Boolean Constraints in SICStus Prolog

by
Mats Carlsson
Boolean Constraints in SICStus Prolog

Mats Carlsson

SICS technical report T91:09
Swedish Institute of Computer Science
Box 1263, S-164 28, KISTA, Sweden

September 1, 1991
Abstract

This report documents the implementation of a Boolean constraint solver and its
integration with a Prolog engine.

The solver comprises built-in predicates for checking consistency and entailment of
a new constraint w.r.t. accumulated constraints and for generating particular solutions
to a set of constraints, and extensions to the Prolog top-level for displaying answer
constraints.

Boolean unification was chosen as the strategy for the consistency check. Boolean
unification fits well with the Prolog execution model, and allows the accumulated con-
straints to be associated in a natural way with the variables being eliminated.

The answer constraints are computed by existentially quantifying in the accumulated
set of constraints all variables not occurring in the user query. The simplified set of
constraints constitutes the answer constraints.

Boolean formulas are internally represented as DAGs. Details are provided on this
representation and the support for it provided by the Prolog Engine.

We have investigated various optimizations of the Boolean unification algorithm,
and a preliminary performance evaluation is included.
1 Introduction

This report documents the implementation of a Boolean constraint solver and its integration with a Prolog engine.

The contributions of this work are: to describe the interactions between the Prolog engine and the constraint solver, to exploit opportunities for optimizations in the Boolean unification algorithm, and a performance study.

The solver comprises built-in predicates for checking consistency and entailment of a new constraint w.r.t. accumulated constraints and for generating particular solutions of a set of constraints, extensions to the Prolog top-level for displaying answer constraints, and a natural connection to the Prolog unifier.

Boolean unification was chosen as the strategy for the consistency check. Boolean unification fits well with the Prolog execution model, and allows the accumulated constraints to be associated in a natural way with the variables being eliminated.

The answer constraints are computed by existentially quantifying in the accumulated set of constraints all variables not occurring in the user query. The simplified set of constraints constitutes the answer constraints.

As in the CHIP implementation [8], Boolean formulas are internally represented as directed acyclic graphs (DAGs). In our implementation, the DAGs are ordinary terms constructed on the Prolog heap. Certain global data structures have been added to support the various operations on them, as well as extensions to the garbage collector and memory manager.

The report is organized as follows: Section 2 defines the necessary concepts; Section 3 defines the Boolean unification problem and describes the usual unification algorithms; Section 4 describes a significant optimization of the DAG structures and its mapping to Prolog terms; Section 5 defines the necessary operations on the DAG representation; Section 6 connects Boolean unification with the DAG representation and describes several optimizations; Section 7 provides details about the connection between the constraint solver and the Prolog engine; and Section 8 gives some performance results. We end with some conclusions.

2 Preliminaries

Our language includes the constructs $p \land q$ (and), $p \lor q$ (inclusive or), $p \oplus q$ (exclusive or), and $\lnot p$ (not). Besides ordinary, propositional variables, represented as Prolog variables, we also have universally quantified variables, represented as Prolog atoms\textsuperscript{1}. These are quite useful for forcing certain variables of an equation to be input values.

Two formulas $f$ and $g$ are antivalent if $f \oplus g = 1$.

A unifier $\sigma$ of two expressions $p$ and $q$ is an idempotent substitution on variables in $p$ and $q$ satisfying $\sigma(p) = \sigma(q)$. A most general unifier (mgu) $\sigma$ of $p$ and $q$ is such that every other unifier $\tau$ of $p$ and $q$ can be written as $\tau = \lambda \sigma$ where $\lambda$ is a substitution.

The cofactors of a function $f$ with respect to a variable $v$, written $f_v$ and $\overline{f_v}$, are the functions resulting from replacing $v$ by the constants 1 and 0 in $f$, respectively. Cofactors are used in Shannon's decomposition theorem [12] which states that any non-constant function can be expressed in terms of a unique pair of functions:

$$f(v_1, v_2 \ldots v_n) = v_1 \cdot f_{v_1}(v_2 \ldots v_n) + \overline{v_1} \cdot f_{\overline{v_1}}(v_2 \ldots v_n)$$

By recursively decomposing a function into cofactors until only the constants 0 and 1 remain, we arrive at the Shannon Normal Form which is a canonical representation, modulo variable order.

\textsuperscript{1}Universally quantified variables are often called constants in the literature, but we have chosen not to do so, to avoid confusion with the constants 0 and 1.
A Binary Decision Diagram (BDD\textsuperscript{2}) is a directed acyclic graph encoding the Shannon normal form, as follows: The only leaf nodes are the constants 0 and 1, and every non-leaf representing a function $f$ is labelled by a variable $v$ and has two daughter nodes representing the cofactors $f_v$ and $f_{\overline{v}}$. Furthermore all isomorphic subgraphs have been merged. BDDs have been described at length in the literature [1, 5, 4, 3, 9].

We shall use the notation $(v, g, h)$ to denote a BDD node labelled by $v$ and with daughter nodes $g, h$, i.e., the function $v \cdot g + \overline{v} \cdot h$.

3 Boolean Unification

3.1 The Problem

The Boolean unification problem is to find an mgu for the ordinary variables $p_1, p_2 \ldots p_n$ which satisfies the following formula no matter what values $q_1, q_2 \ldots q_m$ take:

$$\forall q \exists p \forall f(p_1, p_2 \ldots p_n, q_1, q_2 \ldots q_m) = g(p_1, p_2 \ldots p_n, q_1, q_2 \ldots q_m)$$

In general, an mgu is composed of $q_1, q_2 \ldots q_m$ and some (at most $n$) newly introduced variables $r_1, r_2 \ldots r_n$, i.e.

$$p_1 = \sigma_1(q_1, q_2 \ldots q_m, r_1, r_2 \ldots r_n)$$
$$p_2 = \sigma_2(q_1, q_2 \ldots q_m, r_1, r_2 \ldots r_n)$$
$$\ldots$$
$$p_n = \sigma_n(q_1, q_2 \ldots q_m, r_1, r_2 \ldots r_n)$$

where $r_1, r_2 \ldots r_n$ can take on arbitrary values. In other words, Boolean unification translates a relational description (an equation) into a functional description by introducing new variables. The variables $p_1, p_2 \ldots p_n$ are said to have been eliminated by Boolean unification.

3.2 Algorithms

There are two major Boolean Unification algorithms: Boole’s method and Löwenheim’s method [10, 11]. Boole’s method is recursive and is based on the Shannon expansion of a given formula. Löwenheim’s method constructs an mgu from a set of particular solutions instead of working from a given formula. In this report we shall only treat Boole’s method and its variants. See [11] for a fuller exposition.

