Representing Boolean Functions with If-Then-Else DAGs

Kevin Karplus*

UCSC-CRL-88-28
30 November 1988

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

This article describes the use of binary decision diagrams (BDDs) and if-then-else DAGs for representing and manipulating Boolean functions.

Two-cuts are defined for binary decision diagrams, and a relationship is exhibited between general if-then-else expressions and the two-cuts of a BDD for the same function. An algorithm for computing all two-cuts of a BDD in $O(n^2)$ time is given.

A new canonical form for if-then-else DAGs, analogous to Bryant's canonical form for BDDs, is introduced. The canonical form is based on representing the lowest non-trivial two-cut in the corresponding BDD, while Bryant's canonical form represents the highest two-cut. Expressions in Bryant's canonical form or in the new canonical form are shown to be prime and irredundant.

Some applications of if-then-else DAGs to multi-level logic minimization are given, and the *Printform* transformations for reducing the complexity of if-then-else DAGs are presented.

*This research was partially supported by an IBM Faculty Development Award and by NSF grant DCR-8503262.

1 Boolean expressions

Boolean expressions have found uses in several fields, including artificial intelligence, information retrieval, and VLSI design. The main uses in CAD have been in verification, logic minimization, logic synthesis, and simulation.

Various representations have been used, with three being the most popular:

- sum-of-products form for logic minimizers,
- arbitrary graphs of logic gates for logic synthesis and simulation, and
- binary decision diagrams for verification.

The representations for Boolean expressions can be classified in several ways:

- By the operators used: arbitrary operators, restricted class of operators, or single universal operator.
- By the ease of comparing expressions: canonical versus non-canonical forms.
- By the sharing of subexpressions: no sharing (trees), sharing within an expression, sharing between different expressions.

This paper is primarily about if-then-else DAGs (directed, acyclic graphs), a simple generalization of binary decision diagrams. Like binary decision diagrams, if-then-else graphs are directed and acyclic, and allow substantial sharing of common subexpressions. The if-then-else DAGs have the advantage of being able to represent sum-of-products expressions and arbitrary combinations of logic gates, while being easy to manipulate, to convert to canonical forms, or to factor.

Section 2 will review binary decision diagrams and Bryant's canonical form, then introduce *two-cuts*, which give a natural mapping from binary decision diagrams to if-then-else DAGs.

Section 3 will introduce *two-cut canonical forms* (of which Bryant's canonical form is a special case), and will give a general algorithm for doing binary operations on if-then-else DAGs. Finally, Section 4 will sketch some of the applications of if-then-else DAGs to logic minimization.

2 Binary decision diagrams

Binary decision digrams (BDD's, for short) use a single universal operator, are easily converted to canonical forms, and can share subexpressions either within an expression or across all expressions. The operator in a BDD is the if-then-else operator, with the restriction that the if-part must always be a single variable. The idea has been independently invented several times: early references to if-then-else trees include [Lee59], [Ake77], [Ake78], [Bry86], and [NO79].

The basic operator underlying if-then-else trees and DAGs is the *if-then-else* operator.

Definition 1: *The* if-then-else *operator is a ternary Boolean function, with* (**if** a **then** b **else** c) *defined as* $ab + \neg ac$ *or, equivalently,* $(a + \neg c)(\neg a + b)$.

Common operators are easily defined in terms of the if-then-else operator. For example,

- $\neg a =$ (**if** a **then** FALSE **else** TRUE)
- $ab =$ (**if** a **then** b **else** FALSE)
- a NAND $b =$ (**if** a **then** $\neg b$ **else** TRUE)
- $a + b =$ (**if** a **then** TRUE **else** b)
- $a \oplus b =$ (**if** a **then** $\neg b$ **else** b).

Definition 2: *A* binary decision diagram *is a binary directed acyclic graph with two leaves* TRUE *and* FALSE, *in which each non-leaf node is labeled with an atom and has two out-edges pointing to the* **then**-*part and the* **else**=*part. The meaning of a binary decision diagram is defined recursively as* (**if** *label(node)* **then** *meaning(***then**-*part)* **else** *meaning(***else**-*part)*).

Because BDD's always have an atom as the **if**-part of the if-then-else and TRUE and FALSE as the only leaves, they are easy to evaluate. If no other restrictions are put on binary decision diagrams, they can be difficult to simplify or to compare for equality.

A representation is *canonical* if any two expressions that are logically equivalent are identical. For example, if $ab + a\neg b$ is represented differently from a, then the representation is non-canonical. We can distinguish between *weak canonical forms*, in which logically equivalent expressions have identical structure, but may occur in different locations in memory, and *strong canonical forms*, in which expressions in different locations represent different Boolean functions.

Using canonical forms makes checking for equivalence easier. For a strong canonical form, only one pointer has to be checked for equality, and for other canonical forms, a simple traversal of the data structure (taking $O(n)$ time) suffices. In non-canonical forms, checking for equivalence is usually an NP-hard problem. Unfortunately, conversion from a non-canonical form to canonical form may take a lot of time or memory. Because equivalence checking in canonical form is fast, but equivalence checking in a non-canonical form is equivalent to the the NP-complete problem SATISFIABILITY [GJ79, pages 38–44], we are essentially guaranteed that the conversion to any canonical form is exponential in the worst case. In the common cases, however, a well-chosen canonical form can be small and easy to manipulate, and exponential blow-up is rare.

In some representations, subexpressions can be shared between different expressions, keeping memory usage small. Strong canonical forms are particularly useful, because they guarantee that any explicitly represented subexpression is shared by all expressions that need it.

In most applications, many different Boolean expressions are created, but several expressions have parts that are the same. Considerable savings in memory and processing time is possible if the common subexpressions are shared between different expressions. Tree representations that have no sharing of common subexpressions can often be modified to DAG representations in which repeated parts within a single expression are not duplicated. Bryant's canonical form is an example of sharing subexpressions within a single expression.

2.1 Bryant's canonical form

A common restriction put on binary decision diagrams is that no atom may appear twice on any path from the root to a leaf. One way to guarantee this property is to require that the set of all atoms be ordered, and that the atom at each node of the diagram be earlier in the order than the atoms of the successor nodes.

With one more restriction, we have Randal Bryant's canonical form for binary decision diagrams [Bry86]. The extra restriction is that distinct nodes represent non-equivalent expressions. Within a single DAG, the representation is a strong canonical form, but each expression is handled separately, so two independently built expressions may occupy different memory locations, but be logically equivalent.

Bryant also showed how to manipulate the binary decision diagrams efficiently. When Liisa Raiha and I implemented Bryant's representation, we found several minor improvements over the implementation described in [Bry86]. For example, a small symbol table can be used to cache the results of operations. The table uses triples (an operator and two operands) as a key to look up the result of the operation. The symbol table need not be large, as results are likely to be re-used either soon or not at all. One simple technique is to use a hash table with no provision for handling collisions; when a bucket is already in use, the old cached value is simply discarded.

2.2 Using a symbol table to improve Bryant's representation

After studying Bryant's canonical form and Liisa Raiha's minor improvements, I came up with the concept of a strong canonical form (Section 2), and found a way to make strong canonical forms out of binary decision diagrams. The structure of the diagrams needs no change, but the diagrams are built in a different way. Instead of building binary decision *trees* and reducing them to canonical form, as Bryant did, only canonical diagrams are built. A master symbol table is used to convert triples of the form (atom, left-bdd, right-bdd) into unique pointers to records containing the triple. I passed the idea of a master symbol table on to Randal Bryant, who has since incorporated it into another representation scheme [Bry87].

Because expressions are always built from the bottom up, the left-bdd and right-bdd pointers are guaranteed to be in the strong canonical form, so the check needed to see if the left and right children are equivalent is simply an equality check on pointers.

Bryant's techniques for manipulating BDD's can be used directly, without the costly overhead of repeatedly converting from non-canonical trees. The first implementation I made of the strong-canonical-form BDD's ran several times faster than the best implementation we had of Bryant's canonical form that worked by reducing trees to canonical form.

One disadvantage of this representation is that application programmers cannot free expressions explicitly, because they have no way of telling whether a particular expression is a subexpression of some still active expression. Without explicit freeing, garbage collection is needed to reclaim unused space. My implementations of BDD's give the expressions infinite life, and make no attempt to reclaim storage. A study needs to be made to determine how much space is wasted this way.

After implementing several different variants of Bryant's representation, I observed that his method for performing boolean operations on BDD's can be considerably simplified. Bryant implemented all the standard binary operators by means of a general `Apply()` operator, which took as arguments the operands and a description of what the operator did on the leaves of the BDD. All these Boolean operations can be defined in terms of a single if-then-else operator. The if-then-else operator can be implemented by the same sort of traversal as is used for `Apply()`, but without having to keep track of the operator to be applied at the leaves.

