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

Binary Decision Diagrams (BDDs) are a representation for Boolean formulas which makes many operations, in particular tautology-checking, surprisingly efficient in important practical cases. In contrast to such custom decision procedures, the HOL theorem prover expands all proofs out to a sequence of extremely simple primitive inferences. In this paper we describe how the BDD algorithm may be adapted to comply with such strictures, helping us to understand the strengths and limitations of the HOL approach.

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. However, the problem of tautology checking is co-NP complete; in fact the complementary operation of testing Boolean satisfiability was the original NP-complete problem given by Cook (1971). This means that all known algorithms have exponential worst-case performance.

Despite these discouraging theoretical considerations, there are algorithms which give surprisingly good results in practice, remaining tractable for tautologies involving tens or even hundreds of variables. This makes them very useful in practical situations, most obviously in the verification of digital hardware, e.g. proving that a hand-optimized circuit is equivalent to a naive one produced by automated means. Algorithms worth serious consideration range from the venerable procedure introduced by Davis and Putnam (1960) to the patented algorithm of Stålmarck (1994). However the one which has had the most dramatic impact so far is the method of binary decision diagrams.

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 sub-functions 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 often lead 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 (1959) and Akers (1978).
- Choose a canonical ordering of the variables at the outset, and arrange the graph such that the variables occur in that order down any branch, as proposed by Bryant (1986).

A canonical variable ordering, 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.

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

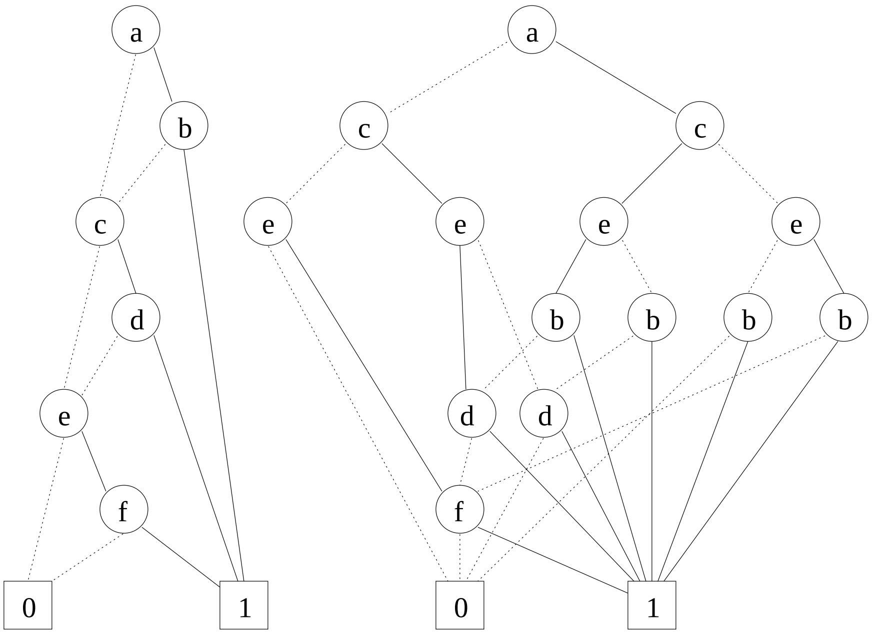

FIGURE 1. The same function with different variable orderings

heuristics; see for example the paper by Butler, Ross, Kapur, and Mercer (1991). Most interesting BDD applications are based on digital circuits, and considering the network topology often allows one to choose a good variable ordering.

On the other hand there are circuits for which no variable ordering will give better than exponential performance. In his original paper, Bryant (1986) 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, Rudell, and Bryant (1990) 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 tag. Furthermore, 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 may contain 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 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. $v \rightarrow t1|t2$ 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:

$$node_1 \wedge 0 \quad = \quad 0$$

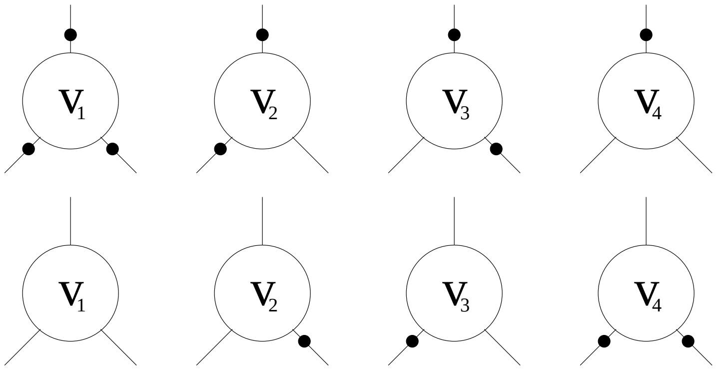

FIGURE 2. Redundancy with complemented edges

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

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:

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

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 pass to:

$$v_1 \to (t_1 \land t_2)|(e_1 \land e_2)$$

If v_1 comes earlier in the canonical ordering we pass to:

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

and if v_2 comes earlier we pass to:

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

This now gives 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 the original treatment by Bryant (1986), 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 (the 'unique 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 (the 'computed table') is maintained, storing 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, Rudell, and Bryant (1990) exploit special properties of the operator to be applied. In the present case of conjunction, we can exploit symmetry by looking up whichever of $b_1 \land b_2$ and $b_2 \land b_1$ 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. HOL

The HOL theorem prover, described by Gordon and Melham (1993), is descended from Edinburgh LCF, described by Gordon, Milner, and Wadsworth (1979), via Cambridge LCF described by Paulson (1987). It proves theorems in a form of Higher Order Logic, hence the name HOL. LCF-style provers are simply programs written in a version of the ML language, where theorems are objects of an abstract type `thm`, and inference rules are implemented as functions which return objects of that type. The only constructors for objects of type `thm` are a few very simple primitive inference rules, e.g. *Modus Ponens* (`MP`):

$$\frac{\Gamma \vdash \phi \quad \Delta \vdash \phi \Longrightarrow \psi}{\Gamma \cup \Delta \vdash \psi}$$

Assumption (`ASSUME`):

$$\overline{\{\phi\} \vdash \phi}$$

and *Discharging assumptions* (DISCH):

$$\frac{\Gamma \vdash \psi}{\Gamma - \{\phi\} \vdash \phi \Longrightarrow \psi}$$

Proof at this level is tedious, but by ML programming, users can build up a repertoire of higher-level *derived rules* which ultimately decompose to these primitives. For example, the following derived rule adds an assumption tm to a theorem th:

```
let ADD_ASSUM tm th =
  MP (DISCH tm th) (ASSUME tm);;
```

This isn't very interesting, but many useful derived rules are included in the HOL system, including rewriting, backward proofs, inductive definitions and free recursive data types. Thus, the user gets the benefits of security and simplicity from the low-level primitives, while at the same time being able to perform proofs at a more palatable level; it is even possible to write customized derived rules should this prove necessary in unusual applications.

There is an apparent disadvantage with this policy, which leads many to dismiss the LCF approach as impractical. Every rule, however sophisticated, must ultimately invoke many primitive inferences, and this is inefficient. However that has not, in the HOL system, turned out to be a major limitation in practice. For a detailed analysis, we refer the reader to Boulton (1993) and Harrison (1995), and content ourselves with two remarks:

1. Patterns of inference leading from E to E' can often (particularly given the flexibility of higher order logic) be encoded as theorems $\vdash E \Longrightarrow E'$. These only need to be proved once, and may then form the basis of a derived rule, with only a modest amount of instantiation followed by a modus ponens step necessary each time.
2. In many theorem proving procedures, *search* for a proof dominates. For example, resolution provers may search through thousands or even millions of clauses for a refutation, but once it is found, the proof discovered is relatively short. The search need not be handicapped by performing primitive inferences, or even performed in HOL itself for that matter; see for example Harrison and Théry (1993).

Despite this, it is not evident that every special proof procedure can be implemented in such a way. In fact the BDD algorithm, involving as it does imperative features and huge dynamic datastructures, seems difficult to shoehorn into the HOL framework. As such, trying to do just that makes an interesting case study in how far the LCF approach can be stretched.

3. 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, because for example tricky low-level optimizations like the above pointer hack are impossible to duplicate, and all array accesses are subject to bounds checks. 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 the AVL trees invented by Adel'son-Vel'skii and Landis (1962) 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.

As noted already, a HOL derived rule does *not* necessarily have to do all its own proof-finding. It would be quite possible to arrange an orthodox C implementation of BDDs to produce a trace which could be imported as a HOL proof. 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 was 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, to allow meaningful comparisons.

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. have both branches identical. If so then the following transformation is applied, and a node already exists for the result.