Büttner and Simonis prove in [6] that the mgu of two expressions is unique, if it exists. They also extract a version of Boole’s method from the proof.

Since $p = q$ is equivalent to $p \oplus q = 0$, the algorithm can be expressed in terms of finding the mgu of an expression $g$ and 0 without loss of generality. If an mgu exists, we say that $g$ is soluble, otherwise $g$ is insoluble. Notice that if $g$ contains universally quantified variables, it might be insoluble even if it is distinct from the constant 1.

3.2.1 Boole’s Algorithm

Two formulations will be given here: an “abstract” one and a procedural one. The abstract formulation appeared in [13]:

\textsuperscript{2}The proper terminology for the BDDs we are using is Reduced Ordered Binary Decision Diagrams (ROBDDs) [5], but in this report, we call them BDDs for short.
If $g$ contains no propositional variables, it must be 0, otherwise no unifier exists.

If $g$ contains a variable $v$, then:

$$g = 0$$

can be rewritten into

$$v * g_v + \overline{v} * g_{\overline{v}} = 0$$

which admits the general solution

$$v := w * g_w + \overline{w} * g_{\overline{w}}$$

iff

$$g_v * g_{\overline{v}} = 0$$

where $w$ is a new variable.

In the procedural formulation, the algorithm constructs the mgu instead of "binding" variables:

```java
Mgu(g)
{
    if (g == 0)
        return \emptyset;
    else if (g == v * g_v + \overline{v} * g_{\overline{v}} and v is not quantified)
    {
        let \sigma = Mgu(g_v * g_{\overline{v}});
        return \{v/\sigma\} \cup \{w * g_w + \overline{w} * g_{\overline{w}}\} \cup \sigma;
    }
    else
        return FAILURE;
}
```

where $w$ is a new variable.

## 4 Typed BDDs

With the BDD representation as described in Section 2, negating of a formula and testing for antivalence cannot be done in constant time. A solution to this problem, suggested in [1] and [9] and elsewhere, consists in *typing* the edges as positive or negative in the BDDs. Unless restrictions are placed on where negated edges can be placed, the representation is not canonical any more, however.

With the typed BDD representation, we can negate a formula and test for antivalence in constant time by simply manipulating the types. It is also up to twice as compact as the untyped BDD representation. When we speak about BDDs in the rest of this report, we mean the typed representation.

In [9] a similar representation called Typed Decision Graphs (TDG) is used. TDGs are not reduced to canonical form (DAGs with maximally shared subtrees), however. The authors claim large performance gains by avoiding the reduction to canonical form.

The ordinary variables that occur in a function $f$ are assigned unique integer indexes. The *top variable* of a function $f$ containing ordinary variables is the smallest such index. The top variable of a function $f$ not containing ordinary variables but containing universally quantified variables is the smallest of the Prolog atoms (in address or lexicographical
order) that represent such variables. Finally, the top variable of the constants 0 and 1 are considered greater than any integer or atom.

We chose to represent BDDs as ordinary, ground terms on the Prolog heap. The root BDD node of a non-constant function \( f \) is represented as the compound term \( \$\text{bdd} \_\text{pos}(I,T,E) \) or \( \$\text{bdd} \_\text{neg}(I,T,E) \), where \( I \) is the top variable of \( f \). If the node is tagged \( \$\text{bdd} \_\text{pos} \), \( T \) and \( E \) are the BDDs representing the cofactors \( f_T \) and \( f_E \), respectively. Otherwise, \( T \) and \( E \) are the BDDs representing the negated cofactors \( \bar{f}_T \) and \( \bar{f}_E \). In either case, \( T \) is restricted to be tagged positive or to be the constant 1, to guarantee canonicity. The same rule is used in [3] and elsewhere.

Thus, we are actually not typing the edges of the graph but the nodes themselves. Although we do not get the space reduction that genuine typed edges would achieve, we can still achieve constant-time negation and antivalence. Implementing typed edges would require major surgery into the tag system of the underlying Prolog.

A new global data structure, the unique-table, was added to ensure that all BDDs are kept canonical. It maps the top variable and daughter nodes of every node to the BDD node having those constituent parts, once such a node has been created. The unique-table is a hash table and can be assumed to operate in constant time. The garbage collector and stack shifter have been extended to notify the unique-table manager so that the table is kept consistent. When a garbage collection has occurred, the contents of the table are restored from the BDD nodes that have survived on the Prolog heap. The unique-table manager also has to take backtracking into account, since nodes may disappear due to backtracking.

5 Operations on BDDs

The operations one wants to perform on BDDs include testing for equivalence and antivalence, applying logical connectives, and applying substitutions.

**equivalence test** This test amounts to mere pointer comparison.

**antivalence test** This is the constant-time operation of testing whether \( f \) and \( g \) are nodes of opposite sign but with identical arguments, or if \( f \) and \( g \) are distinct constants.

**negation** To compute the node \( \bar{f} \) from \( f \), we "create" a node of opposite sign but identical arguments by consulting the unique-table.

**binary connectives** All binary connectives are expressed in terms of the if-then-else function \( \text{Ite}(x, y, z) \) defined as

\[
\text{Ite}(x, y, z) \triangleq x \cdot y + \bar{x} \cdot z
\]

For example,

\[
\begin{align*}
x \cdot y &= \text{Ite}(x, y, 0) \\
x + y &= \text{Ite}(x, 1, y) \\
x \oplus y &= \text{Ite}(x, \bar{y}, y)
\end{align*}
\]

The implementation of \( \text{Ite} \) is described in Section 5.1.

**existential quantification** The existential quantification (also called smoothing) of a variable \( v \) with respect to a formula \( f \) can be computed as

\[
(\exists v) f \triangleq f_v + f_{\bar{v}}
\]
universal quantification The universal quantification of a variable $v$ with respect to a formula $f$ (also called the consensus operator) can be computed as 

$$(\forall v)f \overset{\text{def}}{=} f_v * f_{\overline{v}}$$

substitution The substitution of a variable $v$ by an expression $w$ (not containing $v$) in a formula $f$ can be computed as 

$$f[v/w] \overset{\text{def}}{=} (\exists v)[(v \equiv w) * f]$$

Bryant in [5] introduces the Compose function for the substitution operation. The implementation of Compose is described in Section 5.2.

