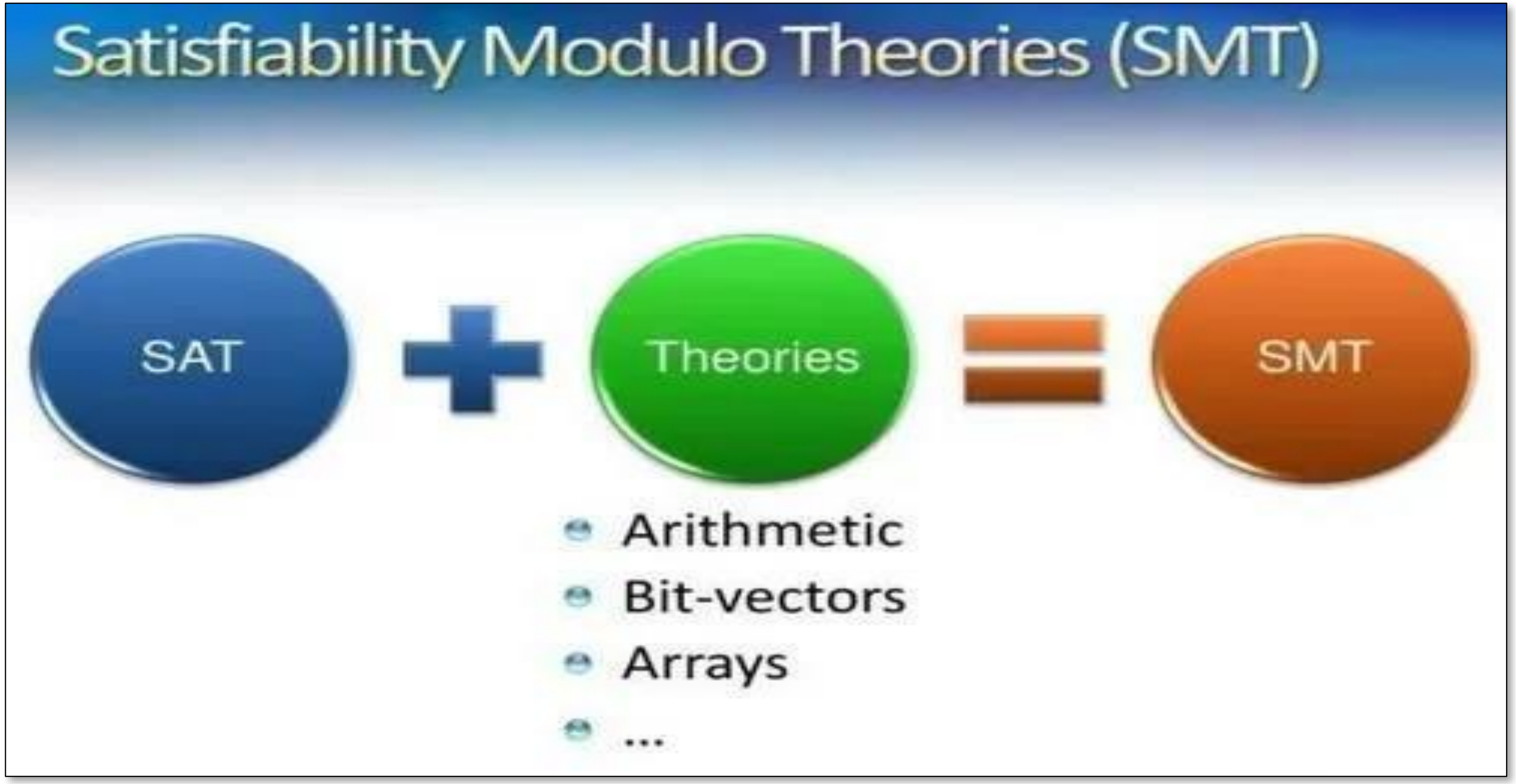
SMT Solving: DPLL(T) and Eager Encoding

Shaowei Cai

Institute of Software, Chinese Academy of Sciences

From Propositional to Quantifier-Free Theories



From Propositional to Quantifier-Free Theories

Example:
$\phi := (x_1 - x_2 \leq 13 \vee x_2 \neq x_3) \wedge (x_2 = x_3 \rightarrow x_4 > x_5) \wedge A \wedge \neg B$

Propositional Skeleton $\mathrm{PS}_\Phi = (b_1 \vee \neg b_2) \wedge (b_2 \rightarrow b_3) \wedge A \wedge \neg B$

$b_1 : x_1 - x_2 \leq 13$

$b_2 : x_2 = x_3$

$b_3 : x_4 > x_5$

From Propositional to Quantifier-Free Theories

Example:

- $\mathrm{a} = \mathrm{b} + 2 \wedge A = write(B, a + 1,4) \wedge (read(A, b + 3) = 2 \vee f(a - 1) \neq f(b + 1))$

- Propositional Skeleton $\mathrm{PS}_{\Phi} = y_1 \wedge y_2 \wedge (y_3 \vee y_4)$

- y_1: $\mathrm{a} = \mathrm{b} + 2$
- y_2: $A = write(B, a + 1,4)$
- y_3: $read(A, b + 3) = 2$
- y_4: $f(a - 1) \neq f(b + 1)$

Interpretation

Example

- F : x + y > z → y > z − x

- We construct a “standard” interpretation I
- The domain is the integers, $\mathbb{Z}$: $D_I = \mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
- α_I: $\{+ \longmapsto +_{\mathbb{Z}}, - \longmapsto -_{\mathbb{Z}}, > \longmapsto >_{\mathbb{Z}}, x \longmapsto 13, y \longmapsto 42, x \longmapsto 1\}$

T-satisfiability

- Given a FOL formula F and interpretation $I: (D_I, \alpha_I)$, we want to compute if F evaluates to true under interpretation I, I ⊨ F, or if F evaluates to false under interpretation I, I ⊭F.
  - I satisfies F: I ⊨ F

- T – interpretation: an interpretation satisfying I ⊨ A for every A $\in \mathcal{A}$.

- A Σ-formula F is satisfiable in T , or T -satisfiable, if there is a T-interpretation I that satisfies F.

