Morten Ib Nielsen

Binary decision diagrams

2. delsprojekt 2006

Contents

1 Introduction

Boolean functions are fundamental in Computer Science if not for anything else then because they are used to reason about and describe digital circuits in hardware. There are several ways to represent Boolean functions amongst others as propositional formulae special cases hereof being *Disjunctive Normal Forms* and *Conjunctive Normal Forms*, as ordered truth tables or as *Reduced Ordered Binary Decision Diagrams*. Each representation has its own characteristics pros and cons. Table 1 taken from [**?**, p. 361] gives a comparing overview of the representations mentioned above.

Table 1 Comparison of efficiency of Boolean representations

Representation	compact	test for satisfiability	validity	operators `and`	`or`	`negate`
Prop. formulae	often	hard	hard	easy	easy	easy
Formulae in DNF	sometimes	easy	hard	hard	easy	hard
Formulae in CNF	sometimes	hard	easy	easy	hard	hard
Ordered truth tables	never	hard	hard	hard	hard	hard
ROBDDs	often	easy	easy	medium	medium	easy

In table 1 *compact* means that the size of the representation is small compared to the number of Boolean variables. *Satisfiability* and *validity* of a Boolean formula is the question of whether there exists an assignment respectively whether every assignment of truth values to the Boolean variables in the formula makes it true. `and`, `or` plus `negate` are (the standard) operators that can be applied to Boolean formulae.

We see that ROBDDs outperform the other representations on average. A big part of the explanation is that Boolean functions have a canonical (unique) form with respect to a given variable ordering when they are represented as ROBDDs (this will be made precise in definition 2.8 and theorem 2.11). As a consequence equivalence testing is reduced to structural equality testing and if we implement ROBDDs using *maximal sharing* (formalized section 3.2) this is further reduced to comparing two pointers for equality.

The work I relate in this report is joint work with Fritz Henglein. It is based on a project proposal by Fritz and has been carried out under his supervision. The goal of the project was to design and implement a bdd package with a well defined interface both using existing techniques based on hashing as well as a hash free approach based on msd[1]. Further we have compared our implementations in practice and with respect to time and space complexities.

This work assumes knowledge of Functional Programming (Standard ML) and Propositional logic.

1.1 Roadmap

The remainder of this report is organized as follows. In section 2 we recap bdd theory. This is followed by design criteria in section 3 which lead to our implementations described in section 4 and 5. Finally we test our implementations in section 6 and we conclude in section 7.

[1]MultiSet Discrimination.

2 Basics

Boolean expressions also called *Propositional logic* are syntactic objects. Formally we can use the inference rules of e.g. *Natural Deduction* to manipulate Boolean expressions and show *sequents* like $\phi_1, \phi_2 \ldots, \phi_n \vdash \psi$, that is from premises $\phi_1, \phi_2 \ldots, \phi_n$ we can infer (prove using Natural Deduction) ψ. The semantics of Boolean expressions can be specified by giving truth tables for the Boolean connectives. This way we can give meaning to compound expressions. The semantic counterpart to sequents is called *semantic entailment*. If it is the case that whenever $\phi_1, \phi_2 \ldots, \phi_n$ evaluate to true then ψ evaluates to true, we write $\phi_1, \phi_2 \ldots, \phi_n \models \psi$, and we say that semantic entailment holds. It is easy to show the following theorem:

Theorem 2.1 (Soundness and Completeness)
Let $\phi_1, \phi_2 \ldots, \phi_n$ and ψ denote Boolean expressions then $\phi_1, \phi_2 \ldots, \phi_n \vdash \psi$ can be proved if and only if $\phi_1, \phi_2 \ldots, \phi_n \models \psi$ holds.

The theorem says that the syntactic and semantic worlds of Boolean expressions are equivalent. For that reason we can allow ourself to be less formal when we work with Boolean expressions and henceforth we won't make a strict distinction between syntax and semantics.

Definition 2.2

1. *A Boolean variable* x is a variable ranging over $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$. We will use $x_1, x_2, \ldots$ and $x, y, z, \ldots$ to denote Boolean variables.

2. *A Boolean expression* is any nonempty expression generated from the following bnf grammar:

$$t ::= x \mid \mathsf{T} \mid \mathsf{F} \mid \neg t \mid t \wedge t \mid t \vee t \mid t \Rightarrow t \mid t \Leftrightarrow t$$

   An always true Boolean expression is often called a *tautology*. In the same style an always false Boolean expression is often called an *absurdity*.

3. *A Boolean function* is a function $f : \mathbb{B}^n \rightarrow \mathbb{B}$, where $n \geq 0$. n is called the *arity* of f. Functions of arity 0 are called *constants* (the only Boolean constants are T and F). In the case $n = 2$ we will often use infix notation as is commonly done. We write $f(x_1, x_2, \ldots, x_n)$ to show that a representation of f only depends on the Boolean variables $x_1, x_2, \ldots x_n$. The notion of tautology and absurdity extends to Boolean functions in the obvious way.

We will use the standard conventions regarding binding priorities. If priorities decrease from left to right we have: $\neg, \wedge, \vee, \Leftrightarrow, \Rightarrow$. Also we omit the prefix *Boolean* when it is clear from the context, that it is a Boolean variable, expression or function we have in mind. Next we define the meaning of the Boolean connectives used in the grammar above.

Definition 2.3 (Boolean connectives)
The boolean connectives $\wedge, \vee, \Rightarrow$ and $\Leftrightarrow$ are binary functions and $\neg$ is an unary function. They are given by the following truth tables:

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\Rightarrow$	0	1
0	1	1
1	0	1

$\Leftrightarrow$	0	1
0	1	0
1	0	1

$\neg$	0	1
	1	0

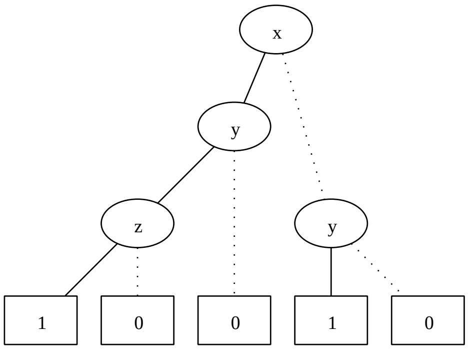


Figure 1: BDT representing the Boolean expression $y \wedge (x \wedge z \vee \neg x)$.

2.1 Binary decision trees and diagrams

In a binary decision tree, henceforth BDT, non-terminals are labeled with Boolean variables and terminals are labeled either T or F. A BDT where all labels on non-terminals belong to $x_1, x_2, \ldots x_n$ represents a unique Boolean function in the following way: Let $v = (v_1, v_2, \ldots, v_n) \in \mathbb{B}^n$ be a truth assignment to $x_1, x_2, \ldots, x_n$ such that $x_i = v_i, 1 \leq i \leq n$. Now start at the root node of the BDT and assume that it is labeled x_j. If $x_j = \mathsf{T}$ we select the root node of the left subtree else (if $x_j = \mathsf{F}$) we select the root node of the right subtree. Now two possibilities exist either the selected node is a non-terminal or the selected node is a terminal. In the first case we repeat the process examining the label of the selected non-terminal and in the latter case the label of the terminal is the function value on the input v. An example is given in figure 1.
Binary decision trees represent boolean expressions in *If-then-else Normal Form*, henceforth INF. This is made precise in the following definition.

Definition 2.4
We define the *if-then-else-operator*, $\mathsf{if}\ x\ t_1\ t_2$, by:

$$\mathsf{if}\ x\ t_1\ t_2 = (x \wedge t_1) \vee (\neg x \wedge t_2)$$

A Boolean expression built using the following bnf grammar is said to be in *If-then-else Normal Form.*

$$\begin{array}{rcl} x & \in & \textit{Boolean variables} \\ i & ::= & \mathsf{if}\ x\ i\ i \mid \mathsf{T} \mid \mathsf{F} \end{array}$$

The following observation is known as *Shannon expansion.*

Lemma 2.5 (Shannon expansion)
For all Boolean expressions t and all Boolean variables x we have:

$$t \equiv \mathsf{if}\ x\ t[\mathsf{T}/x]\ t[\mathsf{F}/x]$$

This leads to the following important theorem.

Theorem 2.6
Any Boolean expression, t, is equivalent with an expression in INF.

PROOF:
The proof is by induction on the number of variables, n, in t. If $n = 0$ then t is equivalent with either T or F which are in INF. Suppose that the number of variables in t is $n > 0$ and assume that any Boolean expression with fewer than n variables is equivalent with an expression in INF. Let x be one of the variables in t and construct $t_1 = t[\mathsf{T}/x]$, $t_2 = t[\mathsf{F}/x]$. t_1 and t_2 are Boolean expressions containing $n-1$ variables. By induction there exist t_1' and t_2' in INF such that $t_1 \equiv t_1'$ and $t_2 \equiv t_2'$. Using Shannon expansion we get:

$$t \equiv \mathsf{if}\ x\ t_1\ t_2 \equiv t', \text{ where } t' = \mathsf{if}\ x\ t_1'\ t_2'$$

Thus we have shown that $t \equiv t'$ and t' is in INF. This concludes the proof.

■

Unfortunately BDTs are redundant e.g. the same variable can occur on the same path more than once. Of course this also means that no canonical INF exists. Another problem is that subtrees in a BDT cannot share subsubtrees e.g. equivalent terminal nodes aren't shared. What we like though is the simple way of representing Boolean expressions solely based on the if-then-else-operator. To cope with these problems and to keep the simple representation of Boolean expressions we introduce BDDs.

Definition 2.7 (BDD)
A *binary decision diagram*, henceforth BDD, is a finite dag with a unique initial node. All terminal nodes are labeled with T or F and all non-terminal nodes are labeled with Boolean variables. A non-terminal node, u, has exactly two out edges, one labeled T and one labeled F (depicted as a solid and dotted line respectively). The node pointed to by the out edge labeled T is called the *high son*, denoted high(u). If we speak of the subBDD rooted by the high son we call it the *high child*. Similarly the node pointed to by the out edge labeled F is called the *low son*, denoted low(u) and the subBDD rooted by the low son is called the *low child*.

It is easy to see that BDTs are special cases of BDDs. Thus every Boolean expression can be represented as a BDD. Four kinds of redundancies can occur in BDDs:

- **R1** The same node might occur several times on a path (from the root).
- **R2** There might be duplicate terminal nodes.
- **R3** There might be duplicate non-terminal nodes.
- **R4** Both out edges of a non-terminal node can point to the same node.

These cases are illustrated in figure 2.
To amend this we introduce *Ordered BDDs* and *Reduced Ordered BDDs*.

Definition 2.8
Let $[x_1, x_2, \ldots, x_n]$ be an ordered list of variables with no duplicates, we shall call this a *variable ordering*. Let B be a BDD. B is called an *Ordered BDD*, henceforth OBDD, with respect to $[x_1, x_2, \ldots, x_n]$ if every variable in B occurs in the ordered list of variables and if for every x_i followed by x_j on any path in B we have $i < j$. Furthermore if redundancies of type R2, R3 and R4 doesn't occur B is called a *Reduced OBDD*, henceforth ROBDD.

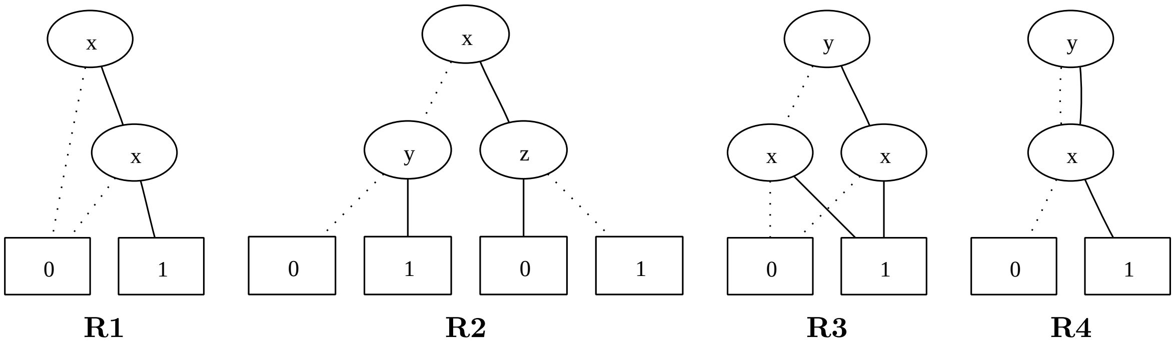


Figure 2: Redundancy in BDDs. R1, R3 and R4 represent the Boolean expression x while R2 represents $(\neg x \wedge y) \vee (x \wedge \neg z)$.

Remark 2.9
In the above definition it is the property of being ordered that ensures that redundancies of type R1 doesn't occur.

2.2 ROBDDs representing functions

It is not hard to modify theorem 2.6 to hold for OBDDs, thus any Boolean expression is equivalent to an OBDD and since removing redundancies doesn't change semantics we also have:

Theorem 2.10
Any Boolean expression, t, is equivalent to a ROBDD.

As was the case with BDTs (R)OBDDs can represent functions. This is done in much the same way, but now the represented function depends explicitly on the variable ordering. Let B be a (R)OBDD with the variable ordering $[x_1, x_2, \ldots, x_n]$ then B represents a unique function $f_B(x_1, x_2, \ldots, x_n)$ from $\mathbb{B}^n$ to $\mathbb{B}$. The important thing to realize is that the variable ordering determines the domain of the function e.g. the variable ordering $[x, y, z]$ together with B_x (see figure 3) represents the function $f(x, y, z) = x$, while the variable ordering $[x]$ together with B_x represents $g(x) = x$.
The crucial property of ROBDDs is the existence of a canonical form, that is, the representation of f_B is unique. This is made precise in the following theorem.

Theorem 2.11 (Canonical forms theorem)
For any function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ there exists exactly one ROBDD, B, with variable ordering $V = [x_1, x_2, \ldots, x_n]$ such that $f(x_1, x_2, \ldots, x_n) = f_B$, where f_B denotes the (unique) function represented by B with respect to V.

PROOF:
The proof is by induction on the number of arguments to f. Suppose $n = 0$ then f represents either T or F. The only ROBDDs without non-terminals (corresponding to the empty variable ordering) are B_1 and B_0. Thus there is exactly one ROBDD representing f in each of the two cases. Now we assume that the theorem holds for all functions of $n-1$ arguments. We must prove it for n. Let $f : \mathbb{B}^n \rightarrow \mathbb{B}$ be a function of n arguments and let $f_{\mathsf{T}}, f_{\mathsf{F}} : \mathbb{B}^{n-1} \rightarrow \mathbb{B}$

be given by:

$$\begin{aligned} f_{\mathsf{T}}(x_2, x_3, \ldots, x_n) &= f(\mathsf{T}, x_2, \ldots, x_n) \\ f_{\mathsf{F}}(x_2, x_3, \ldots, x_n) &= f(\mathsf{F}, x_2, \ldots, x_n) \end{aligned}$$

Since f_{T} and f_{F} are functions of $n-1$ arguments induction gives the existence of unique ROBDDs, $B_{\mathsf{T}}, B_{\mathsf{F}}$ with variable ordering $[x_2, x_3, \ldots, x_n]$ such that:

$$\begin{aligned} f_{\mathsf{T}} &= f_{B_{\mathsf{T}}} \\ f_{\mathsf{F}} &= f_{B_{\mathsf{F}}} \end{aligned} \tag{1}$$

Shannon expansion gives:

$$f(x_1, x_2, \ldots, x_n) = \mathsf{if}\ x_1\ f_{\mathsf{T}}(x_2, x_3, \ldots, x_n)\ f_{\mathsf{F}}(x_2, x_3, \ldots, x_n) \tag{2}$$

At this point the proof is split into two parts. In both cases we need to show that there is exactly one ROBDD, B, with variable ordering $[x_1, x_2, \ldots, x_n]$ such that $f(x_1, x_2, \ldots, x_n) = f_B$.

- $B_{\mathsf{T}} = B_{\mathsf{F}}$:
  From $B_{\mathsf{T}} = B_{\mathsf{F}}$, (1) and (2) we get $f_{B_{\mathsf{T}}} = f_{B_{\mathsf{F}}} = f_{\mathsf{T}} = f_{\mathsf{F}} = f$. $B = B_{\mathsf{T}}$ is a ROBDD representing f with respect to $[x_1, x_2, \ldots, x_n]$. Assume B' is another ROBDD representing f, that is $f = f_{B'}$. From (1), (2) and $f_{B_{\mathsf{T}}} = f_{B_{\mathsf{F}}}$ we get:

  $$f_{B'}[\mathsf{T}/x_1] = f_{B_{\mathsf{T}}} = f_{B_{\mathsf{F}}} = f_{B'}[\mathsf{F}/x_1] \tag{3}$$

  $f_{B'}[\mathsf{T}/x_1]$ and $f_{B'}[\mathsf{F}/x_1]$ are functions of $n-1$ arguments. By induction and (3) we conclude that $B_{\mathsf{T}} = B_{\mathsf{F}}$ is the unique ROBDD representing $f_{B'}[\mathsf{T}/x_1]$ and $f_{B'}[\mathsf{F}/x_1]$. Now it is clear that it cannot be the case that x_1 is the root of B' because that would introduce redundancy (R2, R3 or R4) and then B' wouldn't be a ROBDD. Therefore it can only be the case that $B = B'$.

- $B_{\mathsf{T}} \neq B_{\mathsf{F}}$:
  Let B be the ROBDD with x_1 as root and B_{F} and B_{T} as low and high children respectively. Clearly B is a ROBDD with respect to $[x_1, x_2, \ldots, x_n]$, for x_1 does not occur in B_{T} and B_{F} both of which are ROBDDs with respect to $[x_2, x_3, \ldots, x_n]$. Moreover using (1) and (2) we get $f_B = f$. Assume B' is another ROBDD representing f, that is $f = f_{B'}$. From (1) and (2) we get $f_{B'}[\mathsf{T}/x_1] = f_{B_{\mathsf{T}}}$ and $f_{B'}[\mathsf{F}/x_1] = f_{B_{\mathsf{F}}}$. As above $f_{B'}[\mathsf{T}/x_1]$ and $f_{B'}[\mathsf{F}/x_1]$ are functions of $n-1$ arguments. By induction we conclude that $f_{B'}[\mathsf{T}/x_1]$ and $f_{B'}[\mathsf{F}/x_1]$ are represented uniquely by the ROBDDs B_{T} and B_{F} with respect to $[x_2, x_3, \ldots, x_n]$. Based on the assumption $B_{\mathsf{T}} \neq B_{\mathsf{F}}$ we conclude that x_1 must be root in B'. Hence B' must be the ROBDD with respect to $[x_1, x_2, \ldots, x_n]$ having x_1 as root and B_{F} and B_{T} as low and high children respectively, that is $B' = B$.

■

To see how powerful the canonical forms theorem is consider the following immediate consequences:

- Testing two expressions for semantic equivalence is reduced to structural equivalence testing.

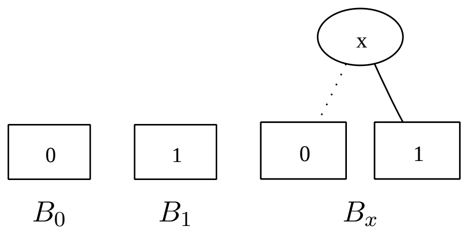


Figure 3: ROBDDs representing the absurdity, the tautology and the Boolean expression x.

- A function is valid if and only if it is represented as B_1 (see figure 3).
- A function is satisfiable if and only if it is not represented as B_0 (see figure 3).
- Let f and g be functions $f, g : \mathbb{B}^n \rightarrow \mathbb{B}$ then $f \Rightarrow g$ if and only if $f \wedge \neg g$ is represented as B_0.

2.3 Limitations

Now that we have introduced ROBDDs it might be good to take a look at the limitations of this wonder tool. It is well known that satisfiability of Boolean expressions is NP-complete yet ROBDDs can answer the question of satisfiability quite easily as we saw above. Consequently we must expect ROBDD construction to be hard. The following example shows that the size of a ROBDD can be exponential in the number of variables in the expression it represents.

Definition 2.12
Let B be a BDD. The *size* of B, denoted $|B|$, is defined to be the number of nodes in B.

Example 2.13
Consider the Boolean expression $t = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2n-1} \wedge x_{2n})$ together with the variable ordering $V = [x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n}]$. Let B be the ROBDD representing t with respect to V. The size of B satisfies $|B| > 2^n$ and thus the size is exponential in the number of variables in t. The case $n = 3$ is depicted in figure 4.

In order to represent Boolean expressions compactly using ROBDDs it is important to choose a *good* variable ordering. In figure 5 the Boolean expression from example 2.13 is represented using the variable ordering $[x_1, x_2, \ldots, x_{2n}]$ in the case $n = 3$. Now the number of nodes is $2n + 2$ (8). How to select a good variable ordering is a research topic in its own right and won't be considered any further here.