The if-then-else operator can be defined recursively using two simpler operations: `UniqTriple` and `split`. `UniqTriple` is a symbol table routine that looks up the if-then-else triple passed as arguments and creates a new entry in the symbol table if the triple is not found. We define `split(a,v,left)` to be the left subDAG `a->left` if `v` is the atom in the root, and the whole DAG `a` otherwise. To change (**if** a **then** b **else** c) to canonical form (with each of a, b, and c already in canonical form), we choose the smallest atom in the three arguments, call it `m`, and return

```
UniqTriple(m, IfThenElse(split(a,m,left),split(b,m,left),split(c,m,left)),
              IfThenElse(split(a,m,right),split(b,m,right),split(c,m,right))
          )
```

We stop the recursion when we get to one of the following special cases:

- If $a =$ TRUE, return b.
- If $a =$ FALSE, return c.
- If $b = c$, return b.
- If $b =$ TRUE and $c =$ FALSE, return a.
- If $c =$ TRUE and $b =$ FALSE, return $\neg a$.
- If (**if** a **then** b **else** c) is already in the symbol table, return it.

In practice, recursion is needed for about half the calls to `IfThenElse()`, and about one-seventh of the triples are already in the symbol table. Another minor speedup can be made by recognizing the following cases:

- If $a = b$, replace b with TRUE.
- If $a = c$, replace c with FALSE.
- If $a = \neg b$, replace b with FALSE.
- If $a = \neg c$, replace c with TRUE.

This if-then-else replacement for Bryant's `Apply()` operator seems to be as fast as the best implementations of `Apply()`, which use a symbol table to make the representation a strong canonical form and a hash table to cache the results of `Apply()`.

Paste MacDraw picture here.

Figure 2.1: Binary Decision Diagram for the expression $ab+cd+ef$, showing the two-cuts.

2.3 Dominators and two-cuts in binary decision diagrams

After developing the strong canonical form, I looked for ways to improve the printing of BDD's. I found that heavily parenthesized if-then-else trees were hard to understand, and that printing all paths to TRUE generated voluminous output, with many more terms than needed, and extra literals in some of the terms. For example, printing all paths to TRUE in $ab + cd + ef$ (represented by the binary decision diagram in Figure 2.1) would produce $ab + a\neg bcd + a\neg bc\neg def + a\neg b\neg cef + \neg acd + \neg ac\neg def + \neg a\neg cef$.

Dipen Moitra and I looked for properties of the graphs that could be used to improve the printing. We identified two such properties—*dominators* and *two-cuts*. These closely-related properties have also turned out to be important in factoring expressions for multi-level logic minimization.

Definition 3: *Vertex v of a rooted* DAG *is a* dominator *of vertex w, if, and only if, every path from the root to w contains v.*

One particularly interesting set of nodes in a BDD is the dominators of the leaf node FALSE. In [KM90], we proved that a BDD with a non-trivial dominator of FALSE (that is, a dominator other than the root or FALSE) can be split into two BDDs that are OR'd together. This *OR-split* is particularly useful for converting BDDs into sum-of-products form. For example, in Figure 2.1, the nodes labeled with c and e are dominators of FALSE, and the expression can be printed as $ab+cd+ef$ as desired. In a similar way, the dominators of TRUE can be used to do an *AND-split*.

The dominators of TRUE and FALSE in a BDD can be easily computed as the BDD is built, usually taking only a constant number of operations per node, but in the worst case taking $O(n^2)$ operations for a BDD with n nodes (see Section 2.4).

Because the dominators of TRUE and FALSE were so useful for printing expressions, we tried to generalize the concept, looking for a more powerful way to reduce the complexity of the expression. The useful property of dominators was that we could cut a BDD into two parts by removing one interior node. A natural generalization was to look at ways to cut a BDD apart by removing two nodes.

Draw this with MacDraw

Figure 2.2: Canonical binary decision diagram for $abc + \neg ad + \neg bd$, showing the two-cuts.

Definition 4: *A pair of vertices* $\{x, y\}$ *is a* two-cut *between the root* r *and a pair of vertices* $\{v, w\}$*, if, and only if, every path from* r *to* v *or* w *contains at least one of* x *or* y*. If a two-cut is mentioned without giving* $\{v, w\}$ *explicitly, then the pair* $\{\text{TRUE}, \text{FALSE}\}$ *is assumed.*

There are two trivial two-cuts in any BDD: the leaves TRUE and FALSE themselves, and the two children of the root. Every dominator of TRUE or FALSE is part of a two-cut (with the other leaf node), but there are other two-cuts. The paper [KM90] describes several useful properties of two-cuts and gives proofs.

The main use of dominators was in simplifying the printing of the if-then-else DAG, because we could ignore part of the DAG while printing another part. To use two-cuts effectively, we need to generalize the OR-split that made dominators useful. If x is a dominator of FALSE, then $\{\text{TRUE}, x\}$ and $\{\text{TRUE}, \text{FALSE}\}$ are both two-cuts, and share a common vertex (TRUE). It turns out that any two-cuts that share a common vertex can be used to simplify the expression. Let's call such pairs of two-cuts *collapsed two-cuts.*

Consider the expression $abc + \neg ad + \neg bd$, whose binary decision diagram is shown in Figure 2.2. The two-cuts of the DAG are $\{2, 4\}$, $\{3, 4\}$, and $\{\text{TRUE}, \text{FALSE}\}$. Notice that the DAG has no dominators of either TRUE or FALSE, so printing using just dominator information yields $abc + a\neg bd + \neg ad$, which has an unnecessary a in the second term. Because two of the two-cuts share a common node ($\{2, 4\}$ and $\{3, 4\}$ share the node 4), we can do better on this expression. The whole expression can be viewed as (**if** ab **then** c **else** d), which is $abc + (\neg a + \neg b)d$, printing neatly as $abc + \neg ad + \neg bd$.

Although the two-cuts were discovered while looking for better ways to print BDD's in sum-of-products form, they provide far more, giving us a partially factored form of the expression.

2.4 Finding two-cuts or dominators is $O(n^2)$

My implementations of BDD's with two-cuts keep track of the cuts as they build expressions. The algorithm used has an $O(n^2)$ worst-case running time for building a a DAG with n nodes, but in practice is rarely worse than linear. The algorithm and an example for which it takes n^2 time are presented below. The computation of dominators of TRUE and FALSE is almost identical to the two-cut computation, and the same example gives worst-case $O(n^2)$ performance.

The algorithm maintains a directed graph whose nodes are the two-cuts of the BDD. Edges in the two-cut graph point from a two-cut to the two-cut immediately below it in some BDD. Because the part of the BDD above a two-cut is irrelevant to the structure of the BDD below the two-cut,

Paste MacDraw picture here.

Figure 2.3: Two-cut graph for the binary decision diagram of Figure 2.1, which represents the expression $ab + cd + ef$.

the next two-cut is unique, and the out-degree of each node of the two-cut graph will be one, except for the two-cut {TRUE, FALSE} which has out-degree zero.

We can easily show that every two-cut in a BDD must be the two children of some vertex. Look at any vertex above the two-cut in the BDD. If one of its children is not one of the two vertices of the two-cut, descend to that child. Keep repeating the descent until the two children are the two-cut. The finiteness of the BDD guarantees that the descent must terminate sometime, and the two-cut property guarantees that the path we descend along cannot skip the vertices in the two-cut.

Because each two-cut is somewhere the two children of a vertex in the BDD, we only need to put a new node and edge in the two-cut graph when we add a new node to the BDD. Whenever a new triple (**if** a **then** B **else** C) is created, the two-cut $\{B, C\}$ may need to be added to the graph, and an edge added pointing to the two-cut immediately below $\{B, C\}$.

The two-cut below the new one may be a collapsed cut (sharing one of B or C with the new cut) or it may be a two-cut of B and a two-cut of C. To look for collapsed cuts, we walk down the cuts of B (follow the edges in the two-cut graph), starting with the two-cut formed by B's children, looking for a cut containing C. We can stop walking down the cuts when both expressions in the cut do not contain the smallest variable of C. To look for a common cut below both B and C, we start at the two-cuts representing the children of B and C and walk down the cuts until we find the place where the paths to the sink {TRUE, FALSE} join. We can always decide which edge in the two-cut graph to traverse next by looking at the smallest variable in the expressions for the two-cuts. Figure 2.3 shows the two-cut graph built for the BDD of Figure 2.1.