Approaches for Solving Single SMT Theory

Two main approaches for SMT

- Lazy Approach

  Integrate a theory solver with a CDCL solver for SAT

- Eager Approach

  Encode the SMT formula to a equ-satisfiable SAT formula

Normalizing T-atoms

- *Drop dual operators*: $(x_1 < x_2), (x_1 \geq x_2) \Longrightarrow \neg(x_1 \geq x_2), (x_1 \geq x_2)$.
- *Exploit associativity*: $(x_1 + (x_2 + x_3) = 1), ((x_1 + x_2) + x_3) = 1) \Longrightarrow (x_1 + x_2 + x_3 = 1)$.
- *Sort*: $(x_1 + x_2 - x_3 \leq 1), (x_2 + x_1 - 1 \leq x_3) \Longrightarrow (x_1 + x_2 - x_3 \leq 1))$.
- *Exploit $\mathcal{T}$-specific properties*: $(x_1 \leq 3), (x_1 < 4) \Longrightarrow (x_1 \leq 3)$ if x_1 represents an integer.

Static Learning

If so, the clauses obtained by negating the literals in such sets (e.g., ¬(x = 0) ∨ ¬(x = 1)) can be added to the formula before the search starts

- *incompatible values* (e.g., $\{x = 0, x = 1\}$),
- *congruence constraints* (e.g., $\{(x_1 = y_1), (x_2 = y_2), f(x_1, x_2) \neq f(y_1, y_2)\}$),
- *transitivity constraints* (e.g., $\{(x - y \leq 2), (y - z \leq 4), \neg(x - z \leq 7)\}$),
- *equivalence constraints* (e.g., $\{(x = y), (2x - 3z \leq 3), \neg(2y - 3z \leq 3)\}$).

Equality logic with Uninterpreted Functions (EUF)

An equality logic formula with uninterpreted functions *and* uninterpreted predicates[2] *is defined by the following grammar:*

$$
\begin{aligned}
\mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \neg \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &: \mathit{term} = \mathit{term} \mid \mathit{predicate\text{-}symbol}\ (\mathit{list\ of\ terms}) \\
\mathit{term} &: \mathit{identifier} \mid \mathit{function\text{-}symbol}\ (\mathit{list\ of\ terms})
\end{aligned}
$$

Using Uninterpreted Functions

$$\models \varphi^{\mathrm{UF}} \implies \models \varphi$$

- Replacing functions with uninterpreted functions in a given formula is a common technique for making it easier to reason about (e.g., to prove its validity).

- At the same time, this process makes the formula weaker, which means that it can make a valid formula invalid.

The only thing uninterpreted functions need to satisfy:

- Functional consistency: Instances of the same function return the same value if given equal arguments.

Using Uninterpreted Functions

```
int power3(int in)
{
  int i, out_a;
  out_a = in;
  for (i = 0; i < 2; i++)
    out_a = out_a * in;
  return out_a;
}
```

(a)

```
int power3_new(int in)
{
  int out_b;

  out_b = (in * in) * in;

  return out_b;
}
```

(b)

$$
\begin{aligned}
out0\_a &= in0_a && \wedge \\
out1\_a &= out0\_a * in0_a && \wedge \\
out2\_a &= out1\_a * in0_a
\end{aligned}
$$

(φ_a)

$$out0\_b = (in0_b * in0_b) * in0_b;$$

(φ_b)

To show that these two piece of codes are actually equivalent, we only need to prove the validity of

$$in0\_a = in0\_b \wedge \varphi_a \wedge \varphi_b \implies out2\_a = out0\_b$$

Using Uninterpreted Functions

$$\begin{aligned} out0\_a &= in0_a && \wedge \\ out1\_a &= out0\_a * in0_a && \wedge \\ out2\_a &= out1\_a * in0_a \end{aligned}$$

(φ_a)

$$out0\_b = (in0_b * in0_b) * in0_b;$$

(φ_b)

$$\begin{aligned} out0\_a &= in0\_a && \wedge \\ out1\_a &= G(out0\_a, in0\_a) && \wedge \\ out2\_a &= G(out1\_a, in0\_a) \end{aligned}$$

$(\varphi_a^{\mathrm{UF}})$

$$out0\_b = G(G(in0\_b, in0\_b), in0\_b)$$

$(\varphi_b^{\mathrm{UF}})$

Using Uninterpreted Functions

```
int mul3(struct list *in)
{
 int i, out_a;
 struct list *a;
 a = in;
 out_a = in -> data;
 for (i = 0; i < 2; i++) {
   a = a -> n;
   out_a= out_a * a -> data;
 }
 return out_a;
}
```

(a)

```
int mul3_new(struct list *in)
{
  int out_b;

  out_b =
    in -> data *
    in -> n -> data *
    in -> n -> n -> data;

  return out_b;
}
```

(b)

```
struct list {
    struct list *n; // pointer to next element
    int data;
};
```

$$
\begin{array}{ll}
a0\_a = in0\_a & \wedge \\
out0\_a = list\_data(in0\_a) & \wedge \\
a1\_a = list\_n(a0\_a) & \wedge \\
out1\_a = G(out0\_a, list\_data(a1\_a)) & \wedge \\
a2\_a = list\_n(a1\_a) & \wedge \\
out2\_a = G(out1\_a, list\_data(a2\_a)) &
\end{array}
$$

$(\varphi_a^{\mathrm{UF}})$

$$
\begin{array}{l}
out0\_b = G(G(list\_data(in0\_b), \\
list\_data(list\_n(in0\_b))), \\
list\_data(list\_n(list\_n(in0\_b))))
\end{array}
$$

$(\varphi_b^{\mathrm{UF}})$

Congruence Closure

$$\varphi^{\mathrm{UF}} := \ x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_5 \wedge x_5 \neq x_1 \wedge F(x_1) \neq F(x_3)\ .$$

Initially, the equivalence classes are

$$\{x_1, x_2\}, \{x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\}\ .$$

Step 1(b) of Algorithm 4.3.1 merges the first two classes:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\}\ .$$

The next step also merges the classes containing $F(x_1)$ and $F(x_3)$, x_1 and x_3 are in the same class:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1), F(x_3)\}\ .$$

In step 2, we note that $F(x_1) \neq F(x_3)$ is a predicate in φ^{UF}, but that $F(x_1)$ and $F(x_3)$ are in the same class. Hence, φ^{UF} is unsatisfiable. ∎

Can be implemented with a union-find data structure, which results in a time complexity of O(n log n)

Congruence Closure

a=b, f(a,f(b,g(a)))=d, g(b)=c, f(a,c)=c, f(a,c)≠d

a=b

f(a,f(b,g(a)))=d

g(b)=c=f(a,c)=g(a)

f(b,g(a))

…merge congruent terms

since a=b and c=g(a), we know f(a,c)=f(b,g(a))

Splitting on demand

- solving problems with general Boolean structure over EUF using the DPLL(T) framework ?