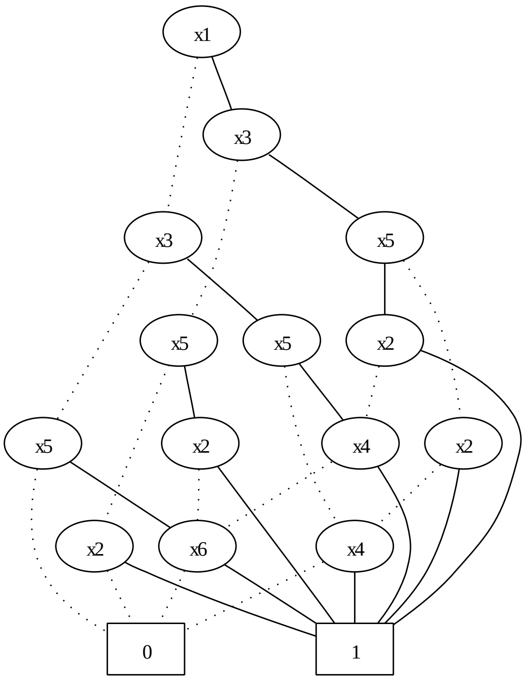


Figure 4: ROBDD representing $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$ with respect to $[x_1, x_3, x_5, x_2, x_4, x_6]$.

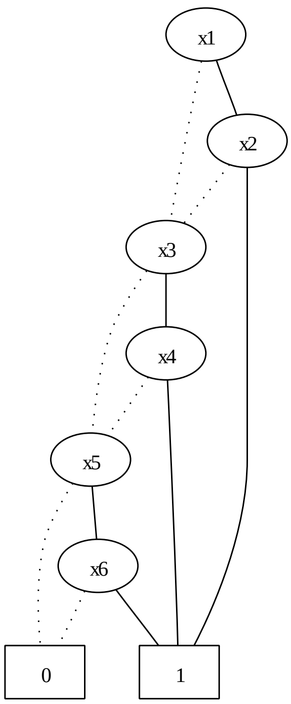


Figure 5: ROBDD representing $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$ with respect to $[x_1, x_2, x_3, x_4, x_5, x_6]$.

3 Designing a BDD package

This section is split into two parts. The first one covers the operations we have implemented while the latter one explains the importance of maximal sharing.

3.1 Operations on BDDs

This section describes the bdd[2] operations we have implemented. In what follows we use `this font` to refer to actual syntax e.g. functions names and type annotations while this font is used to refer to filenames in our implementation. We have chosen Standard ML as implementation language and therefore we present the functions in SML style.
Based on [?] and [?] we have decided to implement the operations show in figure 6. The most important operation is `apply` since many of the others can be implemented in terms hereof, e.g. `neg`$(b) \equiv$ `apply`$(b, true, \mathrm{xor})$. As can be seen bdds are parameterized over the polymorphic type `''voelem`. Voelem is a shorthand for *variable ordering element*. Many presentations of bdds e.g. [?] and [?] assume a static variable ordering e.g. $x_1 < x_2 < \ldots < x_n$ because it makes it easier to explain the bdd operations. However in order to implement `satcount` we need to keep track of which variables belong to the variable ordering, see section 3.1.8. Below we give a short explanation of each of the operations we have implemented.

```
val build     :  ''voelem varorder * ''voelem BEXP -> ''voelem robdd
val toString  :  ''voelem robdd * (''voelem -> string) -> string
val apply     :  ''voelem robdd * ''voelem robdd * operator -> ''voelem robdd
val restrict  :  ''voelem robdd * ''voelem * bool -> ''voelem robdd
val neg       :  ''voelem robdd -> ''voelem robdd
val exists    :  ''voelem robdd * ''voelem -> ''voelem robdd
val forall    :  ''voelem robdd * ''voelem -> ''voelem robdd
val satcount  :  ''voelem robdd -> int
val anysat    :  ''voelem robdd -> (''voelem * int) list
val allsat    :  ''voelem robdd -> (''voelem * int) list list
val equal     :  ''voelem robdd * ''voelem robdd -> bool
```

Figure 6: BDD operations in SML style.

3.1.1 `build`

`build` takes as inputs a variable ordering and a Boolean expression. Every variable occurring in the Boolean expression must also occur in the variable ordering. If so `build` constructs the corresponding bdd.

3.1.2 `toString`

Given a bdd and a function that produces a printable representation of variable ordering elements `toString` creates input (in dot format) to the Graph Visualization Software by Graphviz, see [?]. This way we can obtain visual representations like figure 4 and 5 on page 10.

[2]From here and onwards we take bdd to mean ROBDD.

3.1.3 `apply`

Given two bdds *bdd*1 and *bdd*2 with identical variable orderings and a binary Boolean operator *op* such that *bdd*1 (uniquely) represents the Boolean expression $b1$ and such that *bdd*2 represents $b2$ `apply` creates the (unique) bdd corresponding to the Boolean expression $b1$ *op* $b2$. The variable ordering of the new bdd is the same as for the input bdds.

3.1.4 `restrict`

Given a bdd representing the Boolean expression b and an element, e.g. x_1, from the variable ordering of the bdd together with a truth value, e.g. T, `restrict` produces the bdd corresponding to $b[\mathsf{T}/x_1]$. The result is obtained by forcing x_1 to be T. Thus x_1 is no longer a variable in the result bdd and hence we must remove x_1 from the variable ordering of the result[3].

3.1.5 `neg`

Given a bdd representing b `neg` produces the bdd corresponding to $\neg b$. The variable ordering of the result bdd is the same as for the input bdd.

3.1.6 `exists`

`exists` takes as input a bdd, *bdd*1, representing the Boolean expression b and an element, x_1, from the variable ordering of *bdd*1. It produces the bdd corresponding to $b[\mathsf{T}/x_1] \vee b[\mathsf{F}/x_1]$. The variable ordering of the result bdd is equal to the variable ordering of the input bdd but with x_1 removed.

3.1.7 `forall`

`forall` takes as input a bdd, *bdd*1, representing the Boolean expression b and an element, x_1, from the variable ordering of *bdd*1. It produces the bdd corresponding to $b[\mathsf{T}/x_1] \wedge b[\mathsf{F}/x_1]$. The variable ordering of the result bdd is equal to the variable ordering of the input bdd but with x_1 removed.

3.1.8 `satcount`

Given a bdd, *bdd*1, representing the Boolean expression b `satcount` calculates the number of ways to satisfy b by assigning truth values to all the variables in the variable ordering of *bdd*1. E.g. consider the bdd B_x from figure 3 on page 9 together with the variable ordering $x < y < z$. In order to satisfy B_x the only requirement is $x = \mathsf{T}$, that is y and z can be chosen arbitrarily, thus the number of satisfying variable assignments is 4. If we restrict the variable ordering to $x < y$ the number of satisfying variable assignments is 2.

3.1.9 `anysat`

Given a bdd, *bdd*1, representing the Boolean expression b `anysat` calculates *a set* of strict requirements in order make b true. E.g. In the example above concerning the bdd B_x together with the variable ordering $x < y < z$ the set of strict requirements is $x = \mathsf{T}$.

[3]The variable ordering of a bdd is used in the operation `satcount`.

3.1.10 `allsat`

Given a bdd, $bdd1$, representing the Boolean expression b `allsat` calculates *every set* of strict requirements in order to make b true.

3.1.11 `equal`

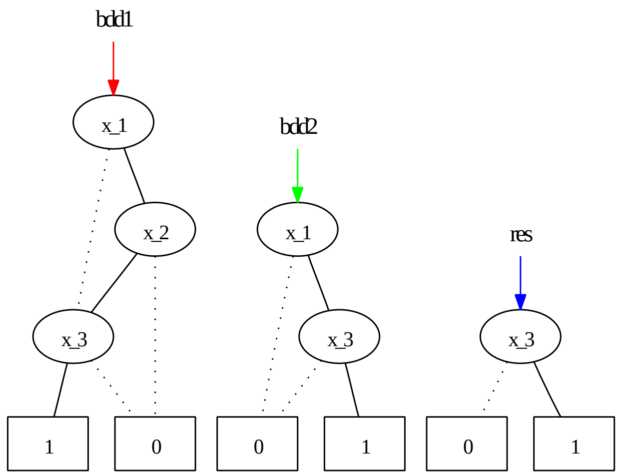
Given two bdds, $bdd1$ and $bdd2$, representing the Boolean expressions $b1$ and $b2$ `equal` checks whether $bdd1$ is structurally[4] equivalent to $bdd2$, which by section 2.2 amounts to semantic equivalence of $b1$ and $b2$.

3.2 Maximal sharing

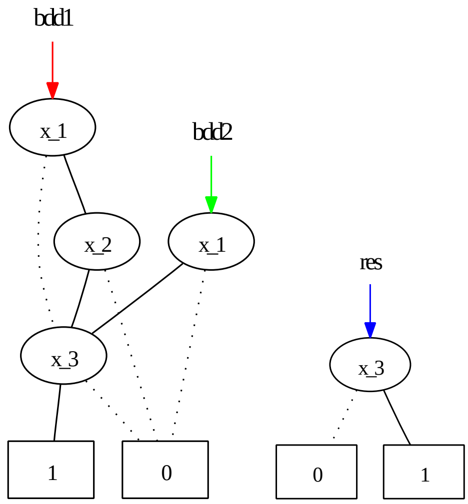
In section 2.3 we saw that the size of a bdd can be exponential in the number of variables in the expression it represents. A system using bdds will typically have references to several bdds at the same time. Therefore we can reduce memory consumption if we make sure that bdds share subbdds wherever possible. Let S be a set of bdds. A subset $T \subseteq S$ is said to be maximally shared if sharing of subbdds is maximal in T. This is illustrated in figure 7.
Maximal sharing has been a driving design principle, but in some cases it comes at a cost. Especially in the case of msd. In that case we allow a relaxation and only demand that the set $\{res, bdd1, bdd2\}$ where $res = \texttt{apply}(bdd1, bdd2, op)$ is maximally shared when `apply` finishes. For performance reasons we also demand that $bdd1$ and $bdd2$ are maximally shared before they can be compared for equality since maximal sharing reduces equality of bdds to pointer or node equivalence.

[4] In the actual implementations this amounts to pointer or node equivalence. Thus there is no need to traverse the data structures of $bdd1$ and $bdd2$.

No sharing:



Some sharing:



Maximal sharing:

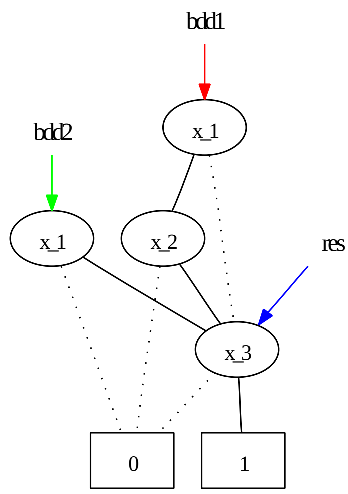


Figure 7: Levels of sharing in bdds.

4 Hash based implementation

The de facto way of implementing bdds is to use hash tables. A priori this can be done with or without maximal sharing. First we could settle for sharing of subbdds only within the same bdd. This way every bdd would have its own data structure thus facilitating automatic garbage collection. The downside is larger memory consumption. The second approach is to use a global data structure to ensure maximal sharing of all sub-bdds. The immediate downside is that garbage collection (maintenance of the data structure) must be done manually. Since we want to minimize memory consumption at all costs we have chosen to implement the second approach. How this can be done is described in elaborate detail in [?]. Our implementation is based on that description. Below we include an overview of some of the key functions. The section is concluded with an overview of running times and space usage for key bdd operations as well as a discussion of some of the problems with hash based implementations.

4.1 Practical stuff

In order to create a uniform user interface to our bdd packages we have extracted some common features that we need in the hash based as well as the msd based implementation. In Binop.{sig | sml} we define a module containing some Binary operators and some functions on them. In Bexp.{sig | sml} we define our representation of Boolean expressions. And finally in Varorder.{sig | sml} we define our representation of variable orderings as well as some functions on them. Below we give a short description of these modules.

4.1.1 Binop

Figure 8 contains an overview of the *Binop* signature. First we define the type `operator`. The functions `isA`, `isC` and `isI` determines whether an operator is associative, commutative or idempotent. `isACI` is the conjunct of these three operations. Finally `AND`, `OR`, `BIMP`, `IMP` and `XOR` are implementations of the operators their names suggest.

4.1.2 Bexp

Figure 9 contains an overview of the *Bexp* signature. The datatype `''voelem BEXP` is used to represent Boolean expressions. The function `subst`$(bexp, t, sub)$ performs a simple syntactic substitution namely $bexp[t/sub]$[5]. Finally the function `prntBexp` is used to create input (in dot format) to the Graph Visualization Software by Graphviz ([?]). This way we can obtain visual representations of Boolean expressions as evaluation trees.

4.1.3 Varorder

Figure 10 contains an overview of the *Varorder* signature. Internally our algorithms assume the variable ordering $x_0 < x_1 < x_2 < \ldots < x_{n-1}$ or just $0 < 1 < 2 < \ldots < n-1$. Thus we represent a variable ordering as a bijection from whatever variable ordering the user of our package might want into $0, 1, 2, \ldots, n-1$, where n is the number of variables in the variable ordering supplied by the user. A variable ordering can be created with `varorderFromLst`. The function maps the ith element in the input list to $i-1$. E.g. `varorderFromLst`$([x_1, x_2, x_3])$

[5]Since we do not allow for any variable qualifiers we do not have to consider variable capture.

```
signature Binop =
sig
  type operator = bool * bool -> bool

  val isA : operator -> bool
  val isC : operator -> bool
  val isI : operator -> bool
  val isACI : operator -> bool

  val AND : operator
  val OR : operator
  val BIMP : operator
  val IMP : operator
  val XOR : operator
end
```

Figure 8: The Binop signature.

```
signature Bexp =
sig
  datatype
    ''voelem BEXP = VAR of ''voelem | F | T | !! of ''voelem BEXP |
    && of ''voelem BEXP * ''voelem BEXP | || of ''voelem BEXP * ''voelem BEXP |
    ==> of ''voelem BEXP * ''voelem BEXP | <==> of ''voelem BEXP * ''voelem BEXP

  val subst : ''voelem BEXP * ''voelem BEXP * ''voelem BEXP -> ''voelem BEXP
  val prntBexp: ''voelem BEXP * (''voelem -> string) -> string
end
```

Figure 9: The Bexp signature.

results in the mapping $[x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 2]$. The function `varOrdElemToInt` takes as input a variable ordering and a variable and returns the integer the variable maps to while the function `intToVarOrdElem` does exactly the opposite. The functions `inOrder` and `idxInOrder` determines whether a variable respectively an index has a mapping in a given variable ordering. The function `checkVarOrder` determines whether a variable ordering really is a bijection as described above. Given a variable ordering and a Boolean expression the function `validate` checks whether all the variables in the Boolean expression occurs in the variable ordering. The functions `maxVar` and `minVar` returns the maximal and minimal index associated with an element in a variable ordering. The function `getNoVars` simply returns the number of variables in a variable ordering. Finally the function `restrictVo` is used to remove a variable from a variable ordering.

```
signature Varorder =
sig
  eqtype ''voelem varorder
  type ''voelem BEXP = ''voelem Bexp.BEXP

  val varorderFromLst : ''voelem list -> ''voelem varorder
  val varOrdElemToInt : (''voelem varorder * ''voelem) -> int option
  val intToVarOrdElem : (''voelem varorder * int) -> ''voelem option
  val inOrder : (''voelem * ''voelem varorder) -> bool
  val idxInOrder : (int * ''voelem varorder) -> bool
  val checkVarOrder : ''voelem varorder -> bool
  val validate : (''voelem varorder * ''voelem BEXP) -> bool
  val maxVar : ''voelem varorder -> int
  val minVar : ''voelem varorder -> int
  val getNoVars : ''voelem varorder -> int
  val restrictVo : ''voelem varorder * ''voelem -> ''voelem varorder
end
```

Figure 10: The Varorder signature.

4.2 Overview of the hash based implementation

Following [?] our implementation uses a table T and a hash table H. Nodes are saved in T and are thus uniquely represented by their index u. A node is a triple (i, l, h) where i indicates which variable the node represents while l and h are indices of the low and high son respectively. Thus T contains the following mapping $T : u \mapsto (i, l, h)$. The hash table H contains the inverse mapping $H : (i, l, h) \mapsto u$. In what follows we explain the key operations in the hash based approach.

4.2.1 Mk

We need to construct reduced ordered bdds. The approach in [?] is to ensure reducedness on the fly. This is done by using the function `mk` whenever we might create a new node. `mk` is shown in figure 11. `mk` first checks whether $l = h$ if so we mustn't create a new node since it

```
fun mk(i,l,h) =
  if l = h then l
  else if member(i,l,h) then lookup(i,l,h)
       else let
              val u = add(i,l,h);
             in
              insert(i,l,h,u); u
             end
```

Figure 11: Function `mk` creates a new node if and only if it doesn't exist already.

would introduce redundancy of type R4 (see figure 2). Instead we should point directly to l

so this is what mk returns. If $l \neq h$ we use H to find out whether the node (i, l, h) already exists. If the node doesn't exist we create it by updating T and H. Finally we return u the index of (i, l, h) in T.

4.2.2 build

There are two approaches to building bdds. Either they are constructed from primitives, B_x, using `apply` or they can be constructed using a dedicated procedure based on the Shannon expansion $t \equiv \mathsf{if}\ x\ t[\mathsf{T}/x]\ t[\mathsf{F}/x]$. In [?] and thus here we use the latter approach. The essential part of the dedicated procedure `build` is shown in figure 12. `build` recursively constructs

```
...
fun build'(t,i) =
  if i > n then
    if t = F then 0 else 1
    else let
      val elemRepVarI = intToVarOrdElem(varorder,i)
         in
           case elemRepVarI of
             SOME el =>
               let
                 val v0 = build'(partBoolEval(subst(F,(VAR el),t)),i+1)
                 val v1 = build'(partBoolEval(subst(T,(VAR el),t)),i+1)
               in
                 mk(i,v0,v1)
               end
           | NONE => ...
```

Figure 12: The essential part of function `build` which creates the bdd uniquely representing the Boolean expression t.

bdds $v0$ and $v1$ representing $t[\mathsf{F}/x]$ and $t[\mathsf{T}/x]$ respectively. Finally `mk` is used to create a node $(i, v0, v1)$ if need be. In all cases $mk(i, v0, v1)$ returns u such that u is the index in T of the root node in the bdd representing t. As mentioned in section 4.1.3 our algorithms use the variable ordering $0 < 1 < 2 < \ldots < n-1$ internally. In figure 12 `build'` should be called with $i = 0$ thus first constructing the root node. In the special case where variable 0 doesn't occur in t we have $v0 \equiv v1$ and thus the overall result returned by `mk` is the index of $v0$ in T.

4.2.3 apply

The implementation of `apply` relies on the following facts:

$$
\begin{aligned}
\mathsf{if}\ x\ t_1\ t_2\ op\ s &= \mathsf{if}\ x\ t_1\ op\ s\ t_2\ op\ s \\
t\ op\ \mathsf{if}\ x\ s_1\ s_2 &= \mathsf{if}\ x\ t\ op\ s_1\ t\ op\ s_2 \\
\mathsf{if}\ x\ t_1\ t_2\ op\ \mathsf{if}\ x\ s_1\ s_2 &= \mathsf{if}\ x\ t_1\ op\ s_1\ t_2\ op\ s_2
\end{aligned}
$$

The idea is to start from the roots of $u1$ and $u2$ and recursively construct the low and high children of the result bdd. The essential part of the function `apply` is shown in figure 13.

Table 2 Time and space complexities for the hash based approach.

Function	**Time**	**Space**
`mk`	$O(1)$	$O(1)$
`build`	$O(2^n)$	$O(1)$
`apply`	$O(\|u1\|\|u2\|) = O(2^n)$	$O(\|u1\|\|u2\|)$

`app` uses memoization to avoid exponential blow-ups. In case we have already calculated $app(u1, u2)$ the result is stored in the memoization hash table and thus we can return the result without further computation. If not we must compute $app(u1, u2)$ and insert the result in the memoization hash table. There are six cases to consider:

$\boldsymbol{u1, u2 \in \{\mathsf{T}, \mathsf{F}\}}$**.** In this case we calculate the result without further recursion.

$\boldsymbol{u1, u2 \notin \{\mathsf{T}, \mathsf{F}\}}$**.** In this case there are three sub-cases:

> $\boldsymbol{var(u1) < var(u2)}$**.** According to the variable ordering $0 < 1 < 2 < \ldots < n - 1$ this node must represent the variable $var(u1)$. Thus we construct the node $(var(u1), app(low(u1), u2), app(high(u1), u2))$.
>
> $\boldsymbol{var(u1) = var(u2)}$**.** In this case we construct the node $(var(u1), app(low(u1), low(u2)), app(high(u1), high(u2)))$.
>
> $\boldsymbol{var(u1) > var(u2)}$**.** The node to construct is $(var(u2), app(u1, low(u2)), app(u1, high(u2)))$.

$\boldsymbol{u1 \in \{\mathsf{T}, \mathsf{F}\}, u2 \notin \{\mathsf{T}, \mathsf{F}\}}$**.** Since $u2 \notin \{\mathsf{T}, \mathsf{F}\}$ we must continue to recurse, thus we construct the node $(var(u2), app(u1, low(u2)), app(u1, high(u2)))$.