With n nodes in a binary decision diagram, we have at most n possible two-cuts, and the longest path in the two-cut graph is at most $n - 1$ long. At worst, we will have to traverse $O(n)$ edges of the two-cut graph when adding a new two-cut, so building the entire graph is $O(n^2)$.

The $O(n^2)$ bound is not necessarily tight for the problem, but is for this particular algorithm. Here is a way to build a BDD that uses $\Omega(n^2)$ steps to build the two-cut graph:

1. Build two BDDs: $X = x_1 + x_2 + \cdots + x_k$ and $Y = y_1 + y_2 + \cdots + y_k$. Note that the two-cut for the children of X begins a path of length k in the two-cut graph, as does the two-cut for the children of Y. The two paths are joined only at {TRUE, FALSE}.
2. Build $2m$ nodes: $R_i = r_i + X$ and $S_i = s_i + Y$.
3. Build m^2 nodes: $M_{i,j}$ = (**if** $m_{i,j}$ **then** r_i **else** s_j). The next two-cut below $\{r_i, s_j\}$ is {TRUE, FALSE}, but it requires traversing $2k + 2$ edges of the two-cut graph to create each new edge.
4. Build tree for each row: B_i = (**if** $b_{i,1}$ **then** (**if** $b_{i,2}$ **then** ... **else** $M_{i,2}$) **else** $M_{i,1}$).
5. Build tree for each column: A = (**if** a_1 **then** (**if** a_2 **then** ... **else** B_2) **else** B_1).

The BDD A contains $2m^2 + 2m + 2k$ nodes, and takes over $2km^2$ operations to build the two-cut graph just for step 3. If m is chosen to be approximately $\sqrt{k}$, then the BDD contains $O(k)$ nodes, but the two-cut graph takes k^2 steps to compute.

Draw this with MacDraw

Figure 3.1: If-then-else DAG for $abc + \neg ad + \neg bd$, factored as (**if** ab **then** c **else** d).

An asymptotically faster algorithm for keeping track of two-cuts probably exists, but the time taken by this algorithm is an insignificant fraction of the time spent manipulating BDD's, so further refinement is not worthwhile.

3 If-then-else DAGs with an expression in the if-part

Finding two-cuts so useful for simplifying binary decision diagrams, I looked for a new representation in which the two-cuts were more naturally represented. We can view a binary decision diagram with a two-cut as having three parts: the DAG from the root to the cut, and the two subDAGs below the cut. For example, for the BDD in Figure 2.2, the parts are ab, c, and d. If we allow arbitrary expressions in the **if**-part of if-then-else expressions, we can represent the two-cut explicitly as (**if** ab **then** c **else** d), as shown in Figure 3.1.

Definition 5: *An* if-then-else DAG *is a ternary directed acyclic graph in which each leaf is labeled with* TRUE, FALSE *or a literal, and each internal node has three out-edges pointing to the* **if**-, **then**-, *and* **else**-*parts. The meaning of a leaf node is the label on the node, and the meaning of an internal node is defined recursively as*

(**if** *meaning(***if**-*part)* **then** *meaning(***then**-*part)* **else** *meaning(***else**-*part)*).

If-then-else DAGs are not new, but they do have several properties that make them more attractive than binary decision diagrams for CAD work. If-then-else DAGs

- provide a single representation scheme for representing binary decision diagrams, sum-of-products, and arbitrary combinations of 1- and 2-input gates. Every 1- or 2-input gate can be represented as an if-then-else triple, as explained on page 1, so networks without feedback can be made by replacing each gate by the appropriate if-then-else triple.
- expose more subexpressions for potential sharing than do binary decision diagrams. The same sharing of **then**- and **else**-parts is possible in both BDD's and if-then-else DAGs, but only the general DAGs allow sharing subexpressions in the **if**-part. For example, the three functions $ab(d + e)$, $c(d + e)$, and abd are represented as (**if** (**if** a **then** b **else** FALSE) **then** (**if** d **then** TRUE **else** e) **else** FALSE), (**if** c **then** (**if** d **then** TRUE **else** e) **else** FALSE), and (**if** (**if** a **then** b **else** FALSE) **then** d **else** FALSE), sharing the subexpressions (**if** a **then** b **else** FALSE) and (**if** d **then** TRUE **else** FALSE).
- have at least two useful canonical forms: Bryant's canonical form and a new canonical form introduced in this paper.
- are a more factored form than BDD's, providing for better printing and logic minimization. With the aid of the transformations described in Section 4.2, good factorings can be found from the canonical forms.
- express Boolean operations naturally as if-then-else triples, so the same symbol table used for storing canonical forms can be used for caching the results of operations.

Draw this with MacDraw

Figure 3.2: If-then-else tree representing the sum-of-products expression $abc + \neg ad + \neg bd$.

A free-form DAG using any subset of the operators AND, OR, XOR, NOT, and IF can easily be converted to a if-then-else DAG having the same structure. Each operator can be converted as follows:

AND abc is converted to (**if** (**if** a **then** b **else** FALSE) **then** c **else** FALSE).

OR $a + b + c$ is converted to (**if** (**if** a **then** TRUE **else** b) **then** TRUE **else** c).

XOR $a \oplus b \oplus c$ is converted to (**if** (**if** a **then** b **else** $\neg b$) **then** c **else** $\neg c$).

NOT $\neg a$ is directly representable by flipping one bit in the pointer to a. $\neg\neg a$ is not representable, but the equivalent expression a is.

IF (**if** a **then** b **else** c) needs no conversion.

This conversion can also be done in the reverse direction. The mapping is not an isomorphism, as some information about the grouping of operands is lost when converting to the if-then-else DAG.

The output of an arbitrary operator can be handled by substituting an expression for the literal corresponding to each input in the defining DAG for the operator. The experimental multi-level logic minimization programs I have been working on accept DAGs of arbitrary operators (in BLIF format [Ber88]), convert them to if-then-else DAGs, transform them to reduce their complexity (as measured by the functions in Section 4.1), then convert to n-input AND, n-input OR, 2-input XOR, and NOT gates.

A sum-of-products expression can be represented exactly using just the conversions for AND and OR, as illustrated in Figure 3.2. Note that a sum-of-products expression allows no sharing of subexpressions, so the if-then-else DAG is always a tree.

Sheldon Akers [Ake78, page 514] suggested using if-then-else trees rather than binary decision diagrams to generate expressions more easily. Nelson and Oppen [NO79] showed how to convert if-then-else trees into binary decision diagrams by recursively using the transformation

$$\begin{aligned}&(\textbf{if}\ (\textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c)\ \textbf{then}\ d\ \textbf{else}\ e) = \\ &\quad (\textbf{if}\ a\ \textbf{then}\ (\textbf{if}\ b\ \textbf{then}\ d\ \textbf{else}\ e)\ \textbf{else}\ (\textbf{if}\ c\ \textbf{then}\ d\ \textbf{else}\ e)).\end{aligned}$$

To get a BDD, leaves labeled with atoms need to be further expanded to (**if** *atom* **then** TRUE **else** FALSE). Neither Akers [Ake77, Ake78] nor Nelson and Oppen [NO79] used the restriction that an atom appear only once on each path. Akers explicitly refers to paths with repeated occurrences of the same atom [Ake78, page 512].

3.1 Two-cut canonical forms

One of the attractive features of binary decision diagrams, especially for verification applications, is the ease of computing a canonical form—Bryant's canonical form. It would be useful to have a similar canonical form for if-then-else DAGs. Of course, binary decision diagrams are a special case of if-then-else DAGs, so we could just use Bryant's canonical form, but in doing so we lose the explicit representation of all two-cuts except the two trivial ones (the two children of the root and the two leaves TRUE and FALSE). Building a separate two-cut graph to find the other two-cuts is an inelegant solution, but we can use the relationship between BDD's and if-then-else DAGs to represent the two-cuts directly.

An if-then-else triple corresponds to a BDD with a two-cut. The **if**-part corresponds to the BDD above the cut, and the **then**- and **else**-parts correspond to the subDAGs below the two-cut. Restricting the **if**-part to simple variables is equivalent to choosing always to represent the topmost two-cut in the BDD. If, instead of choosing the topmost two-cut, we always choose the non-trivial one closest to the leaves, then the triple for the **if**-part of an expression corresponds to the next two-cut up. Following the chain of **if**-parts until we get to a literal gives us the two-cuts in order from the bottom up.

To make these if-then-else DAGs canonical, we must place some restrictions on the expressions allowed in the **if**-, **then**-, and **else**-parts of the structure. There are seven restrictions, the first three of which are modified slightly from corresponding restrictions in Bryant's canonical form.

