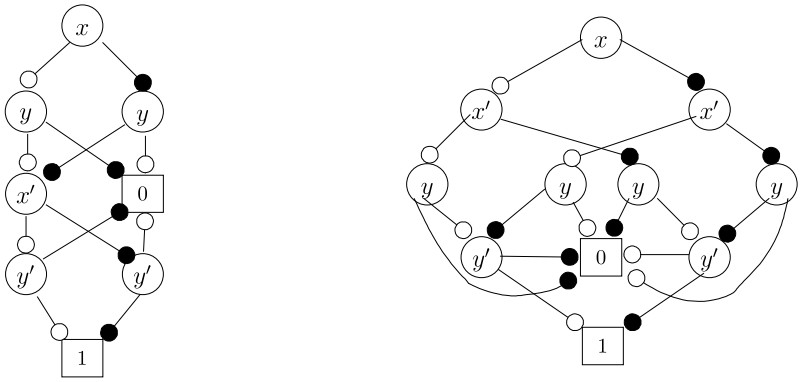

Ordering: $x < y < x' < y'$ *Ordering:* $x < x' < y < y'$

Figure 3.23: Two ROBDDs for $(x = y) \wedge (x' = y')$

$Mult_j$ is guaranteed to grow exponentially with k. More precisely, the following *lower bound* result has been established: there exists an index $0 \leq j < 2k$ such that the ROBDD for $Mult_j$ has at least $2^{k/8}$ vertices.

Shared Data Structure for ROBDDs

Let us turn our attention to implementing regions as ROBDDs. Every vertex of a ROBDD is a ROBDD rooted at that vertex. This suggests that a ROBDD can be represented by an index to a global data structure that stores vertices of all the ROBDDs such that no two vertices are isomorphic. There are two significant advantages to this scheme, as opposed to maintaining each ROBDD as an individual data structure. First, checking isomorphism, and hence equivalence, corresponds to comparing indices and does not require traversal of the ROBDDs. Second, two non-isomorphic ROBDDS may have isomorphic subgraphs and, hence, can share vertices.

Let V be an ordered set of k Boolean variables. The type of ROBDDs is `bdd`, which is either a Boolean constant (denoting a terminal vertex) or a pointer to an entry in the global data structure *BDDPool*. The type of *BDDPool* is `set(bddnode)`, and it stores the (internal) vertices of ROBDDs. An internal vertex records a variable label, which ranges over the type $\texttt{nat}[1, k]$ containing the numbers $\{1, \ldots k\}$, and a left and a right pointer, each of which has type `bdd`. Thus, the vertices of ROBDDs have type `bddnode`, which equals $\texttt{nat}[1, k] \times \texttt{bdd} \times \texttt{bdd}$. The type `bddnode` supports the following operations:

- The operation $\texttt{Label}(u)$, for an internal vertex u, returns the first component of u, which is the number of the variable labeling u.
- The operation $\texttt{Left}(u)$, for an internal vertex u, returns the second component of u, which is either a Boolean constant or a pointer to the left

```
function AddVertex
Input: Variable label j in nat[1, k], ROBDDs B_0, B_1 of type bdd.
Output: ROBDD B such that f(B) is equivalent to (¬ x_j ∧ f(B_0)) ∨
    (x_j ∧ f(B_1)).
if B_0 = B_1 then return B_0;
if Contains((j, B_0, B_1), BDDPool) = 0 then
  Insert((j, B_0, B_1), BDDPool);
return Index((j, B_0, B_1)).
```

Figure 3.24: Creating ROBDD Vertices

successor of u.

- The operation `Right`(u), for an internal vertex u, returns the third component of u, which is either a Boolean constant or a pointer to the right successor of u.

The type `set(bddnode)`, besides usual operations such as `Insert` and `Contains`, also supports

- For an internal vertex u in *BDDPool*, `Index`(u) returns a pointer to u.
- For a pointer B, the operation *BDDPool*$[B]$ returns the vertex that B points to.

For such a representation, given a pointer B of type `bdd`, we write $f(B)$ to denote the Boolean function associated with the ROBDD that B points to. To avoid duplication of isomorphic nodes while manipulating ROBDDs, it is necessary that new vertices are created using the function *AddVertex* of figure 3.24. If no two vertices in the global set *BDDPool* are isomorphic before an invocation of the function *AddVertex*, then even after the invocation, no two vertices in *BDDPool* are isomorphic.