$\boldsymbol{u1 \notin \{\mathsf{T}, \mathsf{F}\}, u2 \in \{\mathsf{T}, \mathsf{F}\}}$**.** Similar to above we construct $(var(u1), app(low(u1), u2), app(high(u1), u2))$.

As can be seen from figure 13 `app` uses `mk` whenever it tries to create a new node. Thus `app` reduces the result bdd on the fly.

4.3 Time and space complexities - the hash based approach

In this section we give time and space complexities of the key functions in the hash based approach. The space complexity expresses how much memory a function uses during computation (memory that is freed when the function returns).

The time and space complexity of `mk` is $O(1)$ since hash operations can be performed in $O(1)$ time (and space). It is easy to see, that `build` generates on the order of 2^n recursive calls. We also observe that no additional memory is used. In the case of apply let $|u1|$ and $|u2|$ be the number of nodes that can be reached from $u1$ and $u2$ respectively. Since we use memoization no more than $|u1||u2|$ recursive calls can be made. We have seen that the number of nodes in a bdd can be exponential in then number of variables in the variable ordering (example 2.13) thus $O(|u1||u2|) = O(2^n)$. The use of memoization introduces a hash table with a maximum of $|u1||u2|$ entries. Thus the space complexity of `apply` is $O(|u1||u2|)$ as well.

The complexities are summarized in table 2.

```
...
fun app(u1,u2) =
  let
    val Gu1u2 = Polyhash.peek Gtbl (u1,u2)
  in
    case Gu1u2 of
      SOME n => n
    | NONE =>
        if (u1=0 orelse u1=1) andalso (u2=0 orelse u2=1) then
          insertInHash(Gtbl,(u1,u2),boolToInt(opr(u1=1,u2=1)))
        else
          let
            val varu1 = var(u1)
            val lowu1 = low(u1)
            val highu1 = high(u1)
            val varu2 = var(u2)
            val lowu2 = low(u2)
            val highu2 = high(u2)
          in
            if varu1=varu2 then
              insertInHash(Gtbl,(u1,u2),
              mk(varu1,app(lowu1,lowu2),app(highu1,highu2)))
            else if (0 <= varu1 andalso varu1 < varu2) orelse varu2 < 0 then
              insertInHash(Gtbl,(u1,u2), mk(varu1,app(lowu1,u2),app(highu1,u2)))
            else (* if (0 <= varu2  andalso varu2 < varu1) orelse varu1 < 0 then *)
              insertInHash(Gtbl,(u1,u2), mk(varu2,app(u1,lowu2),app(u1,highu2)))
          end
  end
...
```

Figure 13: The essential part of function `apply` which creates the bdd corresponding to $b1$ op $b2$ where $u1$ and $u2$ are the bdds representing $b1$ and $b2$ respectively.

4.4 Problems with hash based implementations

A problem with hash based implementations is the hash tables themselves. When using hash tables we want to avoid hash clashes and therefore we want a good hash function that at best distributes the data uniformly in the underlying table. The problem is twofold. On one hand we want to avoid hash clashes and thus we must use a good hash function as well as ensuring a reasonable load factor. That is we waste memory (lower the load factor) in order to avoid hash clashes. On the other hand a good hash function distributes the data uniformly and thus we must expect that the children of a node in a bdd resides in a different place than their parent - that is we loose locality and therefore the advantages of cache access in memory. Another problem is that hash based implementations forces us to do manual garbage collection. As we shall see all of these problems will be alleviated by the msd based implementation.

5 Msd based implementation

In this section we describe our msd based implementation. To the best of our knowledge this approach has never been tried before. Since multiset discrimination in its own right isn't as prevalent as one could wish for we begin this section with an introduction to multiset discrimination focusing on the needs in our implementation. Following that we give a description of some of the key functions. The section is concluded with an overview of running times and space usage for key bdd operations as well as a discussion comparing our hash and msd based implementations.

5.1 MultiSet Discrimination

MultiSet Discrimination is a technique for finding equal or equivalent elements in a multiset. Multiset discrimination doesn't rely on hashing nor comparison based sorting and can be applied to atomic as well as composite types, trees, dags etc.

Example 5.1 (Atomic discrimination of integers)
Suppose we have the multiset of integers $\langle 1, 2, 2, 3, 4, 5, 5 \rangle$. If we discriminate on equality we get $\{\langle 1 \rangle, \langle 2, 2 \rangle, \langle 3 \rangle, \langle 4 \rangle, \langle 5, 5 \rangle\}$. If instead the discriminator is equality modulo two we get $\{\langle 1, 3, 5, 5 \rangle, \langle 2, 2, 4 \rangle\}$.

Discrimination of atomic types[6] e.g. 32-bit integers, as well as composite types e.g. pairs, sets, bags (MultiSet of multisets) and dags is described throughly in [**?**][Chap. 2].

5.1.1 Discrimination of dags

Ultimately we're going to discriminate bdds. As described in section 3.2 maximal sharing has been a driving design principle. Before we get to discrimination of bdds we will explain the principles using dags. So suppose we want to discriminate and maximally share a multiset, S, of dags of type `'a dag`, shown in figure 14, where `'a` is a type we know how to discriminate. If we knew that a set, S', of dags was maximally shared then two equivalent dags (references to an element of type `'a dagVal`) would point to the same element. Thus multiset discrimination of maximally shared dags reduces to multiset discrimination of references (described in [**?**]). Below we show how to maximally share dags. The description is based on [**?**].

```
datatype 'a dagVal =
  Leaf of  'a
| Node of 'a * 'a dag * 'a dag

withtype 'a dag = 'a dagVal ref
```

Figure 14: A simple dag type.

Since we know how to discriminate elements of type `'a` we also known how to discriminate `Leaf` elements. Further we observe that nodes of different heights[7] cannot be equivalent. Now the procedure for maximal sharing is as follows:

[6] Here an atomic type is a type *for which there is a finite number of distinct element values* ([**?**][p. 9]).

[7] The height h of a node n is the *largest* number of edges between n and any leaf node. A leaf node is a node with no outgoing edges ([**?**][p. 27]).

1. Partition all dag-nodes into a set of multisets, $P = \{P_0, P_1, \ldots, P_k\}$, of nodes of equal height such that P_i consists exactly of the nodes of height i.

2. For each P_i in P starting with P_0 discriminate the elements of P_i.

   (a) There are two cases:

   $i = 0$. In this case we use the known discriminator on type `'a` to construct a discriminator for `Leaf` elements.

   $i > 0$. Assuming that the sub nodes pointed to have already been maximally shared we can discriminate a multiset, S, of `Node` elements by first discriminating on the right sub node (using reference discrimination as described in [**?**]). This will result in a partitioning of S which we call S_r consisting of multisets of `Node` elements with equivalent right sub nodes. The second step is to discriminate every multiset in S_r on the left sub node. Thus we obtain a set of sets of multisets with equivalent right and left sub nodes which we flatten to a set of multisets which we call S_{rl}. Finally we discriminate every multiset in S_{rl} using the known discriminator on elements of type `'a`. Again the result is a set of sets of (multi)sets which we flatten to a set of multisets which we call S'. S' is the result of the discrimination on the initial multiset of nodes S thus the elements of S' are equivalence classes according the discrimination process just described.

   (b) For each equivalence class $C \in S'$ select a canonical node, $c \in C$, and update every node that points to a node in C to point to c instead.

Example 5.2
As an example of the approach described above lets carry out the steps needed to go from top to bottom in figure 7. In order to identify nodes uniquely each nodes is given an id. According to the type definition in figure 14 the first component of the content in leafs and non-leaf nodes must be of the same type. Therefore we rename `x_i` to `i`. The overall naming scheme for nodes and leafs is `id:i`. With these small changes we get the dags (bdds) shown in figure 15.

1. The first step is to partition the nodes by height:

$$
\begin{aligned}
P = \{ & \\
& P_0 = \langle (5:1), (4:0), (8:0), (9:1), (11:0), (12:1) \rangle\,, \\
& P_1 = \langle (2:3), (7:3), (10:3) \rangle\,, \\
& P_2 = \langle (3:2), (6:1) \rangle \\
& P_3 = \langle (1:1) \rangle \\
\} &
\end{aligned}
$$

2. Carrying out step two for $i = 0$ identifies the nodes $\{(4:0), (8:0), (11:0)\}$ and $\{(5:1), (9:1), (12:1)\}$. Suppose we choose $(4:0)$ and $(5:1)$ as canonical nodes. Then we obtain figure 16.

   Carrying out step two for $i = 1$ identifies the nodes $\{(2:3), (7:3), (10:3)\}$. Suppose we choose $(2:3)$ as canonical node. Then we obtain figure 17.

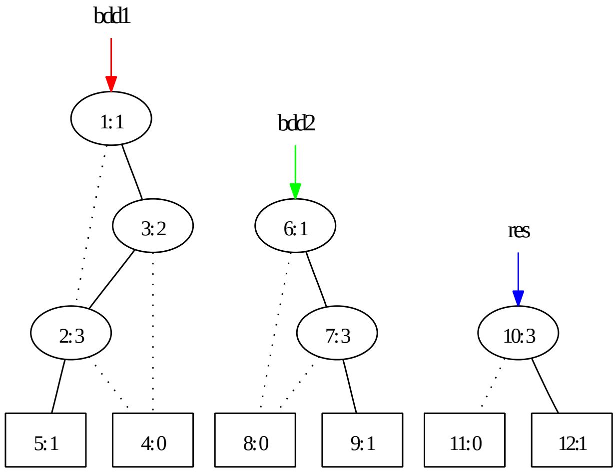


Figure 15: Discrimination step 0.

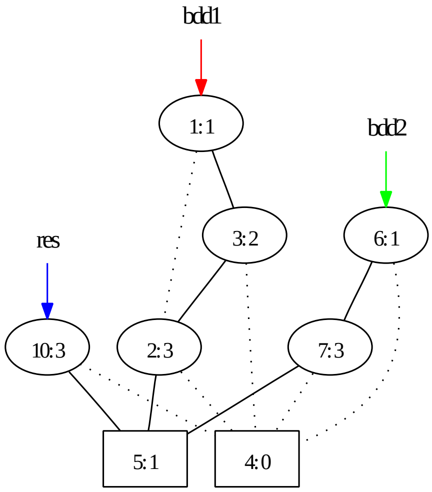


Figure 16: Discrimination step 1.a.

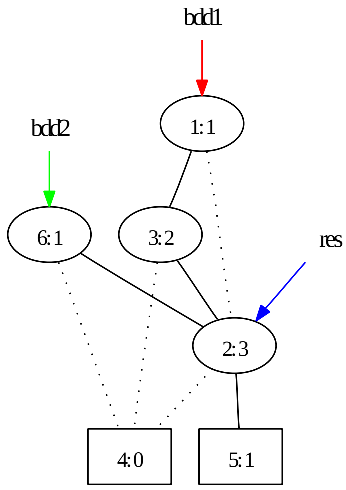


Figure 17: Discrimination step 1.b.

Since nothing is changed by carrying out step two for $i = 2$ and $i = 3$ the dag in figure 17 is the maximally shared equivalent of the dags shown in figure 15.

5.2 Overview of the msd based implementation

The corner stone in our msd based approach is the bdd node of type `node`, shown in figure 18. As in the case of dags a node is a reference to an element of type `nodeVal`. Here we use a

```
datatype nodeVal =
   TRUE
 | FALSE
 | IF of int * node * node
 | APPLY of node * node

withtype node = nodeVal duref
```

Figure 18: The type, `node`, of bdd nodes.

special kind of reference called a *duref* which is short for *discriminatable unionable reference*. Durefs are described in section 5.2.2. The datatype `nodeVal` has four constructors. `TRUE` and `FALSE` are used to create leafs. `IF of int * node * node` is used to create non-leaf bdd nodes (we have selected the name `IF` because a non-leaf node can be interpreted as an if-then-else-operator, see definition 2.4). As described in section 4.1.3 we use the variable ordering $0 < 1 < 2 < \ldots < n-1$ internally. Thus the int-part of an `IF`-node denotes the variable the node represents. The first node-element of an `IF`-node is a (duref) reference to the root of the high child while the second node-element is a reference to the root of the low child.
In what follows we first explain discrimination of atoms, durefs and nodes before we go on and describe our implementation.

```
signature Atom =
sig
  eqtype elmt

  val new : unit -> elmt
  val discriminate : (elmt * 'a) list -> 'a list list
  val equal: elmt * elmt -> bool
end
```

Figure 19: The Atom signature.

5.2.1 Atom

An `Atom.elmt`-element is an element that supports multiset discrimination. Figure 19 contains an overview of the *Atom* signature. As can be seen from the signature the function `discriminate` takes as input a list of elements of type `(elmt * 'a)`. Below we shall refer to the first component as the *equivalence class element* and to the second component as the *value element*. The idea is to group value elements with identical equivalence class elements together - the result being a list of lists of value elements with identical equivalence class elements.
Following [?] an element of type `elmt` represents an *equivalence class* (as suggested above), a data structure that contains a list of value elements which must be empty before and after discrimination.
Then the following very simple algorithm (based on [?]) can be used to discriminate a list, L, of pairs of type `(elmt * 'a)`.

Algorithm 5.3 (Discrimination of Atoms)

1. For each pair $(a, v) \in L$:
   (a) If the list of values in a is empty, add a to the list, U, of used equivalence classes.
   (b) Add v to the list in a.
2. For each equivalence class $a \in U$:
   (a) Add the list stored in a to the result set.
   (b) Clear the list in a so that it is ready for the next discrimination.

Step 1 takes $O(N)$ time where N is the number of elements in L. Similarly step 2 takes worst case $O(N)$ time since the length of U is less than or equal to N. This gives a total running time of $O(N)$. Similarly the accumulated length of all value element lists stored in equivalence class elements is $O(N)$ (each value element is stored exactly once). Since the length of U is $O(N)$ the algorithm uses $O(N)$ space in total.

Remark 5.4
In practice we use `type elmt = exn ref` and a dedicated value `exception EMPTY` to represent the empty list. This is due to the fact that SML doesn't allow free type variables in top level value identifiers. Our implementation of atoms can be found in Atom.sml in appendix A.3.2 on page 51.

5.2.2 Duref

```
signature Duref =
  sig
    eqtype 'a duref

    val duref: 'a -> 'a duref
    val discriminate: ('a duref * 'b) list -> 'b list list
    val equal: 'a duref * 'a duref -> bool
    val !! : 'a duref -> 'a
    val ::= : 'a duref * 'a -> unit
    val unify : ('a * 'a -> 'a) -> 'a duref * 'a duref -> unit
    val union : 'a duref * 'a duref -> unit
    val link : 'a duref * 'a duref -> unit
  end
```

Figure 20: The Duref signature.

Durefs are references with built-in support for union and discrimination. Figure 20 contains an overview of the *Duref* signature. The signature is best explained by comparing it to references (refs). This is done in table 3.

Table 3 Durefs compared to Refs.

Type	`'a ref`	`'a duref`
introduction	`ref`	`duref`
elimination	`!`	`!!`
equality	`=`	`discriminate` (generalized equality)
updating	`:=`	`::=`
unioning		`link, union, unify`

The `link, union, unify` capability is achieved using a Union/Find data structure and algorithms very similar to those described and analyzed in [?][pp. 505-509]. The data structure together with the central `find` function, which compresses instances of the data structure as a side effect when it extracts their *information* (`ECR`-element), are shown in figure 21.
With this in mind it is easy to explain `!!`, `::=` and `equal`. First they all use `find` to locate the references (refs) to the proper `ECR`-records. Second they apply the proper operation (elimination, update, equality).
Discrimination of durefs is also simple, since we can do it in terms of discrimination of Atoms. Given a list, L', of pairs of type `('a duref, 'b)`, we use `find` to construct the corresponding list, L, of pairs of type `(elmt, 'b)`, which we discriminate using `Atom.discriminate`. Since we don't use union by rank elements of type `durefC` are always fully compressed[8]. Thus `find` runs in $O(1)$ time and space. This gives discrimination of durefs an overall complexity of $O(N)$ time and space, where N is the number of elements.

[8] Indirections are only introduced by `link` and no more than one level of indirection will occur. Thus we can ignore it.

```
datatype 'a durefC
  = ECR of 'a * Atom.elmt
  | PTR of 'a duref
withtype 'a duref = 'a durefC ref

fun find (p as ref (ECR _)) = p
  | find (p as ref (PTR p')) =
      let val p'' = find p'
      in p := PTR p''; p''
      end
```

Figure 21: Union/Find data structure and `find` function.

`unify f (e, e')` makes `e` and `e'` equal; if `v` and `v'` are the contents of `e` and `e'`, respectively, before unioning them, then the contents of the unioned element is `f (v, v')`.
`link (e, e')` makes `e` and `e'` equal; the contents of the linked element is the contents of `e'` before the link operation.
Finally `union (e, e')` makes `e` and `e'` equal; the contents of the unioned element is the contents of one of `e` and `e'` before the union operation [9]. After `union (e, e')` elements `e` and `e'` will be congruent in the sense that they are interchangeable in any context.

5.2.3 Node

Figure 22 contains an overview of the *Node* signature. Here we shall only explain the functions `discriminateNodeVal` and the special case `partitionByContent`. As can be seen form the signature `discriminateNodeVal` takes as input a list of elements of type `(nodeVal * 'a)`. Below we shall refer to the first component as the *node element* and to the second component as the *value element*. The idea is to group value elements with equivalent[10] node elements together - the result being a list of lists of value elements with equivalent node elements.
We observe that there are four different kind of elements of type `nodeVal`, one for each of the four type constructors. Nodes can never be equivalent if they are constructed by unequal type constructors. With this setup in mind the following algorithm can be used to discriminate a list, L, of pairs of type `(node * 'a)`.

Algorithm 5.5 (Discrimination of Nodes)

1. Partition the input list, L, into lists L_{TRUE}, L_{FALSE}, L_{IF} and L_{APPLY} containing nodes constructed using the suggested type constructors.

2. Extract the value elements from L_{TRUE} and L_{FALSE} and form the list $LL_{BOOL} = [\texttt{trues}, \texttt{falses}]$.

3. Discriminate the `IF`-nodes in L_{IF} by partitioning on the int-part[11] and then by discriminating first with respect to the left duref (using `Duref.discriminate`) and then with

[9] In our implementation `union` is equal to `link`, since we don't use union by rank ([?][pp. 505-509]).
[10] Elements of type `nodeVal` are equivalent if they are both `TRUE`, `FALSE` or if the durefs they contain are equal according to `Duref.equal` (in the case of `IF` the int-parts must also be equal).
[11] In our implementation this is done beforehand.

```
signature Node =
sig
  eqtype node
  datatype nodeVal =
     TRUE
   | FALSE
   | IF of int * node * node
   | APPLY of node * node

  val !! : node -> nodeVal
  val tt: node
  val ff: node
  val newIf: int * node * node -> node
  val newApply: node * node -> node
  val discriminateNode: (node * 'a) list -> 'a list list
  val equal: node * node -> bool
  val discriminateNodeVal: (nodeVal * 'a) list -> 'a list list
  val partitionByContent: node list -> node list list
  val unify: node list -> unit
end
```

Figure 22: The Node signature.

respect to the right duref. This is similar to step 2.a ($i > 0$) in discrimination of dags in algorithm 5.3. `APPLY`-nodes are discriminated similarly.

4. Concatenate the four result lists.

Since we can discriminate durefs in $O(N)$ time and space where N is the number of elements, it is easy to see that the same is true of nodes. That is discrimination of nodes runs in $O(N)$ time and space where N is the number of elements in L.
`partitionByContent` is a special case of `discriminateNodeVal`. Given a list, L, of nodes it constructs the corresponding list, L', of pairs of type `(nodeVal,node)`. Then L' is discriminated using `discriminateNodeVal`.
Having accounted for discrimination of atoms, durefs and finally nodes and especially `partitionByContent`, which is the discrimination primitive we're going to use, we're ready to describe the key functions in the msd based approach.

5.2.4 build

As mentioned in section 4.2.2 bdds can be constructed from primitives, B_x, using `apply` (and this is what we do here). `build` stages calls to apply by calling itself recursively based on the boolean expression (tree) given as input. A few cases are shown in figure 23.

