**Algorithms for reduced OBDDs**

**The algorithm reduce, apply, restrict, exists**

The reductions C1–C3 are at the core of any serious use of OBDDs, for whenever we construct a BDD we will want to convert it to its reduced form. In this section, we describe an algorithm reduce which does this efficiently for ordered BDDs.

If the ordering of B is [x1, x2,...,xl], then B has at most l + 1 layers. The algorithm reduce now traverses B layer by layer in a bottom-up fashion, beginning with the terminal nodes. In traversing B, it assigns an integer label id(n) to each node n of B, in such a way that the subOBDDs with root nodes n and m denote the same boolean function if, and only if, id(n) equals id(m).

Since reduce starts with the layer of terminal nodes, it assigns the first label (say #0) to the first 0-node it encounters. All other terminal 0-nodes denote the same function as the first 0-node and therefore get the same label (compare with reduction C1). Similarly, the 1-nodes all get the next label, say #1.

Now let us inductively assume that reduce has already assigned integer labels to all nodes of a layer > i (i.e. all terminal nodes and xj -nodes with j>i). We describe how nodes of layer i (i.e. xi-nodes) are being handled.

**Given an xi-node n, there are three ways in which it may get its label:**

* If the label id(lo(n)) is the same as id(hi(n)), then we set id(n) to be that label. That is because the boolean function represented at n is the same function as the one represented at lo(n) and hi(n). In other words, node n performs a redundant test and can be eliminated by reduction C2.
* If there is another node m such that n and m have the same variable xi, and id(lo(n)) = id(lo(m)) and id(hi(n)) = id(hi(m)), then we set id(n) to be id(m). This is because the nodes n and m compute the same boolean function (compare with reduction C3).
* Otherwise, we set id(n) to the next unused integer label.

**The algorithm** **apply**

Another procedure at the heart of OBDDs is the algorithm apply. It is used to implement operations on boolean functions such as +, · , ⊕ and complementation (via f ⊕ 1). Given OBDDs Bf and Bg for boolean formulas f and g, the call apply (op, Bf , Bg) computes the reduced OBDD of the boolean formula f op g, where op denotes any function from {0, 1}×{0, 1} to {0, 1}

The intuition behind the apply algorithm is fairly simple. The algorithm operates recursively on the structure of the two OBDDs:

1. let v be the variable highest in the ordering (=leftmost in the list) which occurs in Bf or Bg.

2. split the problem into two subproblems for v being 0 and v being 1 and solve recursively;

3. at the leaves, apply the boolean operation op directly.

The result will usually have to be reduced to make it into an OBDD. Some reduction can be done ‘on the fly’ in step 2, by avoiding the creation of a new node if both branches are equal (in which case return the common result), or if an equivalent node already exists (in which case, use it).

**The algorithm restrict**

Given an OBDD Bf representing a boolean formula f, we need an algorithm restrict such that the call restrict(0, x, Bf ) computes the reduced OBDD representing f[0/x] using the same variable ordering as Bf . The algorithm for restrict(0, x, Bf ) works as follows. For each node n labelled with x, incoming edges are redirected to lo(n) and n is removed. Then we call reduce on the resulting OBDD. The call restrict (1, x, Bf ) proceeds similarly, only we now redirect incoming edges to hi(n).

**The algorithm exists**

A boolean function can be thought of as putting a constraint on the values of its argument variables. For example, the function x + (y · z) evaluates to 1 only if x is 1; or y is 0 and z is 1 – this is a constraint on x, y, and z.

It is useful to be able to express the relaxation of the constraint on a subset of the variables concerned. To allow this, we write ∃x. f for the boolean function f with the constraint on x relaxed. Formally, ∃x. f is defined as f[0/x] + f[1/x]; that is, ∃x. f is true if f could be made true by putting x to 0 or to 1. Given that ∃x. f def = f[0/x] + f[1/x] the exists algorithm can be implemented in terms of the algorithms apply and restrict as

**apply (+, restrict (0, x, Bf ), restrict (1, x, Bf ))**