1. All the atoms in the **if**-part must be before all atoms in the **then**- and **else**-parts. This restriction is a direct translation of Bryant's restriction that the atom in a node be before the atoms of the subDAGs. A weaker restriction, that the variables of the **if**-part be disjoint from those of the **then**- and **else**-parts, would be enough to eliminate paths with duplicate variables, but not enough to make the form canonical. Non-canonical expressions using this weaker version of the restriction are useful for factoring.
2. The **then**- and **else**-parts of an expression must be distinct Boolean functions—exactly as in Bryant's canonical form.
3. A systematic choice must be made between equivalent expressions (**if** a **then** b **else** c) and (**if** $\neg a$ **then** c **else** b). This corresponds to Bryant's choice of atoms as node labels (never negations of atoms). The preferred choice is not obvious for if-then-else DAGs, so I have experimented with two choices. In one system, I picked an ordering that puts the constants TRUE and FALSE in the **else**- position rather than the **then**-position. The ordering also guaranteed that no atom of the **else**-part comes before the first atom of the **then**-part. This choice of ordering simplifies some of the code in the implementation, but is otherwise arbitrary. A better choice of ordering is described in Section 3.4.
4. Triples of the form (**if** a **then** TRUE **else** FALSE) and (**if** a **then** FALSE **else** TRUE) are prohibited. The first triple should be represented simply as a, and the second one by $\neg a$, which is structured like a, but has the **then**- and **else**-parts negated. (Section 3.4 describes a better way of handling negation.) Any literal can be used as an expression, not just an un-negated atom, so expressions of the form (**if** a **then** FALSE **else** TRUE) are never needed.
5. Triples of the form (**if** TRUE **then** b **else** c) and (**if** FALSE **then** b **else** c) are prohibited, and should be replaced with b and c respectively.
6. In the triple (**if** a **then** b **else** c), b and c must not share both **then**- and **else**-parts. If $b =$ (**if** b_a **then** b_b **else** c_c) and $c =$ (**if** c_a **then** b_b **else** c_c), then the correct representation for the original expression is (**if** (**if** a **then** b_a **else** c_a) **then** b_b **else** c_c).

   Note that the ordering in Condition 3 guarantees that b_b and c_c will be in the same order in both the **then**- and the **else**-part, but if we use the variant of Section 3.4, we also have to look for $b =$ (**if** b_a **then** b_b **else** b_c) and $c =$ (**if** c_a **then** b_c **else** b_b), and convert the original expression to (**if** (**if** a **then** b_a **else** $\neg c_a$) **then** b_b **else** b_c).
7. In the triple (**if** a **then** b **else** c), b must not contain c as a **then**- or **else**-part. If $b =$ (**if** b_1 **then** b_b **else** c) or $b =$ (**if** b_2 **then** c **else** b_c), then the expression should be represented as (**if** (**if** a **then** b_1 **else** FALSE) **then** b_b **else** c) or (**if** (**if** a **then** b_2 **else** TRUE) **then** c **else** b_c). If

c is one of the constants TRUE or FALSE, this condition amounts to choosing left-associativity for commutative AND or OR operations.

Note that the choice of ordering in Condition 3 guarantees that we need not test whether c can be represented as (**if** c_a **then** c_b **else** b) or (**if** c_a **then** b **else** c_c), because the lowest variable in c is no lower than the lowest one in b. With a different ordering, such as that described in Section 3.4, the symmetric test would have to be made.

We can show that imposing the conditions listed above defines a canonical form by exhibiting an isomorphism with Bryant's canonical form.

A general if-then-else DAG can be turned into Bryant's canonical form by repeatedly applying the transformation

$$\begin{aligned}&(\mathbf{if}\ (\mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c)\ \mathbf{then}\ d\ \mathbf{else}\ e) \mapsto\\ &\quad(\mathbf{if}\ a\ \mathbf{then}\ (\mathbf{if}\ b\ \mathbf{then}\ d\ \mathbf{else}\ e)\ \mathbf{else}\ (\mathbf{if}\ c\ \mathbf{then}\ d\ \mathbf{else}\ e)),\end{aligned}$$

and using a symbol table to merge identical subexpressions. Condition 1 guarantees that the variable ordering in the binary decision diagram will be correct.

To do the reverse mapping, we can cut apart an if-then-else DAG in Bryant's canonical form at the lowest two-cut above {TRUE, FALSE}, converting the DAG above the cut for the **if**-part, and the two subDAGs below the cut for the **then**- and **else**-parts (see Section 2.3 for the definition of two-cuts). We may need to negate the **if**-part and swap the **then**- and **else**-parts to meet Condition 3 above. We need to use a symbol table to ensure that identical subexpressions are properly merged.

If we take an expression that meets Conditions 1 through 7, convert it to Bryant's canonical form, and convert it back, we will get the original expression. The proof is straightforward but tedious, so will be left as an exercise for the reader. This isomorphism between Bryant's canonical form and the new form implies that the new form is also canonical.

3.2 General algorithm for reducing to canonical form

We can use essentially the same algorithm for converting to either Bryant's canonical form or the new form described in Section 3.1. Given three expressions, a, b, and c, we construct an expression for (**if** a **then** b **else** c) that is in the desired canonical form.

First, we convert each of the three arguments to canonical form, then transform the triple (**if** a **then** b **else** c) until Conditions 1 through 7 are met. Some of the conditions are easy to meet, but others may require extensive transformation of the expression. The non-trivial ones are Conditions 1, 6, and 7.

The hardest condition to satisfy is Condition 1, which requires that all variables of the **if**-part be before any variables of the **then**- and **else**-parts. Other than some checks for trivial cases, transformations are first applied to satisfy this condition, then transformations that do not affect the order of the variables are applied to satisfy Conditions 6 and 7, and finally, a trivial transformation is applied to satisfy the **then-else** ordering condition (Condition 3). The other conditions are automatically taken care of by the checks for trivial cases and are not re-introduced by any of the transformations.

The main transformation is the one that makes the variable ordering required by Condition 1 correct. The transformation is a generalization of the following simple one:

$$\begin{aligned}&(\mathbf{if}\ (\mathbf{if}\ x\ \mathbf{then}\ a_b\ \mathbf{else}\ a_c)\ \mathbf{then}\ (\mathbf{if}\ x\ \mathbf{then}\ b_b\ \mathbf{else}\ b_c)\ \mathbf{else}\ (\mathbf{if}\ x\ \mathbf{then}\ c_b\ \mathbf{else}\ c_c)) \mapsto\\ &\quad(\mathbf{if}\ x\ \mathbf{then}\ (\mathbf{if}\ a_b\ \mathbf{then}\ b_b\ \mathbf{else}\ c_b)\ \mathbf{else}\ (\mathbf{if}\ a_c\ \mathbf{then}\ b_c\ \mathbf{else}\ c_c)).\end{aligned}$$

Because the three input expressions are in canonical form, the variable ordering is correct in each. In particular, all the variables in x are earlier in the ordering than any variable in a_b, a_c, b_b, b_c, c_b, or c_c, satisfying Condition 1 at this level of the DAG. After the new **then**- and **else**-parts have been converted to canonical form, the entire DAG satisfies the condition.

The transformation is generalized to covers cases when one or two of the input expressions do not include the shared part x, but have only variables that are after those in x. For example, if b and c share a common **if**-part x, but a starts with a variable after those in x, we can think of a as (**if** x **then** a **else** a), and apply the simple transformation.

Of course, the generalization of the simple transformation does not always apply to three arbitrary input expressions. However, there are always equivalent, non-canonical representations of the three expressions to which the transformation can be applied. For the non-canonical representations to be useful, they must still satisfy Condition 1, but we can relax Conditions 6 and 7. The transformation

$$\begin{aligned}&(\textbf{if }(\textbf{if } a_a \textbf{ then } a_b \textbf{ else } a_c) \textbf{ then } b \textbf{ else } c) \mapsto \\ &\quad (\textbf{if } a_a \textbf{ then } (\textbf{if } a_b \textbf{ then } b \textbf{ else } c) \textbf{ else } (\textbf{if } a_c \textbf{ then } b \textbf{ else } c))\end{aligned}$$

can be applied repeatedly to the canonical input expressions to reduce the size of their **if**-parts. This process of peeling off the end of the **if**-part and attaching it to the **then**- and **else**-parts is called *walking up the cuts*. By walking up the cuts of the three expressions, always peeling off variables from the expression with the largest range of variables in its **if**-part, we eventually reach a point where the generalized transformation is applicable.