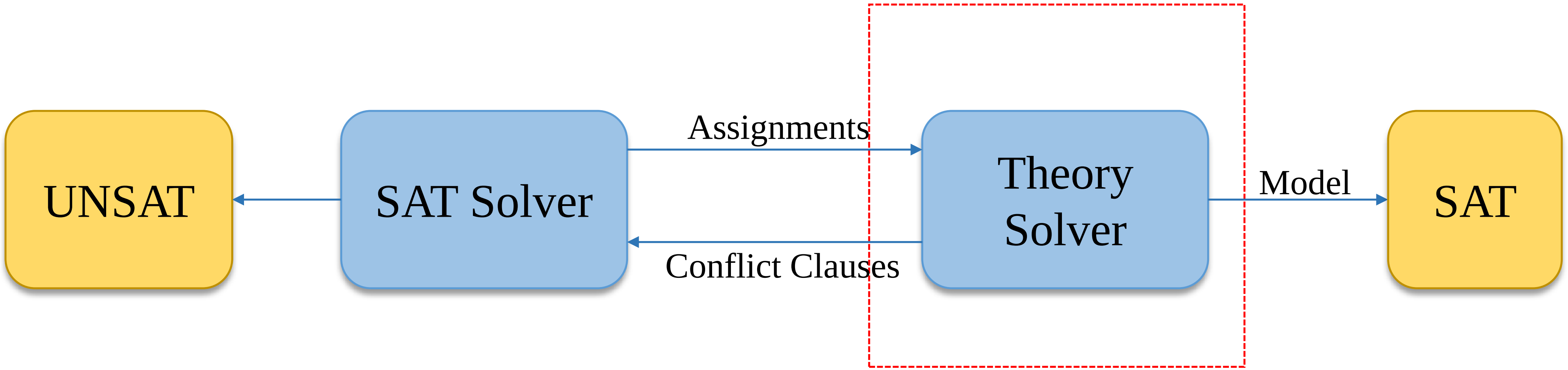
- it is desirable to allow a theory solver T -solver to demand that the DPLL engine do additional case splits before determining the T -consistency of a partial assignment.

Example 26.5.5. In the theory $\mathcal{T}_{\mathcal{A}}$ of arrays introduced in §26.2.2, consider the following set of literals: $\mathit{read}(\mathit{write}(A, i, v), j) = x, \mathit{read}(A, j) = y, x \neq v, x \neq y$. To see that this set is unsatisfiable, notice that if $i = j$, then $x = v$ because the value read should match the value written in the first equation. On the other hand, if $i \neq j$, then $x = \mathit{read}(A, j)$ and thus $x = y$. Deciding the $\mathcal{T}_{\mathcal{A}}$-consistency of larger sets of literals may require a significant amount of such reasoning by cases.

Outline

- SMT Basis
- Lazy Approach --- DPLL(T)
- Eager Approach --- Bit Blasting

DPLL(T)

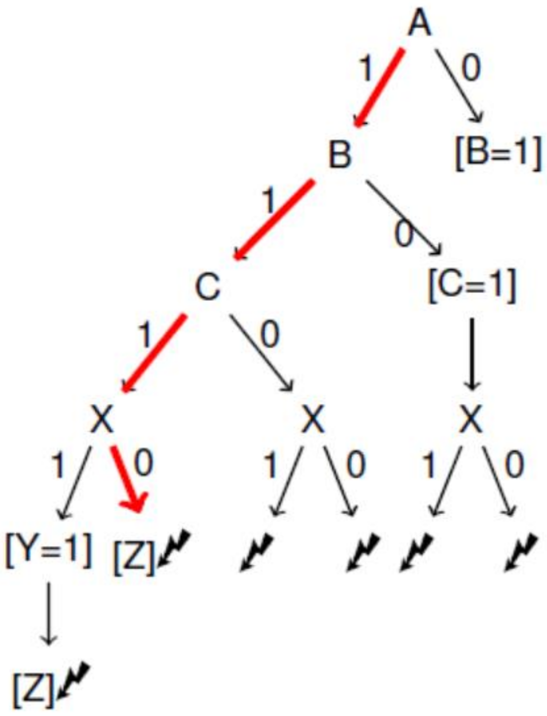


- The method is commonly referred to as DPLL(T), emphasizing that it is parameterized by a theory T.
- The fact that it is called DPLL(T) and not CDCL(T) is attributed to historical reasons only: it is based on modern CDCL solvers”
- ---”Decision Procedures” Daniel Kroening, Ofer Strichman

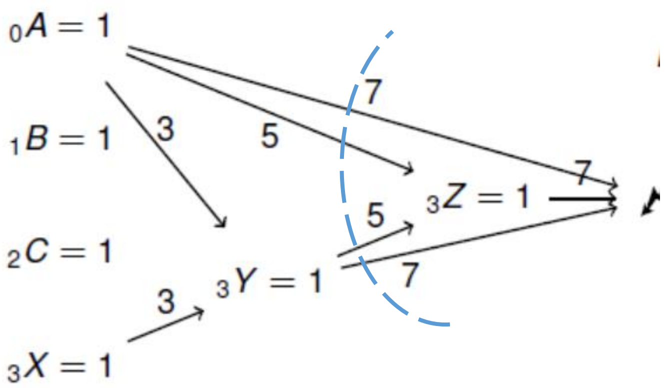
CDCL Review

$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array}$$

$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \\ 8.\ \{\neg A, \neg Y\} \end{array}$$



