A Boolean Simplifier in LISP

Tzewen Wang
Department of Electrical and Computer Engineering
Maseeh College of Engineering and Computer Science
Portland State University
tzewen@cecs.pdx.edu

Abstract

In this paper, a Boolean expression simplifier is presented. This simplifier takes a Boolean expression in prefix-form as an input and simplifies sub-expressions according to the definitions of Boolean operation. This implementation supports three Boolean operators: AND, OR, and NOT. Each operator except NOT can have more than two operands. An operand can be either a symbol or another Boolean expression. The implemented rules are: universal closure, existential closure, Boolean equivalence, and absorbing.

1 Introduction

A Boolean expression may contain several sub-expressions that introduce constants. These sub-expressions cause runtime overhead in evaluating the expression with values. Moreover, the expression itself may be fast evaluated under some special rules of Boolean operations. These evaluations can be performed on symbolic operands. In other words, some expressions may be eliminated without changing the meaning of the entire expression. This is a nothing more than a transformation of a Boolean expression. Thus, a simplifier is presented in this paper to perform transformations.

There are two motivations implementing a Boolean simplifier using LISP. One is LISP’s data structure – atoms and lists. These structures allow LISP to perform symbolic computation. The other motivation is the fact that LISP is a heavily recursive language. Recursion is an essential for operating on tree-like data structure and keeps the program concise. Furthermore, LISP programs are platform independent. The programs can be executed on any platform such as Windows, Linux, and Unix without any modification.

A Boolean expression can be represented in a fashion of tree as the following:

```
      and
     /   /
  or   or
 /     /
A    not
    /   /
   B    not
  /     /
A    B
```

The figure above shows the tree representation of Boolean expression "(a + /b) * (/a + b) * (b + 1) * (/0)" where 0 and 1 are Boolean constants.

The idea of this simplifier is simple. First, the simplifier simplifies the operands at current level and applies all rules according to the Boolean operator. Second, the simplifier visits every operand and performs previous step if an operand is another expression recursively. A level is also simplified in a recursive manner. The expression is considered to be simplest when the results of two simplifications are the same. This is achieved by a recursive simplification starter. Description of the algorithm is given in section 2.

The Boolean expression format must be in prefix-form of a LISP list. The first element in an expression list must be one of the following operators: and, or, and not followed by Boolean operands. The number of operands in an expression list is also being checked in the simplifier. As for operator and and or, there must be
more than two operands. As for operator not, there must be exact one operand. An operand can be either a symbol (LISP atom), or another Boolean expression list. Note that the two special atoms T and NIL in the LISP denote true and false respectively. Therefore, they are treated as Boolean constants in an expression. This format completely takes advantages of LISP internal data structures. The Boolean operator in an expression is also a LISP function. The operands thus are the function arguments. This allows the simplified expression to be used directly in conjunction with other functions which operate on Boolean expressions without any transformation or modification.

The rest of the paper is structured as the followings. In section 2, all Boolean transformation rules and general algorithm are described. Section 3 shows a set of unit tests and their results. Section 4 shows some running examples. Section 5 shows the source code of the simplifier.

2 Rules and Algorithms

2.1 Transformation Rules

The rules implemented in the Boolean simplifier are: 1) universal closure, 2) existential closure, 3) Boolean equivalence, and 4) absorbing. Universal closure is derived from the definition of Boolean AND operation. If there is an operand false in an expression, the entire expression is said to be false. Likewise, existential closure is derived from the definition of Boolean OR operation. If there is an operand true in an expression, the entire expression is said to be true. Absorbing means the redundant operands in a Boolean expression can be safely removed without any effect. Lastly, Boolean equivalence says some constants can be safely removed in certain Boolean operations. For example, constant 0's has no effect on Boolean OR operations. Likewise, constant 1's has no effect on Boolean AND operations. In addition to these rules, the simplifier also simplifies on Boolean NOT operations by attempting removing all inversions. For instance, a double inversion is transformed to no inversion. An inversion of constant 1 is transformed to 0 and vice versa. All rules are summarized by the following figures.

![Operator NOT elimination](image1)

![Boolean equivalence](image2)

![Absorbing](image3)
2.2 Algorithm

In general, a Boolean expression tree can be easily transformed recursively. The idea of this simplifier is simple. A simplification is done by applying the rules described in the previous section accordingly at current level and then traversing or visiting all child nodes recursively. However, this is only one-time traversing which may not result in the optimal or simplest form. Hence, there is a “starter” needed. The starter takes two expressions: 1) preceding simplified expression, 2) the expression to be simplified. The starter will invoke the simplification procedure once and compare the result with preceding result. If two results are identical, the expression is said to be simplest and the recursion stops. In the case that the results are not identical, the starter will be called recursively until the expression is the simplest. The following table summarizes all routines used by the Boolean simplifier.