Walking up the cuts can be made faster if one (or two) of the input expressions does not use the lowest variable used by any of the three. When an expression (say a) does not use the lowest variable, walking up its cuts will eventually convert it to (**if** *don't care* **then** a **else** a), so we can immediately do the conversion, and just walk up the cuts of the other expressions until their **if**-parts use only variables that come before the variables in a.

The slowness of walking up the cuts is the main limitation on the speed of the if-then-else DAG implementations. Walking up the cuts can sometimes be avoided by recognizing special cases. For example, commutative operations like (**if** b **then** a **else** FALSE) and (**if** b **then** a **else** $\neg a$) can be immediately rearranged to get (**if** a **then** b **else** FALSE) and (**if** a **then** b **else** $\neg b$).

Even when walking up the cuts is unavoidable, it need not be as slow as it is in the current implementations. The bottleneck in the current process is storing the intermediate if-then-else triples in the symbol table. By not explicitly building the intermediate expressions, the process could be made much faster. This trick has been used for converting if-then-else DAGs to sum-of-products form, resulting in substantial speedups.

After Condition 1 is satisfied, we can satisfy Conditions 6 and 7 fairly easily. The definitions of the conditions give the necessary tests and transformations. If the **if**-, **then**-, and **else**-parts of an expression are already known to be in canonical form, then the tests for Conditions 6 and 7 are applied only at the top level.

3.3 Two-cut canonical forms are prime and irredundant

Other researchers in multi-level minimization, working primarily with sum-of-products representations, have found the concepts of *primality* and *irredundancy* to be important [BHMS84, page 28], [Bra87, page 202]. Both concepts have natural analogs in if-then-else DAG representations. Both Bryant's canonical form and the new canonical form presented in Section 3.1 can be shown to be prime and irredundant with the definition presented here.

In sum-of-products form, an expression is said to be *prime* if no term could be modified by changing a literal to TRUE without changing the meaning of the expression. Similarly, an expression in sum-of-products form is said to be irredundant if no term can be changed to FALSE without changing the meaning of the expression.

Definition 6: *An if-then-else* DAG *is* prime *if no pointer to a literal, sub*DAG, *or the constant* FALSE *could be replaced with a pointer to* TRUE *without changing the meaning of the expression. An if-then-else* DAG *is* irredundant *if no pointer to a literal, sub*DAG, *or the constant* TRUE *could be replaced with a pointer to* FALSE *without changing the meaning of the expression.*

Paste MacDraw picture here.

Figure 3.3: A prime if-then-else DAG corresponding to the non-prime sum-of-products expression $acd + bcd + b\neg d$. If the shared sub-DAG for cd is duplicated, the resulting if-then-else tree is not prime.

We have to be a little careful about what "changing the meaning of the expression" means. Usually, two expressions are considered to have the same meaning if their values differ only on some specified *don't-care set*, but for this paper, we will restrict ourselves to equality of Boolean functions, which is equivalent to an empty don't-care set.

The new definitions of "prime" and "irredundant" correspond to existing ones for sum-of-products and factored forms. For example, we can use an if-then-else tree (no sharing of parts of terms) to represent a sum-of-products expression as shown in Figure 3.2. If the if-then-else tree is prime or irredundant, then the sum-of-products expression must be, because the substitutions to be tested in the sum-of-products form are a subset of those tested in the if-then-else tree.

Note that an if-then-else DAG that shares parts of different terms may be prime and irredundant without the corresponding sum-of-products expression being prime or irredundant. For example, the expression $acd + bcd + b\neg d$ is not prime, because it is equivalent to $acd + bc + b\neg d$, but the corresponding if-then-else DAG shown in Figure 3.3 is prime.

The extra tests for the if-then-else tree are easily satisfied for trees corresponding to sum-of-products representations. If a sum-of-products expression is prime, then the corresponding if-then-else tree will have its meaning changed if any pointer to a literal is changed to a pointer to TRUE. If changing any literal in a term to TRUE increases the coverage of the term enough to change the meaning of the whole expression, then changing a sequence of literals in one term would change the meaning at least as much. Converting an entire term or sequence of terms to TRUE changes the expression to TRUE, thus changing the meaning of the expression. The if-then-else tree corresponding to a prime sum-of-products expression is, therefore, also prime. An irredundant sum-of-product expression converts to an irredundant tree, because changing a literal or sequence of literals to FALSE is equivalent to changing the term it is in to FALSE.

Factored forms, literals combined with arbitrary combinations of AND and OR operators, can also be represented as if-then-else trees. Brayton's definitions of prime and irredundant for factored forms [Bra87, page 203]) correspond to the definition for if-then-else DAGs.

Having extended the definitions of *prime* and *irredundant* to if-then-else DAGs, let's now show that the two-cut canonical forms are both prime and irredundant. We will show that replacing a pointer to anything else with a pointer to TRUE must change the meaning of the expression, showing that the canonical forms are prime. An exactly analogous proof (changing one pointer to a pointer to FALSE) shows that the forms are irredundant.

For Bryant's canonical form, the proof is trivial. Every non-constant node in a BDD in Bryant's form must have a path to both TRUE and FALSE. In the original expression, follow the path from the root to the node originally pointed to by the changed pointer, setting each variable to 1 if you follow the **then**-branch out of a node, and to 0 if you follow the **else**-branch. Continue along the path from the node to FALSE in the same manner. The variable-order condition for Bryant's canonical form guarantees that no variable has been set twice. This setting of the variables will make the original expression FALSE, but the modified expression will be TRUE, showing that the two are not equivalent, and an expression in Bryant's canonical form is prime. By swapping TRUE and FALSE in the proof, we can show that Bryant's canonical form is irredundant as well.

We can reason in an analogous way for the new canonical form.

Lemma 1: *If* (**if** a **then** b **else** c) *is in canonical form and x is an expression not equivalent to b but using no variables other than those in b, then* (**if** a **then** b **else** c) *is not equivalent to* (**if** a **then** x **else** c).

Proof: There is some setting of the variables of a that make it TRUE, and some setting of the variables of b that distinguishes it from x. By Condition 1 the two sets of variables do not intersect, so the combined setting distinguishes (**if** a **then** b **else** c) from (**if** a **then** x **else** c). The same proof works if we modify c or both b and c.

Lemma 2: *If* (**if** a **then** b **else** c) *is in canonical form and x is an expression not equivalent to a but using no variables other than those in a, then* (**if** a **then** b **else** c) *is not equivalent to* (**if** x **then** b **else** c).

Proof: Because x is not equivalent to a, we can find a way to set the variables of a so that its value is different from the value of x. By Condition 2, we can find a setting of the variables of b and c so that their values are different. By Condition 1, the variables set to distinguish x and a do not overlap with the variables set to distinguish b and c, so the combined setting of the variables will distinguish (**if** a **then** b **else** c) from (**if** x **then** b **else** c).

Theorem 1: *An expression in the new canonical form is both prime and irredundant.*

Proof: We want to show that changing any pointer in the canonical form to a pointer to TRUE or FALSE will result in an expression not equivalent to the original. If the pointer was the root, then the modified expression is clearly not equivalent. Otherwise the pointer is part of some if-then-else triple, and by the lemmas, the original triple is not equivalent to the modified triple. We can continue applying the lemmas to triples higher up the DAG until we reach the root, showing that the meaning of the expression was changed by modifying the pointer.

We have proven that expressions in Bryant's canonical form or the new canonical form presented in Section 3.1 are prime and irredundant.

3.4 Space-efficient if-then-else DAGs

The rather arbitrary order of **then**- and **else**-clauses imposed by Condition 3 of Section 3.1 leads to an inefficient use of storage space or computation time. Because **if**-parts may need to be negated to put the **then**- and **else**-parts in the right order, we either have to build negated expressions on demand (which is slow), or we can build the negation of each expression at the same time as the expression (which doubles the storage cost).

A more space-efficient method for representing negated expressions steals one bit from each pointer to represent negation. If negation is allowed arbitrarily, the representation is not canonical. For example, (**if** a **then** b **else** c) has the same meaning as (**if** $\neg a$ **then** c **else** b), and (**if** a **then** $\neg b$ **else** $\neg c$) has the same meaning as $\neg$(**if** a **then** b **else** c). The space-efficient form is made canonical by requiring that the **if**- and **then**-parts of an expression be pure pointers, with negation allowed only for the **else**-part or the entire expression. Eliminating all negated triples cuts the number of triples in half, and saves one pointer in each triple: the pointer to the negation.

Memory usage can be reduced by using space-efficient symbol tables to store the unique triples. The symbol table that guarantees unique triples can be implemented as a hash table, as a 2-3-tree, or as a balanced binary tree. If linear probing is used in the hash table, rehashing is needed to keep the table between 40% and 80% full; the storage cost is about 1.6 words per triple. Using chained hashing, we can overfill the hash table. If the average chain length is k, then the storage cost for the hash table is $1 + 1/k$ words per triple. The rehashing required for either hash table implementation causes an enormous burst of page faults in a virtual memory environment, and hashing looks much less desirable.