5.2.5 apply

The function `apply` implements a worklist algorithm. The worklist is maintained in a heap, H, (implemented in NodeHeap.{sig | sml}), which we index using the internal representation

```
| build'(x && y) = apply(build'(x),build'(y),AND)
| build'(x || y) = apply(build'(x),build'(y),OR)
```

Figure 23: Example code from build.

of the variable ordering, $0 < 1 < 2 < \ldots < n-1$. The elements in H are called *lazy-pairs* and describe delayed apply-calls. In our implementation a lazy-pair is represented by an `APPLY`-node, see figure 22. The idea is to pass through the worklist from index 0 and onwards until we reach index n which represents leafs (`TRUE` and `FALSE`). For each variable, i, we process the list of lazy-pairs in $H[i]$. The processing of $H[i]$ must identify and unify equivalent lazy-pairs in $H[i]$ and call `applyOp` on exactly one lazy-pair from each equivalence class. `applyOp` updates the worklist and creates a new leaf or `IF`-node.
Our worklist algorithm simulates the recursive behavior of the hash based apply algorithm. The identification (using discrimination) and unification of equivalent lazy-pairs in $H[i]$ ensures that we never compute the same `applyOp`-call twice[12], since `applyOp` is only called on representative elements in $H[i]$ and equal lazy-pairs would have been unified beforehand[13]. In order to justify this claim we also need to observe that there cannot be added additional lazy-pairs to $H[i]$ when we're finished processing $H[i-1]$. This is due to the way `applyOp`-calls are staged and the fact that the input bdds obey the variable ordering.
Before the worklist processing can start the worklist must be initialized with a single lazy-pair. This must be the pair containing the roots of the two bdds we want to apply an operation to. Suppose the variables represented by the roots of the two bdds are mapped to i and i' internally then the lazy-pair must be inserted in H under index j where $j = \min(i, i')$. This holds in general, so suppose we must insert lazy-pair (n_1, n_2) in H where the variables represented by n_1 and n_2 are mapped to i and i' internally. Then (n_1, n_2) must be inserted in H under index $j = \min(i, i')$.
When we're finished processing $H[n]$ we have constructed an ordered bdd describing the the result. This bdd however is not reduced since it might contain redundancies of type R4. These redundancies are removed in an upwards pass through H from n to 0.
Suppose we need to perform `apply(bdd1,bdd2,op)` then the algorithm is as follows:

Algorithm 5.6 (MSD based apply)

1. Initialize the node heap, H, with the needed number of variables (remember to make room for leafs).

2. Insert lazy-pair (n_1, n_2) in H under index j, where $j = \min(i_1, i_2)$ and where i_1 and i_2 are the mappings of the variables represented by n_1 and n_2 respectively.

3. For each i in $\{0, 1, \ldots, n\}$ starting with 0:

    (a) discriminate $H[i]$ using `Node.partitionByContent`. We use S' to refer to the result of the discrimination.

[12] This is what hashing is used for in the hash based approach.

[13] Dependent on the operator it might be possible to use a weaker equivalence than equality, e.g. in the case of `AND` and `OR` we can use equality modulo commutativity instead. We haven't implemented this optimization since it might clutter the picture when we compare with our unoptimized hash based implementation.

```
fun applyOp operator (n1, n2) =
  case (!!n1, !!n2) of
    (FALSE,FALSE) => applyOp' operator(false,false)
  | (FALSE,TRUE) => applyOp' operator(false,true)
  | (TRUE,FALSE) => applyOp' operator(true,false)
  | (TRUE,TRUE) => applyOp' operator(true,true)
  | (IF (i1, n11, n12), IF (i2, n21, n22)) =>
    (case Int.compare (i1, i2) of
      LESS => newIf (i1, lazyApply (n11, n2), lazyApply (n12, n2))
    | EQUAL => newIf (i1, lazyApply (n11, n21), lazyApply (n12, n22))
    | GREATER => newIf (i2, lazyApply (n1, n21), lazyApply (n1, n22))
    )
  | (IF (i1, n11, n12), FALSE) => newIf(i1, lazyApply (n11, n2), lazyApply (n12, n2))
  | (IF (i1, n11, n12), TRUE) => newIf(i1, lazyApply (n11, n2), lazyApply (n12, n2))
  | (FALSE, IF (i2, n21, n22)) => newIf(i2, lazyApply (n1, n21), lazyApply (n1, n22))
  | (TRUE, IF (i2, n21, n22)) => newIf(i2, lazyApply (n1, n21), lazyApply (n1, n22))
  | _ => raise Fail "Impossible: applyOp applied to one or two APPLY-nodes"
```

Figure 24: The `applyOp` function, which is a key part of the `apply` function.

(b) Unify each equivalence class $C \in S'$. That is, for each equivalence class $C \in S'$ select a canonical lazy-pair, $c \in C$, and update every node that points to a lazy-pair in C to point to c instead (this is where durefs play a key role).

(c) For all canonical lazy-pairs, c, call `applyOp(`c`)`. `applyOp` is shown in figure 24.

4. Perform an upwards pass through H from n to 0 removing redundancies of type R4[14].

Example 5.7
We end this section with a small example applying `AND` to the bdds *bdd*1 and *bdd*2 shown in figure 7 with the variable ordering $0 < 1 < 2 < 3$. In order to keep track of the individual nodes we give each node a unique identifier and we map the variables x_i to their internal representation. We use the naming scheme `id:i`, where variable x_{i+1} is mapped to i. Also we rename the leafs to `TRUE` and `FALSE` respectively. Thus we arrive at the bdds shown in figure 25.

1. According to the algorithm the first step is to initialize H. Since the variable ordering is $0 < 1 < 2 < 3$ we must initialize H to size five.

2. Then we insert lazy-pair $(1, 6)$ under $H[0]$.

$$H[0] \quad = \quad \langle (1, 6) \rangle$$

3. For each i in $\{0, 1, 2, 3, 4\}$ starting with 0:

[14] In our implementation this is done by the function `coalesceSameIndex` which can be found in appendix A.3.10.

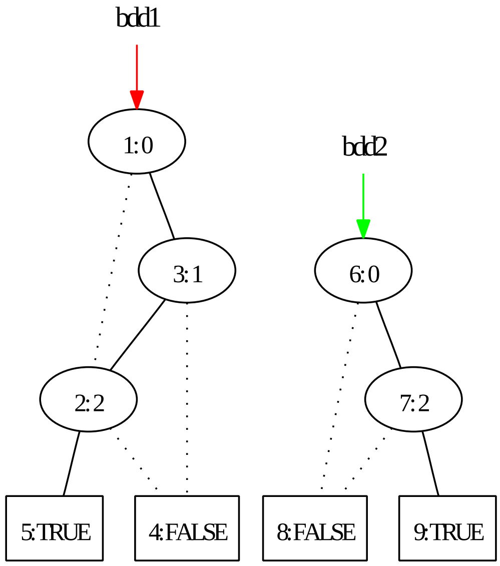


Figure 25: Input bdds to `apply(bdd1,bdd2,AND)`.

$i = 0$:

(a) $S' = \texttt{partitionByContent}(H[0]) = \{\langle(1,6)\rangle\}$.

(b) Nothing to do.

(c) $\texttt{applyOp}((1,6))$.

Now we have:

$$\begin{array}{rcl} H[0] & = & \langle(1,6)\rangle \\ H[1] & = & \langle(3,7)\rangle \\ H[2] & = & \langle(2,8)\rangle \end{array}$$

The partial result obtained thus far is shown in figure 26. The figure uses a slightly different syntax in order to capture history (how was a node created). `IF`-nodes are labeled $\texttt{IF}(i) : (id_1, id_2)$ where i refers to the internal representation of variables and (id_1, id_2) are unique id's (see figure 25) indicating how the `IF`-node was created.

$i = 1$:

(a) $S' = \texttt{partitionByContent}(H[1]) = \{\langle(3,7)\rangle\}$.

(b) Nothing to do.

(c) $\texttt{applyOp}((3,7))$.

Now we have:

$$\begin{array}{rcl} H[0] & = & \langle(1,6)\rangle \\ H[1] & = & \langle(3,7)\rangle \\ H[2] & = & \langle(2,8),(2,9)\rangle \\ H[4] & = & \langle(4,8)\rangle \end{array}$$

The partial result obtained thus far is shown in figure 27.

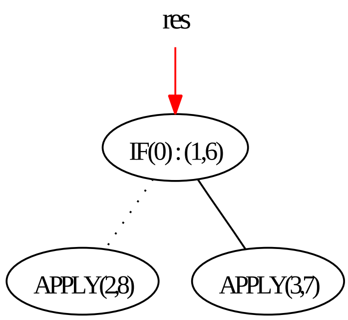


Figure 26: After step 0.

$i = 2$:

(a) $S' = \texttt{partitionByContent}(H[2]) = \{\langle(2,8)\rangle, \langle(2,9)\rangle\}$.

(b) Nothing to do.

(c) $\texttt{applyOp}((2,8))$, $\texttt{applyOp}((2,9))$.

Now we have:

$$\begin{array}{rcl} H[0] & = & \langle(1,6)\rangle \\ H[1] & = & \langle(3,7)\rangle \\ H[2] & = & \langle(2,8),(2,9)\rangle \\ H[4] & = & \langle(4,8),(5,8),(4,8),(5,9),(4,9)\rangle \end{array}$$

The partial result obtained thus far is shown in figure 28.

$i = 3$: Since there are no variables mapped to 3 internally nothing happens in this step.

$i = 4$:

(a) $S' = \texttt{partitionByContent}(H[4]) = \{\langle(4,8),(4,8)\rangle, \langle(5,8)\rangle, \langle(5,9)\rangle, \langle(4,9)\rangle\}$.

(b) We unify S' to $S'' = \{\langle(4,8)\rangle, \langle(5,8)\rangle, \langle(5,9)\rangle, \langle(4,9)\rangle\}$.

(c) $\texttt{applyOp}((4,8)) = \texttt{FALSE}$, $\texttt{applyOp}((5,8)) = \texttt{FALSE}$, $\texttt{applyOp}((5,9)) = \texttt{TRUE}$ and $\texttt{applyOp}((4,9)) = \texttt{FALSE}$.

Now we have:

$$\begin{array}{rcl} H[0] & = & \langle(1,6)\rangle \\ H[1] & = & \langle(3,7)\rangle \\ H[2] & = & \langle(2,8),(2,9)\rangle \\ H[4] & = & \langle(4,8),(5,8),(5,9),(4,9)\rangle \end{array}$$

The partial result obtained thus far is shown in figure 29.

4. The final step is to remove redundancies of type R4. The final result is shown in figure 30.

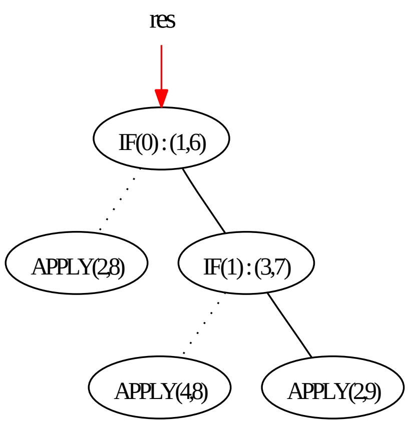


Figure 27: After step 1.

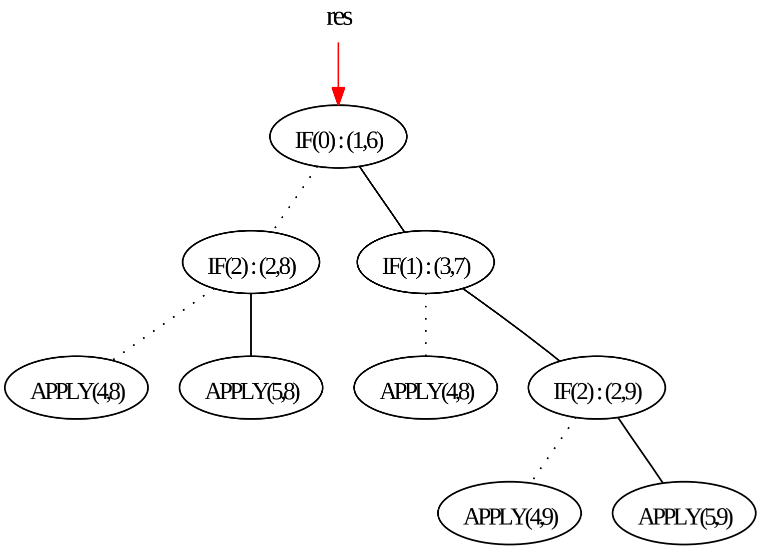


Figure 28: After step 2.

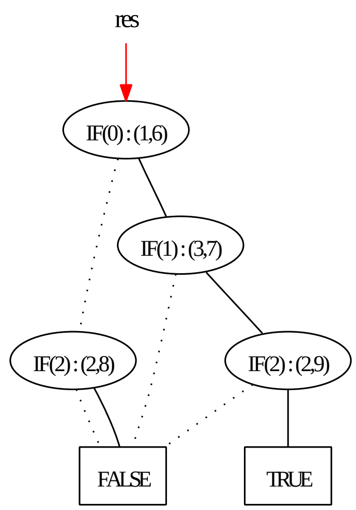


Figure 29: After step 4.

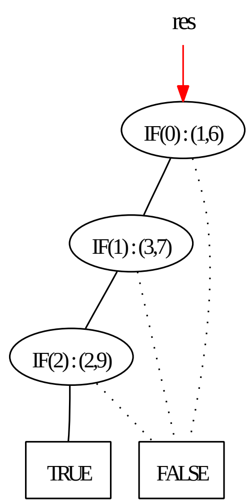


Figure 30: Final result of `apply(bdd1,bdd2,AND)`.

Table 4 Hash VS MSD based implementations.

	Hash	Msd
Garbage collection	Manual	Automatic
Maximal sharing	Yes	Semi
Excess mem. used	In tmp. hash tables	In worklist
Memory localicity	No	Expected
Dedicated `build`	Yes	No

5.3 Time and space complexities - the msd based approach

In this section we show that the msd based `apply` function runs in $O(2^n)$ time and space, where n is the number of variables in the ordering. The running time has two components. First we run through the worklist, H, two times once forwards and once backwards (step 4), this takes $O(n)$ time. The second part is discrimination and unification of the lists of lazy-pairs stored in H. Since we known that discrimination and unification runs in linear time and space in the number of lazy-pairs in the lists stored in H we just need to compute a bound on the maximal number of lazy-pairs stored in H. The height of the bdds given as input is at most n. We observe that apart from the initial lazy-pair all other lazy-pairs are inserted into H by `applyOp` which inserts exactly two lazy-pairs per call. Finally the combined height of the subbdds inserted into H is strictly less than the combined height of their parents. Thus there are at most $2n$ steps, and at step i the maximal number of lazy-pairs inserted into H is 2^i. We now have:

$$\#\text{lazy-pairs in } H \leq \sum_{i=0}^{2n} 2^i = 2^{2n+1} - 1 = O(2^n)$$

All in all we have shown that `apply` runs in time and space $O(2^n)$.

5.4 Hash based VS MSD based implementations

Having explained both the hash and msd based implementations and having seen that the central `apply` function has the same time and space complexity it is time to sum up. The key points are listed in table 4.

As mentioned earlier maximal sharing has been a driving design principle. In the msd based approach max sharing (in general) is ensured by a procedure which takes $O(2^n)$ time and space (very similar to apply), where n is the accumulated number of nodes in all of the bdds to be max shared. Evidently it is too expensive to ensure max sharing all the time and therefore we only do it when it is necessary (before comparing for equality). Thus the msd based implementation wastes some space here. In order to alleviate this one could apply some kind of amortized analysis to ensure that memory usage is never worse than the optimal amount times a constant factor. Another key to excess memory usage is the temporary data structures used in e.g. `apply`. In the hash based implementation the hash tables must be larger than the number of nodes stored in order to avoid hash clashes (see section 4.4) and in the msd based approach we insert more nodes in the worklist than will actually appear in the result bdd (e.g. node $\texttt{IF}(2):(2,8)$ in figure 29). What is worst is hard to say, but due to the way nodes are created in the msd based approach we may hope for memory locality (parents and children are stored close to each other in memory) which is not the case in the hash based approach since a good hash function should distribute data uniformly in the hash table. Having said this it is time to compare our implementations in practice.

6 Benchmarks

As mentioned in section 2.3 the selection of a good variable ordering is extremely important with respect to performance. We had hoped to be able to use standard benchmarks to test our implementations, but we haven't been able to find any benchmarks that specify which variable ordering to use and since our implementations don't do anything to select a good variable ordering they aren't suited for the benchmarks we did find. Instead we'll use a bdd based implementation (following [?][pp. 27-28]) of the well-known *N-Queens problem* to compare our implementations. Our solution of the N-Queens problem specifies a fixed variable ordering and therefore differences in performance cannot be refered to different choises of variable orderings. The bdd based solution of the N-Queens problem is by no means efficient and should only be seen as a way of generating benchmarks of varying sizes. The code can be found in NQueen.sml in appendix A.3.11.

6.1 MLton

We use MLton (see [?]) to perform our benchmarks. MLton is an optimizing whole-program compiler for Standard ML and is known to produce good code. Benchmarking is done using MLtons profiling capability which is described in some detail on the web ([?]). Amongst others MLton supports profilling of memory allocation which is the one we're going to use.

6.1.1 How we used MLton

In this section we list the commands we used to generate the benchmarks presented in the following section. We only show the command used in the msd case with $N = 6$ since the other cases are similar.

Memory :

1. `mlton -profile alloc MltonMsdQueen.sml`
2. `./MltonMsdQueen`
3. `mlprof -raw true MltonMsdQueen mlmon.out > NQ_6MSD.out`

As can be seen the first step is to compile MltonMsdQueen.sml which contains all the code needed by the msd based implementation[15]. Next we run the program and finally we use `mlprof` to extract the information we need.

6.2 Results

We ran our tests on a PC running debian GNU/Linux[16]. Unfortunately our hash based implementation didn't scale well. Compiled with MLton it wasn't able to solve the case $N = 5$ within reasonable time (15 minutes) and thus we only have one measurement of the hash based implementation. Table 5 summarizes the tests we did run and lists names on files containing the corresponding test results - the files can be found in appendix B.

[15] How MltonMsdQueen.sml is generated can be seen in our Makefile in appendix A.1.1.

[16] The PC had a 1400MHz AMD Athlon CPU with 256KB cache and 512MB of main memory.

Table 5 Overview of benchmark measurements.

N	Hash	Msd
4	NQ_4HASH.out	NQ_4MSD.out
5		NQ_5MSD.out
6		NQ_6MSD.out
7		NQ_7MSD.out
8		NQ_8MSD.out

6.3 Interpretation of results

In the case $N = 4$ the msd based implementation allocates 6,6MB of memory while the hash based implementation allocates 320MB (a factor $48,6$ more). It is interesting that 99,7% of this memory is allocated by the function `mk` (see section 4.2.1) which suggests it must be space allocated by the global hash table.
Looking at the tests of the msd based approach for $N \in \{4, 5, 6, 7, 8\}$ we see that the percentages of memory allocation for the functions involved doesn't change in any way worth mentioning. Thus `discriminateNodeVal'` accounts for roughly 20% of the allocated memory etc.
Since we haven't been able to run more tests we won't comment any further on the test results.

7 Conclusion

We have shown that *bdds can be implemented without hashing using the msd technique.* The benefits are automatic garbage collection which makes the code very easy to maintain and the possibly smaller memory consumption which our tests seemed to confirm. Having said that it is important to realize that our hash based as well as our msd based implementations do not employ any advanced optimizations and thus they shouldn't be compared to anything else but each other.

The next step in developing our msd based implementation would be to add state of the art algorithms to select good variable orderings as well as performing some obvious optimizations, e.g. discrimination modulo commutativity as mentioned in section 5.2.5, before running some standard benchmarks to get a better feel for the potential of our method.