5.1 Implementation of the Ite function

It is easy to see that a recursive formulation of the ITE function is:

\[
\begin{align*}
\text{Ite}(F, 1, 0) &= F \\
\text{Ite}(1, G, H) &= G \\
\text{Ite}(0, G, H) &= H \\
\text{Ite}(F, G, H) &= \langle v, \text{Ite}(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}}), \text{Ite}(F, G_{\overline{v}}, H_{\overline{v}}) \rangle
\end{align*}
\]

where $v$ is the top variable of $F$, $G$, and $H$. The efficiency of this highly recursive function can be vastly improved by a number of techniques [4]. These techniques are described below.

5.1.1 Ite Function Memory

The most significant device is a function memory for Ite which maps the three argument nodes $F$, $G$, $H$ to the result node Ite($F$, $G$, $H$) once this result has been computed. The function memory is called the computed-table. Similarly to the unique-table, the computed-table manager is notified when garbage collections and stack shifts occur and takes backtracking into account. When a garbage collection has occurred, the part of the computed-table that corresponds to the segment of the heap that was subject to garbage collection [2] is erased. With the computed-table we can formulate Ite as in [4]:

\[
\text{Ite}(F, G, H)
\]

\[
\begin{cases}
\text{if } (Z = \text{base case}) \\
\quad \text{return } Z; \\
\text{else if } (Z = \text{computed-table}(\{F, G, H\}) \\
\quad \text{return } Z; \\
\text{else} \\
\quad \{ \\
\quad \quad \text{let } v = \text{TopVar}(\{F, G, H\}); \\
\quad \quad \text{let } T = \text{Ite}(F_v, G_{\overline{v}}, H_v); \\
\quad \quad \text{let } E = \text{Ite}(F_{\overline{v}}, G_v, H_{\overline{v}}); \\
\quad \quad \{ \\
\quad \quad \quad \text{if } (T = E) \\
\quad \quad \quad \quad \text{let } Z = T; \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \quad \text{let } Z = \text{unique-table}(\{v, T, E\}); \\
\quad \quad \}\end{cases}
\]

5
let computed-table(\{F, G, H\}) = Z;
return Z;
}
}

where TopVar(B) stands for the top variable of B.

The complexity of the above formulation is shown in [4] to be $O(|F| \cdot |G| \cdot |H|)$ but is claimed by the authors to be closer to the size of the resulting function.

5.1.2 Standard Triples

For given parameters $F_1, F_2, F_3$ there exist in general a large number of triples $G_1, G_2, G_3$ such that $F_i \neq G_i$ for some $i$ and Ite($F_1, F_2, F_3$) = Ite($G_1, G_2, G_3$). To improve the hit rate of the computed-table and to reduce its redundancy the authors of [4] define a set of rewrite rules on Ite calls that transforms them to a standard form. The rewrite conceptually takes place at entry to the Ite function. In the function body, the rewritten arguments are used instead of the original ones.

Below, we list our version of the rewrite rules. The rewrite rules are based on recognizing the constants 0 and 1, equivalences, and antivalences between parameters, and machine address order. For clarity, we have chosen to present the rules as mutually exclusive cases. We assume that 0, 1, $F, F, G, G, H, H$, are all distinct functions.

\[
\begin{align*}
\text{Ite}(0, -, 0) & \rightarrow 0 \\
\text{Ite}(0, -, 1) & \rightarrow 1 \\
\text{Ite}(0, -, F) & \rightarrow F \\
\text{Ite}(1, 0, -) & \rightarrow 0 \\
\text{Ite}(1, 1, -) & \rightarrow 1 \\
\text{Ite}(1, F, -) & \rightarrow F \\
\text{Ite}(-, 0, 0) & \rightarrow 0 \\
\text{Ite}(-, 1, 1) & \rightarrow 1 \\
\text{Ite}(-, F, F) & \rightarrow F \\
\text{Ite}(F, 0, 1) & \rightarrow F \\
\text{Ite}(F, 0, F) & \rightarrow 0 \\
\text{Ite}(F, 0, F) & \rightarrow 0 \\
\text{Ite}(F, 0, F) & \rightarrow 0 \\
\text{Ite}(F, 0, G) & \rightarrow \text{Ite}(F, G, 0) \\
\text{Ite}(F, 1, 0) & \rightarrow F \\
\text{Ite}(F, 1, F) & \rightarrow F \\
\text{Ite}(F, 1, F) & \rightarrow 1 \\
\text{Ite}(F, F, 0) & \rightarrow F \\
\text{Ite}(F, F, 1) & \rightarrow 1 \\
\text{Ite}(F, F, F) & \rightarrow 1 \\
\text{Ite}(F, F, G) & \rightarrow \text{Ite}(F, 1, G) \\
\text{Ite}(F, F, F) & \rightarrow 0 \\
\text{Ite}(F, F, 1) & \rightarrow F \\
\text{Ite}(F, F, F) & \rightarrow 0 \\
\text{Ite}(F, F, G) & \rightarrow \text{Ite}(F, 1, G) \\
\end{align*}
\]
\begin{align*}
\text{Ite}(F, G, 0) & \rightarrow \text{Ite}(G, F, 0), F > G \\
\text{Ite}(F, G, 1) & \rightarrow \text{Ite}(F, 1, G) \\
\text{Ite}(F, G, F) & \rightarrow \text{Ite}(F, G, 0) \\
\text{Ite}(F, G, \overline{F}) & \rightarrow \text{Ite}(F, G, 1) \\
\text{Ite}(F, G, \overline{G}) & \rightarrow \text{Ite}(G, F, \overline{F}), F > G \\
\text{Ite}(F, G, H) & \rightarrow \text{Ite}(\overline{F}, H, G), G > H
\end{align*}

The rules below were used in [4]. We chose not to use them, since preliminary measurements indicated that the decrease in computed-table growth was outweighed by an increase in the number of nodes created. This is probably due to our lack of genuine typed edges.

\begin{align*}
\text{Ite}(F, G, H) & \rightarrow \overline{\text{Ite}(F, G, H)} \\
\text{Ite}(\overline{F}, G, H) & \rightarrow \overline{\text{Ite}(F, H, G)}
\end{align*}

Finally, the rules below prevent a lot of shallow recursion and so save space in the computed-table. A special case of the first rule was quoted in [4].