A 2-3-tree implementation requires 5 words per node of the tree. If half the nodes are full, then 1.5 triples per node implies a storage cost of 3.33 words per triple. The depth of the 2-3-tree can be reduced by having a separate 2-3-tree for each set of triple with the same **if**-part. Each triple e has a pointer to the symbol table for the set $\{(a, b, c) | a = e\}$. Breaking the symbol table into pieces greatly reduces the lookup time, at a cost of one more word per triple.

If we use a balanced binary tree, instead of a 2-3-tree, the pointers for the symbol table can be included in the data structure for the expression, requiring only 2 words per triple. Breaking up the symbol table in the same way as for 2-3-trees increases the cost of the symbol table to 3 words per node. Although the balanced binary trees are slightly deeper than the corresponding 2-3-trees, access is slightly faster, since insertion can be done with an iterative algorithm, instead of a recursive one. The average depth of an expression in the balanced binary trees is empirically estimated at about $0.5 \ln(\text{total number of triples})$.

Because the total number of triples that can be stored is limited by the size of the swap space, the best performance is obtained by using a large hash table with chaining, and never rehashing.

4 Current work—multi-level logic optimization

Logic synthesis usually consists of several somewhat separable stages. My current work has been applying if-then-else DAGs to one of those stages—technology-independent multi-level logic optimization. In this stage of the process, the main goal is to reduce the complexity of a logic network, so that technology mappers can find good implementations.

If-then-else DAGs are particularly attractive for representing logic networks, because they can explicitly represent shared subexpressions, and can easily be manipulated to minimize the network. Single-output functions can be represented as one DAG, and multiple-output networks either as multiple DAGs or as a single DAG with multiple roots. Because my implementations use a symbol table to store if-then-else triples uniquely, the two ways of representing multiple-output networks are equivalent.

4.1 Expression complexity, counting literals

When doing logic minimization, the first question is "what exactly is being minimized?" Usually there are constraints on signal delays and on implementation costs, and the goal is to find the best (fastest or cheapest) design that meets the constraints. Unfortunately, both signal delay and implementation cost are very dependent on the technology used, and optimizations tied to a particular cell library quickly become obsolete. Just as compiler writers have found code optimizations that work well independent of the target machine, we look for logic optimizations that will work well independent of the target technology. After doing what optimization we can in a technology-independent way, a *technology mapper* generates a specific implementation. Some optimizations are done by the mapper, equivalent to the peephole optimizations done by the code generator of a compiler.

For technology-independent minimization to work, we need measures that are not dependent on any particular cell library, but that roughly approximate the cost or speed obtained by a technology mapper. Technology-independent delay estimates are hard to come up with, so most research has concentrated on size minimization, leaving the delay minimization to the technology mapper. Other researchers have used

- the number of literals in sum-of-products form,
- the number of literals in the factored form,
- the number of distinct literals the function depends on, or
- the number of distinct variables that the function depends on

as an estimate of the complexity of a complex gate, and added the size estimates for all gates in a network to get a size estimate for the network [Bra87, page 235]. Note that with any of these methods, inverters are essentially free, as any input to a gate can be inverted and the gate definition changed without changing the size of the gate. Because optimal signal polarity is technology-dependent, this hiding of inverters is usually considered a good feature for technology-independent optimization.

The measures described above are useful when a network has been decomposed into gates, but are not directly applicable to a Boolean network described as an if-then-else DAG. New measures are needed that are as useful for estimating circuit size.

If we are interested primarily in the storage cost of the representation, the following definition seems natural.

Definition 7: *The size of an if-then-else* DAG *is the number of nodes in the* DAG*, counting the if-then-else triples and the literals, but not the constants* TRUE *and* FALSE*.*

If we use space-efficient if-then-else DAGs, inverters can be inserted arbitrarily without changing the cost of the expression, as no new nodes are created. This definition of size is excellent for use in inductive proofs of properties of the DAGs, or for determining the storage costs in a program manipulating the DAGs, but may not be the best estimator of circuit size when mapped to typical technologies.

Another size measure I have experimented with is intended to be roughly proportional to the number of transistors needed in a multi-level MOS implementation. It is computed by a depth-first search of the DAG:

$$
\begin{aligned}
\text{count}(\text{TRUE}) &= \text{count}(\text{FALSE}) = 0 \\
\text{count}(\text{literal}) &= 1 \\
\text{count}(\text{internal node}) &= 1, \text{if node previously visited} \\
\text{count}((\textbf{if } a \textbf{ then } b \textbf{ else } c)) &= \text{count}(a) + \text{count}(b), \text{if } c \text{ is TRUE or FALSE} \\
\text{count}((\textbf{if } a \textbf{ then } b \textbf{ else } c)) &= \text{count}(a) + \text{count}(c), \text{if } b \text{ if TRUE or FALSE} \\
\text{count}((\textbf{if } a \textbf{ then } b \textbf{ else } c)) &= 1 + \text{count}(a) + \text{count}(b) + \text{count}(c), \text{otherwise}
\end{aligned}
$$

In the "count" measure, multiple uses of a literal are counted separately, so the "count" measure provides a much better estimate of circuit size than the "size" measure. Adding one for general if-then-else triples, but not for triples with a constant **then**- or **else**-part, reflects the additional cost of selectors and XOR gates over AND, OR, NAND, NOR, and AND-OR-INVERT gates in most technologies. Other meeasures could be devised to get better matches to the costs of particular technologies.

I am primarily interested in technology-independent optimization, and so have been using the technology mapper in misII [DGR+87], rather than developing a new technology mapper. To use this mapper, it has been necessary to convert if-then-else DAGs into networks of gates, each specified in sum-of-products form. My conversion routine generates n-input ANDs and ORs, inverters, 2-input XORs, and if-then-else selectors. The "count" measure is almost identical to the number of literals

expression	size	count	pcount	sum-of-products
$abcd$	7	4	4	4
$a+b+c+d$	7	4	4	4
$a \oplus b \oplus c \oplus d$	10	10	4	32
$ad+be+cf$	11	6	6	6
$a(c+d+e+g)+b(c+d+e+f)+$ $c(e+f)+d(f+g)$	27	14	16	24
$(a+b+c+d)(\neg a+\neg b+\neg c+\neg d)$	12	8	8	12
$(b+c)(a(e+g)(\neg e+f+\neg g))+h+i)+$ $d(e+g)(\neg e+f+\neg g)$	20	12	16	41
$a(c+d+e+g)+b(c+d+e+f)+$ $(c+d)(dg+e+f)$	35	13	16	29

Table 4.1: Different ways to measure the size of an if-then-else DAG illustrated on some simple expressions. The first three size measures are for an if-then-else DAG that corresponds to the expression as factored in the first column. The last size measure is the number of literals in a sum-of-products representation for the same function. The last four functions are taken from [Bra87], but are factored to minimize the *count* measure.

reported by misII minus the number of nodes in the decomposition. Note that extra inverters, which should have zero cost, add one node and one literal to misII's count. Because misII's technology mapper adds and removes inverters, and removes many of the intermediate nodes from an AND/OR decomposition, my cost estimate seems quite reasonable for this mapper.

Other minor variants of the count measure are useful for other purposes. For example, we may want to count the number of literals that would be printed if we expanded with AND, OR, XOR, and NOT operators. A general if-then-else triple (**if** a **then** b **else** c) can be expanded as either $ab+(\neg a)c$ or $(a+c)(\neg a+b)$, whichever is more convenient. For this measure, we want to count shared subexpressions fully each time, as they will have to be printed repeatedly.

$$\begin{aligned}
\text{pcount}(\textsc{true}) &= \text{pcount}(\textsc{false}) = 0\\
\text{pcount}(\text{literal}) &= 1\\
\text{pcount}((\textbf{if } a \textbf{ then } b \textbf{ else } c)) &= \text{pcount}(a)+\text{pcount}(b), \text{if } c=\neg b,\ c=\textsc{true}, \text{ or } c=\textsc{false}\\
\text{pcount}((\textbf{if } a \textbf{ then } b \textbf{ else } c)) &= \text{pcount}(a)+\text{pcount}(c), \text{if } b=\textsc{true} \text{ or } b=\textsc{false}\\
\text{pcount}((\textbf{if } a \textbf{ then } b \textbf{ else } c)) &= 2\text{pcount}(a)+\text{pcount}(b)+\text{pcount}(c), \text{otherwise}
\end{aligned}$$

Table 4.1 shows the sizes of several simple expressions, comparing the different measures and the number of literals needed in the equivalent sum-of-products representation. Other measures of expression complexity are easily devised, and studies are needed to determine which measures correlate best with circuit size and delay after mapping to various technologies.