Chronological Backtracking

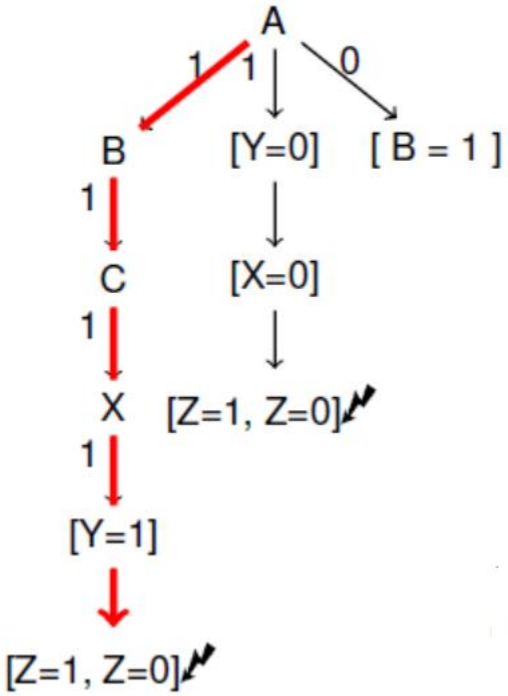


Conflict Analysis

Conflicting Clause: $\{\neg A, \neg Y, \neg Z\}$

Learnt Clause(1UIP): $\{\neg A, \neg Y\}$

Clause Learning



Non-Chronological Backtracking

Propositional Skeleton

<u>Abstract the skeleton</u>:
Given atom a, we associate with it a unique Boolean variable e(a), which we call the Boolean encoder of this atom.

$$\varphi := \ x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z)\ .$$

The propositional skeleton of φ is

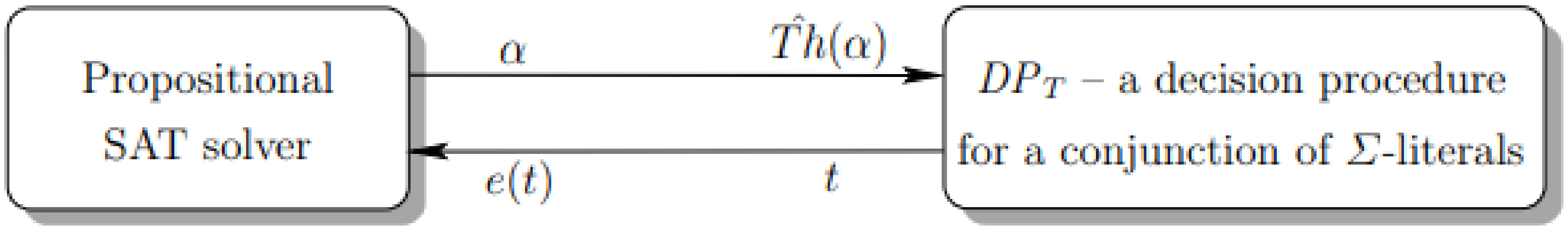
$$e(\varphi) := \ e(x = y) \wedge ((e(y = z) \wedge \neg e(x = z)) \vee e(x = z))\ .$$

Let $\mathcal{B}$ be a Boolean formula, initially set to $e(\varphi)$, i.e.,

$$\mathcal{B} := \ e(\varphi)\ .$$

$$\alpha := \ \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{TRUE},\ e(x = z) \mapsto \text{FALSE}\}\ .$$

DPLL(T)



Propositional
SAT solver
α
$\hat{Th}(\alpha)$
$e(t)$
t
DP_T – a decision procedure
for a conjunction of Σ-literals

A basic lazy approach

$$\varphi := x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z) .$$

The propositional skeleton of φ is

$$e(\varphi) := e(x = y) \wedge ((e(y = z) \wedge \neg e(x = z)) \vee e(x = z)) .$$

Let $\mathcal{B}$ be a Boolean formula, initially set to $e(\varphi)$, i.e.,

$$\mathcal{B} := e(\varphi) .$$

- Call SAT solver to solve $e(\varphi)$, find

$$\alpha := \{e(x = y) \mapsto \mathrm{TRUE},\ e(y = z) \mapsto \mathrm{TRUE},\ e(x = z) \mapsto \mathrm{FALSE}\} .$$

- →Call decision procedure DP_T to check the conjunction corresponding to α, denoted by $\widehat{Th}(\alpha)$, $\widehat{Th}(\alpha) \coloneqq x{=}y \wedge y = z \wedge \neg(x{=}z)$ → the result: $\widehat{Th}(\alpha)$ is unsat.

A basic lazy approach

$$\varphi := \; x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z)\ .$$

The propositional skeleton of φ is

$$e(\varphi) := \; e(x = y) \wedge ((e(y = z) \wedge \neg e(x = z)) \vee e(x = z))\ .$$

Let $\mathcal{B}$ be a Boolean formula, initially set to $e(\varphi)$, i.e.,

$$\mathcal{B} := \; e(\varphi)\ .$$

- $e(\neg\widehat{Th}(\alpha))$ is conjoined into B, the Boolean encoding of this tautology.
  - $e\left(\neg\widehat{Th}(\alpha)\right) := \neg e(x{=}y) \vee \neg e(y = z) \vee e(x{=}z)$ --- blocking clause(s)
  - This clause contradicts the current assignment, and hence blocks it from being repeated
  - In general, we denote by t the lemma returned by DP_T.

A basic lazy approach

$$\varphi := \ x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z) \ .$$

The propositional skeleton of φ is

$$e(\varphi) := \ e(x = y) \wedge ((e(y = z) \wedge \neg e(x = z)) \vee e(x = z)) \ .$$

Let $\mathcal{B}$ be a Boolean formula, initially set to $e(\varphi)$, i.e.,

$$\mathcal{B} := \ e(\varphi) \ .$$

- →After the blocking clause has been added, the SAT solver is invoked again and suggests another assignment
- →Then invoke DP_T again to check the conjunction of the literals corresponding to the new assignment.

A Basic Lazy Approach: Example

$$\Phi \coloneqq \mathrm{g}(a) = c \wedge (f(g(a)) \neq f(c) \vee g(a) = d) \wedge c \neq d$$

- $\mathrm{PS}_\Phi = y_1 \wedge (\neg y_2 \vee y_3) \wedge y_4)$

- $y_1 \colon \mathrm{g}(a) = c$
- $y_2 \colon f(g(a)) = f(c)$
- $y_3 \colon g(a) = d$
- $y_4 \colon c = d$

Send$\{1, \bar{2} \vee 3, \bar{4}\}$ to SAT
SAT solver returns model $\{1, \bar{2}, \bar{4}\}$
UF-solver find concretization of $\{1, \bar{2}, \bar{4}\}$ UNSAT
Send $\{1, \bar{2} \vee 3, \bar{4}, \neg(1 \wedge \bar{2} \wedge \bar{4})\}$ to SAT
Send $\{1, \bar{2} \vee 3, \bar{4}, \bar{1} \vee 2 \vee 4\}$ to SAT
SAT solver returns model $\{1,3, \bar{4}\}$
UF-solver find concretization of $\{1,3, \bar{4}\}$ UNSAT
Send $\{1, \bar{2} \vee 3, \bar{4}, \bar{1} \vee 2 \vee 4, \bar{1} \vee \bar{3} \vee 4\}$ to SAT
SAT solver returns UNSAT; Original formula is UNSAT in UF

