BOOLEAN: Computing with boolean expressions

H. Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin – Dahlem
Federal Republic of Germany
E–mail: melenk@zib.de

1 Introduction

The package **Boolean** supports the computation with boolean expressions in the propositional calculus. The data objects are composed from algebraic expressions (“atomic parts”, “leafs”) connected by the infix boolean operators **and**, **or**, **implies**, **equiv**, and the unary prefix operator **not**. **Boolean** allows you to simplify expressions built from these operators, and to test properties like equivalence, subset property etc. Also the reduction of a boolean expression by a partial evaluation and combination of its atomic parts is supported.

2 Entering boolean expressions

In order to distinguish boolean data expressions from boolean expressions in the REDUCEprogramming language (e.g. in an **if** statement), each expression must be tagged explicitly by an operator **boolean**. Otherwise the boolean operators are not accepted in the REDUCEalgebraic mode input. The first argument of **boolean** can be any boolean expression, which may contain references to other boolean values.

```
    boolean (a and b or c);
    q := boolean(a and b implies c);
    boolean(q or not c);
```

Brackets are used to override the operator precedence as usual. The leafs or atoms of a boolean expression are those parts which do not contain a leading boolean operator. These are considered as constants during the boolean evaluation. There are two pre-defined values:

- **true**, **t** or **1**
- **false**, **nil** or **0**

These represent the boolean constants. In a result form they are used only as **1** and **0**.

By default, a **boolean** expression is converted to a disjunctive normal form, that is a form where terms are connected by **or** on the top level and each term is set of leaf expressions, eventually preceded by **not** and connected by **and**. An operators **or** or **and** is omitted if it would have only one single operand. The result of the transformation is again an expression with leading operator **boolean** such that the boolean expressions remain separated from other algebraic data. Only the boolean constants **0** and **1** are returned untagged.

On output, the operators **and** and **or** are represented as `/\` and `\/`, respectively.

```
boolean(true and false);    ->   0
boolean(a or not(b and c)); -> boolean(not(b) \/ not(c) \/ a)
boolean(a equiv not c);     -> boolean(not(a)/\c \/ a/\not(c))
```

3 Normal forms

The **disjunctive** normal form is used by default. It represents the "natural" view and allows us to represent any form free or parentheses. Alternatively a **conjunctive** normal form can be selected as simplification target, which is a form with leading operator **and**. To produce that form add the keyword **and** as an additional argument to a call of **boolean**.

```
boolean (a or b implies c);
                    ->
     boolean(not(a)/\not(b) \/ c)
```

```
boolean (a or b implies c, and);
                           ->
       boolean((not(a) \/ c)/\(not(b) \/ c))
```

Usually the result is a fully reduced disjunctive or conjuntive normal form, where all redundant elements have been eliminated following the rules

$a \wedge b \vee \neg a \wedge b \longleftrightarrow b$

$a \vee b \wedge \neg a \vee b \longleftrightarrow b$

Internally the full normal forms are computed as intermediate result; in these forms each term contains all leaf expressions, each one exactly once. This unreduced form is returned when you set the additional keyword **full**:

```
boolean (a or b implies c, full);
                          ->
boolean(a/\b/\c \/ a/\not(b)/\c \/ not(a)/\b/\c \/ not(a)/\not(b)/\c

            \/ not(a)/\not(b)/\not(c))
```

The keywords **full** and **and** may be combined.

4 Evaluation of a boolean expression

If the leafs of the boolean expression are algebraic expressions which may evaluate to logical values because the environment has changed (e.g. variables have been bound), you can re–investigate the expression using the operator `testbool` with the boolean expression as argument. This operator tries to evaluate all leaf expressions in REDUCEboolean style. As many terms as possible are replaced by their boolean values; the others remain unchanged. The resulting expression is contracted to a minimal form. The result **1** (= true) or **0** (=false) signals that the complete expression could be evaluated.

In the following example the leafs are built as numeric greater test. For using $>$ in the expressions the greater sign must be declared operator first. The error messages are meaningless.

```
operator >;
fm:=boolean(x>v or not (u>v));
          ->
```

```
    fm := boolean(not(u>v) \/ x>v)

v:=10$

testbool fm;

   ***** u - 10 invalid as number
   ***** x - 10 invalid as number

        ->
   boolean(not(u>10) \/ x>10)

x:=3$
testbool fm;

   ***** u - 10 invalid as number

        ->
   boolean(not(u>10))

x:=17$

testbool fm;

   ***** u - 10 invalid as number

        ->
    1
```