<table>
<thead>
<tr>
<th>Function Symbol</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>simplify</td>
<td>This is the main routine of the simplifier. It takes a Boolean expression as on input and simplifies it. The output of this function is the simplest Boolean expression. It may be simply a constant.</td>
</tr>
<tr>
<td>simplify-expr</td>
<td>This is the starter routine which invokes one-time simplification procedure recursively until the expression is optimal or simplest. The function takes two expressions as inputs and returns a simplified Boolean expression. This routine is called by simplify-expr, simplify-operand, or simplify routine.</td>
</tr>
<tr>
<td>simplify-boolean-expr</td>
<td>This is the dispatcher routine which applies a Boolean expression to certain rules according to the Boolean operator. The function takes an expression as an input and returns a simplified Boolean expression. The routine is called by simplify-expr routine.</td>
</tr>
<tr>
<td>simplify-and-expr</td>
<td>Simplifies a Boolean AND expression. All rules related to the Boolean AND operation are being applied to the expression within this routine. The routine takes a list of AND-operands and returns either a simplified AND expression or a Boolean constant. The routine is called by simplify-boolean-exp routine.</td>
</tr>
<tr>
<td>simplify-or-expr</td>
<td>Simplifies a Boolean OR expression. All rules related to the Boolean OR operation are being applied to the expression within this routine. The routine takes an list of OR-operands and returns either a simplified OR expression or a Boolean constant. The routine is called by simplify-boolean-exp routine.</td>
</tr>
<tr>
<td>simplify-not-expr</td>
<td>Simplifies a Boolean NOT expression. All rules related to the Boolean NOT operation are being applied to the expression within this routine. The routine takes a NOT-operand and returns either a simplified NOT expression or a Boolean constant. The routine is called by simplify-boolean-exp routine.</td>
</tr>
</tbody>
</table>
simplify-operand

Traverses or visits all child nodes or operands in a Boolean expression recursively. This routine takes a list of operands as an input and returns a list of simplified operands as an output. The routine is called by simplify-and-exp, simplify-and-exp, simplify-and-exp, or simplify-operand routine.

The following figure shows program control flow graph of the Boolean simplifier among routines.

3 Unit Tests

3.1 simplify-and-expr

This routine should apply the AND rules to a list of AND-operands and return either a simplified and-expression or a Boolean constant.

> (simplify-and-expr '(a b))
(AND A B)
> (simplify-and-expr '(a b c))
(AND A B C)
> (simplify-and-expr '(a b c a))
(AND B C A)
> (simplify-and-expr '(a b c a T))
(AND B C A)
> (simplify-and-expr '(a b c a NIL))
NIL
> (simplify-and-expr '(a b c a T NIL))
NIL

3.2 simplify-or-expr

This routine should apply the OR rules to a list of OR-operands and return either a simplified or-expression or a Boolean constant.

> (simplify-or-expr '(a b))
> (simplify-or-expr `(a b c))
(OR A B C)
> (simplify-or-expr `(a b c a))
(OR B C A)
> (simplify-or-expr `(a b c a NIL))
(OR B C A)
> (simplify-or-expr `(a b c a T))
T
> (simplify-or-expr `(a b c a T NIL))
T

3.3 simplify-not-expr

This routine should apply the NOT rules to a list of NOT-operands and return either a simplified not-expression or a Boolean constant.

> (simplify-not-expr `(T))
NIL
> (simplify-not-expr `(NIL))
T
> (simplify-not-expr `(a))
(NOT A)
> (simplify-not-expr `((and a b)))
(NOT (AND A B))
> (simplify-not-expr `((not NIL)))
NIL
> (simplify-not-expr `((not T)))
T
> (simplify-not-expr `((not a)))
A
> (simplify-not-expr `((not (and a b))))
(AND A B)

3.4 simplify-operand

This routine should traverse all operands in a given operand list and apply the rules recursively to all operands accordingly.

