We start with a binary decision diagram. For example, following is a decision diagram (tree) for the formula \((x_1 = y_1) \land (x_2 = y_2)\):

```
  x1
  /  \
 y1 y1
  /  \
 y2 y2
  /  \
 y2 y2
  /  \
 0 1 1 0
```

In general, it requires an exponential number of nodes.

**Optimize**

- **Identify** identical subgraphs.
- **Remove redundant** tests.

Yielding:

```
  x1
  /  \
 y1 y1
  /  \
 y2 y2
  /  \
 y2 y2
  /  \
 0 1
```

**Definitions**

A binary decision diagram BDD is a rooted, directed acyclic graph with

- One or two nodes of out-degree zero (leaves) labeled 0 or 1, and
- A set of variable nodes \(v\) of out-degree 2. The two outgoing edges are given by the functions \(\text{low}(v)\) and \(\text{high}(v)\). A variable \(\text{var}(v)\) is associated with each node.

A BDD is ordered (OBDD) if the variables respect a given linear order \(x_1 < x_2 < \cdots < x_n\) on all paths through the graph. An OBDD is reduced (ROBDD) if it satisfies:

- **Uniqueness** – no two distinct nodes are the roots of isomorphic subgraphs.
- **No redundant tests** – \(\text{low}(v) \neq \text{high}(v)\) for all nodes \(v\) in the graph.

For simplicity, we will refer to ROBDD simply as BDDs.
**Canonicity**

**Claim 12.** For every function \( f : \text{Bool}^n \to \text{Bool} \) and variable ordering \( x_1 < x_2 < \cdots < x_n \), there exists exactly one BDD representing this function.

**Sensitivity to Variable Ordering**

The complexity of BDD representation is very sensitive to the variable ordering. For example, the BDD representation of \( (x_1 = y_1) \land (x_2 = y_2) \) under the variable ordering \( x_1 < x_2 < y_1 < y_2 \) is:

```
x_1 x_2 y_1 y_1 y_1 y_1 y_1 y_1
```

**Implementation of BDD Packages**

**Types and Variables:**

- \( \text{node} = \text{naturals} \)
- \( \text{var_num} = \text{naturals} \)
- \( \text{node_rec} = \text{record of} \)
  - \( \text{var} : \text{var_num} \)
  - \( \text{low, high} : \text{node} \)
- \( \text{end_record} \)
- \( T : \text{node} \to \text{node_rec} \)
- \( H : \text{node_rec} \to \text{node} \cup \{ \bot \} \)

**Operations:**

- \( \text{init}(T) \) Initialize \( T \) to contain only 0 and 1
- \( u := \text{new}(T, i, \ell, h) \) allocate a new node \( u \), such that \( T(u) = \langle i, \ell, h \rangle \)
- \( \text{init}(H) \) initialize \( H \) to \( \bot \)

\( H \) is the inverse of \( T \). That is, \( H(T(u)) = u \), for every \( u \in \text{dom}(T) \).

We will write \( \text{var}(u) \), \( \text{low}(u) \), \( \text{high}(u) \), and \( H(i, \ell, h) \) as abbreviations for \( T(u).\text{var}, T(u).\text{low}, T(u).\text{high}, \) and \( H(\langle i, \ell, h \rangle) \).

**Internal Representation**

```
<table>
<thead>
<tr>
<th>T : u → ⟨i, ℓ, h⟩</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
```
Making or Retrieving a node_id

Function $Mk(i : var\_num; \ell, h : node) : node$

1: if $\ell = h$ then return $\ell$
2: if $H(i, \ell, h) \neq \bot$ then return $H(i, \ell, h)$
3: $u := new(i, \ell, h)$
4: $H(i, \ell, h) := u$
5: return $u$

Restriction (Substitution)

Function $Rest(u : node; j : var\_num; b : Bool) : node$

function $res(u : node) : node$

1: if $var(u) > j$ then return $u$
2: if $var(u) < j$ then
3: return $Mk(var(u), res(low(u)), res(high(u)))$
4: (*var(u) = j*) if $b = 0$ then return $low(u)$
5: else return $high(u)$
6: end
7: return $res(u)$

Restriction is the same as substitution. We denote by $t[x \mapsto b]$ the result of substituting $b$ for $x$ in assertion $t$.

Applying a Binary Boolean Operation to Two BDD’s

Let $op : Bool \times Bool \rightarrow Bool$ be a binary boolean operation. The following function uses the auxiliary dynamic array $G : node \times node \rightarrow node$.

Function $Apply(op : u_1, u_2 : node) : node$

$G := \bot$

function $Apply(u_1, u_2 : node) : node$

1: if $G[u_1, u_2] \neq \bot$ then $u := G[u_1, u_2]$
2: else if $u_1 \in \{0, 1\} \land u_2 \in \{0, 1\}$ then $u := op(u_1, u_2)$
3: else if $var(u_1) = var(u_2)$ then
4: $u := Mk(var(u_1).App(low(u_1), low(u_2)), App(high(u_1), high(u_2)))$
5: else if $var(u_1) < var(u_2)$ then
6: $u := Mk(var(u_1), App(low(u_1), u_2), App(high(u_1), u_2))$
7: else (*var(u_1) > var(u_2)*)
8: $u := Mk(var(u_2), App(u_1, low(u_2)), App(u_1, high(u_2)))$
9: $G[u_1, u_2] := u$
10: end $Apply$
11: return $Apply(u_1, u_2)$

Quantification

Existential quantification can be computed, using the equivalence

$$\exists x : t \sim t[x \mapsto 0] \lor t[x \mapsto 1]$$

Universal quantification can be computed dually:

$$\forall x : t \sim t[x \mapsto 0] \land t[x \mapsto 1]$$
Application of BDD’s to Symbolic Model Checking

Let $V$ be the state variables for the FDS $D$. Taking a vocabulary $U = V \cup V'$, we represent the state formulas $\Theta$, $J$ for each $J \in \mathcal{J}$, $p_i$, $q_i$, for each $\langle p_i, q_i \rangle \in \mathcal{C}$, and the INV symbolic working variables $\text{new}$ and $\text{old}$ as BDD’s over $U$ which are independent of $V'$.

The transition relation $\rho$ is represented as a BDD over $U$ which may be fully dependent on both $V$ and $V'$.

All the boolean operations used in the INV algorithm can be implemented by the Apply function. Negation can be computed by $\neg t = t \oplus 1$, where $\oplus$ is sum modulo 2.

To check for equivalence such as $\text{old} = \text{new}$ we compute $t := (\text{old} \leftrightarrow \text{new})$ and then verify that the result is the singleton BDD $1$.

The existential pre-condition transformer is computed by

$$\rho \otimes \psi = \exists V' : \rho(V, V') \land \psi(V')$$

Priming an assertion $\psi$ is performed by

$$\text{prime}(\psi) = \exists V : \psi(V) \land V' = V$$