\begin{align*}
\text{Ite}(\langle f, 1, 0 \rangle, G, H) & \rightarrow \langle f, G_f, H_f \rangle, f \leq g, f \leq h \\
\text{Ite}(\langle f, 0, 1 \rangle, G, H) & \rightarrow \langle f, H_f, G_f \rangle, f \leq g, f \leq h \\
\text{Ite}(F, \langle g, 1, 0 \rangle, 0) & \rightarrow \langle g, F_g, 0 \rangle, g \leq f \\
\text{Ite}(F, \langle g, 0, 1 \rangle, 0) & \rightarrow \langle g, 0, F_g \rangle, g \leq f \\
\text{Ite}(F, \langle h, 1, 0 \rangle) & \rightarrow \langle h, 1, F_h \rangle, h \leq f \\
\text{Ite}(F, \langle h, 0, 1 \rangle) & \rightarrow \langle h, F_h, 1 \rangle, h \leq f \\
\text{Ite}(F, \langle g, 1, 0 \rangle, \langle g, 0, 1 \rangle) & \rightarrow \langle g, F_g, \overline{F_g} \rangle, g \leq f \\
\text{Ite}(F, \langle g, 0, 1 \rangle, \langle g, 1, 0 \rangle) & \rightarrow \langle g, \overline{F_g}, F_g \rangle, g \leq f \\
\text{Ite}(\langle f, P, Q \rangle, \langle f, Q, P \rangle) & \rightarrow \text{Ite}(P, 1, Q) \\
\text{Ite}(\langle f, P, Q \rangle, \langle f, Q, P \rangle, 0) & \rightarrow \text{Ite}(P, Q, 0) \\
\text{Ite}(\langle f, P, Q \rangle, \langle f, Q, P \rangle, \langle f, \overline{Q}, \overline{P} \rangle) & \rightarrow \text{Ite}(P, Q, \overline{Q}) \\
\text{Ite}(\langle f, P, Q \rangle, \langle f, \overline{Q}, \overline{P} \rangle, \langle f, Q, P \rangle) & \rightarrow \text{Ite}(P, \overline{Q}, Q)
\end{align*}

where \( f, g, h \) are the top variables of respectively \( F, G, H \).

### 5.2 The Compose function

The function \( \text{Compose}(f, v, g) \) computes the results of replacing the variable \( v \) by the function \( g \) in the function \( f \). It is easy to arrive at a recursive formulation in terms of \( \text{Ite} \) as:

\begin{align*}
\text{Compose}(F, v, G) & = F, f > v \\
\text{Compose}(F, v, G) & = \text{Ite}(G, F_f, F_f), f = v \\
\text{Compose}(F, v, G) & = \langle f, \text{Compose}(F_f, v, G), \text{Compose}(F_f, v, G) \rangle, f < v
\end{align*}

where \( f \) is the top variable of \( F \).
It turns out that for the purpose of applying a substitution obtained from Boolean unification, a generalization of the Compose function is more useful than the basic function and can be much more efficient. We therefore introduce the function $\text{Apply}(F, S)$ which computes the result of applying the substitution $S$ on the function $F$. The recursive formulation of $\text{Apply}$ is:

$$\begin{align*}
\text{Apply}(F, S) &= F, f > s \\
\text{Apply}(F, S) &= \text{Ite}(G, \text{Apply}(F_f, S), \text{Apply}(F_f, S)), f/G \in S \\
\text{Apply}(F, S) &= \langle f, \text{Apply}(F_f, S), \text{Apply}(F_f, S) \rangle, f/G \notin S
\end{align*}$$

where $s$ is the top variable of $S$ and $f$ is the top variable of $F$.

As with the Ite function, the performance of the Apply function can be vastly improved if it is equipped with a function memory. So, we introduce the apply-table which maps the argument node $F$ to the result node $\text{Apply}(F, S)$ once this result has been computed. Notice that the apply-table is only valid w.r.t. a particular substitution $S$ and so has to be explicitly purged between top-level calls to Apply. It is also affected by garbage collections and stack shifts like the computed-table. We assume that the substitution is represented as a list of pairs $v/G$ ordered in $v$ by increasing values. With the apply-table we can formulate Apply as:

```plaintext
Apply(F, S)
{
  if (F is a constant or f is an atom)
    return F;
  while (S = v/G, S' ∧ v < f)
    let S = S';
  if (F is tagged positive)
    {
      let z = 0;
      let F' = F;
    }
  else
    {
      let z = 1;
      let F' = F;
    }
  if (S = φ)
      return F' ⊕ z;
  else if (v = f)
    {
      let G' = G;
      let S = S';
      if (F' = (f, 1, 0))
        return G ⊕ z;
      if (G = 1)
        return Apply(F'_f, S) ⊕ z;
      if (G = 0)
        return Apply(F'_f, S) ⊕ z;
    }
}
```
else
  {
    let \( G' = (f, 1, 0) \);

    if (\( F' = (f, 1, 0) \))
      return \( F' \oplus z \);
  }

if (\( Z = \text{apply-table}(F') \))
  return \( Z \oplus z \);

if (\( F'_f = \overline{F'_f} \))
  {
    let \( Z = \text{Apply}(F'_f, S) \);
    let \( Z' = \text{apply-table}(F') = \text{Ite}(G', Z, \overline{Z}) \);
    return \( Z' \oplus z \);
  }
else
  {
    let \( Z = \text{Ite}(G', \text{Apply}(F'_f, S), \text{Apply}(F'_f, S)) \);
    let \( \text{apply-table}(F') = Z \);
    return \( Z \oplus z \);
  }
}

where \( f \) is the top variable of \( F \).

6 Boolean Unification with BDDs

It should by now be clear to the reader how the Boolean unification algorithm can be expressed in terms of BDD operations.

In the procedural version of Boolean's method (Section 3.2.1), the variable \( v \) is simply chosen as the top variable of the function \( g \). To apply the substitution \( \sigma \), we use the Apply function.

There are a number of opportunities for optimizing the Boolean unification algorithm. The rest of this section is devoted to some optimizations that we have identified and exploited.

6.1 Reusing Variable Indices

It is worth noting that the each introduced variables \( (w) \) can be assigned the same BDD variable index as the eliminated variable \( (f) \). This leads to significant space and time savings. This optimization was discovered independently by Pascal Van Hentenryck.