> (simplify-operand `(a b c d))
(A B C D)
> (simplify-operand `(a b c d T))
(A B C D T)
> (simplify-operand `(a b c d NIL))
(A B C D NIL)
> (simplify-operand `(a b c d (not T)))
(A B C D (NOT T))
> (simplify-operand `(a b c d (not A)))
(A B C D (NOT A))
> (simplify-operand `(a b c d (and A B)))
(A B C D (AND A B))
> (simplify-operand `(a b c d (and a b) (OR a b)))
(A B C D (AND A B) (OR A B))
> (simplify-operand `(a b c d (and a b NIL) (OR a b)))
(A B C D NIL (OR A B))
> (simplify-operand `(a b c d (and a b NIL) (OR a b T)))
(A B C D T)

3.5 simplify-boolean-expr

This routine should correctly apply rules to the expression according to different operators.

> (simplify-boolean-expr `(and a b))
(AND A B)
> (simplify-boolean-expr `(and a b c))
(AND A B C)
> (simplify-boolean-expr `(and a b c a))
(AND B C A)
> (simplify-boolean-expr `(and a b c a T))
  (AND B C A)
> (simplify-boolean-expr `(and a b c a NIL))
  NIL
> (simplify-boolean-expr `(or a b))
  (OR A B)
> (simplify-boolean-expr `(or a b c))
  (OR A B C)
> (simplify-boolean-expr `(or a b c a))
  (OR B C A)
> (simplify-boolean-expr `(or a b c a T))
  T
> (simplify-boolean-expr `(or a b c a NIL))
  (OR B C A)
> (simplify-boolean-expr `(not T))
  NIL
> (simplify-boolean-expr `(not NIL))
  T
> (simplify-boolean-expr `(not a))
  (NOT A)
> (simplify-boolean-expr `(not (and a b)))
  (NOT (AND A B))
> (simplify-boolean-expr `(not (not NIL)))
  NIL
> (simplify-boolean-expr `(not (not T)))
  T
> (simplify-boolean-expr `(not (not (and a b))))
  (AND A B)
> (simplify-boolean-expr `(and a b c (or a b c a) a (not NIL) (and a b c a NIL))
  (AND B C (OR B C A) A T NIL)

3.6 simplify-expr

This routine should be able to return the simplest Boolean expression.

> (simplify-expr NIL `(not (not (and a b)))))

1. Trace: (SIMPLIFY-EXPR 'NIL '(NOT (NOT (AND A B)))))
2. Trace: (SIMPLIFY-EXPR '(NOT (NOT (AND A B))) 'AND A B))
3. Trace: SIMPLIFY-EXPR ==>(AND A B)
2. Trace: SIMPLIFY-EXPR ==>(AND A B)
1. Trace: SIMPLIFY-EXPR ==>(AND A B)
   (AND A B)
   > (simplify-expr NIL `(and a b c (or a b c a) a (not NIL) (and a b c a NIL))
   (AND B C (OR B C A) A T NIL)

1. Trace: (SIMPLIFY-EXPR 'NIL '(AND A B C (OR A B C A) A (NOT NIL) (AND A B C A NIL)))))
2. Trace: (SIMPLIFY-EXPR 'NIL '(OR A B C A))
3. Trace: (SIMPLIFY-EXPR '(OR B C A) '(OR B C A))
3. Trace: SIMPLIFY-EXPR ==>(OR B C A)
2. Trace: SIMPLIFY-EXPR ==>(OR B C A)
2. Trace: (SIMPLIFY-EXPR 'NIL '(NOT NIL))
3. Trace: (SIMPLIFY-EXPR '(NOT NIL) 'T)
3. Trace: SIMPLIFY-EXPR => T
2. Trace: SIMPLIFY-EXPR => T
2. Trace: (SIMPLIFY-EXPR 'NIL 'AND A B C A NIL)
3. Trace: (SIMPLIFY-EXPR '(AND A B C A NIL) 'NIL)
3. Trace: SIMPLIFY-EXPR => NIL
2. Trace: SIMPLIFY-EXPR => NIL
2. Trace: (SIMPLIFY-EXPR '(AND A B C (OR A B C A) A (NOT NIL) (AND A B C A NIL)) '(AND B C (OR B C A) A T NIL))
3. Trace: (SIMPLIFY-EXPR '(AND B C (OR B C A) A T NIL) 'NIL)
3. Trace: SIMPLIFY-EXPR => NIL
2. Trace: SIMPLIFY-EXPR => NIL
1. Trace: SIMPLIFY-EXPR ==> NIL
   NIL