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

   Next, 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:

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

   Now the node is looked up in the unique table, and inserted if it is not already there.
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:

   $$\begin{array}{rcl} x \vee y & = & \neg(\neg x \wedge \neg y) \\ x \Longrightarrow y & = & \neg(x \wedge \neg y) \\ x \equiv y & = & (x \Longrightarrow y) \wedge (y \Longrightarrow x) \\ x \to y|z & = & (x \wedge y) \vee (\neg x \wedge z) \end{array}$$

   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 $x \to y|\neg y$, 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 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 version of the algorithm which performs primitive inference is intended to convert a conventional HOL Boolean expression E into a theorem $\vdash E = b$ where b represents a BDD in the manner chosen. In HOL jargon, this is a *conversion*. But we need to decide just what the HOL representation of BDD nodes is to be. 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 adopt the following scheme. For all BDD nodes entered in the 'unique' 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. One solution to this problem is to use an explicit 2-pass organization; given a Boolean expression, the algorithm can make one pass over it, deciding which definitions will be needed, and then can make a second pass performing inference. The defect of this policy is inflexibility; we might want to manipulate BDDs in various ways, rather than just translate a single expression once and for all.

Instead, we adopt a 'lazy' approach, in the style of Boulton (1993). A BDD contains concrete data about the node, and also a function closure which may be called in appropriate circumstances to generate a theorem. As operations are performed on BDDs, the computed table is filled with theorems giving the results of conjunction operations, but these theorems have assumptions which need to be discharged against previous theorems in the same table. Meanwhile, the BDD function closures will pick out the appropriate element of the computed table to give the result. The process of realizing a BDD as a HOL theorem consists of three parts:

1. Collect together all the node definitions, form a theorem with a conjunctive assumption and split it up into its conjuncts.
2. Run in sequence through the computed table, eliminating the assumptions from theorems by using the appropriate node definitions and previous entries in the computed table.
3. Evaluate the function closure, giving it a pair consisting of the new unique and computed tables as argument.

There is a choice to be made of how much of the computation to perform 'live', and how much to save till the evaluation of the function closure. We choose to do as much inference as possible 'live'. The reason is psychological; while the user is interactively working with BDDs, the computation naturally gets spread out into manageable pieces, so there is never a long delay. There are arguments for the opposite position too; in particular the user may decide that a given BDD is no longer of interest and need not be realized (e.g. something turns out not to be a tautology as expected).

In order to avoid recomputation of function closures if they are evaluated at several points in another expression, we use special abstraction and evaluation functions, following Boulton (1993). A 'lazy theorem' is an element of the following Standard ML datatype:

```
datatype lazythm =
   Realized of thm
 | Unrealized of thm array * thm array -> thm;
```

Now a function closure is tagged as 'unrealized' using the following function:

```
fun wrap f = ref (Unrealized f);
```

Evaluation of a realized theorem is easy: just read it. After evaluating an unrealized theorem, we store it as a realized one, ready for next time:

```
fun eval(r,e) =
  case !r of
    Unrealized f =>
      let val th = f(e)
       in (r := Realized th; th)
      end
  | Realized th => th;
```

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 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 APPLY recursions happen. Hence by arranging the conjunction in order we can always peel off the conjuncts in sequence.

Now consider how to eliminate such a 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 Δ, 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 $O(n^2)$ rate-determining step of the procedure. The largest example below proved impossible to do in a reasonable time using this method.

This problem can be compared to the insistence of some programming languages (like Standard ML!) on performing bounds checks on array references even when the programmer knows they are unnecessary because of the logic of the program.

There is an alternative, which is a good example of how theorems can be used to justify awkward inference patterns. The assumption that we want to eliminate consists of a conjunction of equations; we can think of this as a set of recursion equations for the node variables. Now in general, these recursion equations might be unsatisfiable, e.g:

$$(n_1 = v \to n_2 | \neg n_2) \land (n_2 = v \to \neg n1 | n1)$$

or

$$(n_1 = v \to n_2 | \neg n_2) \land (n_1 = v \to \neg n_2 | n_2)$$

In our case we know that the first situation cannot arise because the canonical ordering we have chosen ensures that the graph is acyclic. And the second cannot arise because we do not define any nodes twice. The task now is to encode these facts into a theorem asserting that a suitably restricted set of recursion equations is satisfiable.