6.2 Special Cases

At each recursion level in Boolean's method, we have either a base case or an equation

\[ v \star g_v + \overline{v} \star g_{\overline{v}} = 0 \]

from which we must compute a substitution

\[ v := w \star g_v + \overline{w} \star g_{\overline{v}} \]
and a residual equation

\[ g_v + g_{\overline{v}} = 0 \]

We identify the following special cases of the above steps:

<table>
<thead>
<tr>
<th>( g_v )</th>
<th>( g_{\overline{v}} )</th>
<th>( v := w * g_v + w * g_{\overline{v}} )</th>
<th>( g_v * g_{\overline{v}} = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>( v := 0 )</td>
<td>0 = 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( v := 1 )</td>
<td>0 = 0</td>
</tr>
<tr>
<td>1</td>
<td>( r )</td>
<td>( v := 0 )</td>
<td>( r = 0 )</td>
</tr>
<tr>
<td>0</td>
<td>( r )</td>
<td>( v := w + r )</td>
<td>0 = 0</td>
</tr>
<tr>
<td>( q )</td>
<td>0</td>
<td>( v := w * \overline{q} )</td>
<td>0 = 0</td>
</tr>
<tr>
<td>( q )</td>
<td>1</td>
<td>( v := 1 )</td>
<td>( q = 0 )</td>
</tr>
<tr>
<td>( q )</td>
<td>( \overline{q} )</td>
<td>( v := \overline{q} )</td>
<td>0 = 0</td>
</tr>
<tr>
<td>( q )</td>
<td>( r )</td>
<td>( v := w * \overline{q} + \overline{w} * r )</td>
<td>( q * r = 0 )</td>
</tr>
</tbody>
</table>

It is worth noting that in the cases where \( g_v = 1 \), we can immediately deduce \( v := 0 \) instead of constructing a substitution which eventually will reduce to 0 anyway, when the mgu of the residual equation is applied. Similarly for \( g_{\overline{v}} = 1 \).

### 6.3 Solving Sets of Equations

We exploit the tautology

\[ g_1 + \ldots + g_n = 0 \equiv g_1 = 0 \land \ldots \land g_n = 0 \]

as follows: Instead of translating the formula to a single BDD, we try to split it into a disjunction \( g_1 + \ldots + g_n \), translate each \( g_i \) to a BDD, and apply Boolean unification to the resulting set of equations. The advantage of this is that failures are detected earlier: as soon as an obviously insoluble \( g_i \) is found, \( g \) is insoluble. Also, it is hoped that space and time savings result from avoiding constructing a BDD for the whole formula. In Section 8 some data to substantiate this claim are presented.

To adapt Boole’s algorithm for handling a set of equation, we add the requirement that the top variables of the \( g_i \) must all be distinct. If two disjuncts \( g_i, g_j \) have the same top variable, they are simply replaced by \( g_i + g_j \) instead. This is captured in the EqnMerge procedure below. The procedure “MguSet” modifies Boole’s algorithm for operating on a set of equations:

**MguSet**(\( S \))

\[
\begin{align*}
\text{if } (S = \emptyset) & \text{ return } \emptyset; \\
\text{let } S = \{g\} \cup S' \text{ chosen such that} & \\
& v = \text{TopVar}(g) < \text{TopVar}(S'); \\
& \text{let } \sigma = \text{MguSet}(\text{EqnMerge}(g_v * g_{\overline{v}}, S')); \\
& \text{return } \{v/\sigma(w * g_v + \overline{w} * g_{\overline{v}})\} \cup \sigma;
\end{align*}
\]

**EqnMerge**(\( g, S \))

\[
\begin{align*}
\text{if } (g = 0) & \text{ return } S; \\
\text{else if } (g = 1 \text{ or TopVar}(g) \text{ is quantified}) & \text{ return FAILURE;} \\
\text{else if } (\text{TopVar}(g) \neq \text{TopVar}(S)) & \text{ return } S \cup \{g\};
\end{align*}
\]
else
    return EqnMerge(g + g', S \ {g'}) chosen such that
    TopVar(g) = TopVar(g');

6.4 The Cardinality Operator

One of the constraints of our language is the cardinality operator \text{card}(I, F) where I is a set of integers and F is a set of formulas. The interpretation of this operator is the constraint that the number of true formulas in F is a member of I. This is a trivial generalization of the cardinality operator as defined in [14]. An implementation has to face the problem of translating a cardinality formula to a BDD.

A straightforward approach would be to expand the formula to disjunctive normal form. For example,

\text{card}([2], \{f,g,h\})

could be expanded to

\begin{align*}
    f \cdot g \cdot h + f \cdot \overline{g} \cdot h + \overline{f} \cdot g \cdot h
\end{align*}

but this approach rapidly breaks down due to combinatorial explosion.

Gunnar Stålmarck pointed out to me that it is possible to construct a BDD for \text{card}(I, F) in time and space linear in |F| for a given I by building a “counter network”. A counter network for n variables is a BDD with n + 1 leaf nodes arranged such that the value of the BDD is the i\text{th} leaf node if exactly i + 1 of the variables are true. By assigning 0s and 1s to the leaf nodes, and substituting formulas for the variables, any cardinality formula can be expressed as a counter network.

For example, the formula

\text{card}([2,3], \{a,b,c,d,e\})

translates to counter network is shown in Figure 1. After normalizing, we obtain the BDD shown in Figure 2.

![Counter Network](image_url)

**Figure 1**: Counter network for \text{card}([2,3], \{a,b,c,d,e\})
6.5 Existential Quantification

Although it is straightforward to use the definition of existential quantification of a variable \( v \) with respect to a formula \( f \) as is, i.e.

\[
(\exists v)f \equiv f_v + f_{\overline{v}}
\]

this can be expensive. It is much more efficient to arrange the variable order such that \( v = \text{TopVar}(f) \), in which case \( f_v \) and \( f_{\overline{v}} \) can be computed in constant time as the daughter nodes of the top node of the BDD representing \( f \). This optimization becomes especially important if several variables are to be quantified at once, for example when computing answer constraints (see Section 7.4).

7 Built-In Operations

7.1 Consistency Check

The consistency check is expressed as the built-in predicate \( \text{bool:sat}(E) \). Its purpose is to check the consistency of the Boolean expression \( E \) and the accumulated constraints \( C \), that is, to check the satisfiability of \( C \cup \{E\} \). If the check succeeds, \( E \) is added to the accumulated constraints.

The consistency check is performed by translating \( \overline{E} \) to a BDD \( E' \) and applying Boolean unification to \( E' = 0 \). The translation process involves assigning variable indexes to the variables in \( E \) and possibly dereferencing them as described below. Variable indexes are assigned locally for each invocation of the constraint solver, instead of globally. This increases the amount of sharing of BDD nodes, since variable indexes are reused over and over again.

As a by-product, the obtained mgu provides the natural "storage location" for storing the new constraint, namely the eliminated variables. To add the new constraint to the system, the formula in each right-hand side of the mgu is stored under its corresponding left-hand-side \( X_i \), and will be retrieved again if and when \( X_i \) occurs in a new constraint.