4.2 Factoring

The process of reducing the complexity of an expression is usually called *factoring*, because the main techniques used by other researchers involve finding shared parts of terms in a sum-of-products representation, and factoring them out. For example, $abc+ad+cd$ might be factored as $a(bc+d)+cd$.

When dealing with if-then-else DAGs, *factoring* means any transformation of the DAG that preserves the meaning while reducing the size (as measured by the "count"function in Section 4.1.

The most effective factoring techniques I have found for if-then-else DAGs involve transformations that change the order of the variables, either locally for one part of the DAG, or for the entire DAG. A complete description of the transformations used is beyond the scope of this paper, but I will illustrate the concept with a particularly simple set of transformations. These transformations were originally intended to be applied to if-then-else DAGs in my new canonical form, to make them easier to read when printed, and so are called *Printform* transformations.

The Printform transformations do crude factoring while maintaining a weakened version of Condition 1. The variables in the **if**-part are required to be disjoint from the variables in the **then**- and **else**-parts, but are not required to be before them in the variable ordering. This weakened form of Condition 1 is still enough to guarantee that an expression is prime and irredundant, using the same proof as in Section 3.3.

Another set of transformations (the *local factoring* transformations) are similar, but allow some duplication of variables between **if**-part and the **then**- and **else**-parts. The local factoring transformations are far more powerful, and provide most of the improvements obtained by my factoring programs, but do not guarantee that the result is prime and irredundant. They will be described in a later paper.

By rearranging the if-then-else DAGs (converting them to a non-canonical form), we can increase the number of times the constants TRUE and FALSE appear, without increasing the number of nodes in the DAG, thus decreasing the size of the DAG by any of the measures. The transformations currently used are shown in Table 4.2. The transformations are first applied to the component parts, then to the expression as a whole. The last two transformations in Table 4.2 are applied only if the size of the DAG would be reduced, that is, if the new **then**- or **else**-part simplifies. For example, in the second-from-the-last transformation, if $b_b = c_b$, then $(\textbf{if } a \textbf{ then } b_b \textbf{ else } c_b) = b_b$, and the transformation can be applied. As another example, if $b_b = \text{TRUE}$ and $c_c = \text{FALSE}$ in the last transformation, then $(\textbf{if } a \textbf{ then } b_b \textbf{ else } c_c) = a$, and the transformation is acceptable for the size measure, but not for the count measure.

The Printform transformations may actually increase the size and count measures of an if-then-else DAG, because rearranging the **then**-part may reduce the amount of sharing with the **else**-part. For example, the canonical form for $(c+d)(dg+e+f)$ is $(\textbf{if } c \textbf{ then } (\textbf{if } d \textbf{ then } e+f+g \textbf{ else } e+f)$ $\textbf{else } d(e+f+g))$, which has size 13, count 10, and pcount 13. After applying the Printform transformations, we get $(\textbf{if } c \textbf{ then } dg+e+f \textbf{ else } d(e+f+g))$, which has size 16, count 10, and pcount 10. The pcount measure, which estimates the complexity of the printed form, is always reduced by the transformations, as one would expect from transformations originally intended for improving printing. The Printform transformation is still valuable as a factoring tool, as it enables the more powerful local factoring transformations to find the factoring $(c+d)(dg+e+f)$.

Table 4.3 shows the sizes of the results of applying the Printform transformations to the canonical forms for the examples in Table 4.1.

The procedure that does the transformations makes sure that Conditions 2, 3, 4, 5, and 7 are not violated, but allows Conditions 1 and 6 to be violated. Note that the second set of transformations (those involving 4 subDAGs) are simply ways to maintain Condition 7.

Condition 1 is not removed entirely, but is weakened—variables in the **if**-part are not necessarily earlier in the variable ordering than the variables in the **then**- and **else**-parts, but are disjoint from them. The transformations can re-order the variables differently in the **then**- and **else**-parts, and are thus potentially more powerful than simply re-ordering variables (a technique suggested by Randal Bryant [Bry86, page 26]). An exponential algorithm for finding the best ordering for BDD's is given in [FS87]. A combination approach using transformations and heuristics for doing complete variable re-ordering of if-then-else DAGs has yielded the most powerful factoring techniques, as will be reported in a later paper.

The violation of Condition 6 is illustrated by

$$(\textbf{if } \neg a \textbf{ then } (\textbf{if } b \textbf{ then } c \textbf{ else } \neg c) \textbf{ else } (\textbf{if } (\textbf{if } b \textbf{ then } c \textbf{ else } \neg c) \textbf{ then } d \textbf{ else } \neg d)),$$

Existing expression		Transformed expression
Transformations involving 3 subDAGs:		
$(\textbf{if}\ a\ \textbf{then}\ (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ \mathrm{FALSE})\ \textbf{else}\ b_a)$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ a\ \textbf{then}\ b_b\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ b_a\ \textbf{else}\ \mathrm{FALSE})$
$(\textbf{if}\ a\ \textbf{then}\ (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ \neg b_b)\ \textbf{else}\ b_a)$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ a\ \textbf{then}\ b_b\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ b_a\ \textbf{else}\ \neg b_a)$
$(\textbf{if}\ a\ \textbf{then}\ (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ \mathrm{TRUE})\ \textbf{else}\ \neg b_a)$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ a\ \textbf{then}\ \neg b_b\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ \neg b_a\ \textbf{else}\ \mathrm{TRUE})$
$(\textbf{if}\ a\ \textbf{then}\ (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ \neg b_b)\ \textbf{else}\ \neg b_a)$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ a\ \textbf{then}\ \neg b_b\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ \neg b_a\ \textbf{else}\ b_a)$
$(\textbf{if}\ a\ \textbf{then}\ c_a\ \textbf{else}\ (\textbf{if}\ c_a\ \textbf{then}\ c_b\ \textbf{else}\ \mathrm{FALSE}))$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ \neg a\ \textbf{then}\ c_b\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ c_a\ \textbf{else}\ \mathrm{FALSE})$
$(\textbf{if}\ a\ \textbf{then}\ c_a\ \textbf{else}\ (\textbf{if}\ c_a\ \textbf{then}\ c_b\ \textbf{else}\ \neg c_b))$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ \neg a\ \textbf{then}\ c_b\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ c_a\ \textbf{else}\ \neg c_a)$
$(\textbf{if}\ a\ \textbf{then}\ \neg c_a\ \textbf{else}\ (\textbf{if}\ c_a\ \textbf{then}\ c_b\ \textbf{else}\ \mathrm{TRUE}))$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ \neg a\ \textbf{then}\ \neg c_b\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ c_a\ \textbf{else}\ \mathrm{TRUE})$
$(\textbf{if}\ a\ \textbf{then}\ \neg c_a\ \textbf{else}\ (\textbf{if}\ c_a\ \textbf{then}\ c_b\ \textbf{else}\ \neg c_b))$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ \neg a\ \textbf{then}\ \neg c_b\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ \neg c_a\ \textbf{else}\ c_a)$
Transformations involving 4 subDAGs:		
$(\textbf{if}\ a\ \textbf{then}\ (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ b_c)\ \textbf{else}\ b_c)$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ a\ \textbf{then}\ b_a\ \textbf{else}\ \mathrm{FALSE})\ \textbf{then}\ b_b\ \textbf{else}\ b_c)$
$(\textbf{if}\ a\ \textbf{then}\ (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ b_c)\ \textbf{else}\ b_b)$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ a\ \textbf{then}\ b_a\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ b_b\ \textbf{else}\ b_c)$
$(\textbf{if}\ a\ \textbf{then}\ c_b\ \textbf{else}\ (\textbf{if}\ c_a\ \textbf{then}\ c_b\ \textbf{else}\ c_c))$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ \neg a\ \textbf{then}\ c_a\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ c_b\ \textbf{else}\ c_c)$
$(\textbf{if}\ a\ \textbf{then}\ c_c\ \textbf{else}\ (\textbf{if}\ c_a\ \textbf{then}\ c_b\ \textbf{else}\ c_c))$	$\Longrightarrow$	$(\textbf{if}\ (\textbf{if}\ \neg a\ \textbf{then}\ c_a\ \textbf{else}\ \mathrm{FALSE})\ \textbf{then}\ c_b\ \textbf{else}\ c_c)$
Transformations applied only if result is simpler:		
$(\textbf{if}\ a\ \textbf{then}\ (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ b_c)\ \textbf{else}\ (\textbf{if}\ b_a\ \textbf{then}\ c_b\ \textbf{else}\ c_c))$	$\Longrightarrow$	$(\textbf{if}\ b_a\ \textbf{then}\ (\textbf{if}\ a\ \textbf{then}\ b_b\ \textbf{else}\ c_b)\ \textbf{else}\ (\textbf{if}\ a\ \textbf{then}\ b_c\ \textbf{else}\ c_c))$
$(\textbf{if}\ a\ \textbf{then}\ (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ b_c)\ \textbf{else}\ (\textbf{if}\ \neg b_a\ \textbf{then}\ c_b\ \textbf{else}\ c_c))$	$\Longrightarrow$	$(\textbf{if}\ b_a\ \textbf{then}\ (\textbf{if}\ a\ \textbf{then}\ b_b\ \textbf{else}\ c_c)\ \textbf{else}\ (\textbf{if}\ a\ \textbf{then}\ b_c\ \textbf{else}\ c_b))$