First, instead of using single variables for the node names, we use a function B applied to a pair of numeric arguments; the first argument corresponds to the node number, and the second is the index number of the variable involved in its definition. Thus each definition is of the form:

$$B(n, i) = v \to b | b'$$

where the variable v has index i in the canonical variable ordering. The crucial fact is that terms $B(m, j)$ appearing (possibly negated) as b or b' have a lower index, i.e. $j < i$ (for convenience, we regard the ordering of the variables the other way round). With a few constant definitions, we can state this as a proforma theorem.

```
(DEF f n [] = T) /\
(DEF f n (CONS h t) =
    UNIQUE f n h /\ DEF f (SUC n) t)
```

where `UNIQUE` is defined as follows:

```
UNIQUE f n (i,pj,qk,v,b1,b2) =
  (f(n,i) = (v => (b1 => ~f(pj) | f(pj))
                | (b2 => ~f(qk) | f(qk))))
```

That is, in `DEF f n l`, each element of the list is a tuple which indicates, in a suitable stylized way, how the appropriate node is defined. The recursion equation for `DEF` increments `n` at each conjunct, so we are guaranteed that no node is defined twice. However we still need to ensure wellfoundedness, so we define:

```
(OK [] = T) /\
(OK (CONS (i,(p,j),(q,k),v,b1,b2) l) =
   j < i /\ k < i /\ OK l)
```

This asserts that the definitions all obey our strictures. Now it is a straightforward matter, using the theorems about wellfounded recursion already proved in HOL, the following theorem (the extra conjunct in the consequent allows us to use `B(0,0)` in place of `T` and hence use the terminal node in the same stylized way).

```
|- !l. OK(l) ==> ?f. (f(0,0) = T) /\ DEF f 1 l
```

We are almost home now; we simply need to fold the recursion equations down to the canonical form by repeated rewriting with `DEF` and `UNIQUE`, then appeal to this theorem, and finally, prove all the inequalities. The inequalities are proved acceptably fast by existing facilities, though if there were a lot of variables, it might be worth cacheing some useful inequalities.

However our problems are not over! In general, some unfamiliar problems arise when one starts manipulating very large terms. To give an example, specializing the theorem $\forall x\ y.\ P[x, y]$ with a large term t is very slow, because a complete traversal of t is necessary to ensure that it contains no instances of y to be captured. Generally speaking, anything which involves doing full term traversals is likely to be troublesome when large terms are involved. In our case, such a traversal is hidden inside a call to the transitivity rule `TRANS`:

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t' = v}{\Gamma \cup \Delta \vdash s = v}$$

Here t and t' need not be equal, merely alpha-equivalent. An equality test would, in the cases we are concerned with, rapidly 'short circuit' down to a pointer comparison, since the terms are built up from the same components, except for the top few layers of the term tree. But this is not the case for alpha-equivalence; the test has no shortcuts and always provokes a full structural comparison (merely ignoring the names of bound variables in the de Bruijn implementation used in `hol90`). Unfortunately in our case the terms have size $O(n)$ and so testing for α-equivalence also takes time $O(n)$ for each unfolding of the recursion equation for `DEF`, i.e. $O(n^2)$ overall. For the purposes of these tests, we modified the primitive rules `TRANS` and `EQ_MP`, which led to a dramatic improvement in this phase.

4. RESULTS

In this section we present some results on tautology-checking examples. These are taken from a public-domain suite of hardware verification examples produced by Verkest and Claesen (1990) at IMEC in Belgium. See Table 1 for an explanation of these circuits. Table 2 shows, for each circuit:

- Nodes: The number of nodes in the unique table.
- C: Time for the C implementation. The C version was compiled with the optimizer switched off. The runtimes are in any case so small that they are difficult to measure accurately.
- ML: Time for a Standard ML implementation which does not perform inference, but is a close analogue of the C implementation.
- Inf(1a): Time for the ML version performing inference to perform its initial work, creating function closures.
- Inf(1b): Time for the ML version performing inference to realize the BDD as a theorem with no hypotheses.
- Inf(1): Overall time for ML version performing inference.
- Inf(2): Overall time for ML version performing inference without modification of primitive rules.

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.

5. CONCLUSION

We have shown that it is possible to implement BDDs as a HOL derived rule, and that inference causes a slowdown of something like a factor of 50 over a direct Standard ML implementation.