Each formula is stored as the frozen [7] goal \( \text{boolvalue}(X_i, G, M) \), where \( G \) is the "skeleton" of the formula, and \( M \) is its "binding environment" which maps BDD variable indexes to Prolog terms. Whenever \( X_i \) occurs again in a new constraint, it is dereferenced by retrieving this frozen goal and renaming the variables in \( M \) to the indexes being assigned for the current invocation of the constraint solver. The renaming is efficiently performed.
with the Apply function. This process may need to recursively dereference the variables in $M$.

Notice that in the absence of universally quantified variables, the consistency check per se would just amount to constructing $E'$ and comparing it to the constant 1.

### 7.2 Entailment Check

The entailment check is expressed as the built-in predicate `bool:taut(E,T)`. Its purpose is to check whether $E$ or its negation is entailed by the accumulated constraints $C$, that is, whether $C \models E$ or $C \models \neg E$. The variable $T$ takes on the value 1 in the previous case and 0 in the latter case. If neither case holds, the predicate simply fails.

The entailment check is performed by translating $\neg E$ to a BDD $E'$ and applying to it a variant of Boolean unification, where the obtained mgu is simply ignored. The following cases are possible:

- $E'$ is the constant 0, (the check succeeds with $T = 1$);
- $E'$ is soluble but $E' \neq 0$, (the check fails); or
- $E'$ is insoluble, (the check succeeds with $T = 0$).

Notice that in the absence of universally quantified variables, there would be scope for significant optimizations of this algorithm, using a variant of the Ite function specialized to return 0 or 1 only [4].

### 7.3 Generating Particular Solutions

The constraint solver never commits itself to any particular solution of the accumulated constraints. If such solutions are needed, the built-in predicate `bool:labeling(L)` is available, which instantiates the list of variables $L = x_1 \ldots x_n$ to 0s and 1s satisfying the accumulated constraints.

A possible implementation of this primitive would be to combine any constraints on $x_1 \ldots x_n$ into a single function of $x_1 \ldots x_n$ represented as a BDD. Finding a particular solutions then consists in a depth-first search of the BDD. In the absence of universally quantified variables, such a search would find the first particular solution in time linear in $n$ [5]. If universally quantified variables appear in the BDD, however, the search could require exponential time due to arbitrary amounts of backtracking. For this reason, and because it was feared that the memory requirements of this approach would be prohibitive, another solution was chosen.

To find a particular solution, $x_1 \ldots x_n$ are first split into two parts: $L_1$, the variables without constraints, and $L_2$, the variables with constraints. $L_1$ is first assigned some combination of 0s and 1s (enumerating all combinations by backtracking). Then an arbitrary variable $y \in L_2$ is chosen, and in two different branches of the computation, the new constraints $y = 0$ and $y = 1$ are checked for consistency and added, respectively, which can lead to the constraints on $L_2 \setminus \{y\}$ being updated. In each branch, the whole process is repeated on $L_2 \setminus \{y\}$ until no more constrained variables remain.

### 7.4 Prolog Top-Level Extensions

Any CLP implementation is faced with the problem of presenting the answer constraints in a way that is meaningful to the user. In particular, if the constraint solver is complete, any auxiliary constraints not involving the query variables are irrelevant to the answer constraints, since such auxiliary constraints are always satisfiable.

The only reasonable way of presenting the answer constraints seems to be to "project" the accumulated constraints onto the query variables. This is precisely what existential quantification is useful for.
To achieve the projection onto the query variables, we solve the constraint \( (\exists X) Q \), where \( X \) is the set of constrained variables which do not occur in the query, and \( Q \) is the set of constraints constraining the query variables. The residual constraints extracted from the mgu are stored on the query variables as usual, and are displayed to the user as polynomials composed of variables and the connectives \( \ast \) and \( \oplus \).

### 7.5 Prolog Unification Extension

There is a connection between the constraint solver and the Prolog unifier via the mechanism for storing a constraint on a variable. This mechanism is simply the usual freeze mechanism for delaying a Prolog goal until a variable is bound. If a variable \( X \) subject to a constraint is bound to a value \( V \), a goal \texttt{boolvalue}(V,G,M) will be woken. The result of the woken goal will be almost as if \texttt{bool: sat}(X \equiv V) had been called, with one important exception:

- Binding \( X \) to \( V \) may temporarily violate certain internal invariants, and the constraint solver may have to repair this situation. In particular, such variable bindings may cause infinite dereference chains, and the solver’s dereference mechanism has to exercise extra care to cope with such cases. This situation is not very satisfactory, and these problems could be avoided if the freeze mechanism were generalized so that variable bindings were not made immediately but executed as part of the execution of the woken goals. This idea is due to Christian Holzbaur.

Thus, the frozen \texttt{boolvalue}/3 goals play double roles. First, they are a repository for constraints on variables. Secondly, they enable a natural connection between the constraint solver and the Prolog unifier.

### 8 Performance

In this section we present performance results from running the solver on some selected examples. We will first briefly describe the benchmarks:

- **pigeon** \( i \times j \) The problem of placing \( j \) pigeons into \( i \) holes, such that each pigeon-hole holds only one pigeon.

- **schur** \( N \) Schur’s lemma: how to distribute the numbers \( 1 \ldots N \) into three boxes such that
  - (i) no box can contain the numbers \( i \) and \( 2i \), and
  - (ii) no box can contain the numbers \( i \), \( j \), and \( i + j \).

  For \( N \leq 13 \) the problem admits solution; beyond, it doesn’t.

- **ramsey** \( i/j \) Ramsey problems: for a complete graph with \( i \) vertices, color the edges with two colors so that at most \( j \) complete subgraphs with three edges become monochrome.

  For \( i = 6 \), there are solutions if \( j \geq 2 \). For \( i = 7 \), there are solutions if \( j \geq 4 \).

- **ulm216r2** This is a problem in 3SAT form from Prof. Deussen from University of Karlsruhe. It contains 480 clauses and is satisfiable.

Two tables are shown below. The first table contains examples that are expressed as non-singleton conjunctions of constraints. The solver translates such conjunctions into sets of equations (see Section 6.3). This table does not include the Ramsey problems, since they are all expressed as cardinality constraints.

In the second table, all examples are stated as formulas which do not have \( \ast \) as the principal connective. The solver translates such formulas into a single BDD before applying Boolean unification.

The columns are:

- **size** \texttt{formula} The number of variables in the formula, and the size of the BDD representation in nodes.
parse  Time spent and number of nodes created while translating the formula to a set of equations or to a single equation.

sat  Time spent and number of nodes created while computing the mgu of the translated formula.

taut  Time spent and number of nodes created while checking whether the translated formula is valid or unsatisfiable.

Times are in seconds (Sun-4/60, compiled with -O1) The notation n/a means that the available memory was exceeded.