Integration into CDCL

Algorithm LAZY-CDCL

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

function LAZY-CDCL
  ADDCLAUSES($cnf(e(\varphi))$);
  while (TRUE) **do**
    while (BCP() = "conflict") **do**
      backtrack-level := ANALYZE-CONFLICT();
      if *backtrack-level* < 0 **then return** "Unsatisfiable";
      else BackTrack(*backtrack-level*);
    if $\neg$DECIDE() **then** ▷ Full assignment
      $\langle t, res \rangle$:=DEDUCTION($\hat{Th}(\alpha)$); ▷ α is the assignment
      if res="Satisfiable" **then return** "Satisfiable";
      ADDCLAUSES($e(t)$);

Improving the Basic Lazy Approach

- Incremental SAT solving

Let B^{i} be the formula B in the i-th iteration of the loop in basic lazy algorithm. $B^{\text{i}+1}$ is strictly stronger than B^{i} for all $i \geq 1$, because blocking clauses are added but not removed between iterations.

It is not hard to see that this implies that any conflict clause that is learned while solving B^{i} can be reused when solving B^{j} for $i < j$.

This, in fact, is a special case of **incremental satisfiability**, which is supported by most modern SAT solvers.

Hence, invoking an incremental SAT solver can increase the efficiency of the algorithm.

Still not clever enough…

- Consider, for example, a formula ϕ that contains literals
$$x_1 \geq 10,\ \ x_1 < 0,$$
where x_1 is an integer variable.

- Assume that the CDCL procedure assigns $e(x_1 \geq 10) \mapsto$ true and $e(x_1 < 0) \mapsto$ true. Inevitably, any call to Deduction results in a contradiction between these two facts.

- However, Algorithm Lazy-CDCL does not call Deduction until a full satisfying assignment is found. // waste time to complete the assignment.

Theory Propagation

Theory Propagation

- Deduction is invoked after BCP stops.
- It finds T-implied literals and communicates them to the CDCL part of the solver in the form of a constraint t.

Example. Consider the two encoders $e(x_1 \geq 10)$ and $e(_1 < 0)$.

- After $e(x_1 \geq 10)$ has been set to true, Deduction detects that $\neg(x_1 < 0)$ is implied.

- In other words, t := $\neg(x_1 \geq 10) \vee \neg(x_1 < 0)$ is T-valid.
- The corresponding encoded (asserting) clause
  e(t) := $\neg e(x_1 \geq 10) \vee \neg e(x_1 < 0)$

- e(t) is added to B, which leads to an immediate implication of $\neg e(x_1 < 0)$, and possibly further implications.

The DPLL(T) Approach

Algorithm DPLL(T)