A Code

A.1 Common code

A.1.1 Makefile

```
# Unix Makefile stub for separate compilation with Moscow ML.
MOSMLC=mosmlc -c
MOSMLL=mosmlc
MOSMLLEX=mosmllex
MOSMLYACC=mosmlyac

MLTONPREAMBLE=MltonPreamble.sml
MLTONMSDPREAMBLE=MltonMsdPreamble.sml
MLTONHASHPREAMBLE=MltonHashPreamble.sml
MLTONMSDTESTONE=MltonMsdTest1.sml
MLTONHASHTESTONE=MltonHashTest1.sml
MLTONMSDQUEEN=MltonMsdQueen.sml
MLTONHASHQUEEN=MltonHashQueen.sml

.SUFFIXES : .sig .sml .ui .uo .dot .png

.PHONY: clean png modules mltonpreamble mltonmsdpreamble mltonhashpreamble mltontest1 mltonqueen

all: modules

modules: RobddMsd.uo RobddHash.uo CnfReader.uo

png: test0.png robdd0gen.png fig3og6input.png fig3gen.png fig6gen.png apply.png\
 resrobddinput.png resrobddoutput.png existsfig6.20.png forall.png negfig6gen.png\
 satcount-beforeres.png satcount-afterres.png expsize.png linsize.png large.png app1.png app2.png

mltonpreamble:
	cat Bexp.sig > $(MLTONPREAMBLE); cat Bexp.sml >> $(MLTONPREAMBLE);\
cat Varorder.sig >> $(MLTONPREAMBLE); cat Varorder.sml >> $(MLTONPREAMBLE);\
cat Binop.sig >> $(MLTONPREAMBLE); cat Binop.sml >> $(MLTONPREAMBLE);\
cat Polyhash.sig >> $(MLTONPREAMBLE); cat Polyhash.sml >> $(MLTONPREAMBLE)

mltonmsdpreamble: mltonpreamble
	cat $(MLTONPREAMBLE) > $(MLTONMSDPREAMBLE);\
cat Atom.sig >> $(MLTONMSDPREAMBLE); cat Atom.sml >> $(MLTONMSDPREAMBLE);\
cat SimpleDURef.sig >> $(MLTONMSDPREAMBLE); cat SimpleDURef.sml >> $(MLTONMSDPREAMBLE);\
cat Node.sig >> $(MLTONMSDPREAMBLE); cat Node.sml >> $(MLTONMSDPREAMBLE);\
cat NodeHeap.sig >> $(MLTONMSDPREAMBLE); cat NodeHeap.sml >> $(MLTONMSDPREAMBLE);\
cat RobddMsd.sig >> $(MLTONMSDPREAMBLE); cat RobddMsd.sml >> $(MLTONMSDPREAMBLE);\
cat MsdTestPreamble.sml >> $(MLTONMSDPREAMBLE)

mltonhashpreamble: mltonpreamble
	cat $(MLTONPREAMBLE) > $(MLTONHASHPREAMBLE);\
cat Dynarray.sig >> $(MLTONHASHPREAMBLE); cat Dynarray.sml >> $(MLTONHASHPREAMBLE);\
cat RobddHash.sig >> $(MLTONHASHPREAMBLE); cat RobddHash.sml >> $(MLTONHASHPREAMBLE);\
cat HashTestPreamble.sml >> $(MLTONHASHPREAMBLE)


mltontest1: mltonmsdpreamble mltonhashpreamble
	cat $(MLTONMSDPREAMBLE) > $(MLTONMSDTESTONE);\
cat RobddTestSuite.sml >> $(MLTONMSDTESTONE);\
cat $(MLTONHASHPREAMBLE) > $(MLTONHASHTESTONE);\
cat RobddTestSuite.sml >> $(MLTONHASHTESTONE)

mltonqueen: mltonmsdpreamble mltonhashpreamble NQueen.sml
	cat $(MLTONMSDPREAMBLE) > $(MLTONMSDQUEEN);\
cat NQueen.sml  >> $(MLTONMSDQUEEN);\
mlton $(MLTONMSDQUEEN);\
cat $(MLTONHASHPREAMBLE) > $(MLTONHASHQUEEN);\
cat NQueen.sml >> $(MLTONHASHQUEEN);\
mlton $(MLTONHASHQUEEN)

queens.dot: modules
	mosml test.sml

queens: queens.png

.sig.ui:
	$(MOSMLC) $<

.sml.uo:
	$(MOSMLC) $<
```

```
.dot.png:
	dot -Tpng -o $*.png $*.dot

clean:
	-rm *.ui *.uo *.png *.dot *.ps Makefile.bak *~ $(MLTONPREAMBLE) $(MLTONMSDPREAMBLE) /
	$(MLTONHASHPREAMBLE) $(MLTONMSDQUEEN) $(MLTONHASHQUEEN) Mlton*.sml Mlton*

#Dependencies
Node.uo: Atom.uo SimpleDURef.uo Node.ui
SimpleDURef.uo: SimpleDURef.ui
Atom.uo: Atom.ui
Bexp.uo: Bexp.ui
Varorder.uo: Bexp.uo Varorder.ui
Binop.uo: Binop.ui
NodeHeap.uo: NodeHeap.ui
RobddMsd.uo: Bexp.uo Varorder.uo Binop.uo Node.uo NodeHeap.uo RobddMsd.ui
RobddHash.uo: Bexp.uo Varorder.uo Binop.uo RobddHash.ui
CnfReader.uo: Bexp.uo CnfReader.ui
```

A.1.2 Bexp.sig

```
signature Bexp =
sig
  datatype
    ''voelem BEXP = VAR of ''voelem | F | T | negate of ''voelem BEXP |
   && of ''voelem BEXP * ''voelem BEXP | || of ''voelem BEXP * ''voelem BEXP |
   ==> of ''voelem BEXP * ''voelem BEXP | <==> of ''voelem BEXP * ''voelem BEXP
  (* bexp[t/sub] *)
  val subst : ''voelem BEXP * ''voelem BEXP * ''voelem BEXP -> ''voelem BEXP

  (* print bexp in Graphviz dot format *)
  val prntBexp: ''voelem BEXP * (''voelem -> string) -> string
end
```

A.1.3 Bexp.sml

```
structure Bexp:> Bexp =
struct
  infix 6 &&
  infix 5 ||
  infix 4 <==>
  infix 3 ==>

  datatype
    ''voelem BEXP = VAR of ''voelem | F | T | negate of ''voelem BEXP |
   && of ''voelem BEXP * ''voelem BEXP | || of ''voelem BEXP * ''voelem BEXP |
   ==> of ''voelem BEXP * ''voelem BEXP | <==> of ''voelem BEXP * ''voelem BEXP

  (* Substitute sub for t in bexp - bexp[t/sub] *)

  fun subst(t, sub, bexp) =
      let
        fun subst_hlp(t,sub, VAR x) = VAR x
          | subst_hlp(t, sub, F) = F
          | subst_hlp(t, sub, T) = T
          | subst_hlp(t, sub, negate x) = negate (subst(t,sub,x))
          | subst_hlp(t, sub, x && y) = subst(t,sub,x) && subst(t,sub,y)
          | subst_hlp(t, sub, x || y) = subst(t,sub,x) || subst(t,sub,y)
          | subst_hlp(t, sub, x ==> y) = subst(t,sub,x) ==> subst(t,sub,y)
          | subst_hlp(t, sub, x <==> y) = subst(t,sub,x) <==> subst(t,sub,y)
      in
          if sub = bexp then t else subst_hlp(t, sub, bexp)
      end

  (* print a boolean expression in the graphviz dot format *)
  fun prntBexp(bexpr,elemToStr) =
      let
        val preStr = "digraph G {\n"
        val postStr = "}\n"
        fun p(VAR x,n) =
            let
              val strX = elemToStr(x)
              val strN = Int.toString(n)
            in
                ("node"^strN^" [label=\""^strX^"\"];\n",n)
            end
          | p(F,n) =
          let
            val strN = Int.toString(n)
          in
              ("node"^strN^" [label=\"0\" shape=box];\n",n)
          end

          | p(T,n) =
          let
            val strN = Int.toString(n)
          in
              ("node"^strN^" [label=\"1\" shape=box];\n",n)
          end
          | p(negate x,n) =
          let
            val (strX,N1) = p(x,n)
            val strN1 = Int.toString(N1)
            val strN1p1 = Int.toString(N1+1)
            val strNOT = "node"^strN1p1^" [label=\"!!\"];\n"^
                "node"^strN1p1^" -> node"^strN1^" [arrowhead=none];\n"
          in
```

```
                (strX^strNOT,N1+1)
            end
            | p(x && y,n) =
            let
              val (strX,N1) = p(x,n)
              val (strY,N2) = p(y,N1+1)
              val strN1 = Int.toString(N1)
              val strN2 = Int.toString(N2)
              val strN2p1 = Int.toString(N2+1)
              val strAND = "node"^strN2p1^" [label=\"&&\"];\n"^
                  "node"^strN2p1^" -> node"^strN1^" [arrowhead=none];\n"^
                  "node"^strN2p1^" -> node"^strN2^" [arrowhead=none];\n"
            in
                (strX^strY^strAND, N2+1)
            end
            | p(x || y,n) =
            let
              val (strX,N1) = p(x,n)
              val (strY,N2) = p(y,N1+1)
              val strN1 = Int.toString(N1)
              val strN2 = Int.toString(N2)
              val strN2p1 = Int.toString(N2+1)
              val strOR = "node"^strN2p1^" [label=\"||\"];\n"^
                  "node"^strN2p1^" -> node"^strN1^" [arrowhead=none];\n"^
                  "node"^strN2p1^" -> node"^strN2^" [arrowhead=none];\n"
            in
                (strX^strY^strOR, N2+1)
            end
            |  p(x ==> y,n) =
               let
                 val (strX,N1) = p(x,n)
                 val (strY,N2) = p(y,N1+1)
                 val strN1 = Int.toString(N1)
                 val strN2 = Int.toString(N2)
                 val strN2p1 = Int.toString(N2+1)
                 val strIMP = "node"^strN2p1^" [label=\"==>\"];\n"^
                     "node"^strN2p1^" -> node"^strN1^" [arrowhead=none];\n"^
                     "node"^strN2p1^" -> node"^strN2^" [arrowhead=none];\n"
               in
                   (strX^strY^strIMP, N2+1)
               end
            | p(x <==> y,n) =
               let
                 val (strX,N1) = p(x,n)
                 val (strY,N2) = p(y,N1+1)
                 val strN1 = Int.toString(N1)
                 val strN2 = Int.toString(N2)
                 val strN2p1 = Int.toString(N2+1)
                 val strBIMP = "node"^strN2p1^" [label=\"<==>\"];\n"^
                     "node"^strN2p1^" -> node"^strN1^" [arrowhead=none];\n"^
                     "node"^strN2p1^" -> node"^strN2^" [arrowhead=none];\n"
               in
                   (strX^strY^strBIMP, N2+1)
               end
        in
            preStr^(#1(p(bexpr,0)))^postStr
        end
end
```

A.1.4 Varorder.sig

```
signature Varorder =
sig
  eqtype ''voelem varorder
  (* type ''voelem varorder = (''voelem * int) list *)
  type ''voelem BEXP = ''voelem Bexp.BEXP

  val varorderFromLst : ''voelem list -> ''voelem varorder
  val varOrdElemToInt : (''voelem varorder * ''voelem) -> int option
  val intToVarOrdElem : (''voelem varorder * int) -> ''voelem option
  val inOrder : (''voelem * ''voelem varorder) -> bool
  val idxInOrder : (int * ''voelem varorder) -> bool
  val checkVarOrder : ''voelem varorder -> bool
  val validate : (''voelem varorder * ''voelem BEXP) -> bool
  val maxVar : ''voelem varorder -> int
  val minVar : ''voelem varorder -> int
  val getNoVars : ''voelem varorder -> int
  val restrictVo : ''voelem varorder * ''voelem -> ''voelem varorder
end
```

A.1.5 Varorder.sml

```
(*
A varorder is a representation of a set of variables Vset as numbers between
0 and #Vset-1
*)

structure Varorder :> Varorder =
struct

  open Bexp (* This module depends on the Bexp module *)
  infix 6 &&
  infix 5 ||
  infix 4 <==>
  infix 3 ==>

  type ''voelem varorder = (''voelem * int) list

  (* Find idx corresponding to elem in varorder *)
  fun varOrdElemToInt(varorder,elem) =
      let
        fun getIdx((var,idx)::vars,elem) =
            if var=elem then SOME idx else getIdx(vars,elem)
          | getIdx([],_) = NONE
      in
          getIdx(varorder,elem)
      end

  (* Find elem corresponding to idx in varorder *)
  fun intToVarOrdElem(varorder,i) =
      let
        fun getVar((var,idx)::vars,varidx) =
            if idx=varidx then SOME var else getVar(vars,varidx)
          | getVar([],_) = NONE
      in
          getVar(varorder,i)
      end

  (* Determine whether elem is in varorder *)
  fun inOrder(elem,[]) = false
    | inOrder(elem,(v,_)::vars) = if elem=v then true else inOrder(elem,vars)

  (* Determine whether idx is in varorder *)
  fun idxInOrder(idx,[]) = false
    | idxInOrder(idx,(_,i)::vars) = if idx=i then true else idxInOrder(idx,vars)

  (* Makes sure that varorder and idx are unique and that 0<= idxes < #Vset *)
  fun checkVarOrder(vo) =
      let
        val sizeVO = length vo
        fun checkVarOrderHlp([]) = true
          | checkVarOrderHlp((v,i)::vs) = not(inOrder(v,vs)) andalso not(idxInOrder(i,vs))
          andalso 0<= i andalso i < sizeVO andalso checkVarOrderHlp(vs)
      in
          checkVarOrderHlp(vo)
      end

  (* Makes sure every var used in bexp occurs in varorder *)
```

```
  fun validate(varorder,bexp) =
      let
        fun checkBexp(VAR x) = inOrder(x,varorder)
          | checkBexp(F) = true
          | checkBexp(T) = true
          | checkBexp(negate x) = checkBexp(x)
          | checkBexp(x && y) = checkBexp(x) andalso checkBexp(y)
          | checkBexp(x || y) = checkBexp(x) andalso checkBexp(y)
          | checkBexp(x ==> y) = checkBexp(x) andalso checkBexp(y)
          | checkBexp(x <==> y) = checkBexp(x) andalso checkBexp(y)
    in
        checkBexp(bexp)
    end

  (* Create a varOrder from a list *)
  fun varorderFromLst(varOrdLst) =
      let
        fun varorderFromLst_hlp([],_) = []
          | varorderFromLst_hlp(v::vs,n) =
          (v,n):: varorderFromLst_hlp(vs,n+1)
      in
          varorderFromLst_hlp(varOrdLst,0)
      end

  (* Get maximal idx used in varorder *)
  fun maxVar(varorder) =
      let
        fun maxVar_hlp([],max) = max
          | maxVar_hlp((_,idx)::vs,max) =
          if idx > max then maxVar_hlp(vs,idx)
          else maxVar_hlp(vs,max)
      in
          maxVar_hlp(varorder,0)
      end

  (* Get minimal idx used in varorder *)
  fun minVar(varorder) =
      let
        fun minVar_hlp([],min) = min
          | minVar_hlp((v,idx)::vs,min) =
          if idx < min then minVar_hlp(vs,idx)
          else minVar_hlp(vs,min)

        val (v,i) = hd varorder;
      in
          minVar_hlp(tl varorder,i)
      end

  (* Get number of vars in varorder *)
  local
    fun getNoVarsHlp([],res) = res
      | getNoVarsHlp(x::xs,res) = getNoVarsHlp(xs,res+1)
  in
    fun getNoVars(varorder) = getNoVarsHlp(varorder,0)
  end

  fun restrictVo(v1, elem) = List.filter (fn (v,i) => v <> elem) v1
end
```

A.1.6 Binop.sig

```
signature Binop =
sig
  type operator = bool * bool -> bool

  val isA : operator -> bool
  val isC : operator -> bool
  val isI : operator -> bool
  val isACI : operator -> bool

  val AND : operator
  val OR : operator
  val BIMP : operator
  val IMP : operator
  val XOR : operator
end
```

A.1.7 Binop.sml

```
structure Binop :> Binop =
struct
  type operator = bool * bool -> bool

  local
    fun testA operator (a,b,c) =
          operator(operator(a,b),c) = operator(a,operator(b,c))
  in
    fun isA operator =
        let val testOp = testA operator
        in
            testOp (false,false,false) andalso
            testOp (false,false,true) andalso
            testOp (false,true,false) andalso
            testOp (false,true,true) andalso
            testOp (true,false,false) andalso
            testOp (true,false,true) andalso
            testOp (true,true,false) andalso
            testOp (true,true,true)
        end
  end

  fun isC operator = operator(true,false) = operator (false,true)

  fun isI operator = operator(true,true) = true andalso operator(false,false) = false

  fun isACI operator = isA operator andalso isC operator andalso isI operator

  fun AND(x,y) = x andalso y;
  fun OR(x,y) = x orelse y;
  fun BIMP(x,y) = x=y;
  fun IMP(x,y) = not(x) orelse y;
  fun XOR(x,y) = (not(x) andalso y) orelse (x andalso not(y));
end
```

A.2 Hash code

A.2.1 RobddHash.sig

```
signature RobddHash =
sig
  type ''voelem BEXP = ''voelem Bexp.BEXP
  type ''voelem varorder = ''voelem Varorder.varorder
  type operator = Binop.operator
  type ''voelem robdd (* = int ref * ''voelem varorder *)

  (* Build robdd from BEXP using varorder *)
  val build : ''voelem varorder * ''voelem BEXP -> ''voelem robdd

  (* print robdd in Graphviz dot format *)
  val toString : ''voelem robdd * (''voelem -> string) -> string

  (* robdd1 operator robdd2 *)
  val apply : ''voelem robdd * ''voelem robdd * operator -> ''voelem robdd
  (* restrict this robdd *)
  val restrict : ''voelem robdd * ''voelem * bool -> ''voelem robdd
  (* construct negated robdd *)
  val neg : ''voelem robdd -> ''voelem robdd
  (* create exists robdd *)
  val exists : ''voelem robdd * ''voelem -> ''voelem robdd
  (* create forall robdd *)
  val forall : ''voelem robdd * ''voelem -> ''voelem robdd
  (* find number of satisfying variable assignments *)
  val satcount : ''voelem robdd -> int
  (* find a variable assignment - only strict requirements are returned *)
  val anysat : ''voelem robdd -> (''voelem * int) list
  (* find all satisfying assignments - only strict requirements are returned *)
  val allsat : ''voelem robdd -> (''voelem * int) list list
  (* test two robdds for equality *)
 val equal : ''voelem robdd * ''voelem robdd -> bool

end
```

