasoning. Hence the main algorithm can now be implemented with only a constant factor slowdown (albeit a significant constant factor) over an implementation which performs no inference.

### How to get started

There is an obvious problem with the above scheme. We want to set up a big conjunction of all the necessary definitions at the beginning, and this entails knowing *in advance of running the algorithm* what all those definitions should be. Consequently a 2-pass implementation is required. First, the Standard ML version which does not do primitive inference is run; as it runs, it records all the nodes in its own node table. Then these are translated into theorems for the inference version, which can be run afterwards. Here it is crucial that the underlying steps of the two implementations are identical, or the nodes required might differ.

Although this 2-pass organization is an unwelcome increase in the complexity of the implementation, all the necessary code is at hand, and it proved quite easy to write the necessary interface. There is some advantage in that failures will tend to be trapped early by the first pass.

### How to finish

A less obvious and more serious problem is how to get rid of all the BDD node definitions when the algorithm is finished. Of course, we are considering here cases where the given expression has been proved to be a tautology, i.e. the theorem produced by the BDD conversion is of the form:

$$\Gamma \vdash E = T$$

Logically, there is no difficulty in eliminating the assumptions. Recall that the BDD graph is acyclic. This means that among any set of nodes there will be a topmost one, which is not used in any other definition (and of course none of the node variables appear in the conclusion of the theorem). In fact we can always choose the latest node to be created, because of the bottom-up way in which the BDD was constructed. Hence by arranging the conjunction in order (we put the newest node as the leftmost conjunct in fact) we can always peel off the conjuncts in sequence.

Now consider how to eliminate such a topmost definition. It is a simple matter logically to discharge it and leave the remaining conjuncts as the new assumption:

$$\Delta \vdash (n_i = v \to b_i | b_j) \Longrightarrow (E = T)$$

Since we have assumed that $n_i$ is not free in $\Delta$, we may instantiate $n_i$ to $v \to b_i | b_j$ in this theorem. Since $n_i$ is also not free in the initial expression $E$, this yields:

$$\Delta \vdash (v \to b_i | b_j = v \to b_i | b_j) \Longrightarrow (E = T)$$

Now from reflexivity:

$$\vdash v \to b_i | b_j = v \to b_i | b_j$$

and Modus Ponens gives:

$$\Delta \vdash E = T$$

This process can be iterated, and so all the assumptions can be eliminated, leaving the bare theorem $\vdash E = T$ as required.

The above process obviously takes $O(n)$ primitive inferences, where $n$ is the number of nodes. However a quadratic complexity in $n$ lurks beneath the surface, since each instantiation must *check* that $n_i$ is not free in the hypotheses. And the single hypothesis has size $O(n)$, all of which must be traversed! Consequently, this final innocuous cleanup has turned out to be the rate-determining step of the procedure. This is dramatically reflected in the results given below.

## 3 A logical application

Fast tautology checking is most useful for verification of circuits at a low level. In general theorem proving, tautology-checking applications are usually sufficiently simple that a naive case analysis algorithm is fast enough to prove them in an acceptable time. Nevertheless there are a few situations where fast tautology checking can be useful in a more abstract domain, and we address one such situation here by describing a decision procedure for a certain class of logical formulas.

We will consider first-order formulae involving no function symbols (but relation-symbols of arbitrary arities are allowed) and no equality symbol, with all universal quantifiers preceding existential ones, i.e.

$$\forall x_1 \ldots x_m.\ \exists y_1 \ldots y_n.\ \ P[x_1, \ldots, x_m, y_1, \ldots, y_n]$$

where $P[x_1, \ldots, x_m, y_1, \ldots, y_n]$ is quantifier-free. We will show how a decision procedure originally described by Bernays and Schönfinkel [3] may be implemented reasonably efficiently and applied to simple inequality reasoning. Our exposition will follow the nice discussion given by Hilbert and Ackermann [10].

### Validity and satisfiability in finite domains

We will call a first-order formula $\phi$ *valid*, and write $\models \phi$, if it is true in all interpretations (in the usual sense). Now for any positive integer $n$, we shall call it *n-valid*, and write $\models_n \phi$, if it is valid in all interpretations with a domain size $n$ (the domain being the set over which the quantifiers range). Now take a first-order formula $\phi$ conforming to the above restrictions:

$$\forall x_1 \ldots x_m.\ \exists y_1 \ldots y_n.\ \ P[x_1, \ldots, x_m, y_1, \ldots, y_n]$$

and consider the disjunction:

$$P[x_1, \ldots, x_m, x_{p_1}, \ldots, x_{p_n}] \vee \ldots \vee P[x_1, \ldots, x_m, x_{q_1}, \ldots, x_{q_n}]$$

where $x_{p_1}, \ldots, x_{p_n}$ run through all the possible choices of $n$ elements from the set $x_1, \ldots, x_m$. Let us denote this disjunction by $Q[x_1, \ldots, x_m]$. We claim that the following four assertions are equivalent:

$$\models \ \forall x_1 \ldots x_m. \ \exists y_1 \ldots y_n. \ \ P[x_1, \ldots, x_m, y_1, \ldots, y_n] \tag{1}$$