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
function DPLL(T)
    AddClauses(cnf(e(φ)));
    while (TRUE) do
        repeat
            while (BCP() = "conflict") do
                backtrack-level := Analyze-Conflict();
                if backtrack-level < 0 then return "Unsatisfiable";
                else BackTrack(backtrack-level);
            ⟨t, res⟩:=Deduction(T̂h(α));
            AddClauses(e(t));
        until t ≡ TRUE;
        if α is a full assignment then return "Satisfiable";
        Decide();
```

- When α is partial, Deduction checks
  - if there is a contradiction on the theory side,
  - and if not, it performs theory propagation.

not mandatory, only for efficiency

Performance, Performance…

- For performance, it is frequently better to run an approximation for finding contradictions.
  - This does not change the completeness of the algorithm, since a complete check is performed when α is full.

E.g. integer linear arithmetic:
Deciding the conjunctive fragment of this theory is NP-complete

- consider the real relaxation of the problem, which can be solved in polynomial time.
- Deduction sometimes misses a contradiction and hence not return a lemma

Performance, Performance…

- Exhaustive theory propagation refers to a procedure that finds and propagates all literals that are implied in T by $\widehat{Th}(\alpha)$.

- A simple generic way (called "plunging") to perform theory propagation

  Given an unassigned theory atom at_i , check whether $\widehat{Th}(\alpha)$ implies either at_i or $\neg at_i$.

  The set of unassigned atoms that are checked in this way depends on how exhaustive we want the theory propagation to be.

- In many cases a better strategy is to perform only cheap propagations
  - E.g. LIA: to search for simple-to-find implications, such as "if x > c holds, where x is a variable and c a constant, then any literal of the form x > d is implied if d < c"

Running A DPLL(LIA) Example

(x>y $\vee$ x>z) $\wedge$(x+1<y $\vee$ ¬x>y) $\wedge$(x>y $\vee$ z>y)

- DPLL(LIA) algorithm
  - Decide x>y → true
  - Propagate x+1<y →true
  - Invoke theory solver for LIA on: { x>y, x+1<y }

Context

x>y^d
x+1<y

Running A DPLL(LIA) Example

(x>y ∨ x>z) ∧(x+1<y ∨ ¬x>y) ∧(x>y ∨ z>y) ∧
(¬ x>y ∨ ¬ x+1<y)

Conflicting clause!
…backtrack on a decision

Context

$x>y^d$
x+1<y

- DPLL(LIA) algorithm
  - Decide x>y → true
  - Propagate x+1<y →true
  - Invoke theory solver for LIA on: { x>y,x+1<y}
    - x>y ∧ x+1<y is LIA-unsatisfiable,add(¬ x>y ∨ ¬ x+1<y)

Running A DPLL(LIA) Example

$(x>y \lor x>z) \land (x+1<y \lor \neg x>y) \land (x>y \lor z>y) \land$
$(\neg x>y \lor \neg x+1<y)$

- DPLL(LIA) algorithm
  - Backtrack : $x>y \rightarrow$ false
  - Propagate : $x>z \rightarrow$ true
  - Propagate : $z>y \rightarrow$ true
  - Invoke theory solver for LIA on: $\{\neg x>y, x>z, z>y\}$

Context

$\neg x>y$

$x>z$

$z>y$

Running A DPLL(LIA) Example

(x>y ∨ x>z) ∧(x+1<y ∨ ¬x>y) ∧(x>y∨ z>y) ∧
(¬ x>y ∨ ¬ x+1<y) ∧ (x>y ∨ ¬ x>z ∨ ¬ z>y)

- DPLL(LIA) algorithm
    - Backtrack : x>y → false
    - Propagate : x>z → true
    - Propagate : z>y → true
    - Invoke theory solver for LIA on: {¬ x>y, x>z, z>y }
        - ¬ x>y ∧ x>z ∧ z>y is LIA-unsatisfiable,add(x>y ∨ ¬ x>z ∨ ¬ z>y)

Conflicting clause!
...no decision to backtrack

Input is LIA-UNSAT

Context

¬ x>y
x>z
z>y

Another Example

$$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$$

- DPLL(LIA) algorithm

Invoke DPLL(T) for theory T = LIA (linear integer arithmetic)

Another Example

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$

- DPLL(LIA) algorithm
    - Propagate : $x+y>0 \rightarrow$ false
    - Propagate : $x+1>0 \rightarrow$ true
    - Decide : $x<0 \rightarrow$ true

➡ Unlike propositional SAT case, we must check T-satisfiability of context

Context

$\neg x+y>0$

$x+1>0$

$x<0^d$

Another Example

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on: $\underbrace{\{x+1>0, \neg x+y>0, x<0\}}$

Context is LIA-unsatisfiable! → one of $x+1>0$, $x<0$ must be false

Context

$\neg x+y>0$

$x+1>0$

$x<0^d$

Another Example

(x+1>0 ∨ x+y>0) ∧(x<0 ∨ x+y>4) ∧ ¬x+y>0 ∧ (¬x+1>0 ∨ ¬x<0)

Conflicting clause!
...backtrack on a decision

Context

¬x+y>0
x+1>0
$x<0^d$

- DPLL(LIA) algorithm
  - Propagate : x+y>0 → false
  - Propagate : x+1>0 → true
  - Decide : x<0 → true
  - Invoke theory solver for LIA on: {x+1>0, ¬ x+y>0, x<0}
    - Add theory lemma (¬x+1>0 ∨ ¬x<0)

Another Example

(x+1>0 ∨ x+y>0) ∧(x<0 ∨ x+y>4) ∧ ¬x+y>0 ∧
(¬ x+1>0 ∨¬ x<0)

- DPLL(LIA) algorithm
  - Propagate : x+y>0 → false
  - Propagate : x+1>0 → true
  - Propagate : x<0 → false

Context

¬x+y>0
x+1>0
¬ x<0

Another Example

(x+1>0 ⋁ x+y>0) ⋀(x<0 ⋁ x+y>4) ⋀ ¬x+y>0 ⋀
(¬ x+1>0 ⋁¬ x<0)

- DPLL(LIA) algorithm
  - Propagate : x+y>0 → false
  - Propagate : x+1>0 → true
  - Propagate : x<0 → false
  - Propagate : x+y>4 → true
  - Invoke theory solver for LIA on: { x+1>0, ¬ x+y>0, ¬x<0， x+y>4 }

Context

¬x+y>0
x+1>0
¬ x<0
x+y>4

Another Example

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0 \land$
$(\neg x+1>0 \lor \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: $\{ x+1>0, \neg x+y>0, \neg x<0, x+y>4 \}$

Context is LIA-unsatisfiable! $\rightarrow$ one of
$\neg x+y>0$, $x+y>4$ must be false

Context

$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

Another Example

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0 \land$
$(\neg x+1>0 \lor \neg x<0) \land (x+y>0 \lor \neg x+y>4)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: $\{ x+1>0, \neg x+y>0, \neg x<0, x+y>4 \}$
    - Add theory lemma $(x+y>0 \lor \neg x+y>4)$

Conflicting clause!
...no decision to backtrack

Input is unsat

Context

$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T)

- DPLL(T) algorithm for satisfiability modulo T
  - Extends DPLL (indeed CDCL) algorithm to incorporate reasoning about a theory T
  - Basic Idea:
    - Use CDCL algorithm to find assignments for propositional abstraction of formula

      Use off-the-shelf SAT solver
    - Check the T-satisfiability of assignments found by SAT solver

      Use Theory Solver for T
    - Perform contradiction detection and theory propagation at partial assignments in CDCL

      Use Theory Solver for T

DPLL(T) Theory Solver

- Input : A set of T-literals M
- Output : either

1. M is T-satisfiable
   - Return model, e.g. { x → 2, y → 3, z → -3, … }

   → Should be *solution-sound*
     - Answers "M is T-satisfiable" only if M is T-satisfiable
2. $\{l_1, \dots, l_n\} \subseteq$ M is T-unsatisfiable // $l_1 \land \cdots \land l_n$
   - Return conflict clause ($\neg l_1 \lor \dots \lor \neg l_n$)

   → Should be *refutation-sound*
     - Answers "$\{l_1, \dots, l_n\}$ is T-unsatisfiable" only if $\{l_1, \dots, l_n\}$ is T-unsatisfiable
3. Don't know
   - Return lemma

→ If solver is solution-sound, refutation-sound, and *terminating*,
- Then it is a *decision procedure* for T

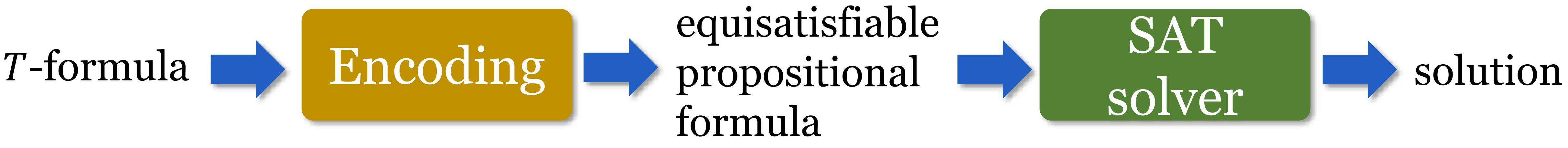
Design of DPLL(T) Theory Solvers

- A DPLL(T) theory solver:
  - Should be solution-sound, refutation-sound, terminating
  - Should produce models when M is T-satisfiable
  - Should produce T-conflicts of minimal size when M is T-unsatisfiable
  - Should be designed to work incrementally
    - M is constantly being appended to/backtracked upon
  - Can be designed to check T-satisfiability either:
    - Eagerly: Check if M is T-satisfiable immediately when any literal is added to M
    - Lazily: Check if M is T-satisfiable only when M is complete
  - Should cooperate with other theory solvers when combining theories
    - (see later)

Outline

- SMT Basis
- Lazy Approach --- DPLL(T)
- Eager Approach --- Bit Blasting

Eager Approach



Perform a full reduction of a T-formula to an equisatisfiable propositional formula in ***one-step***. A ***single run*** of the SAT solver on the propositional formula is then sufficient to decide the original formula.

Eliminating Function Applications

Ackermann's method

Eliminate applications of function and predicate symbols of non-zero arity.

These applications are replaced by new propositional symbols, and also encode information to maintain functional consistency (the congruence property).

Suppose that function symbol f has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$. First, we generate three fresh constant symbols xf_1, xf_2, and xf_3, one for each of the three different terms containing f, and then we replace those terms in F_{norm} with the fresh symbols.

The result is the following set of functional consistency constraints for f:

$$\left\{a_1 = a_2 \implies xf_1 = xf_2,\ a_1 = a_3 \implies xf_1 = xf_3,\ a_2 = a_3 \implies xf_2 = xf_3\right\}$$

Eliminating Function Applications

The Bryant-German-Velev method

eliminate function applications using a nested series of ITE expressions.

f has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$, then we would generate three new constant symbols xf_1, xf_2, and xf_3. We would then replace all instances of $f(a_1)$ by xf_1, all instances of $f(a_2)$ by $ITE(a_2 = a_1, xf_1, xf_2)$, and all instances of $f(a_3)$ by $ITE(a_3 = a_1, xf_1, ITE(a_3 = a_2, xf_2, xf_3))$. It is easy to see that this preserves functional consistency.

Small-domain encodings

- an enumerative approach

$$\sum_{j=1}^{n} a_{i,j} x_j \geq b_i$$

- the coefficients and the constant terms are integer constants and the variables are integer-valued.

If there is a satisfying solution to a formula, there is one whose size, measured in bits, is polynomially bounded in the problem size [BT76, vzGS78, KM78, Pap81]. Problem size is traditionally measured in terms of the parameters m, n, $\log a_{\max}$, and $\log b_{\max}$, where m is the total number of constraints in the formula, n is the number of variables (integer-valued constant symbols), and $a_{\max} = \max_{(i,j)} |a_{i,j}|$ and $b_{\max} = \max_i |b_i|$ are the maximums of the absolute values of coefficients and constant terms respectively.

Small-domain encodings

- Given a formula F_{Z} , we first compute the polynomial bound S on solution size, and then search for a satisfying solution to F_{Z} in the bounded space $\{0,1,\ldots,2^S-1\}$
- S is O(log m + log b_{max} + m[log m + log a_{max}])

Improving Small-domain encoding

Equalities

- Theorem. For an equality logic formula with n variables, S = log n

- The key proof argument is that any satisfying assignment can be translated to the range {0, 1, 2, . . . , n – 1}, since we can only tell whether variable values differ, not by how much.

- Get compact search space by constraint graph
  - representing equalities and disequalities between variables in the formula
  - Connected components of this graph correspond to equivalence classes

Improving Small-domain encoding

Difference Logic

$$x_i - x_j \bowtie b_t$$

x_0 is a special "variable" denoting zero.

- Build constraint graph

1. A vertex v_i is introduced for each variable x_i, including for x_0.
2. For each difference constraint of the form $x_i - x_j \geq b_t$, we add a directed edge from v_i to v_j of weight b_t.

Improving Small-domain encoding

Theorem 26.3.3. Let F_{diff} be a DL formula with n variables, excluding x_0. Let $b_{\max}$ be the maximum over the absolute values of all difference constraints in F_{diff}. Then, F_{diff} is satisfiable if and only if it has a solution in $\{0, 1, 2, \ldots, d\}^n$ where $d = n \cdot (b_{\max} + 1)$.

- any satisfying assignment for a formula with constraints represented by G can have a spread in values that is at most the weight of the longest path in G.
- This path weight is at most n $\cdot(b_{max} + 1)$. The bound is tight, the “+1” in the second term arising from a “rounding” of inequalities from strict to non-strict.

Bit Vector

Many compilers have this sort of bug

overflow?

$(x - y > 0) \Leftrightarrow (x > y)$

What is the output? (44)