A.2.2 RobddHash.sml

```
structure RobddHash :> RobddHash =
struct
  infix 6 &&
  infix 5 ||
  infix 4 <==>
  infix 3 ==>

  open Bexp
  open Varorder
  open Binop

  type ''voelem BEXP = ''voelem Bexp.BEXP
  type ''voelem varorder = ''voelem Varorder.varorder
  type operator = Binop.operator

  type ''voelem robdd = int ref * ''voelem varorder

  val HINT_UPPER = 1000; (* Hint on size of Ttbl and Htbl *)

  (* used to initialize Polyhash *)
  fun sameKey(x,y) = x=y

(* pair is stolen from Henrik Reif Andersen *)
  fun pair(i,j) =
       let fun realmod (x, q) = floor(x-q*real(floor(x/q)))
         fun p(i,j) = realmod ((i+j)*(i+j+1.0)/2.0 + i, 536870912.0)
       in  p(real i, real j) end
  handle Overflow =>
     (print("Overflow in pair: i=" ^ Int.toString(i) ^
              ", j=" ^ Int.toString(j) ^"\n");0)

(*
  fun pair(i,j) = ((i+j)*(i+j+1)) div 2 + i
       handle Overflow => 1
*)

  (* hash used for memoization *)
  fun hashHtbl(i,v0,v1) =
       (*    (pair(i,pair(v0,v1)) mod HASH_PRIME) mod HASH_MOD; *)
```

```
        pair(i,pair(v0,v1))

    exception entryNotInHash

    (* inserts value in hashTbl and returns value *)
    fun insertInHash(hashTbl,key,value) =
        (Polyhash.insert hashTbl (key,value); value)

    fun init() =
        let
          (* Impl of table T: u -> (i,l,h) *)
          val Ttbl = Dynarray.array(HINT_UPPER, (0,0,0))
          (* Impl of table H (i,l,h) -> u *)
          val Htbl = Polyhash.mkTable(hashHtbl,sameKey) (HINT_UPPER,entryNotInHash)
        in
            (Dynarray.update(Ttbl,0,(~1,~1,~1)); (* Init 0 var(0)=~1 *)
             Dynarray.update(Ttbl,1,(~2,~1,~1)); (* Init 1 var(1)=~2 *)
             (ref 2,Ttbl,Htbl)) (* initial noted init to 0 *)
        end

    val (nxtIdxRef,Ttbl,Htbl) = init() (* global data store *)

    (* Insert (i,l,h) in Array Ttbl in next free slot *)
    fun add(i,l,h) =
        (Dynarray.update(Ttbl,!nxtIdxRef,(i,l,h));
         nxtIdxRef := !nxtIdxRef + 1; !nxtIdxRef-1)

    (* return all values assoc with u *)
    fun all(u) =
        let
          val (i,l,h) = Dynarray.sub(Ttbl,u);
        in
            (i,l,h)
        end

    (* return var u *)
    fun var(u) =
        let
          val (i,_,_) = Dynarray.sub(Ttbl,u)
        in
            i
        end

    (* return low son *)
    fun low(u) =
        let
          val (_,l,_) = Dynarray.sub(Ttbl,u)
        in
            l
        end

    (* return high son *)
    fun high(u) =
        let
          val (_,_,h) = Dynarray.sub(Ttbl,u)
        in
            h
        end

    fun member(i,l,h) =
        let
          val peek = Polyhash.peek Htbl (i,l,h)
        in
          case peek of
            SOME n => true
          | NONE => false
        end

    fun lookup(i,l,h) = Polyhash.find Htbl (i,l,h) (* raises entryNotInHash *)

    fun insert(i,l,h,u) = Polyhash.insert Htbl ((i,l,h),u)

    fun mk(i,l,h) =
        if l = h then l
        else if member(i,l,h) then lookup(i,l,h)
             else let
                    val u = add(i,l,h);
                  in
                      insert(i,l,h,u); u
                  end
```

```
  (* evaluate  b1 op b2 were b1,b2 \in {T,F} *)
  fun partBoolEval(VAR x) = VAR x
    | partBoolEval(F) = F
    | partBoolEval(T) = T
    | partBoolEval(negate x) =
    let
      val Ex = partBoolEval(x)
    in
        if Ex=T then F else if Ex=F then T else negate Ex
    end
    | partBoolEval(x && y) =
    let
      val Ex = partBoolEval(x)
      val Ey = partBoolEval(y)
    in
        if Ex=T andalso Ey=T then T
        else if Ex=F orelse Ey=F then F
             else Ex && Ey
    end
    | partBoolEval(x || y) = partBoolEval(negate(negate x && negate y)) (* de morgan *)
    | partBoolEval(x ==> y) = partBoolEval(negate x || y)
    | partBoolEval(x <==> y) = partBoolEval( (x ==> y) && (y ==> x))

  (* it took a long time to solve x ==> (y && y) ==> x sigh *)

  (* Create robdd from a variable order and a bexp *)
  fun build(varorder,bexp) =
      if not(checkVarOrder(varorder)) then raise Fail "Varorder is inconsistent\n"
      else if not(validate(varorder,bexp)) then raise Fail "All vars not in varorder\n"
      else
        let
            val n = maxVar(varorder)
            val robdd as (u,varord) = (ref ~1,varorder)
            fun build'(t,i) =
                if i > n then
                    if t = F then 0 else 1
                else let
                        val elemRepVarI = intToVarOrdElem(varorder,i)
                    in
                        case elemRepVarI of
                            SOME el =>
                            let
                                val v0 = build'(partBoolEval(subst(F,(VAR el),t)),i+1)
                                val v1 = build'(partBoolEval(subst(T,(VAR el),t)),i+1)
                            in
                                mk(i,v0,v1)
                            end
                          | NONE =>
                           let
                                val v0 = build'(t,i+1)
                                val v1 = build'(t,i+1)
                            in
                                print("WARNING: idx "^Int.toString(i)^" not in varorder\n");
                                mk(i,v0,v1)
                            end

                    end
        in
            u:= build'(bexp,0); robdd
        end

(* hash used for memoization *)
fun hashApply(i,j) = pair(i,j)

fun boolToInt(true) = 1
  | boolToInt(false) = 0

(* apply *)
fun apply(robdd1 as (u1,varorder1),robdd2 as (u2,varorder2), opr) =
    if varorder1<>varorder2 then raise Fail "Robdd ordering mismatch\n"
    else
        let
            val newRobdd as (u,_) = (ref ~1,varorder1)
            val HINT_GTBL_SIZE = 100
            val Gtbl = Polyhash.mkTable(hashApply,sameKey) (HINT_GTBL_SIZE,entryNotInHash)
            fun app(u1,u2) =
                let
                    val Gu1u2 = Polyhash.peek Gtbl (u1,u2)
                in
```

```
                    case Gu1u2 of
                        SOME n => n
                      | NONE =>
                       if (u1=0 orelse u1=1) andalso (u2=0 orelse u2=1) then
                           insertInHash(Gtbl,(u1,u2),boolToInt(
                                   opr(u1=1,u2=1)))
                       else let
                               val varu1 = var(u1)
                               val lowu1 = low(u1)
                               val highu1 = high(u1)
                               val varu2 = var(u2)
                               val lowu2 = low(u2)
                               val highu2 = high(u2)

                           in
                               if varu1=varu2 then
                                   insertInHash(Gtbl,(u1,u2),
                                   mk(varu1,app(lowu1,lowu2),app(highu1,highu2)))
                               else if
                                   (0 <= varu1 andalso varu1 < varu2) orelse varu2 < 0 then
                                   insertInHash(Gtbl,(u1,u2),
                                           mk(varu1,app(lowu1,u2),app(highu1,u2)))
                                 else (* if
                                     (0 <= varu2  andalso varu2 < varu1) orelse varu1 < 0 then *)
                                     insertInHash(Gtbl,(u1,u2),
                                             mk(varu2,app(u1,lowu2),app(u1,highu2)))
                           end
                end
        in
            u := app(!u1,!u2); newRobdd (* update init node and return new robdd *)
        end

(* restrict this robdd *)
fun restrict(robdd as (u,varorder),varX,b) =
    if not(inOrder(varX,varorder)) then raise Fail "Restrict impossible variable not i varorder"
    else
        let
            val varOrdElemIdx = varOrdElemToInt(varorder,varX)
            val newVarorder = restrictVo(varorder,varX)
        in
            case varOrdElemIdx of
                SOME j =>
                let
                    val newRobdd as (newU,_) = (ref ~1,newVarorder)

                    fun res(u) =
                        let
                            val varu = var(u)
                            val lowu = low(u)
                            val highu = high(u)
                        in
                            if varu > j orelse varu < 0 then u
                            else if varu < j then mk(varu,res(lowu),res(highu))
                            else (* var(u)=j *) if b = false then res(lowu)
                                                else (* var(u)=j, b=true *) res(highu)
                        end
                in
                    newU:= res(!u); newRobdd
                end
              | NONE =>
                robdd (* no var to restrict *)
        end


(* hash used for memoization *)
fun hashSatCount(i) = i

(* satcount *)
fun satcount(r1 as (n1, v1)) =
    let
        val n = maxVar(v1)
        val HINT_GTBL_SIZE = 100
        val Gtbl = Polyhash.mkTable(hashSatCount,sameKey) (HINT_GTBL_SIZE,entryNotInHash)

        (* varu1idx <= varu2idx : computes the number of free vars between the two *)
        fun freeBetween(varu1idx, varu2idx) =
            let
              fun freeBetween'(testNow,free) =
```

```
                  if testNow < varu2idx then
                      if idxInOrder(testNow,v1) then freeBetween'(testNow+1,free+1)
                      else freeBetween'(testNow+1,free)
                  else free
              in
                  freeBetween'(varu1idx+1,0)
              end

          fun getVarNum(node) =
              let
                  val varNode = var(node)
              in
                  if varNode < 0 then (* this is a leaf node (true/false) *) n+1
                  else varNode
              end

          fun count(node) =
              let
                  val Gu = Polyhash.peek Gtbl node
              in
                  case Gu of
                      SOME n => n
                    | NONE =>
                     if node=0 then insertInHash(Gtbl,node,0)
                     else if node=1 then insertInHash(Gtbl,node,1)
                     else
                         let
                           val varNode = var(node)
                           val lowNode = low(node)
                           val highNode = high(node)

                           val countHigh = count(highNode)
                           val countLow = count(lowNode)
                       in
                           if countHigh <> 0 then
                               if countLow <> 0 then
                                   insertInHash(Gtbl,node,
                                    floor(Math.pow(2.0,real(
                                    freeBetween(varNode,getVarNum(lowNode)))))*countLow+
                                    floor(Math.pow(2.0,real(
                                    freeBetween(varNode,getVarNum(highNode)))))*countHigh)
                               else insertInHash(Gtbl,node,
                                    floor(Math.pow(2.0,real(
                                    freeBetween(varNode,getVarNum(highNode)))))*countHigh)
                           else
                               if countLow <> 0 then
                                   insertInHash(Gtbl,node,
                                    floor(Math.pow(2.0,real(
                                    freeBetween(varNode,getVarNum(lowNode)))))*countLow)
                               else insertInHash(Gtbl,node,0)
                       end
              end
      in
          floor(Math.pow(2.0,real(freeBetween(minVar(v1)-1,getVarNum(!n1)))))*count(!n1)
      end


(* anysat only strict requirements are computed *)
fun anysat(robdd as (n1,v1)) =
      let
          fun anysat'(node) =
              if node=0 then raise Fail "No satisfying truth assignment exists\n"
              else if node=1 then []
              else let
                      val lowNode = low(node)
                      val highNode = high(node)
                      val varNode = var(node)
                      val varOrdElem = intToVarOrdElem(v1,varNode)
                  in
                      case varOrdElem of
                          SOME varOE =>
                           if lowNode=0 then (varOE,1):: anysat'(highNode)
                           else (varOE,0) :: anysat'(lowNode)
                        | NONE => raise Fail "Error in anysat'\n"
                  end
      in
          anysat'(!n1)
      end

(* all sat *)
```

```
exception allSatInternalError
fun allsat(robdd as (ref u,varorder)) =
    let
        fun allsat'(u) =
            if u=0 then []
            else if u=1 then [[]]
            else let
                    fun consmap elem lst = elem::lst
                    val lowu = low(u)
                    val highu = high(u)
                    val varu = var(u)
                    val varOrdElem = intToVarOrdElem(varorder,varu)
                in
                    case varOrdElem of
                        SOME varOE =>
                        map (consmap (varOE,0)) (allsat'(lowu))   @
                        map (consmap (varOE,1)) (allsat'(highu))
                      | NONE => raise allSatInternalError
                end
    in
        allsat'(u)
    end


(* hash used for memoization *)
fun hashNeg(i) = i

(* neg: negate robdd *)
fun neg(robdd as (ref u,varorder)) =
    let
        val HINT_GTBL_SIZE = 100
        val Gtbl = Polyhash.mkTable(hashNeg,sameKey) (HINT_GTBL_SIZE,entryNotInHash)
        fun neg'(u1) =
            let
                val Gu = Polyhash.peek Gtbl u1
            in
                case Gu of
                    SOME n => n
                  | NONE =>
                    if u1=0 then insertInHash(Gtbl,u1,1)
                    else if u1=1 then insertInHash(Gtbl,u1,0)
                    else let
                            val varu = var(u1)
                            val lowu = low(u1)
                            val highu = high(u1)
                            val newLowU = neg'(lowu)
                            val newHighU = neg'(highu)
                        in
                            insertInHash(Gtbl,u1,mk(varu,newLowU,newHighU))
                        end
            end
    in
        (ref (neg'(u)),varorder)
    end


(* exists *)
fun exists(robdd,x) =
    let
        val lowResRobdd = restrict(robdd,x,false)
        val highResRobdd = restrict(robdd,x,true)
    in
        apply(lowResRobdd,highResRobdd, OR)
    end

(* forall *)
fun forall(robdd,x) =
    let
        val lowResRobdd = restrict(robdd,x,false)
        val highResRobdd = restrict(robdd,x,true)
    in
        apply(lowResRobdd,highResRobdd, AND)
    end


(* hash used for memoization *)
fun hashEqual(i,j) = pair(i,j)

infix xor
```

```
fun x xor y = (x andalso not(y)) orelse (not(x) andalso y)

(* Since we have full sharing equality is a question of rootNode equality *)
fun equal(robdd1 as (u1,varorder1), robdd2 as (u2,varorder2)) =
    if varorder1 <> varorder2 then raise Fail "Robdd Order mishatch"
    else !u1 = !u2

(* End Core Functions *)

(* hash used for memoization *)
fun hashPrntRobdd(i) = i

(* print a robdd in graphviz dot format. *)
fun toString(robdd as (ref u,varorder),elemToStr) =
    let
        val preStr = "digraph G {\n"
        val postStr = "}\n"
        val HINT_GTBL_SIZE = 100
        val Gtbl = Polyhash.mkTable(hashPrntRobdd,sameKey) (HINT_GTBL_SIZE,entryNotInHash)
        fun prnt'(node) =
            let
                val printed = Polyhash.peek Gtbl node
            in
                case printed of
                    SOME n => ""
                  | NONE =>
                    let
                        val strU = Int.toString(node)
                    in
                        if node=0 then (insertInHash(Gtbl,node,true);
                        "node"^strU^" [label=\"0\" shape=box];\n")
                        else if node=1 then (insertInHash(Gtbl,node,true);
                        "node"^strU^" [label=\"1\" shape=box];\n")
                        else let
                                val varOrdElem = intToVarOrdElem(varorder,var(node))
                            in
                                case varOrdElem of
                                    SOME elem =>
                                    let
                                        val lowStr  = prnt'(low(node))
                                        val highStr = prnt'(high(node))
                                        val strLowU = Int.toString(low(node))
                                        val strHighU = Int.toString(high(node))
                                        val strVarU = elemToStr(elem)
                                        val thisStr = "node"^strU^" [label=\""^strVarU^"\"];\n"^
                                                      "node"^strU^" -> node"^strLowU^
                                                      " [arrowhead=none style=dotted];\n"^
                                                      "node"^strU^" -> node"^strHighU^
                                                      " [arrowhead=none];\n"
                                    in
                                        insertInHash(Gtbl,node,true); lowStr^highStr^thisStr
                                    end
                                  | NONE =>
                                    raise Fail "No var for int"
                            end
                    end
            end
    in
        preStr^prnt'(u)^postStr
    end

end
```

A.3 MSD code

A.3.1 Atom.sig

```
signature Atom =
sig

  eqtype elmt

  val new : unit -> elmt
  val discriminate : (elmt * 'a) list -> 'a list list
  val equal: elmt * elmt -> bool
end
```

A.3.2 Atom.sml

```
(* Atom: Type of dynamically generated atoms with discrimination *)

structure Atom :> Atom =
struct

  type elmt = exn ref

  exception EMPTY
  fun new () = ref EMPTY

  fun discriminate nil = nil
    | discriminate [(_, v)] = [[v]]
    | discriminate (args: (elmt * 'v) list): 'v list list =
        let exception VAL of 'v list
            val atoms =
                List.foldl (fn ((atom as ref EMPTY, v), atoms) => (atom := VAL [v]; atom :: atoms)
                             | ((atom as ref (VAL vs), v), atoms) => (atom := VAL (v :: vs); atoms)
                             | ((ref exn, _), _) => raise exn) nil args
        in List.map (fn (atom as ref (VAL vs)) => (atom := EMPTY; vs)
                      | ref exn => raise exn) atoms
        end

  val equal = op=
end
```

A.3.3 SimpleDURef.sig

```
(* Discriminatable unionable references (durefs)
 *
 * Interface to UnionFind package with support for discrimination.
 *
 * Author:
 *     Fritz Henglein
 *     DIKU, University of Copenhagen
 *     henglein@diku.dk
 *
 * DESCRIPTION
 *
 * Union/Find data type with ref-like interface
 * and support for discrimination.  A Union/Find structure
 * consists of a type constructor 'a duref with operations for
 * making an element of 'a duref (duref), getting the contents of
 * an element (!!), updating the contents (::=),
 * discriminating (discriminate), and
 * for joining two elements (union, link, unify).
 * duref is analogous to ref as expressed in the following table:
 *
 * -------------------------------------------------------------------
 * type                   'a ref              'a duref
 * -------------------------------------------------------------------
 * introduction           ref                 duref
 * elimination            !                   !!
 * equality               =                   discriminate (generalized equality)
 * updating               :=                  ::=
 * unioning                                   link, union, unify
 * -------------------------------------------------------------------
 *
 * The main difference between 'a ref and 'a duref is in the unioning
 * operations and support for linear time discrimination (generalized
 * equality).  Without union 'a ref and 'a duref can be used basically
 * interchangebly.  An assignment to a reference changes only the
 * contents of the reference, but not the reference itself.  In
 * particular, any two pointers that were different (in the sense of the
 * equality predicate = returning false) before an assignment will still
 * be so.  Their contents may or may not be equal after the assignment,
 * though.  In contrast, applying the union operations (link, union,
 * unify) to two duref elements makes the two elements themselves
 * 'equal' (in the sense of the predicate equal returning true).  As a
 * consequence their contents will also be identical: in the case of link
 * and union it will be the contents of one of the two unioned elements,
 * in the case of unify the contents is determined by a binary
 * function parameter.  The link, union, and unify functions return true
 * when the elements were previously NOT equal.
 *)

signature SimpleDURef =
  sig

    eqtype 'a duref (* TODO: made duref an equality type *)
        (* type of duref-elements with contents of type 'a *)

    val duref: 'a -> 'a duref
        (* duref x creates a new discriminable, unionable reference with contents x *)

    val discriminate: ('a duref * 'b) list -> 'b list list
        (* discriminate [(d1, v1), ..., (dn, vn)] partitions the vi according to
         * equality on corresponding di's *)

    val equal: 'a duref * 'a duref -> bool
        (* equal (e, e') returns true if and only if e and e' are either made by
         * the same call to duref or if they have been unioned (see below);
         * equal (d1, d2) is equivalent to length (discriminate [(d1, ()), (d2, ())]) = 1
         *)

    val !! : 'a duref -> 'a
        (* !!e returns the contents of e.
         * Note: if 'a is an equality type then !!(duref x) = x, and
         * equal(duref (!!x), x) = false.
         *)

    val ::= : 'a duref * 'a -> unit
        (* ::= (e, x) updates the contents of e to be x *)

    val unify : ('a * 'a -> 'a) -> 'a duref * 'a duref -> unit
        (* unify f (e, e') makes e and e' equal; if v and v' are the
         * contents of e and e', respectively, before unioning them,
```

```
          * then the contents of the unioned element is f (v, v').
          *)

    val union : 'a duref * 'a duref -> unit
        (* union (e, e') makes e and e' equal; the contents of the unioned
         * element is the contents of one of e and e' before the union operation.
         * After union (e, e') elements e and e' will be congruent in the
         * sense that they are interchangeable in any context.
         *)

    val link : 'a duref * 'a duref -> unit
        (* link (e, e') makes e and e' equal; the contents of the linked
         * element is the contents of e' before the link operation.
         *)

  end; (* DUREF *)
```

A.3.4 SimpleDURef.sml

```
(* simple-uref.sml
 *
 * UNIONFIND DATA STRUCTURE WITH PATH COMPRESSION
 * AND MULTISET DISCRIMINATION
 *
 * Author:
 *     Fritz Henglein
 *     DIKU, University of Copenhagen
 *     henglein@diku.dk
 *)

structure SimpleDURef :> SimpleDURef =
  struct

    datatype 'a durefC
      = ECR of 'a * Atom.elmt
      | PTR of 'a duref
    withtype 'a duref = 'a durefC ref

(*
    datatype 'a durefC
      = ECR of 'a * Atom.elmt
      | PTR of 'a durefC ref

   type 'a duref = 'a durefC ref
*)
    fun find (p as ref (ECR _)) = p
      | find (p as ref (PTR p')) =
          let val p'' = find p'
          in p := PTR p''; p''
          end

    fun duref x = ref (ECR (x, Atom.new()))

    fun !! p = let val ref (ECR (x, _)) = find p
               in x
               end

fun discriminate ns = (* fun needed due to mlton *)
    (Atom.discriminate o
    map (fn (p, v) => let val ref (ECR (_, a)) = find p in (a, v) end)) ns
(*
    val discriminate =
        Atom.discriminate o
        map (fn (p, v) => let val ref (ECR (_, a)) = find p in (a, v) end)
*)
    fun equal (p, p') = (find p = find p')

    fun ::= (p, x) =
        let val p' as ref (ECR (_, a)) = find p
        in p' := ECR (x, a)
        end

    fun link (p, q) =
        let val p' = find p
            val q' = find q
        in if p' = q' then () else p' := PTR q'
        end

    val union = link

    fun unify f (p, q) =
```

```
        let val v = f(!!p, !!q)
        in union (p, q) before ::= (q, v)
        end

  end (* SimpleDURef *)
```

A.3.5 Node.sig

```
(*
 * NODE: BDD-nodes, consisting of terminal nodes (Booleans true and false),
 * IF-nodes (decisions) and delayed nodes.
 * A node contains a node value.  New nodes can be generated dynamically.
 * Nodes are updatable;
 * updateability is only used to implement efficient unification of multiple
 * nodes (setting their contents to be the same).
 * Two nodes are _equal_ if they are represented by the same reference.
 * Two nodes are _content equivalent_ at some point in time during exeuction
 * if the node values they contain at that time are structurally equal; that is, if
 * the respective node values are equal in the sense of SML-equality.
 * Note that both content equivalence and node equality (induced by the representation
 * of nodes by references) are required in the implementation:
 * content equivalence is the "standard" equivalence on nodes;
 * structural equality on node values, however, relies on node equality.
 *)

signature Node =
sig

  eqtype node                          (* nodes *)
  datatype nodeVal =                  (* node values (contents of nodes) *)
     TRUE                             (* true *)
   | FALSE                            (* false *)
   | IF of int * node * node          (* IF-node value with given variable index and child nodes *)
   | APPLY of node * node             (* delayed node ("apply node") with implicit operator *)

  val !! : node -> nodeVal            (* Get contents of node *)

  val tt: node                        (* Node containing true *)
  val ff: node                        (* Node containing false *)
  val newIf: int * node * node -> node   (* Generate new IF-node with given variable index and nodes *)
  val newApply: node * node -> node   (* Generate new delayed node with given pair of nodes *)

  val discriminateNode: (node * 'a) list -> 'a list list
                                      (* Discriminate nodes according to note equality *)
  val equal: node * node -> bool      (* Decide node equality *)
  val discriminateNodeVal: (nodeVal * 'a) list -> 'a list list
                                      (* Discriminate nodeVals according to structural equality *)
  val partitionByContent: node list -> node list list
                                      (* Partitions nodes by structural equality on their contents *)
  val unify: node list -> unit        (* Set all nodes equal, with contents of first element in
                                       node list; has no effect if input list is empty *)
end
```

A.3.6 Node.sml

```
structure Node :> Node =
struct

  open SimpleDURef

  datatype nodeVal =
     TRUE
   | FALSE
   | IF of int * nodeVal duref * nodeVal duref
   | APPLY of nodeVal duref * nodeVal duref

   type node = nodeVal duref

  val !! = !!

  val tt = duref TRUE
  val ff = duref FALSE
  val newIf = duref o IF
  val newApply = duref o APPLY

  val discriminateNode = discriminate
  val equal = equal

  fun discriminateNodeVal' (trues, falses, ifs, applies) ((TRUE, v) :: rest) =
```

```
          discriminateNodeVal' (v :: trues, falses, ifs, applies) rest
      | discriminateNodeVal' (trues, falses, ifs, applies) ((FALSE, v) :: rest) =
          discriminateNodeVal' (trues, v :: falses, ifs, applies) rest
      | discriminateNodeVal' (trues, falses, ifs, applies) ((IF (_, n1, n2), v) :: rest) =
          discriminateNodeVal' (trues, falses, (n1, (n2, v)) :: ifs, applies) rest
      | discriminateNodeVal' (trues, falses, ifs, applies) ((APPLY (n1, n2), v) :: rest) =
          discriminateNodeVal' (trues, falses, ifs, (n1, (n2, v)) :: applies) rest
      | discriminateNodeVal' (trues, falses, ifs, applies) nil =
           let val ifsDisc = List.concat (map discriminateNode (discriminateNode ifs))
               val appliesDisc = List.concat (map discriminateNode (discriminateNode applies))
           in List.filter (not o null) [trues, falses] @ ifsDisc @ appliesDisc
           end

    fun discriminateNodeVal args = discriminateNodeVal' (nil, nil, nil, nil) args

    val partitionByContent = discriminateNodeVal o map (fn n => (!!n, n))


    fun unify nil = ()
      | unify (n :: ns) = List.app (fn n' => (link (n', n); ())) ns

end
```