On the negative side, this is a very substantial slowdown, even more spectacular in comparison with a C implementation. The HOL rule is likely to be inadequate for the problems close to the limits of conventional BDD implementations. Furthermore, even to get this performance necessitated changing HOL's primitive rules slightly.

On the positive side, we see that BDDs, chosen because of the apparent difficulty in implementing them as a derived rule, can be forced into the HOL framework with only a constant factor slowdown. Furthermore, 'stress-testing' the LCF approach is a good way of discovering hidden problems, and has led us to interesting conclusions about the nature of the primitive rules we should have.

As far as practicality goes, the BDD tool is certainly superior to any existing tautology-checkers implemented in HOL, and sticks to the LCF approach, producing a proof as a logician understands the term. It could be of real use in automatic verification of circuits which, while not at the frontiers of research, are quite substantial, and possibly as a submodule of other logical decision procedures.

If something better is required, the LCF approach needs to be modified. One possibility, explored by Joyce and Seger (1993a), is simply to make BDDs a new primitive rule; though contrary to the LCF philosophy of making the primitive rules simple, this may be a good practical choice. Alternatively one could try to *verify* an implementation of the BDD algorithm and thus justify adding it to HOL via a so-called *reflection principle*. For example, Moore (1994) gives an implementation of BDDs which is entirely applicative (though it depends on hashing) and suggests that its verification may be tractable. The issues involved in using reflection principles are further explored by Harrison (1995).

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 Engineering and Physical

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

TABLE 1. Summary of the test circuits

Example	Nodes	C	ML	Inf(1a)	Inf(1b)	Inf(1)	Inf(2)
add1.be	425	0.01	0.23	4.48	7.84	12.32	24.94
add2.be	1784	0.06	0.63	21.63	32.90	54.53	295.48
add3.be	5471	0.40	3.47	68.66	106.01	174.67	2572.52
add4.be	11251	0.80	8.71	141.52	231.88	373.40	10074.00

TABLE 2. Results of BDD implementations

Sciences Research Council and the Isaac Newton Trust for financial support.

REFERENCES

Adel'son-Vel'skii, G. M. and Landis, E. M. (1962) An algorithm for the organization of information. *Soviet Mathematics Doklady*, **3**, 1259–1262.

Akers, S. B. (1978) Binary decision diagrams. *ACM Transactions on Computers*, **C-27**, 509–516.

Boulton, R. J. (1993) Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK. Author's PhD thesis.

Brace, K. S., Rudell, R. L., and Bryant, R. E. (1990) Efficient implementation of a BDD package. In *Proceedings of 27th ACM/IEEE Design Automation Conference*, pp. 40–45.

Bryant, R. E. (1986) Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, **C-35**, 677–691.

Butler, K. M., Ross, D. E., Kapur, R., and Mercer, M. R. (1991) Heuristics to compute variable orderings for efficient manipulation of Ordered Binary Decision Diagrams. In *Proceedings of 28th ACM/IEEE Design Automation Conference*, pp. 417–420.

Cook, S. A. (1971) The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on the Theory of Computing*, pp. 151–158.

Davis, M. and Putnam, H. (1960) A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, **7**, 201–215.

Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979) *Edinburgh LCF: A Mechanized Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Harrison, J. (1995) Metatheory and reflection in theorem proving: A survey and critique. Technical report, SRI Cambridge, Millers Yard, Cambridge, UK. To appear.

Harrison, J. and Théry, L. (1993) Extending the HOL theorem prover with a computer algebra system to reason about the reals. See Joyce and Seger (1993b), pp. 174–184.

Joyce, J. and Seger, C. (1993a) The HOL-Voss system: Model-checking inside a general-purpose theorem-prover. See Joyce and Seger (1993b), pp. 185–198.

Joyce, J. J. and Seger, C. (eds.) (1993b) *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, Volume 780 of *Lecture Notes in Computer Science*, UBC, Vancouver, Canada. Springer-Verlag.

Lee, C. Y. (1959) Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, **38**, 985–999.

Moore, J (1994) Introduction to the OBDD algorithm for the ATP community. *Journal of Automated Reasoning*, **12**, 33–45.

Paulson, L. C. (1987) *Logic and computation: interactive proof with Cambridge LCF*. Number 2 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Stålmarck, G. (1994) System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula. United States Patent number 5,276,897.

Verkest, D. and Claesen, L. (1990) Special benchmark session on tautology checking. In *Formal VLSI Correctness Verification*, pp. 81–82. Elsevier Science Publishers.