```
unsigned char number = 200;
number = number + 100;
printf ("Sum: %d\n", number );
```

- Bitwise operator frequently occur in system-level software
  - left-shift, right-shift
  - and, or, xor

Complexity

- Satisfiability is undecidable for an unbounded width, even without arithmetic.

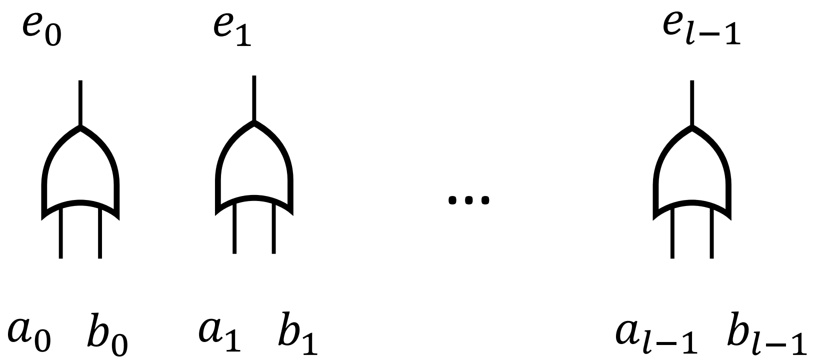
- It is NP-complete otherwise.

Operator to Circuit

Bitwise operators (l-bits): $a|b$

Introduce new bitvector variable e for $a|b$, such that foreach i

$$(a_i \lor b_i) \Leftrightarrow e_i$$

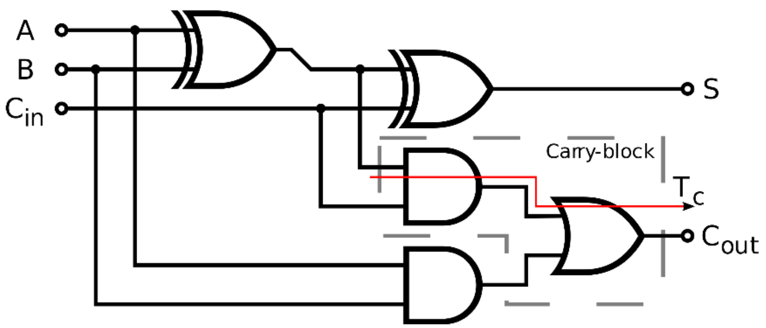


Other bitwise operators is similar

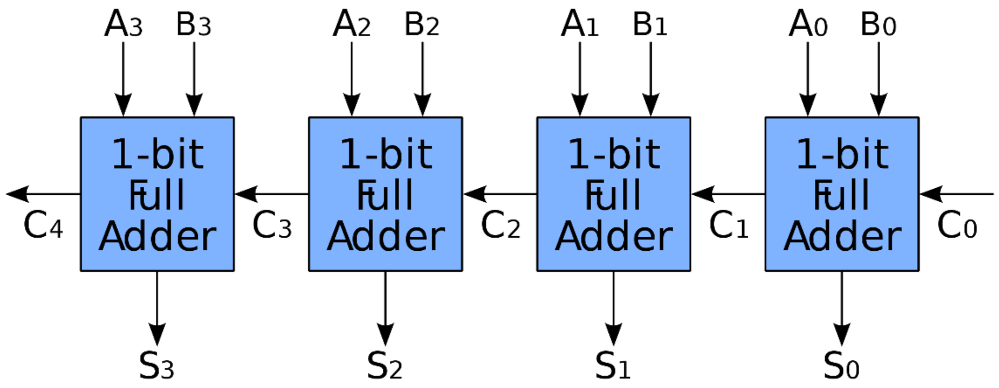
Operator to Circuit

$$a + b$$

one-bit Full adder



four-bits Full adder



How about 32-bits or 64-bits

Operator to Circuit

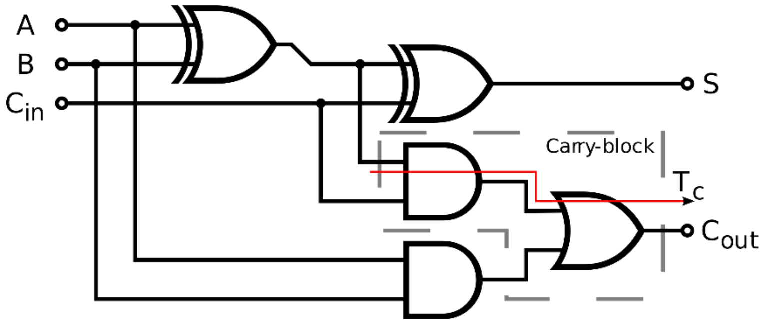
Complement(补码)
for negative numbers:

$$-b \to \sim b + 1$$

$\sim b$: invert each bits of b

$$a - b = (a + \sim b + 1)$$

one-bit Full adder