$$\models_n \ \forall x_1 \ldots x_m. \ \exists y_1 \ldots y_n. \ \ P[x_1, \ldots, x_m, y_1, \ldots, y_n] \tag{2}$$

$$\models_n \ \forall x_1 \ldots x_m. \ Q[x_1, \ldots, x_m] \tag{3}$$

$$\models \ \forall x_1 \ldots x_m. \ Q[x_1, \ldots, x_m] \tag{4}$$

Obviously 1 implies 2 (it is a more general statement). Furthermore 4 implies 1, because one of the sets of terms in the disjunction provides a witness for the existential quantifier. It is a fact that 3 implies 4 for any quantifier-free statement $Q[x_1, \ldots, x_m]$ at all, not just of the special form we have arrived at. To see this, suppose 4 is false. Then we can find an interpretation and elements $a_1, \ldots, a_m$ such that

$$\models \neg Q[a_1, \ldots, a_m]$$

But then by restricting the interpretation to $\{a_1, \ldots, a_m\}$ it would follow that the original statement is not $m$-valid. (If the $a_1, \ldots, a_n$ are not all distinct, we can add new elements and identify them with $a_1$ as far as the action of the interpretation is concerned. Using this method we can always expand a domain while retaining a formula's validity — this depends of course on the fact that we do not have the special interpretation of the equality symbol.) Using this fact, it is similarly easy to show that 2 implies 3.

This means that to prove a theorem of the form 1, we need only prove one of the simpler form 4. Since we can choose the interpretation of predicates without restriction, we can assign all instances of the predicates independently. Consequently a statement of the form 4 is true iff its body is a substitution instance of a propositional tautology. Thus the whole problem is reduced to tautology checking; in general the tautology will be quite big, since there will be $m^n$ disjuncts in our reduced form.

Apart from providing a ready source of tautologies to test, it is worth asking whether this sort of decision procedure is useful in everyday proofs. Certainly the form is quite restricted, but we believe it may be useful for simple forms of inequality reasoning, which arise quite often. For example, if we have an implication with universally quantified antecedent and consequent, it can be transformed into the above form. The basic properties of a partial or total order are all universal, and so this fits into the framework. Although we are not allowed a specially interpreted equality predicate, it is easy enough to simulate it by asserting some predicate's reflexivity, symmetry, transitivity and congruence properties, all universal sentences.

Note however that because of the lack of function symbols, it is not possible to decide algebraic theories such as group theory (in fact [13] no decision procedure exists for group theory). Certainly the axioms can all be rendered in relational form, i.e. $m(x, y, z)$ for $x.y = z$ etc., but the fact that the corresponding function is total:

$$\forall x\ y.\ \exists z.\ m(x, y, z)$$

is not a universal statement and so when negated (as it effectively will be, occurring as the antecedent of an implication) it does not fall within the compass of our procedure.

## 4 Results

In this section we present some results on tautology-checking examples. One of these examples was derived by applying the above logical decision procedure to a simple fact of inequality reasoning. This is the deduction of

$$\forall a\ b\ c.\ a \leq b \wedge c \not\leq b \Longrightarrow a < c$$

from the set of 'axioms' for a total order (much larger examples could be done; the axioms are the dominant part of this term). The other four were taken from a public-domain suite of hardware verification examples produced by Diederik Verkest and others at IMEC in Belgium [14]. These circuits are as follows:

| Name | Explanation | Number of inputs |
|---|---|---|
| add1.be | 4-bit adder-subtractor | 9 |
| add2.be | 4-bit ALU with 5 control signals | 13 |
| add3.be | 8-bit ALU with 5 control signals | 21 |
| add4.be | 12-bit ALU with 5 control signals | 29 |

The fields of the table are as follows:

- Nodes: The number of nodes in the BDD generated (all algorithms give the same result, which is an independent check on their similarity).
- C: Time for the C implementation. The C version was compiled with the optimizer switched off.
- ML (initial): Time for the first ML pass which does no inference. Although it is only used to record the nodes for the inference pass, this pass does almost exactly the same as the C program, and can meaningfully be compared.
- ML (main): Time for the second ML pass doing inference to produce a tautology proof still decorated with assumptions.
- ML (cleanup): Time for the final cleanup phase, removing all the assumptions from the BDD theorem.

All timings are in CPU seconds on a Sparc 10. There were 1000 hash chains in the unique and computed tables; for the later examples this might profitably have been increased, but was kept fixed throughout.

| Example | Nodes | C | ML (initial) | ML (main) | ML (cleanup) |
|---|---|---|---|---|---|
| add1.be | 426 | 0.01 | 0.23 | 5.22 | 18.12 |
| Inequality | 1463 | 0.04 | 0.44 | 22.83 | 165.62 |
| add2.be | 1785 | 0.06 | 0.63 | 24.20 | 481.38 |
| add3.be | 5472 | 0.40 | 3.47 | 98.49 | 7425.85 |
| add4.be | 11252 | 0.80 | 8.71 | 354.84 | (unable?) |

**Ideas for improving efficiency**

The inference version was by no means implemented in the most efficient way possible. For example, all the lookup in the computed table was replicated, rather than recording results from the first pass. There is considerable potential for optimization.