It is worth noting that the parse time for all examples in the first table is much smaller than in table two, whereas the time for checking consistency or entailment for the same examples is much larger than in table two (except for ulm216r2 which could not be run at all in “grouped” form). Thus, the benchmark data do not give an unequivocal answer to the question whether the technique of solving a set of equations is a worthwhile optimization. But we chose to implement that technique, since some examples cannot be run otherwise.

Note also the difference in performance between consistency and entailment checks on ramsey6/2. This illustrates the extra work performed by the consistency check in computing the mgu.

<table>
<thead>
<tr>
<th>ungrouped name</th>
<th>size formula variables</th>
<th>size formula nodes</th>
<th>parse cpu time</th>
<th>parse nodes</th>
<th>sat cpu time</th>
<th>sat nodes</th>
<th>taut cpu time</th>
<th>taut nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>pigeon8x9</td>
<td>72</td>
<td>269</td>
<td>0.37</td>
<td>617</td>
<td>26.7</td>
<td>87545</td>
<td>26.7</td>
<td>87545</td>
</tr>
<tr>
<td>schur13</td>
<td>39</td>
<td>323</td>
<td>0.33</td>
<td>455</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>schur14</td>
<td>42</td>
<td>369</td>
<td>0.4</td>
<td>518</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>ulm216r2</td>
<td>184</td>
<td>960</td>
<td>3.29</td>
<td>1616</td>
<td>1.26</td>
<td>1544</td>
<td>0.95</td>
<td>1352</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grouped name</th>
<th>size formula variables</th>
<th>size formula nodes</th>
<th>parse cpu time</th>
<th>parse nodes</th>
<th>sat cpu time</th>
<th>sat nodes</th>
<th>taut cpu time</th>
<th>taut nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>pigeon8x9</td>
<td>72</td>
<td>0</td>
<td>4.86</td>
<td>25214</td>
<td>0.0</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>schur13</td>
<td>39</td>
<td>219</td>
<td>8.71</td>
<td>40151</td>
<td>2.1</td>
<td>374</td>
<td>0.1</td>
<td>262</td>
</tr>
<tr>
<td>schur14</td>
<td>42</td>
<td>0</td>
<td>11.1</td>
<td>51644</td>
<td>0.0</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>ulm216r2</td>
<td>184</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>ramsey6/1</td>
<td>20</td>
<td>n/a</td>
<td>0.92</td>
<td>3927</td>
<td>0.0</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>ramsey6/2</td>
<td>20</td>
<td>548</td>
<td>1.7</td>
<td>7837</td>
<td>9.0</td>
<td>15813</td>
<td>0.31</td>
<td>513</td>
</tr>
<tr>
<td>ramsey7/3</td>
<td>35</td>
<td>0</td>
<td>15.6</td>
<td>58707</td>
<td>0.0</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>ramsey7/4</td>
<td>35</td>
<td>2183</td>
<td>25.4</td>
<td>90795</td>
<td>n/a</td>
<td>n/a</td>
<td>3.86</td>
<td>2777</td>
</tr>
</tbody>
</table>

9 Conclusions

We have described how a complete and efficient Boolean constraint solver has been built into an existing Prolog engine. The bulk of the implementation is a package for Binary Decision Diagrams (BDDs) integrated into the Prolog memory management. Details on the implementation and its performance have been given.

We have described the interactions between the Prolog engine and the constraint solver. These consist of built-in predicates for checking consistency and entailment and for generating particular solutions, support for displaying answer constraints, and a natural connection to the Prolog unifier. The algorithms used by the constraint solver for these different functions have been outlined.

Boolean unification was chosen as the strategy for the consistency check. We have described the algorithm and its implementation as operations on BDDs and opportunities for optimizations, not previously described in the literature.
We have pointed out how some opportunities for optimizing the algorithms for entailment check and generating particular solutions are spoilt by the existence of universally quantified variables.

The freeze mechanism for delaying a Prolog goal until a variable is bound was used to establish a link between Prolog unification and the constraint solver. We have shown that this idea leads to some problems, and that a more flexible mechanism for waking goals is probably needed in CLP systems implemented on top of Prolog engines.

We believe that the general structure of our implementation can be generalized to at least some other domains.
A C Types and Functions

This section documents the data types and functions used in the C part of the BDD package.

A.1 Hash Tables

Three hash tables are used in the implementation:

bdd_node_table
   This is the unique-table.

bdd_current_node
   The slot in the unique-table holding the most recent entry.

bdd_univ_table
   This is the computed-table.

bdd_current_univ
   The slot in the computed-table holding the most recent entry.

bdd_subst_table
   This is the apply-table.

bdd_current_node
   The slot in the apply-table holding the most recent entry.

These hash tables are operated in a similar way: no separate overflow tables are used, and the entries of a table form a linked list (most recent entry first). The linked lists are needed for deleting entries which have become invalid due to backtracking or memory management. Whenever a hash table is accessed, a check is first made to see whether such entries need to be deleted.

A.2 Support Functions

bdd_begin()
   This initializes the hash tables and certain global variables.

bdd_end()
   This deallocates all memory currently in use by the BDD package.

bdd_reset_node(h)
   This deletes entries from the unique-table which have become invalid due to backtracking, stack shifting or garbage collection. The argument h is the Prolog heap pointer.

bdd_reset_univ(h)
   This deletes entries from the computed-table which have become invalid due to backtracking, stack shifting or garbage collection. The argument h is the Prolog heap pointer.

bdd_reset_subst(h)
   This deletes entries from the apply-table which have become invalid due to backtracking. The argument h is the Prolog heap pointer.

bdd_rehash(newsize,tablep,curp)
   This expands a hash table that has become 50% full, or after stack shifting. The argument newsize is the new size, tablep is a pointer to the hash table, and curp is a pointer to the most recent entry in it. The linked list of entries, in order of recency, is preserved in the expanded table as well.
bdd_heap_overflow(reloc,newh)
This is called by the stack shifter after it has relocated the contents of the Prolog heap. The argument reloc is the quantity by which heap pointers in the hash tables have to be incremented, and newh is the new Prolog heap pointer. All three hash tables are rehashed.

bdd_garbage_collect(oldh,newh)
This is called by the garbage collector after it has collected garbage in the Prolog heap. The argument oldh is the lower limit of the segment subjected to garbage collection, and newh is the new heap pointer. All table entries more recent than oldh are invalidated. The garbage collected segment is then scanned for surviving BDD nodes, which are reentered into the unique-table.

bdd_lookup_internal(w,tablep,curp,func,tvar,then,telse,val,insert)
This is the generic hash table lookup and insertion routine. The arguments tablep and curp are pointers to the hash table and most recent entry, func,tvar,then,telse make up the hash key, value is the value to store, and insert is 1 if a value should be stored.