As an illustrative example, let us examine the snapshot of the global data structure *BDDPool* shown in figure 3.25. Each row shows an internal vertex stored in this data structure. For example, the ROBDD B_0 points to a vertex labeled with variable x_4 whose left successor is the terminal vertex 0 and right successor is the terminal vertex 1, and the ROBDD B_4 points to a vertex labeled with variable x_1 whose left successor is the terminal vertex 0 and right successor is the ROBDD B_2. The Boolean functions corresponding to each of the vertices are listed below:

$$\begin{aligned} f(B_0) &= x_4, \\ f(B_1) &= x_2 \vee x_4, \\ f(B_2) &= \neg x_3 \vee x_4, \\ f(B_3) &= x_1 \wedge (x_2 \vee x_4), \end{aligned}$$

Index	Label	Left	Right
B_0	4	0	1
B_1	2	B_0	1
B_2	3	1	B_0
B_3	1	0	B_1
B_4	1	0	B_2

Figure 3.25: Illustrative Snapshot of the Data Structure *BDDPool*

$$f(B_4) \quad = \quad x_1 \ \wedge \ (\neg\, x_3 \ \vee \ x_4).$$

Observe that the ROBDDs B_3 and B_4 share the ROBDD B_0.

Operations on ROBDDs

To construct a ROBDD representation of a given Boolean formula and implement the primitives of the symbolic reachability algorithm, we need a way to compute conjunctions and disjunctions of ROBDDs. We give a recursive algorithm for obtaining conjunction of ROBDDs. The algorithm is shown in figure 3.26.

Consider two ROBDDs, B and B', and suppose we wish to compute the conjunction $f(B) \wedge f(B')$. If one of them is a Boolean constant, then the result can be determined immediately. For instance, if B is the terminal constant 0, then the conjunction is also the terminal constant 0. If B is the terminal constant 1, then the conjunction is equivalent to $f(B')$, and thus the algorithm can return B'. Also, note that when both the ROBDDs are the same, we can use the fact $f \wedge f$ always equals f, and thus the result coincides with the input argument.

The interesting case is when both ROBDDs are pointers to distinct internal vertices, say u and u', respectively. Let j be the minimum of the indices labeling u and u'. Then x_j is the least variable that the function $f(u) \wedge f(u')$ can depend on. The label of the root of the conjunction is j, the left successor is the ROBDD for $(f(u) \wedge f(u'))[x_j \mapsto 0]$, and the right successor is the ROBDD for $(f(u) \wedge f(u'))[x_j \mapsto 1]$. Let us consider the left successor. Observe the equivalence

$$(f(u) \wedge f(u'))[x_j \mapsto 0] \ \equiv \ f(u)[x_j \mapsto 0] \ \wedge \ f(u')[x_j \mapsto 0].$$

If u is labeled with j, the ROBDD for $f(u)[x_j \mapsto 0]$ is the left successor of u. If the label of u exceeds j, then the function $f(u)$ does not depend on x_j, and the ROBDD for $f(u)[x_j \mapsto 0]$ is u itself. The ROBDD for $f(u')[x_j \mapsto 0]$ is computed similarly, and then the function *Conj* is applied recursively to compute the conjunction according to the expression above.

Let us apply this scheme to compute the conjunction of the ROBDDs B_3 and B_4 from figure 3.25. The vertex corresponding to B_3 has label x_1, left successor 0, and right successor B_1, while the vertex corresponding to B_4 has label x_1, left successor 0, and right successor B_2. As a result,

$$\mathit{Conj}(B_3, B_4) \;=\; \mathit{AddVertex}(1, \mathit{Conj}(0, 0), \mathit{Conj}(B_1, B_2)).$$

The call $\mathit{Conj}(0, 0)$ returns 0 using the rule for the constant ROBDDs. To compute the conjunction of B_1 and B_2, the algorithm examines the corresponding vertices: the vertex corresponding to B_1 has label x_2, left successor B_0, and right successor 1, while the vertex corresponding to B_2 has a higher label x_3. This leads to:

$$\mathit{Conj}(B_1, B_2) \;=\; \mathit{AddVertex}(2, \mathit{Conj}(B_0, B_2), \mathit{Conj}(1, B_2)).$$

This generates two recursive calls to *Conj* again: the second call $\mathit{Conj}(1, B_2)$ returns immediately with the answer B_2 using rules for simplification when one of the arguments is a constant. The first call $\mathit{Conj}(B_0, B_2)$ requires examination of the corresponding vertices: the vertex corresponding to B_0 has label x_4, while the vertex corresponding to B_2 has label x_3, left successor 1, and right successor B_0. This leads to:

$$\mathit{Conj}(B_0, B_2) \;=\; \mathit{AddVertex}(3, \mathit{Conj}(B_0, 1), \mathit{Conj}(B_0, B_0)).$$