However the most important object of attack is the final cleanup phase. As remarked above, there is a subtle quadratic dependency lurking here, and results manifest it in an unpleasant way.

There are plenty of ways in which this phase could be done more efficiently. The following idea has been started, but not finished, at time of writing. The difficulty is justifying the consistency of the set of node definitions, which in general might make incompatible recursive assignments. It is only because we have stuck to a variable ordering that we know the result to be admissible. Thus we have tried to perform a proof of this by generating a proforma theorem asserting that a list of definitions of a restricted form is admissible.

First, instead of using single variables for the node names, we use a function applied to a numeric argument; what we have written as $n_i$ actually becomes $B(i)$ in HOL. Now we prove a proforma theorem by list recursion asserting that a list of such definitions may be satisfied subject to conditions. The list is a set of tuples which code the right hand sides of the definitions. The interpretation is that the $i$th element of the list defines $B(i)$, so automatically assuring us that no node is defined twice. Now the list is admissible provided that the tuple which is the $i$th element of the list has both its numeric fields less than $i$. This can be expressed as a simple list recursion, and to justify a given definition requires $O(n)$ steps.

Of course at each step we need to do some arithmetic reasoning on numerals, but that can be done in $O(log(n))$ time by representing the numbers as lists as in Tim Leonard's numeral library. The overall complexity is bounded by $O(n\ log(n))$, and since in practice $n$ will certainly be bounded by the memory capacity of the machine, we effectively have an $O(n)$ bound over all feasible ranges. Consequently it should be possible to fix the problem of inefficiency in the cleanup phase.

## 5 Conclusion

We have shown that it is possible to implement BDDs as a HOL derived rule, which, except for the final phase, is tolerably fast. The slowness of this final cleanup phase is disappointing, but we believe that an approach like that outlined in the last section will alleviate it. In any case, the algorithms described here are much more efficient on large examples than naive tautology-checking algorithms based on repeated case splits. Almost certainly, `add3.be` is the largest tautology ever proved in HOL.

The derived rule can be used not only as a 'one-shot' tautology checker but in order to implement boolean expressions directly as BDDs and manipulate them efficiently in that form: this is what is done in mainstream uses of BDDs. It would be worth programming other operations on BDDs efficiently.

It is clear that a carefully optimized C implementation will always be much more efficient, and from a pragmatic point of view there is considerable merit in hardwiring BDDs in as a fundamental principle, as in Carl Seger's Voss system. Nevertheless we believe that implementing BDDs as a HOL derived rule is not a hopeless idea. For certain applications a constant factor of even several orders of magnitude in performance may be outweighed by the increased security of having a precise proof as a logician understands the word.

## Acknowledgements

I would like to thank Mark Linderman, of Cornell University, who explained BDDs to me, as well as everyone else who made my stay there so enjoyable and stimulating. I am also very grateful to Catia Angelo for sending me the IMEC benchmarks, and to my supervisor Mike Gordon for his constant support and encouragement. Thanks are also due to the Natural Sciences and Engineering Research Council and the Isaac Newton Trust for financial support.

## References

1. G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1262, 1962.
2. S. B. Akers. Binary decision diagrams. *ACM Transactions on Computing*, C-27:509–516, 1978.
3. P. Bernays and M. Schönfinkel. Zum entscheidungsproblem der mathematischen logik. *Mathematische Annalen*, 99:401–419, 1928.
4. Karl S. Brace, Richard L. Rudell, and Randall E. Bryant. Efficient implementation of a bdd package. In *Proceedings of 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.
5. Randall E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
6. S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on the Theory of Computing*, pages 151–158, 1971.
7. M. Fujita, H. Fujisawa, and N. Kawato. Evaluations and improvements of a boolean comparison program based on binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 2–5. IEEE, 1988.

8. John Harrison and Laurent Théry. Extending the hol theorem prover with a computer algebra system to reason about the reals. In Jeffrey J. Joyce and Carl Seger, editors, *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG '93*, volume 780 of *Lecture Notes in Computer Science*, pages 174–184, UBC, Vancouver, Canada, 1993. Springer-Verlag.
9. John Harrison and Laurent Théry. Reasoning about the reals: the marriage of hol and maple. In Andrei Voronkov, editor, *Logic programming and automated reasoning: proceedings of the 4th international conference, LPAR '93*, volume 698 of *Lecture Notes in Computer Science*, pages 351–353, Hotel Olgino, St. Petersburg, Russia, 1993. Springer-Verlag.
10. D. Hilbert and W. Ackermann. *Principles of Mathematical Logic*. Chelsea, 1950.
11. C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.
12. S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proceedings of the International Conference on Computer-Aided Design*, pages 6–9. IEEE, 1988.
13. A. Tarski. *Undecidable Theories*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1953.
14. Diederik Verkest and Luc Claesen. Special benchmark session on tautology checking. In *Formal VLSI Correctness Verification*, pages 81–82. Elsevier Science Publishers, 1990.

This article was processed using the LaTeX macro package with LLNCS style