For the unique-table, func will be one of the functors $\text{bdd_pos}$ and $\text{bdd_neg}$ and value will be 0, since no extra information is to be associated with nodes. For the computed-table, the keys are constructed as Prolog terms $\text{bdd_key}(I,T,E)$. For the apply-table, func will be the same as for the unique-table.

bdd_build_node(w,func,tvar,then,telse)
Looks up or inserts into the unique-table a BDD node with the given constituents.

bdd_build_cofactors(w,unit,then,telse)
If unit is a node $(v,1,0)$ and then,telse are nodes $f,g$ and $v \leq \text{TopVar}(f,g)$, this returns the node $(v,f_v,g_v)$. Similarly if unit is $(v,0,1)$, the node $(v,g_v,f_v)$ is returned.

bdd_antiv_internal(t,u)
Returns TRUE if t,u are antivalent, and FALSE otherwise.

bdd_transv_internal(t,u)
Let t,u be the nodes $f,g$. Returns 1 if

\[ v = \text{TopVar}(f) = \text{TopVar}(g) \land f_v = g_v \land f_v = g_v \]

or -1 if

\[ v = \text{TopVar}(f) = \text{TopVar}(g) \land f_v = \overline{g_v} \land f_v = \overline{g_v} \]

or 0 otherwise.

bdd_find_value(tree)
Returns the frozen goal holding a Boolean constraint for a given constrained variable.

compile_bdd()
Compiles a BDD which was encountered while asserting a dynamic clause to a sequence of operands of the new WAM instruction GET_BDD, which was introduced to support this case. The format of the operands is documented in the code.

A.3 Built-in Predicates
The following predicates are implemented as C functions:

$\text{bdd_build}(+\text{Var},+T,+E,-\text{Node})$
Constructs as Node the node $(\text{Var},T,E)$.
$\text{bdd\_store}(+I,+T,+E,+Val)$
Records $Val$ as the value of $\text{Ite}(I, T, E)$ in the computed-table.

$\text{bdd\_parts}(+\text{Node},-\text{Var},-T,-E)$
Decomposes a node into its constituent parts.

$\text{bdd\_negate}(+\text{Pos},-\text{Neg})$
Negates a node.

$\text{bdd\_equiv}(+S,+T)$
Tests whether the two nodes $S$ and $T$ are equivalent. Not currently in use.

$\text{bdd\_antiv}(+S,+T)$
Tests whether the two nodes $S$ and $T$ are antivalent. Not currently in use.

$\text{bdd\_transv}(+S,+T,-I)$
Returns in $I$ the value of $\text{bdd\_transv\_internal}(S, T)$. Not currently in use.

$\text{bdd\_type}(+S,+T,-I)$
Returns in $I$ which case the combination of $S, T$ represents in certain situations. The case is encoded as:

<table>
<thead>
<tr>
<th>S</th>
<th>T</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q$</td>
<td>$q$</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$r$</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>$r$</td>
<td>3</td>
</tr>
<tr>
<td>$q$</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>$q$</td>
<td>$q$</td>
<td>5</td>
</tr>
<tr>
<td>$q$</td>
<td>$r$</td>
<td>6</td>
</tr>
<tr>
<td>$q$</td>
<td>$r$</td>
<td>7</td>
</tr>
</tbody>
</table>

$\text{bdd\_subst\_reset}$
The apply-table is purged.

$\text{bdd\_subst\_store}(+\text{Key},+\text{Val})$
Records $Val$ as the value of $\text{Key}$ in the apply-table.

$\text{bdd\_univ\_case}(+F,+G,+H,-C,-X,-Y,-Z)$
Performs the case analysis of the $\text{Ite}(F, G, H)$ function and handles all the cases that don’t need recursive calls. The particular subcase is returned in $C$ which can take the values:

1  The value is $X$.

11 The value is $\text{Ite}(X, Y, Z)$ and $\text{TopVar}(Z) > \text{TopVar}(X) < \text{TopVar}(Y)$.

12 The value is $\text{Ite}(X, Y, Z)$ and $\text{TopVar}(Z) > \text{TopVar}(Y) < \text{TopVar}(X)$.

13 The value is $\text{Ite}(X, Y, Z)$ and $\text{TopVar}(X) > \text{TopVar}(Z) < \text{TopVar}(Y)$.

14 The value is $\text{Ite}(X, Y, Z)$ and $\text{TopVar}(X) = \text{TopVar}(Y) < \text{TopVar}(Z)$.
The value is $\text{Ite}(X, Y, Z)$ and $\text{TopVar}(X) = \text{TopVar}(Z) < \text{TopVar}(Y)$.

The value is $\text{Ite}(X, Y, Z)$ and $\text{TopVar}(X) < \text{TopVar}(Y) = \text{TopVar}(Z)$.

The value is $\text{Ite}(X, Y, Z)$ and $\text{TopVar}(X) = \text{TopVar}(Y) = \text{TopVar}(Z)$.

$\text{bdd\_subst\_case}(\sigma, +X, -C, -\sigma', -V, -Q, -R)$

Performs the case analysis of the $\text{Apply}(X, \sigma)$ function and handles all the cases that don't need recursive calls. The particular subcase is returned in $C$ which can take the values:

0

The value is $R$.

1

The value is $\overline{R}$.

10

The value is $\langle V, \text{Apply}(Q, \sigma'), \text{Apply}(R, \sigma') \rangle$.

11

The value is $\overline{\langle V, \text{Apply}(Q, \sigma'), \text{Apply}(R, \sigma') \rangle}$.

20

The value is $\langle V, \text{Apply}(Q, \sigma'), \overline{\text{Apply}(Q, \sigma')} \rangle$.

21

The value is $\langle V, \text{Apply}(Q, \sigma'), \text{Apply}(Q, \sigma') \rangle$.

$\text{bdd\_tell\_value}(+\text{Var}, +\text{BDD}, +\text{Map})$

Stores the frozen goal $\text{boolvalue}($Var, BDD, Map$)$ under the variable Var. If such a goal already exists, the new goal replaces the old, and the old goal is restored on backtracking.

$\text{bdd\_ask\_value}(+\text{Var}, -\text{BDD}, -\text{Map})$

Retrieves the frozen goal $\text{boolvalue}($Var, BDD, Map$)$ under the variable Var. If such a goal does not exist, BDD and Map are both unified with $\Box$ instead.
References


