optimization of if-then-else chains

<table>
<thead>
<tr>
<th>original C code</th>
<th>optimized C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(!a &amp;&amp; !b) h(); else if(!a) g(); else f();</td>
<td>if(a) f(); else if(b) g(); else h();</td>
</tr>
<tr>
<td>=&gt;</td>
<td>=&gt;</td>
</tr>
<tr>
<td>if(!a) { if(a) f(); if(!b) h(); else g(); } else f();</td>
<td>if(!a) { if(a) f(); if(!b) h(); else g(); }</td>
</tr>
</tbody>
</table>

How to check that these two versions are equivalent?

1. represent procedures as independent boolean variables

   original := optimized :=

   if ¬a ∧ ¬b then h else if ¬a then g else f

   if a then f else if ¬b then g else if b then g else h

2. compile if-then-else chains into boolean formulae

   compile(if x then y else z) ≡ (x ∨ ¬z)

3. check equivalence of boolean formulae

   compile(original) ⇔ compile(optimized)

How to Check (In)Equivalence?

Reformulate it as a satisfiability (SAT) problem:

Is there an assignment to a, b, f, g, h, which results in different evaluations of original and optimized?

or equivalently:

Is the boolean formula compile(original) ↔ compile(optimized) satisfiable?

such an assignment would provide an easy to understand counterexample

Note: by concentrating on counterexamples we moved from Co-NP to NP (this is just a theoretical note and not really important for applications)
SAT Example: Circuit Equivalence Checking

SAT (Satisfiability) the classical NP complete Problem:

Given a propositional formula $f$ over $n$ propositional variables $V = \{x, y, \ldots\}$.

Is there an assignment $\sigma : V \rightarrow \{0, 1\}$ with $\sigma(f) = 1$?

SAT belongs to NP

There is a non-deterministic Touring-machine deciding SAT in polynomial time:

guess the assignment $\sigma$ (linear in $n$), calculate $\sigma(f)$ (linear in $|f|$)

Note: on a real (deterministic) computer this would still require $2^n$ time

SAT is complete for NP (see complexity / theory class)

Implications for us:
general SAT algorithms are probably exponential in time (unless NP = P)

Negation Normal Form

Assumption: we only have conjunction, disjunction and negation as operators.

a formula is in Negation Normal Form (NNF) if negations only occur in front of variables

if negations only occur in front of variables

in this case the result can be exponentially larger (see parity example later).

Conjunctive Normal Form

Definition

a formula in Conjunctive Normal Form (CNF) is a conjunction of clauses

$$C_1 \land C_2 \land \ldots \land C_n$$

each clause $C$ is a disjunction of literals

$$C = L_1 \lor \ldots \lor L_m$$

and each literal is either a plain variable $x$ or a negated variable $\overline{x}$.

Example

$$(a \lor b \lor c) \land (\overline{a} \lor \overline{b}) \land (\overline{a} \lor \overline{c})$$

Note 1: two notions for negation: in $\overline{x}$ and $\neg x$ as in $\neg x$ for denoting negation.

Note 2: the original SAT problem is actually formulated for CNF

Note 3: SAT solvers mostly also expect CNF as input
NNF Algorithm

```
Formula
formula2nnf (Formula f, Bool sign)
{
    if (is_variable (f))
        return sign ? new_not_node (f) : f;
    if (op (f) == AND || op (f) == OR)
    {
        l = formula2nnf (left_child (f), sign);
        r = formula2nnf (right_child (f), sign);
        flipped_op = (op (f) == AND) ? OR : AND;
        return new_node (sign ? flipped_op : op (f), l, r);
    }
    else
    {
        assert (op (f) == NOT);
        return formula2nnf (child (f), !sign);
    }
}
```