A.3.7 NodeHeap.sig

```
(*
 * NODEHEAP: Structure for generating imperative maps
 * from segment [0..n-1] to lists of nodes (may be any type).
 * Supported operations are: looking up node list associated
 * with a particular index (lookup); adding a node to the node list
 * at a given index (add); iterating over all node lists in ascending
 * and in descending order.
 *)

signature NodeHeap =
sig
  type node = Node.node

  val new: int ->                             (* generate new imperative map from [0..n-1]
                                                 with n given by the argument; all node lists in map
                                                 are initialized to nil *)
                   (int -> node list)         (* return node list at given index *)
                 * (int * node list -> unit)  (* set node list at given index to given node list *)
                 * (int * node -> unit)       (* add given node to node list at given index *)
                 * ((node list -> node list) -> unit)
                                              (* update node lists by applying given function
                                                 in ascending order of index; function may be
                                                 side-effecting *)
                 * ((node list -> node list) -> unit) (* apply function to max-1..0 *)
                                              (* update node lists by applying given function
                                                 in descending order of index; function may be
                                                 side-effecting *)
end
```

A.3.8 NodeHeap.sml

```
structure NodeHeap :> NodeHeap =
struct
  type node = Node.node
  fun new n =
    let val m = Array.array (n, nil)
        fun lookup i = Array.sub (m, i)
        fun update (i, elmts) = Array.update (m, i, elmts)
        fun add (i, elmt) = Array.update (m, i, elmt :: Array.sub (m, i))
        fun app f =
          let fun loop i = if i >= n then () else (update (i, f (lookup i)); loop (i+1))
          in loop 0
          end
        fun revapp f =
          let fun loop i = if i < 0 then () else (update (i, f (lookup i)); loop (i-1))
          in loop (n-1)
          end
    in (lookup, update, add, app, revapp)
    end

end
```

A.3.9 RobddMsd.sig

```
signature RobddMsd =
sig
  type ''voelem BEXP = ''voelem Bexp.BEXP
  type ''voelem varorder = ''voelem Varorder.varorder
  type operator = Binop.operator
  type ''voelem robdd (* = Node.node * ''voelem varorder *)


  (* Build robdd from BEXP using varorder *)
  val build : ''voelem varorder * ''voelem BEXP -> ''voelem robdd

  (* print robdd in Graphviz dot format *)
  val toString : ''voelem robdd * (''voelem -> string) -> string

  (* robdd1 operator robdd2 *)
  val apply : ''voelem robdd * ''voelem robdd * operator -> ''voelem robdd
  (* restrict this robdd *)
  val restrict : ''voelem robdd * ''voelem * bool -> ''voelem robdd
  (* construct negated robdd *)
  val neg : ''voelem robdd -> ''voelem robdd
  (* create exists robdd *)
  val exists : ''voelem robdd * ''voelem -> ''voelem robdd
  (* create forall robdd *)
  val forall : ''voelem robdd * ''voelem -> ''voelem robdd
  (* find number of satisfying variable assignments *)
    val satcount : ''voelem robdd -> int
  (* find a variable assignment - only strict requirements are returned *)
  val anysat : ''voelem robdd -> (''voelem * int) list
  (* find all satisfying assignments - only strict requirements are returned *)
  val allsat : ''voelem robdd -> (''voelem * int) list list
  (* test two robdds for equality *)
 val equal : ''voelem robdd * ''voelem robdd -> bool

(* debug crap below *)
(*
 val maxShare : ''voelem robdd list -> unit
 val isBDD : ''voelem robdd -> bool
*)

end
```

A.3.10 RobddMsd.sml

```
structure RobddMsd :> RobddMsd =
struct
  infix 6 &&
  infix 5 ||
  infix 4 <==>
  infix 3 ==>

  open Bexp
  open Varorder
  open Binop
  open Node
  open NodeHeap

  type ''voelem BEXP = ''voelem Bexp.BEXP
  type ''voelem varorder = ''voelem Varorder.varorder
  type operator = Binop.operator
  type ''voelem robdd = Node.node * ''voelem varorder
  type node = Node.node

  fun coalesceSameIndex ns =
      (List.app (fn ns => unify
                 (case !!(hd ns) of
                    IF (_, n1, n2) => if Node.equal (n1, n2) then n1 :: ns else ns (* removes
                                                                                   redundancy R4 *)
                  | TRUE => nil
                  | FALSE => nil
                  | APPLY _ => raise Fail "Impossible: APPLY node in coalesceSameIndex argument"))
      (partitionByContent ns);
      nil)

  (* Ensures maximal sharing among a list of robdds *)
  fun maxShare nil = ()
    | maxShare (ns) =
      let
```

```
        val maxVarInLst = foldl (fn (a,b) => if a>b then a else b) ~1 (map (maxVar o (fn (n, v) => v)) ns)

        val (lookup, update, add, app, revapp) = NodeHeap.new (maxVarInLst + 1 +1)

      fun varIndex node =
            case !!node of
              TRUE => maxVarInLst + 1
            | FALSE => maxVarInLst + 1
            | IF (i, _, _) => i
            | APPLY _ => raise Fail "Impossible: varIndex of APPLY-node"

            fun insert node = (* Inefficient worst case exponential *)
                case !!node of
                  TRUE => add ((varIndex node), node)
                | FALSE => add ((varIndex node), node)
                | IF (i, n11, n12) => (add ((varIndex node), node); insert n11; insert n12)
                | APPLY _ => raise Fail "insert used on APPLY-node"
      in
          (List.app (insert o (fn (n, v) => n)) ns;
           revapp coalesceSameIndex)
      end

  fun new varNum =
      let
        open Node
        val (lookup, update, add, app, revapp) = NodeHeap.new (varNum + 1)

        fun varIndex node =
        case !!node of
          TRUE => varNum
        | FALSE => varNum
        | IF (i, _, _) => i
        | APPLY _ => raise Fail "Impossible: varIndex of APPLY-node"

        fun lazyApply (n1, n2) =
            let val newNode = newApply (n1, n2)
            in add (Int.min (varIndex n1, varIndex n2), newNode);
                newNode
            end

        fun applyOp' operator (t1,t2) = if operator(t1,t2) then tt else ff

        fun applyOp operator (n1, n2) =
        case (!!n1, !!n2) of
          (FALSE,FALSE) => applyOp' operator(false,false)
        | (FALSE,TRUE) => applyOp' operator(false,true)
        | (TRUE,FALSE) => applyOp' operator(true,false)
        | (TRUE,TRUE) => applyOp' operator(true,true)
        | (IF (i1, n11, n12), IF (i2, n21, n22)) =>
           (case Int.compare (i1, i2) of
              LESS => newIf (i1, lazyApply (n11, n2), lazyApply (n12, n2))
            | EQUAL => newIf (i1, lazyApply (n11, n21), lazyApply (n12, n22))
            | GREATER => newIf (i2, lazyApply (n1, n21), lazyApply (n1, n22))
           )
(* Naive approach:
        | (IF (i1, n11, n12), FALSE) => newIf(i1, lazyApply (n11, n2), lazyApply (n12, n2))
        | (IF (i1, n11, n12), TRUE) => newIf(i1, lazyApply (n11, n2), lazyApply (n12, n2))
        | (FALSE, IF (i2, n21, n22)) => newIf(i2, lazyApply (n1, n21), lazyApply (n1, n22))
        | (TRUE, IF (i2, n21, n22)) => newIf(i2, lazyApply (n1, n21), lazyApply (n1, n22))
*)
(* Stop calling lazyApply if possible *)
        | (IF (i1, n11, n12), FALSE) =>
           if operator(true,false) = operator(false,false) then if operator(true,false) then tt else ff
                                        else newIf(i1, lazyApply (n11, n2), lazyApply (n12, n2))
        | (IF (i1, n11, n12), TRUE) =>
           if operator(true,true) = operator(false,true) then if operator(true,true) then tt else ff
                                        else newIf(i1, lazyApply (n11, n2), lazyApply (n12, n2))
        | (FALSE, IF (i2, n21, n22)) =>
           if operator(false,true) = operator(false,false) then if operator(false,true) then tt else ff
                                        else newIf(i2, lazyApply (n1, n21), lazyApply (n1, n22))
        | (TRUE, IF (i2, n21, n22)) =>
           if operator(true,true) = operator(true,false) then if operator(true,true) then tt else ff
                                      else newIf(i2, lazyApply (n1, n21), lazyApply (n1, n22))

        | _ => raise Fail "Impossible: applyOp applied to one or two APPLY-nodes"

        fun applyEquivs binop (ns as (fstNode :: otherNodes)) =
            (case !!fstNode of
```

```
                APPLY (n1, n2) => let val newNode = applyOp binop (n1, n2)
                                  in unify (newNode :: ns); newNode
                                  end
          | _ => raise Fail "Impossible: Non-APPLY node in applyEquivs argument")
          | applyEquivs binop nil = raise Fail "Impossible: empty argument to applyEquivs"

      fun applySameIndex binop =
           List.map (applyEquivs binop) o
           partitionByContent


      fun apply binop (n1, n2) =
          lazyApply (n1, n2) before
            (app (applySameIndex binop);
             revapp coalesceSameIndex)

      in
          apply
      end

  fun apply (r1 as (n1, v1), r2 as (n2, v2), operator) =
      if v1 <> v2 then raise Fail "Robdd ordering mismatch\n"
      else
          let
            val apply' = new (maxVar(v1)+1)(* (getNoVars v1) *)
          in
              (apply' operator (n1,n2),v1)
          end


      fun neg (n1, v1) = apply((n1,v1),(tt,v1),XOR)


  fun build (varorder,bexp) =
      let
        fun build'(VAR x) =
            let
              val varElemInt = varOrdElemToInt(varorder,x)
            in
              case varElemInt of
                SOME j => (newIf(j,tt,ff),varorder)
              | NONE => raise Fail "Error in build'\n"
            end
          | build'(T) = (tt,varorder)
          | build'(F) = (ff,varorder)
          | build'(negate x) = apply (build'(x),(tt,varorder),XOR)
          | build'(x && y) = apply(build'(x),build'(y),AND)
          | build'(x || y) = apply(build'(x),build'(y),OR)
          | build'(x ==> y) = apply(build'(x),build'(y),IMP)
          | build'(x <==> y) = apply(build'(x),build'(y),BIMP)
      in
          if (not o checkVarOrder) varorder then raise Fail "Varorder is inconsistent\n"
          else if (not o validate) (varorder,bexp) then raise Fail "All vars not in varorder\n"
          else build'(bexp)
      end

  local
    (* used to initialize Polyhash *)
    fun sameKey(x,y) = x=y
    exception entryNotInHash

    (* inserts value in hashTbl and returns value *)
    fun insertInHash(hashTbl,key,value) =
        (Polyhash.insert hashTbl (key,value); value)

    fun hashPrntRobdd(IF (i, _, _)) = i
      | hashPrntRobdd TRUE = 1
      | hashPrntRobdd FALSE = 0 (* TODO FIX many collisions - impl without hash *)
      | hashPrntRobdd _ = raise Fail "Apply-nodes cannot be hashed"
  in
    fun toString(r1 as (n1, v1), elemToStr) =
        let
          val preStr = "digraph G {\n"
          val postStr = "}\n"
          val HINT_GTBL_SIZE = 100
          val Gtbl = Polyhash.mkTable(hashPrntRobdd,sameKey) (HINT_GTBL_SIZE,entryNotInHash)

          fun prntRobdd'(nodeVal, nxtFree) =
              let
                val printed = Polyhash.peek Gtbl nodeVal
```

```
                in
                  case printed of
                    SOME n => ("",n,nxtFree)
                  | NONE =>
                      let
                        val strNode = Int.toString(nxtFree);
                      in
                        case nodeVal of
                          n as TRUE => (insertInHash(Gtbl, n, nxtFree);
                                        ("node"^strNode^" [label=\"1\" shape=box];\n",nxtFree,nxtFree+1))
                        | n as FALSE => (insertInHash(Gtbl, n, nxtFree);
                                         ("node"^strNode^" [label=\"0\" shape=box];\n",nxtFree,nxtFree+1))
                        | n as (IF (i1, n11, n12)) =>
                            let
                              val varOrdElem = intToVarOrdElem(v1,i1)
                            in
                              case varOrdElem of
                                SOME elem =>
                                  let
                                    val (lowStr,nodeNumLow,nxtFreeLow) = prntRobdd'(!!n12,nxtFree+1)
                                    val (highStr,nodeNumHigh,nxtFreeHigh) = prntRobdd'(!!n11,nxtFreeLow)
                                    val nodeNumLowStr = Int.toString(nodeNumLow)
                                    val nodeNumHighStr = Int.toString(nodeNumHigh)
                                    val varStr = elemToStr(elem)
                                    val thisStr = "node"^strNode^" [label=\""^varStr^"\"];\n"^
                                        "node"^strNode^" -> node"^nodeNumLowStr^
                                        " [arrowhead=none style=dotted];\n"^
                                        "node"^strNode^" -> node"^nodeNumHighStr^
                                        " [arrowhead=none];\n"
                                  in
                                    insertInHash(Gtbl,nodeVal,nxtFree);
                                    (lowStr^highStr^thisStr,nxtFree,nxtFreeHigh)
                                  end
                              | NONE => raise Fail "noVarForInt"
                            end
                        | (APPLY _) => raise Fail "APPLY-nodes cannot be printed"
                      end
                end
        in
            preStr^(#1(prntRobdd'(!!n1, 1)))^postStr
        end
  end

  (* Builds a new robdd by restricting varX to b. The result is maximally shared with the input r1 *)
  fun restrict(r1 as (n1, v1), varX, b) =
      if (not o inOrder) (varX, v1) then raise Fail "Restrict impossible variable not i varorder"
      else
          let
            val varOrdElemIdx = varOrdElemToInt(v1, varX)
            val newVarorder = restrictVo(v1, varX)

            fun restrict'(n1,j,b) =
              case (!!n1) of
                TRUE => n1
              | FALSE => n1
              | IF (i,n11,n12) =>
                  if i=j then
                      if b then n11 else n12
                  else newIf(i, restrict'(n11,j,b),restrict'(n12,j,b))
              | APPLY _ => raise Fail "restrict cannot be applied to an APPLY-node"
          in
            case varOrdElemIdx of
              SOME j =>
                  let
                    val result = (restrict'(n1,j,b), newVarorder)
                  in
                     maxShare([result, r1]); result
                  end
            | NONE => r1 (* no var to restrict *)
          end

  (* exists *)
  fun exists(robdd,x) =
      let
        val lowResRobdd = restrict(robdd,x,false)
        val highResRobdd = restrict(robdd,x,true)
      in
          apply(lowResRobdd,highResRobdd, OR)
      end
```

```
(* forall *)
fun forall(robdd,x) =
    let
      val lowResRobdd = restrict(robdd,x,false)
      val highResRobdd = restrict(robdd,x,true)
    in
        apply(lowResRobdd,highResRobdd, AND)
    end

(* compare for equality. First we ensure maximal sharing between r1 and r2 - then we comare root nodes *)
  fun equal(r1 as (n1, v1), r2 as (n2, v2)) = (maxShare([r1, r2]); Node.equal(n1,n2))


local
  (* hash used for memoization *)
  fun hashSatCount n =
      case !!n of
        TRUE => 1
      | FALSE => 0
      | IF(i,_,_) => i
      | APPLY _ => raise Fail "hashSatCount cannot be used on APPLY-nodes"

  (* used to initialize Polyhash *)
  fun sameKey(x,y) = x=y

  exception entryNotInHash

  (* inserts value in hashTbl and returns value *)
  fun insertInHash(hashTbl,key,value) =
      (Polyhash.insert hashTbl (key,value); value)
in
  (* satcount only defined if there are variables to assign to *)
    fun satcount(r1 as (n1, v1)) =
        let
          val n = maxVar(v1)
          val HINT_GTBL_SIZE = 100
          val Gtbl = Polyhash.mkTable(hashSatCount,sameKey) (HINT_GTBL_SIZE,entryNotInHash)

          (* varu1idx <= varu2idx : computes the number of free vars between the two *)
          fun freeBetween(varu1idx, varu2idx) =
              let
                fun freeBetween'(testNow,free) =
                    if testNow < varu2idx then
                        if idxInOrder(testNow,v1) then freeBetween'(testNow+1,free+1)
                        else freeBetween'(testNow+1,free)
                    else free
              in
                  freeBetween'(varu1idx+1,0)
              end

          fun getVarNum(node) =
          case !!node of
            TRUE => n+1
          | FALSE => n+1
          | IF (i,_,_) => i
          | APPLY _ => raise Fail "getVarNum cannot be applied to an APPLY-node"

          fun count(node) =
              let
                val Gnode = Polyhash.peek Gtbl node
              in
                case Gnode of
                  SOME n => n
                | NONE =>
                case !!node of
                  TRUE => insertInHash(Gtbl,tt,1)
                | FALSE => insertInHash(Gtbl,ff,0)
                | IF(i1,n11,n12) =>
                   let
                     val countn11 = count(n11)
                     val countn12 = count(n12)
                   in
                       if countn11 <> 0 then
                           if countn12 <> 0 then
                               insertInHash(Gtbl,node,
                                floor(Math.pow(2.0,real(freeBetween(i1,getVarNum(n12)))))*countn12+
                                floor(Math.pow(2.0,real(freeBetween(i1,getVarNum(n11)))))*countn11)
                           else insertInHash(Gtbl,node,
                                floor(Math.pow(2.0,real(freeBetween(i1,getVarNum(n11)))))*countn11)
                       else
```

```
                                  if countn12 <> 0 then
                                      insertInHash(Gtbl,node,
                                        floor(Math.pow(2.0,real(freeBetween(i1,getVarNum(n12))))*countn12)
                                  else insertInHash(Gtbl,node,0)
                          end
                      | _ => raise Fail "count used on APPLY-node\n"
                    end
            in
                floor(Math.pow(2.0,real(freeBetween(minVar(v1)-1,getVarNum(n1))))*count(n1))
            end
  end

  fun anysat(r1 as (n1, v1)) =
      let
        fun anysat' (node) =
        case !!node of
          TRUE => []
        | FALSE => raise Fail "No satisfying truth assignment exists\n"
        | IF (i1, n11, n12) =>
           let
             val varOrdElem = intToVarOrdElem(v1, i1)
           in
             case varOrdElem of
               SOME varOE =>
                   if equal((n12,v1),(ff,v1)) then (varOE,1):: anysat'(n11)
                   else (varOE,0) :: anysat'(n12)
             | NONE => raise Fail "Error in anysat'\n"
           end
        | _ => raise Fail "anysat' cannot be used on APPLY-nodes"


      in
        anysat'(n1)
    end

  fun allsat(r1 as (n1, v1)) =
      let
        fun allsat' (node) =
              case !!node of
                FALSE => []
              | TRUE => [[]]
              | APPLY _ => raise Fail "allsat' used on APPLY-node\n"
              | IF(i1, n11, n12) =>
                  let
                    fun consmap elem lst = elem::lst
                    val varOrdElem = intToVarOrdElem(v1,i1)
                  in
                    case varOrdElem of
                      SOME varOE =>
                          map (consmap (varOE,0)) (allsat'(n12))   @
                          map (consmap (varOE,1)) (allsat'(n11))
                    | NONE => raise Fail "Err in allsat'\n"
            end
    in
        allsat'(n1)
    end

(*
  fun varIdx (ref (BOOL b)) = 1073741823
    | varIdx (ref (IF (i,_,_,_))) = i
    | varIdx (ref (APPLY _)) = raise Fail "Impossible: varIndex of APPLY-node"

  fun isBDD' (ref (BOOL b)) = true
    | isBDD' (ref (IF (i, n11, n12, _))) =
                let
                    val varIdxHigh = varIdx(n11)
                    val varIdxLow = varIdx(n12)
                in
                    i<varIdxHigh andalso i<varIdxLow andalso isBDD'(n11) andalso isBDD'(n12)
                end
    | isBDD' (ref (APPLY _)) = raise Fail "Impossible: varIndex of APPLY-node"

  fun isBDD (n1,_) = isBDD' n1

*)
end
```