In this case, both the recursive calls to `Conj` return immediately: $\mathit{Conj}(B_0, 1)$ returns B_0 and $\mathit{Conj}(B_0, B_0)$ also returns B_0 using the rule for the conjunction of identical ROBDDs. As a result, a call is made to $\mathit{AddVertex}(3, B_0, B_0)$. This does not create a new vertex since the reduction rules do not allow both left and right successors to be the same. The call $\mathit{AddVertex}(3, B_0, B_0)$ simply returns B_0:

$$\mathit{Conj}(B_0, B_2) \;=\; \mathit{AddVertex}(3, B_0, B_0) \;=\; B_0.$$

Now, $\mathit{Conj}(B_1, B_2)$ calls $\mathit{AddVertex}(2, B_0, B_2)$. The data structure *BDDPool* does not contain a vertex with label 2, left successor B_0, and right successor B_2. As a result, *AddVertex* will create a new entry $(2, B_0, B_2)$, with the index B_5:

$$\mathit{Conj}(B_1, B_2) \;=\; \mathit{AddVertex}(2, B_0, B_2) \;=\; B_5.$$

Finally, $\mathit{Conj}(B_3, B_4)$ calls $\mathit{AddVertex}(1, 0, B_5)$. Again, *BDDPool* does not contain such a vertex, so a new entry, indexed by B_6, is created and is the desired result, namely, the ROBDD representation of the conjunction of the functions represented by B_3 and B_4:

$$\mathit{Conj}(B_3, B_4) \;=\; \mathit{AddVertex}(1, 0, B_5) \;=\; B_6.$$

Avoiding Recomputation

The recursive algorithm described so far may call the function *Conj* repeatedly with the same two arguments. To avoid unnecessary computation, a table is

```
Input: bdd B, B'.
Output: bdd B'' such that f(B'') is equivalent to f(B) ∧ f(B').

table[(bdd × bdd) × bdd] Done = EmptyTable

return Conj(B, B').

bdd Conj(bdd B, B')
  bddnode u, u'; bdd B'', B0, B1, B'0, B'1; nat[1, k] j, j'

  if ( B = 0 ∨ B' = 1 ) then return B;
  if ( B = 1 ∨ B' = 0 ) then return B';
  if B = B' then return B;
  if Done[(B, B')] ≠⊥ then return Done[(B, B')];
  if Done[(B', B)] ≠⊥ then return Done[(B', B)];
  u := BDDPool[B]; u' := BDDPool[B'];
  j := Label(u); B0 := Left(u); B1 := Right(u);
  j' := Label(u'); B'0 := Left(u'); B'1 := Right(u');
  if j = j' then B'' := AddVertex(j, Conj(B0, B'0), Conj(B1, B'1));
  if j < j' then B'' := AddVertex (j,Conj(B0,B'),Conj(B1,B'));
  if j > j' then B'' := AddVertex (j',Conj(B,B'0),Conj(B,B'1));
  Done[(B, B')] := B'';
  return B''.
```

Figure 3.26: Algorithm for Taking Conjunction of ROBDDs

used to store the arguments and the corresponding result of each invocation of *Conj*. When *Conj* is invoked with input arguments B and B', it first consults the table to check whether the conjunction of $f(B)$ and $f(B')$ was previously computed. The actual recursive computation is performed only the first time, and the result is stored into the table.

A *table* data structure stores values that are indexed by keys. If the type of values stored is `value` and the type of the indexing keys is `key`, then the type of the table is `table`[`key` × `value`]. This data type supports the retrieval and update operations like arrays: $D[k]$ is the value stored in the table D with the key k, and the assignment $D[k] := m$ updates the value stored in D for the key k. The constant table `EmptyTable` has the default value $\perp$ stored with every key. Tables can be implemented as arrays or hash-tables. The table used by the algorithm uses a pair of ROBDDs as a key and stores ROBDDs as values.

Let us analyze the time complexity of the algorithm of figure 3.26. Suppose the ROBDD pointed to by B has n vertices and the ROBDD pointed to by B' has n' vertices. Let us assume that the implementation of the set *BDDPool* supports constant time membership tests and insertions and the table *Done* supports constant-time creation, access, and update. Then within each invocation of *Conj*, all the steps, apart from the recursive calls, are performed in constant time. Thus, the time complexity of the algorithm is the same, within a constant