Revision: 1.12

Simple Translation of Formula into CNF

```
Formula
formula2cnf_aux (Formula f)
{
    if (is_cnf (f))
        return f;
    if (op (f) == AND)
    {
        l = formula2cnf_aux (left_child (f));
        r = formula2cnf_aux (right_child (f));
        return new_node (AND, l, r);
    }
    else
    {
        assert (op (f) == OR);
        l = formula2cnf_aux (left_child (f));
        r = formula2cnf_aux (right_child (f));
        return merge_cnf (l, r);
    }
}
```

Revision: 1.12

Merging two CNFs

```
Formula
formula2cnf (Formula f)
{
    return formula2cnf_aux (formula2nnf (f, 0));
}
```

Revision: 1.12

Why are Sharing / Circuits / DAGs important?

```
DAG may be exponentially more succinct than expanded Tree

Examples: adder circuit, parity, mutual exclusion
```

Revision: 1.12
Parity Example

Boole parity (Boole a, Boole b, Boole c, Boole d, Boole e, Boole f, Boole g, Boole h, Boole i, Boole j)
{
    tmp0 = b ? !a : a;
    tmp1 = c ? !tmp0 : tmp0;
    tmp2 = d ? !tmp1 : tmp1;
    tmp3 = e ? !tmp2 : tmp2;
    tmp4 = f ? !tmp3 : tmp3;
    tmp5 = g ? !tmp4 : tmp4;
    tmp6 = h ? !tmp5 : tmp5;
    tmp7 = i ? !tmp6 : tmp6;
    return j ? !tmp7 : tmp7;
}

Eliminate the tmp... variables through substitution.

What is the size of the DAG vs the Tree representation?

How to detect Sharing

- through caching of results in algorithms operating on formulas (examples: substitution algorithm, generation of NNF for non-monotonic ops)
- when modeling a system: variables are introduced for subformulae (then these variables are used multiple times in the toplevel formula)
- structural hashing: detects structural identical subformulae (see Signed And Graphs later)
- equivalence extraction: eg. BDD sweeping, Stålmarcks Method (we will look at both techniques in more detail later)

Example of Tseitin Transformation: Circuit to CNF

CNF

Algorithmic Description of Tseitin Transformation

1. for each non input circuit signal s generate a new variable x_s
2. for each gate produce complete input / output constraints as clauses
3. collect all constraints in a big conjunction

the transformation is satisfiability equivalent: the result is satisfiable iff and only the original formula is satisfiable
not equivalent in the classical sense to original formula: it has new variables
extract satisfying assignment for original formula, from one of the result (just project satisfying assignment onto the original variables)
Tseitin Transformation: Input / Output Constraints

Negation: $x \leftrightarrow \overline{y}$
$\Leftrightarrow (x \rightarrow \overline{y}) \land (\overline{y} \rightarrow x)$
$\Leftrightarrow (\overline{x} v \overline{y}) \land (y v x)$

Disjunction: $x \leftrightarrow (y v z)$
$\Leftrightarrow (y \rightarrow x) \land (z \rightarrow x) \land (x \rightarrow (y v z))$
$\Leftrightarrow (y v x) \land (z v x) \land (y v z v x)$

Conjunction: $x \leftrightarrow (y \land z)$
$\Leftrightarrow (x \rightarrow y) \land (x \rightarrow z) \land ((y \land z) \rightarrow x)$
$\Leftrightarrow (\overline{x} v y) \land (\overline{x} v z) \land (y v z v x)$

Equivalence: $x \leftrightarrow (y \leftrightarrow z)$
$\Leftrightarrow (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x)$
$\Leftrightarrow (x \rightarrow ((y \rightarrow z) \land (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x)$
$\Leftrightarrow (y v x) \land (z v x) \land (y v z v x)$

Optimizations for Tseitin Transformation

- goal is smaller CNF (less variables, less clauses)
- extract multi argument operands (removes variables for intermediate nodes)
- half of AND, OR node constraints may be removed for unnegated nodes
- a node occurs negated if it has an ancestor which is a negation
- half of the constraints determine parent assignment from child assignment
- those are unnecessary if node is not used negated
- those have to be carefully applied to DAG structure
  (compare with the implementation of the BMC tool from CMU)