A.3.11 NQueen.sml

```
local
  fun genVar (i, j) = "x"^Int.toString(i)^"_"^Int.toString(j)

  fun genVars' (res, i, j, n) =
      if i <= n then if j <= n then
          genVars' (genVar (i, j) :: res, i, j+1, n)
                     else genVars' (res, i+1, 1, n)
      else
          res

  fun genVarorder n = varorderFromLst (genVars' ([], 1, 1, n))

in
  fun NQueenBDD n =
      let
          val varorder = genVarorder n
          local
            fun genCond1' (res, i, j, n, l) =
                if 1 <= l andalso l <= n then
                    if l <> j then
                        genCond1'(apply(
                                        res,neg(build(varorder, VAR (genVar (i, l)))), AND
                                        ),
                                  i, j, n, l+1)
                    else (* l = j *) genCond1'(res, i, j, n, l+1)
                else res
          in
            fun genCond1'' (i, j, n) = apply(build(varorder, VAR (genVar (i, j))),
                                             genCond1' (build(varorder, T), i, j, n, 1),
                                             IMP)
          end
          local
              fun genCond2' (res, i, j, n, k) =
                  if 1 <= k andalso k <= n then
                      if k <> i then
                          genCond2'(apply(
                                          res,neg(build(varorder, VAR (genVar (k, j)))), AND
                                        ),
                                  i, j, n, k+1)
                    else (* k = j *) genCond2'(res, i, j, n, k+1)
                  else res
          in
            fun genCond2'' (i, j, n) = apply(build(varorder, VAR (genVar (i, j))),
                                             genCond2' (build(varorder, T), i, j, n, 1),
                                             IMP)

          end
          local
              fun genCond3' (res, i, j, n, k) =
                  if 1 <= k andalso k <= n then
                      if 1 <= j+k-i andalso j+k-i <= n andalso k <> i then
                          genCond3'(apply(
                                          res,neg(build(varorder, VAR (genVar (k, j+k-i)))), AND
                                        ),
                                  i, j, n, k+1)
                      else genCond3'(res, i, j, n, k+1)
                  else res
          in
            fun genCond3'' (i, j, n) = apply(build(varorder, VAR (genVar (i, j))),
                                             genCond3' (build(varorder, T), i, j, n, 1),
                                             IMP)
          end
          local
              fun genCond4' (res, i, j, n, k) =
                  if 1 <= k andalso k <= n then
                      if 1 <= j+i-k andalso j+i-k <= n andalso k <> i then
                          genCond4'(apply(
                                          res,neg(build(varorder, VAR (genVar (k, j+i-k)))), AND
                                        ),
                                  i, j, n, k+1)
                    else genCond4'(res, i, j, n, k+1)
                  else res
          in
            fun genCond4'' (i, j, n) = apply(build(varorder, VAR (genVar (i, j))),
                                             genCond4' (build(varorder, T), i, j, n, 1),
                                             IMP)
          end
```

```
            local
                fun genCond5' (res, i, n, k) =
                    if 1 <= k andalso k <= n then
                        genCond5'(apply(
                                        res,build(varorder, VAR (genVar (i, k))), OR), i, n, k+1)
                    else res
            in
                fun genCond5'' (i, n) = genCond5' (build(varorder, F), i, n, 1)
            end
            local
              fun genCondX' (res, condX, i, j, n) =
                  if i <= n then if j <= n then
                      genCondX' (apply(
                                        res,condX (i, j, n), AND), condX, i, j+1, n)
                                 else genCondX' (res, condX, i+1, 1, n)
                  else
                      res
            in
              fun genCondX (condX, n) = genCondX' (build(varorder, T), condX, 1, 1, n)
            end

            fun genCond5''' (res, i, n) =
                if 1 <= i andalso i <= n then
                    genCond5'''(apply(
                                       res, genCond5'' (i, n), AND), i+1, n)
                else res

        fun genCond1 n = genCondX (genCond1'', n)
        fun genCond2 n = genCondX (genCond2'', n)
        fun genCond3 n = genCondX (genCond3'', n)
        fun genCond4 n = genCondX (genCond4'', n)
        fun genCond5 n = genCond5''' (build(varorder, T), 1, n)

        val bdd1 = genCond1 n
        val bdd2 = genCond2 n
        val bdd3 = genCond3 n
        val bdd4 = genCond4 n
        val bdd5 = genCond5 n

        in
            (apply( apply( apply( apply (bdd1, bdd2, AND), bdd3, AND), bdd4, AND), bdd5, AND),
             bdd1,bdd2,bdd3,bdd4,bdd5)
        end
end

val n = 6;
val (eightQueen,bdd1,bdd2,bdd3,bdd4,bdd5) = NQueenBDD n;
val sol = List.map (List.filter (fn (_,b) => b=1)) (allsat eightQueen);
val bogus = print("Total number of solutions to the "^Int.toString(n)^" queens problem: "^
                  Int.toString(satcount eightQueen)^".\n");
```

B Benchmarks

B.1 Memory

B.1.1 NQ_4MSD.out

```
6,550,196 bytes allocated (45,656 bytes by GC)
                 function                               cur        raw
-----------------------------------------------------  -----  -----------
Node.discriminateNodeVal'                              23.6%  (1,555,212)
SimpleDURef.discriminate                               13.2%    (868,464)
Atom.discriminate.anon                                 12.9%    (853,260)
SimpleDURef.find                                        9.9%    (653,792)
General.o                                               7.1%    (470,628)
SimpleDURef.discriminate.anon                           6.6%    (434,232)
SimpleDURef.duref                                       5.6%    (366,540)
Atom.discriminate                                       5.3%    (352,180)
Node.anon                                               3.6%    (235,284)
NodeHeap.new.app.loop                                   2.7%    (177,432)
NodeHeap.new.add                                        2.2%    (146,568)
RobddMsd.new.lazyApply                                  2.2%    (146,568)
SimpleDURef.link                                        1.6%    (105,544)
RobddMsd.new.applyOp                                    1.4%     (94,064)
<gc>                                                    0.7%     (45,656)
NodeHeap.new                                            0.6%     (36,480)
Sequence.append                                         0.2%     (12,148)
Array.ArraySlice.vector                                 0.1%      (7,968)
<main>                                                  0.1%      (5,908)
List.foldl.loop                                         0.1%      (3,948)
List.@                                                  0.1%      (3,720)
RobddMsd.build.build'                                   0.1%      (3,712)
RobddMsd.neg                                            0.0%      (1,824)
NQueenBDD.genCondX'                                     0.0%      (1,584)
RobddMsd.allsat.allsat'                                 0.0%      (1,500)
Polyhash.insert.look                                    0.0%      (1,220)
NQueenBDD.genCond4''                                    0.0%      (1,152)
NQueenBDD.genCond3''                                    0.0%      (1,152)
NQueenBDD.genCond1''                                    0.0%      (1,152)
NQueenBDD.genCond1'                                     0.0%      (1,152)
NQueenBDD.genCond2''                                    0.0%      (1,152)
NQueenBDD.genCond2'                                     0.0%      (1,152)
NQueenBDD.genCond3'                                     0.0%        (672)
NQueenBDD.genCond4'                                     0.0%        (672)
Polyhash.mkTable                                        0.0%        (536)
Varorder.varorderFromLst.varorderFromLst_hlp            0.0%        (384)
RobddMsd.allsat.allsat'.consmap                         0.0%        (384)
NQueenBDD.genCond5''                                    0.0%        (240)
genVars'                                                0.0%        (212)
NQueenBDD.genCond5'                                     0.0%        (192)
NQueenBDD                                               0.0%        (104)
NQueenBDD.genCond5'''                                   0.0%         (48)
NQueenBDD.genCondX                                      0.0%         (48)
NQueenBDD.genCond5                                      0.0%         (12)
```

B.1.2 NQ_4HASH.out

```
320,675,368 bytes allocated (246,664 bytes by GC)
                    function                          cur        raw
----------------------------------------------------  -----  -------------
RobddHash.mk                                          99.7%  (319,928,880)
Polyhash.mkTable                                       0.1%      (251,508)
<gc>                                                   0.1%      (246,664)
Polyhash.insert.look                                   0.1%      (199,640)
Polyhash.growTable.copy                                0.0%       (99,840)
<unknown>                                              0.0%       (99,252)
Polyhash.growTable                                     0.0%       (40,584)
Sequence.append                                        0.0%       (12,148)
Dynarray.expand                                        0.0%        (8,012)
Array.ArraySlice.vector                                0.0%        (7,968)
<main>                                                 0.0%        (5,900)
Dynarray.array                                         0.0%        (4,020)
RobddHash.build                                        0.0%        (2,440)
RobddHash.apply                                        0.0%        (2,432)
NQueenBDD.genCondX'                                    0.0%        (1,632)
RobddHash.allsat.allsat'                               0.0%        (1,500)
RobddHash.neg                                          0.0%        (1,216)
NQueenBDD.genCond2''                                   0.0%        (1,152)
NQueenBDD.genCond1''                                   0.0%        (1,152)
NQueenBDD.genCond3''                                   0.0%        (1,152)
NQueenBDD.genCond4''                                   0.0%        (1,152)
NQueenBDD.genCond1'                                    0.0%          (576)
NQueenBDD.genCond2'                                    0.0%          (576)
Varorder.varorderFromLst.varorderFromLst_hlp           0.0%          (384)
RobddHash.allsat.allsat'.consmap                       0.0%          (384)
NQueenBDD.genCond3'                                    0.0%          (336)
NQueenBDD.genCond4'                                    0.0%          (336)
NQueenBDD.genCond5''                                   0.0%          (240)
List.foldl.loop                                        0.0%          (228)
genVars'                                               0.0%          (212)
NQueenBDD.genCond5'                                    0.0%          (192)
NQueenBDD                                              0.0%          (104)
General.o                                              0.0%           (60)
NQueenBDD.genCond5'''                                  0.0%           (48)
NQueenBDD.genCondX                                     0.0%           (48)
TextIO.print                                           0.0%           (28)
StreamIOExtra.flushOut                                 0.0%           (12)
NQueenBDD.genCond5                                     0.0%           (12)
RobddHash.satcount                                     0.0%           (12)
```

B.1.3 NQ_5MSD.out

```
30,407,132 bytes allocated (86,496 bytes by GC)
                  function                            cur       raw
-------------------------------------------------   -----   ----------
Node.discriminateNodeVal'                           22.3%  (6,812,160)
Atom.discriminate.anon                              14.2%  (4,320,540)
SimpleDURef.discriminate                            13.6%  (4,160,640)
SimpleDURef.find                                    10.5%  (3,194,520)
General.o                                            7.1%  (2,161,932)
SimpleDURef.discriminate.anon                        6.8%  (2,080,320)
SimpleDURef.duref                                    5.7%  (1,742,540)
Atom.discriminate                                    4.9%  (1,503,588)
Node.anon                                            3.5%  (1,080,936)
NodeHeap.new.app.loop                                2.5%    (768,000)
RobddMsd.new.lazyApply                               2.3%    (696,936)
NodeHeap.new.add                                     2.3%    (696,936)
SimpleDURef.link                                     1.7%    (512,368)
RobddMsd.new.applyOp                                 1.5%    (457,632)
NodeHeap.new                                         0.3%    (101,384)
<gc>                                                 0.3%     (86,496)
Sequence.append                                      0.1%     (22,808)
Array.ArraySlice.vector                              0.0%     (15,072)
RobddMsd.allsat.allsat'                              0.0%     (11,196)
List.foldl.loop                                      0.0%      (9,000)
List.@                                               0.0%      (7,380)
RobddMsd.build.build'                                0.0%      (7,120)
Polyhash.insert.look                                 0.0%      (6,900)
<main>                                               0.0%      (5,908)
RobddMsd.neg                                         0.0%      (3,840)
RobddMsd.allsat.allsat'.consmap                      0.0%      (3,000)
Polyhash.growTable.copy                              0.0%      (2,560)
NQueenBDD.genCondX'                                  0.0%      (2,448)
NQueenBDD.genCond1'                                  0.0%      (2,400)
NQueenBDD.genCond2'                                  0.0%      (2,400)
NQueenBDD.genCond1''                                 0.0%      (2,100)
NQueenBDD.genCond2''                                 0.0%      (2,100)
NQueenBDD.genCond3''                                 0.0%      (2,100)
NQueenBDD.genCond4''                                 0.0%      (2,100)
NQueenBDD.genCond3'                                  0.0%      (1,440)
NQueenBDD.genCond4'                                  0.0%      (1,440)
Polyhash.growTable                                   0.0%      (1,048)
Varorder.varorderFromLst.varorderFromLst_hlp         0.0%        (600)
Polyhash.mkTable                                     0.0%        (536)
NQueenBDD.genCond5''                                 0.0%        (360)
genVars'                                             0.0%        (320)
NQueenBDD.genCond5'                                  0.0%        (300)
NQueenBDD                                            0.0%        (104)
NQueenBDD.genCond5'''                                0.0%         (60)
NQueenBDD.genCondX                                   0.0%         (48)
NQueenBDD.genCond5                                   0.0%         (12)
```

B.1.4 NQ_6MSD.out

```
128,794,596 bytes allocated (112,836 bytes by GC)
                 function                               cur       raw
-------------------------------------------------------  -----  ------------
Node.discriminateNodeVal'                               21.9%  (28,199,880)
Atom.discriminate.anon                                  14.8%  (19,076,628)
SimpleDURef.discriminate                                13.9%  (17,896,176)
SimpleDURef.find                                        10.7%  (13,765,176)
General.o                                                7.1%   (9,123,564)
SimpleDURef.discriminate.anon                            6.9%   (8,948,088)
SimpleDURef.duref                                        5.8%   (7,471,980)
Atom.discriminate                                        4.7%   (6,008,612)
Node.anon                                                3.5%   (4,561,752)
NodeHeap.new.app.loop                                    2.4%   (3,146,160)
RobddMsd.new.lazyApply                                   2.3%   (2,988,672)
NodeHeap.new.add                                         2.3%   (2,988,672)
SimpleDURef.link                                         1.7%   (2,232,896)
RobddMsd.new.applyOp                                     1.5%   (1,980,496)
NodeHeap.new                                             0.2%     (239,040)
<gc>                                                     0.1%     (112,836)
Sequence.append                                          0.0%      (38,452)
Array.ArraySlice.vector                                  0.0%      (25,504)
List.foldl.loop                                          0.0%      (13,440)
List.@                                                   0.0%      (12,744)
RobddMsd.build.build'                                    0.0%      (12,160)
RobddMsd.neg                                             0.0%       (6,960)
RobddMsd.allsat.allsat'                                  0.0%       (6,768)
<main>                                                   0.0%       (5,908)
Polyhash.insert.look                                     0.0%       (5,260)
NQueenBDD.genCond1'                                      0.0%       (4,320)
NQueenBDD.genCond2'                                      0.0%       (4,320)
NQueenBDD.genCondX'                                      0.0%       (3,504)
NQueenBDD.genCond1''                                     0.0%       (3,456)
NQueenBDD.genCond2''                                     0.0%       (3,456)
NQueenBDD.genCond3''                                     0.0%       (3,456)
NQueenBDD.genCond4''                                     0.0%       (3,456)
NQueenBDD.genCond3'                                      0.0%       (2,640)
NQueenBDD.genCond4'                                      0.0%       (2,640)
Polyhash.growTable.copy                                  0.0%       (2,560)
RobddMsd.allsat.allsat'.consmap                          0.0%       (1,728)
Polyhash.growTable                                       0.0%       (1,048)
Varorder.varorderFromLst.varorderFromLst_hlp             0.0%         (864)
Polyhash.mkTable                                         0.0%         (536)
NQueenBDD.genCond5''                                     0.0%         (504)
genVars'                                                 0.0%         (452)
NQueenBDD.genCond5'                                      0.0%         (432)
NQueenBDD                                                0.0%         (104)
NQueenBDD.genCond5'''                                    0.0%          (72)
NQueenBDD.genCondX                                       0.0%          (48)
NQueenBDD.genCond5                                       0.0%          (12)
```

B.1.5 NQ_7MSD.out

```
504,965,700 bytes allocated (228,056 bytes by GC)
                    function                         cur      raw
--------------------------------------------------- ----- -------------
Node.discriminateNodeVal'                           21.0% (106,083,396)
Atom.discriminate.anon                              15.2%  (76,960,104)
SimpleDURef.discriminate                            14.1%  (71,220,144)
SimpleDURef.find                                    10.9%  (54,952,168)
General.o                                            7.1%  (35,994,732)
SimpleDURef.discriminate.anon                        7.0%  (35,610,072)
SimpleDURef.duref                                    5.9%  (29,699,040)
Atom.discriminate                                    4.5%  (22,616,760)
Node.anon                                            3.6%  (17,997,336)
NodeHeap.new.app.loop                                2.4%  (12,235,776)
RobddMsd.new.lazyApply                               2.4%  (11,879,448)
NodeHeap.new.add                                     2.4%  (11,879,448)
SimpleDURef.link                                     1.8%   (8,990,368)
RobddMsd.new.applyOp                                 1.6%   (7,900,784)
NodeHeap.new                                         0.1%     (499,472)
<gc>                                                 0.0%     (228,056)
RobddMsd.allsat.allsat'                              0.0%      (85,608)
Sequence.append                                      0.0%      (60,056)
Polyhash.insert.look                                 0.0%      (44,780)
Array.ArraySlice.vector                              0.0%      (39,904)
Polyhash.growTable.copy                              0.0%      (38,400)
List.foldl.loop                                      0.0%      (29,688)
RobddMsd.allsat.allsat'.consmap                      0.0%      (23,520)
List.@                                               0.0%      (20,160)
RobddMsd.build.build'                                0.0%      (19,152)
Polyhash.growTable                                   0.0%      (15,456)
RobddMsd.neg                                         0.0%      (11,424)
NQueenBDD.genCond2'                                  0.0%       (7,056)
NQueenBDD.genCond1'                                  0.0%       (7,056)
<main>                                               0.0%       (5,908)
NQueenBDD.genCond2''                                 0.0%       (5,292)
NQueenBDD.genCond1''                                 0.0%       (5,292)
NQueenBDD.genCond3''                                 0.0%       (5,292)
NQueenBDD.genCond4''                                 0.0%       (5,292)
NQueenBDD.genCondX'                                  0.0%       (4,752)
NQueenBDD.genCond3'                                  0.0%       (4,368)
NQueenBDD.genCond4'                                  0.0%       (4,368)
Varorder.varorderFromLst.varorderFromLst_hlp         0.0%       (1,176)
NQueenBDD.genCond5''                                 0.0%         (672)
genVars'                                             0.0%         (608)
NQueenBDD.genCond5'                                  0.0%         (588)
Polyhash.mkTable                                     0.0%         (536)
NQueenBDD                                            0.0%         (104)
NQueenBDD.genCond5'''                                0.0%          (84)
NQueenBDD.genCondX                                   0.0%          (48)
NQueenBDD.genCond5                                   0.0%          (12)
```

B.1.6 NQ_8MSD.out

```
1,907,237,716 bytes allocated (362,456 bytes by GC)
                        function                         cur        raw
_______________________________________________________ _____ ______________
Node.discriminateNodeVal'                               20.8% (397,424,328)
Atom.discriminate.anon                                  15.5% (294,757,104)
SimpleDURef.discriminate                                14.2% (270,671,424)
SimpleDURef.find                                        10.9% (208,190,760)
General.o                                                7.1% (136,225,452)
SimpleDURef.discriminate.anon                            7.1% (135,335,712)
SimpleDURef.duref                                        5.9% (112,815,320)
Atom.discriminate                                        4.2%  (80,876,080)
Node.anon                                                3.6%  (68,112,696)
NodeHeap.new.app.loop                                    2.4%  (45,973,584)
RobddMsd.new.lazyApply                                   2.4%  (45,125,904)
NodeHeap.new.add                                         2.4%  (45,125,904)
SimpleDURef.link                                         1.8%  (34,705,712)
RobddMsd.new.applyOp                                     1.6%  (30,055,936)
NodeHeap.new                                             0.0%     (952,000)
<gc>                                                     0.0%     (362,456)
RobddMsd.allsat.allsat'                                  0.0%     (246,444)
Polyhash.insert.look                                     0.0%      (99,900)
Sequence.append                                          0.0%      (88,568)
Polyhash.growTable.copy                                  0.0%      (79,360)
RobddMsd.allsat.allsat'.consmap                          0.0%      (70,656)
Array.ArraySlice.vector                                  0.0%      (58,912)
List.foldl.loop                                          0.0%      (55,752)
Polyhash.growTable                                       0.0%      (31,864)
List.@                                                   0.0%      (29,940)
RobddMsd.build.build'                                    0.0%      (28,416)
RobddMsd.neg                                             0.0%      (17,472)
NQueenBDD.genCond1'                                      0.0%      (10,752)
NQueenBDD.genCond2'                                      0.0%      (10,752)
NQueenBDD.genCond4''                                     0.0%       (7,680)
NQueenBDD.genCond1''                                     0.0%       (7,680)
NQueenBDD.genCond2''                                     0.0%       (7,680)
NQueenBDD.genCond3''                                     0.0%       (7,680)
NQueenBDD.genCond4'                                      0.0%       (6,720)
NQueenBDD.genCond3'                                      0.0%       (6,720)
NQueenBDD.genCondX'                                      0.0%       (6,192)
<main>                                                   0.0%       (5,908)
Varorder.varorderFromLst.varorderFromLst_hlp             0.0%       (1,536)
NQueenBDD.genCond5''                                     0.0%         (864)
genVars'                                                 0.0%         (788)
NQueenBDD.genCond5'                                      0.0%         (768)
Polyhash.mkTable                                         0.0%         (536)
NQueenBDD                                                0.0%         (104)
NQueenBDD.genCond5'''                                    0.0%          (96)
NQueenBDD.genCondX                                       0.0%          (48)
NQueenBDD.genCond5                                       0.0%          (12)
```