Table 4.2: The Printform transformations, which do a crude job of factoring if-then-else DAGs. Each transformation decreases the complexity of the if-then-else DAG, using the *pcount* measure.

which is transformed to

$$(\textbf{if}\ (\textbf{if}\ a\ \textbf{then}\ d\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ (\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ \neg c)\ \textbf{else}\ (\textbf{if}\ \neg b\ \textbf{then}\ c\ \textbf{else}\ \neg c)).$$

To satisfy Condition 6, the result should be

$$(\textbf{if}\ (\textbf{if}\ (\textbf{if}\ a\ \textbf{then}\ d\ \textbf{else}\ \mathrm{TRUE})\ \textbf{then}\ b\ \textbf{else}\ \neg b)\ \textbf{then}\ c\ \textbf{else}\ \neg c).$$

The problem occurs only with exclusive-or functions, and amounts to choosing left- or right-associativity for the exclusive-or. Although the violation of Condition 6 can be viewed as a bug in the current set of transformations, it does not appear to have any consequences in the current applications. If the transformations are to be used for many purposes, it may become important to fix this bug.

The set of transformations would be more useful if they had the *Church-Rosser* property—that is, when no more transformations can be applied to the DAG, the result is the same independent of

expression	canonical form			Printform		
	size	count	pcount	size	count	pcount
$abcd$	7	4	4	7	4	4
$a+b+c+d$	7	4	4	7	4	4
$a \oplus b \oplus c \oplus d$	10	10	4	10	10	4
$ad+be+cf$	29	24	25	11	6	6
$a(c+d+e+g)+b(c+d+e+f)+$ $c(e+f)+d(f+g)$	32	24	30	36	23	23
$(a+b+c+d)(\neg a+\neg b+\neg c+\neg d)$	12	8	8	12	8	8
$(b+c)(a(e+g)(\neg e+f+\neg g))+h+i)+$ $d(e+g)(\neg e+f+\neg g)$	27	20	37	21	14	19
$a(c+d+e+g)+b(c+d+e+f)+$ $(c+d)(dg+e+f)$	30	23	31	38	24	24

Table 4.3: The change in size that results from applying the Printform transformations to the canonical forms of several expressions. The variable ordering used is alphabetical.

the order in which the transformations were applied. If the transformations had the Church-Rosser property, then the transformed expressions would also be a canonical form, but one which is smaller than the the if-then-else DAGs with fixed variable order. Unfortunately, counter-examples are easily found. For example,

$$(\textbf{if } a \textbf{ then } (\textbf{if } b \textbf{ then } (\textbf{if } c \textbf{ then } d \textbf{ else } \text{FALSE}) \textbf{ else } c) \textbf{ else } (\textbf{if } \neg b \textbf{ then } c \textbf{ else } e))$$

can be converted in two different ways, depending whether the left subDAG is converted using the first transformation, or the whole DAG is converted using the last transformation. Transforming the left subDAG, we get

$$(\textbf{if } a \textbf{ then } (\textbf{if } (\textbf{if } b \textbf{ then } d \textbf{ else } \text{TRUE}) \textbf{ then } c \textbf{ else } \text{FALSE}) \textbf{ else } (\textbf{if } \neg b \textbf{ then } c \textbf{ else } e)),$$

which cannot be transformed further, and transforming at the root first we get

$$(\textbf{if } b \textbf{ then } (\textbf{if } a \textbf{ then } (\textbf{if } c \textbf{ then } d \textbf{ else } \text{FALSE}) \textbf{ else } e) \textbf{ else } c),$$

which also cannot be further transformed. One interesting direction for further research is to find a more complete set of transformations that has the Church-Rosser property.

5 Conclusions and current work

This paper has presented a new way of looking at if-then-else DAGs, based on two-cuts in a corresponding binary decision diagram. The relationship between the two representations leads to a natural canonical form for if-then-else DAGs.

If-then-else DAGs are attractive for VLSI CAD work, because they provide sharing of subexpressions, a compact canonical form for tautology checking, easy manipulation for all the standard Boolean operators, straightforward representation of networks of gates, and factoring by simple transformations of the DAG.

Several implementations of binary decision diagrams and if-then-else DAGS have been written, and used for switch-level verification tasks [Kar86] and for multi-level logic minimization. Current work focusses on using transformations of if-then-else DAGs to do logic minimization. In addition to the Printform transformations described in this paper, more powerful transformations, which violate even the weakened version of Condition 1, have been implemented. These transformations, together with heuristics for finding good variable orderings for arbitrary if-then-else DAGs, have given some promising results, but are still under intensive development.

Other researchers have found that using don't-care sets provides significant improvement in multi-level minimizers. A transformation that simplifies expressions modulo don't-care sets has been implemented and is being debugged and evaluated. The transformation is intended to guarantee that an if-then-else DAG is prime and irredundant with respect to the don't-care set—that is, that any change in meaning occurs only for cases that we know are unimportant.

Several size measures, including the three presented in this paper, are being evaluated to determine which ones provide the best predictors of circuit size and delay after technology mapping. Ideally, we would like to generate an appropriate measuring function directly from a description of the technology.

Because sum-of-products representations are so important for PLA implementation, we are working on methods to convert prime, irredundant if-then-else DAGs into prime, irredundant sum-of-products representations.

Acknowledgements

I would like to thank Liisa Raiha, for implementing a version of Bryant's canonical form; Dipen Moitra, for examining the graph theoretic properties of binary decision diagrams, particularly identifying dominators as an interesting feature; Habib Krit, for studying my undocumented code for if-then-else DAGs and suggesting several improvements, and for reading early drafts of this paper; and Martine Schlag, for providing several useful comments on a draft of this paper.

References

[Ake77] Sheldon B. Akers. On the specification and analysis of large digital functions. In *7th Annual International Conference on Fault Tolerant Computing*, pages 88–93, Los Angeles, CA, 28–30 June 1977. IEEE.

[Ake78] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[Ber88] UC Berkeley. Berkeley logic interchange format (BLIF). In Rick Spickelmier, editor, *Oct Tools Distribution 2.1*. Electronics Research Laboratory, University of California, Berkeley, 25 March 1988.

[BHMS84] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[Bra87] Robert K. Brayton. Algorithms for multi-level logic synthesis and optimization. In G. De Micheli, Alberto Sangiovanni-Vincentelli, and P. Antognetti, editors, *Design Systems for VLSI Circuits—Logic Synthesis and Silicon Compilation*, pages 197–247. Martinus Nijhoff Publishers, 1987.

[Bry86] Randal Everitt Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry87] Randal Everitt Bryant. Algorithmic aspects of symbolic switch network analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(4):618–633, July 1987.

[DGR+87] Ewald Detjens, Gary Gannot, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert Wang. Technology mapping in MIS. In *IEEE International Conference on Computer-Aided Design ICCAD-87*, pages 116–119, Santa Clara, CA, 9–12 November 1987. IEEE Computer Society Press.

[FS87] Steven J. Friedman and Kenneth J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *ACM IEEE 24^{th} Design Automation Conference Proceedings*, pages 348–355, Miami Beach, FL, 28 June–1 July 1987.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, CA, 1979.

[Kar86] Kevin Karplus. Exclusion constraints, a new application of graph algorithms to VLSI design. In *4th MIT Conference on Advanced Research in VLSI*, pages 123–139, Cambridge, MA, April 7–9 1986.

[KM90] Kevin Karplus and Dipen Moitra. Using dominators for Boolean function manipulation. Technical report, Department of Computer Science, Cornell University, Ithaca, NY 14853, forthcoming, probably 1990.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, July 1959.

[NO79] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.