4 Execution Sample

> (simplify `(a b))
*** - (A B) is not a valid boolean expression or has an unknown operator.
> (simplify `(or a b c d e f a (and b (or b (not (and a b)) a) T) g)))
*** - (NOT (AND A B) A T) is not a valid boolean expression or has an unknown operator.
> (simplify `(or a b c d e f a (and b (or b nil) a T) g h nil))
(OR B C D E F A (AND B A) G H)
> (simplify `(or a b c d e f a (and b (or b (not (and a b)) a)) g))
T
> (simplify `(or a b c d e f a (and b (or b (not (and a b)) a)) g))
(OR B C D E F A (AND B (OR B (NOT (AND A B)) A)) G)

5 Source code

;/*
 * A Boolean Simplifier by Tzewen Wang.
 * *
 * See the paper for detail description and discussion.
 */

(defun simplify (EXPR)
  (simplify-expr NIL EXPR))

(defun simplify-expr (EXPR1 EXPR2)
  (cond ;;; Terminates if (EXPR1 == EXPR2 || CONSTANT).
    ((or (atom EXPR2) (equal EXPR1 EXPR2)) EXPR2)
    ;;; Continues simplifying, otherwise.
    (T (simplify-expr EXPR2 (simplify-boolean-expr EXPR2)))))

(defun simplify-boolean-expr (EXPR)
  (cond ;;; Simplifies AND expression.
    ((and (equal (first EXPR) `and) (>= (length EXPR) 3)) (simplify-and-expr (rest EXPR)))
    ;;; Simplifies OR expression.
    ((and (equal (first EXPR) `or) (>= (length EXPR) 3)) (simplify-or-expr (rest EXPR)))
    ;;; Simplifies NOT expression.
    ((and (equal (first EXPR) `not) (= (length EXPR) 2)) (simplify-not-expr (rest EXPR)))
    ;;; More rules can be implemented here:
    ;;; (....)
    ;;; Anything else is considered to be ill-formed.
    (T (error "~S is not a valid boolean expression or has an unknown operator." EXPR))))

(defun simplify-and-expr (EXPR)
  ;; Removes all 1's constants and duplicates (RULE: equivalence and absorbing).
  (let ((SIMPLIFIED_EXPR (remove `T (remove-duplicates EXPR))))
    (cond ;;; Returns with 1's constant.
      ((null SIMPLIFIED_EXPR) `T)
    ;;; Returns with 0's constant (RULE: universal closure).
      ((member `NIL SIMPLIFIED_EXPR) `NIL)
    ;;; Returns with the only operand.
      (T (first SIMPLIFIED_EXPR)))))
More rules can be implemented here:

Simplifies every operand recursively.
(T (cons `and (simplify-operand SIMPLIFIED_EXPR)))

(defun simplify-or-expr (EXPR)
  ;; Removes all 0's constants and duplicates (RULE: equivalence and absorbing).
  (let ((SIMPLIFIED_EXPR (remove `NIL (remove-duplicates EXPR))))
    (cond ;; Returns with 0's constant.
      ((null SIMPLIFIED_EXPR) `NIL)
      ;; Returns with 1's constant (RULE: existential closure).
      ((member `T SIMPLIFIED_EXPR) `T)
      ;; Returns with the only operand.
      ((null (second SIMPLIFIED_EXPR)) (first SIMPLIFIED_EXPR))
      ;; More rules can be implemented here:
      ;; (...)
    ;; Simplifies every operand recursively.
    (T (cons `or (simplify-operand SIMPLIFIED_EXPR))))))

(defun simplify-not-expr (EXPR)
  (cond ;; Simplifies constants.
    ((equal (first EXPR) `NIL) `T)
    ((equal (first EXPR) `T) `NIL)
    ;; Simplifies double inversions (RULE).
    ((and (listp (first EXPR)) (equal (first (first EXPR)) `not)) (first (rest (first EXPR))))
    ;; More rules can be implemented here:
    ;; (...)
    ;; Does nothing, otherwise.
    (T (cons `not (simplify-operand EXPR))))))

(defun simplify-operand (OPERAND_LIST)
  (cond ;; Terminates with NIL.
    ((null OPERAND_LIST) NIL)
    ;; Simplifies the rest of operands.
    ((atom (first OPERAND_LIST)) (cons (first OPERAND_LIST) (simplify-operand (rest OPERAND_LIST))))
    ;; Simplifies a sub-expression and the rest of operands.
    (T (cons (simplify-expr NIL (first OPERAND_LIST)) (simplify-operand (rest OPERAND_LIST))))))

Notes

This Boolean Simplifier is implemented in ANSI Common LISP language and tested on GNU CLisp implementation [1]. The CLisp implementation is installed on the Unix machines in College of Engineering and Computer Science, Portland State University. To use this simplifier, simply invoke CLisp environment by “clisp” under a Unix box and load with the source code as file.

References