```
6 - 3 ==> 6 + (-3)
  0000 0110 // 6(补码)
+ 1111 1101 // -3(补码)
----------------------
  0000 0011 // 3(补码)
```

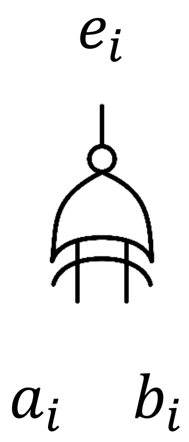
CNF: How many variables and clauses?

XOR | $C = A \oplus B$ | $(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee B \vee C)$

Operator to Circuit

$a = b$

$a_i = b_i \Leftrightarrow e_i$



$(a - b = (2^l - b) + a)_{mod\ 2^l}$

If $c_{out} = 1$, then in RHS, the subtract part b is less than the addition part a, i.e. $b < a$

unsigned $a < b$

$\langle a \rangle_U < \langle b \rangle_U \Leftrightarrow \neg add(a, \sim b, 1).c_{out}$

$2 - 3 \Rightarrow 010 - 011 = 010 + 101, c_{out} = 0$

$3 - 2 \Rightarrow 011 - 010 = 011 + 110, c_{out} = 1$

signed $a < b$

$\langle a \rangle_S < \langle b \rangle_S \Leftrightarrow (a_{l-1} \Leftrightarrow b_{l-1}) \oplus add(a, \sim b, 1).\ cout$

Operator to Circuit

$$a \ll b$$

n-stage (n is the width of b)

stage 1: for each bit i

$$e_i \Leftrightarrow \begin{cases} a_i & : b_0 = 0 \\ a_{i-1} & : i \geq 1 \wedge b_0 \\ 0 & : otherwise \end{cases}$$

stage 2: for each bit i

$$e'_i \Leftrightarrow \begin{cases} e_{i-2^1} & : i \geq 2^1 \wedge b_1 \\ e_i & : b_1 = 0 \\ 0 & : otherwise \end{cases}$$

...

if ($i < 1$)

$ite(b_0, (e_i \Leftrightarrow 0), (e_i \Leftrightarrow a_i))$

if ($i \geq 1$)

$ite(b_0, (e_i \Leftrightarrow a_{i-1}), (e_i \Leftrightarrow a_i))$

$$1011011 \ll 101$$

Stage 1:

$0110110 \Leftarrow 1011011 \ll 001$

Stage 2:

$0110110 \Leftarrow 0110110 \ll 000$

Stage 3:

$1100000 \Leftarrow 0110110 \ll 100$

Operator to Circuit

$$a \times b$$

n-stage (shift-and-add):

$(l-1)$ adder

$$mul(a, b, -1) \doteq 0$$
$$mul(a, b, i) \doteq mul(a, b, i-1) + (b_i ? (a \ll i) : 0)$$

$$\begin{array}{rl} 1001 & \\ \times\, 0101 & \\ \text{------}\ \text{-} & \\ 1001 & b_0 = 1 \to a \ll 0 \\ 0000\# & b_1 = 0 \to 0 \\ 1001\#\# & b_2 = 1 \to a \ll 2 \\ 0000\#\#\# & b_3 = 0 \to 0 \\ \text{-------}\ \text{-} & \end{array}$$

Operator to Circuit

$$a \div b$$

Implemented by adding two constraints:

$$\begin{aligned} &b \neq 0 \Longrightarrow e \times b + r = a, \\ &b \neq 0 \Longrightarrow r < b \end{aligned}$$

If $b = 0$, $a \div b$ is set to a special value.

Rewrite before Bit-Blasting

n	Number of variables	Number of clauses
8	313	1001
16	1265	4177
24	2857	9529
32	5089	17057
64	20417	68929

Fig. The size of the constraint for an n-bit multiplier expression after Tseitin's transformation

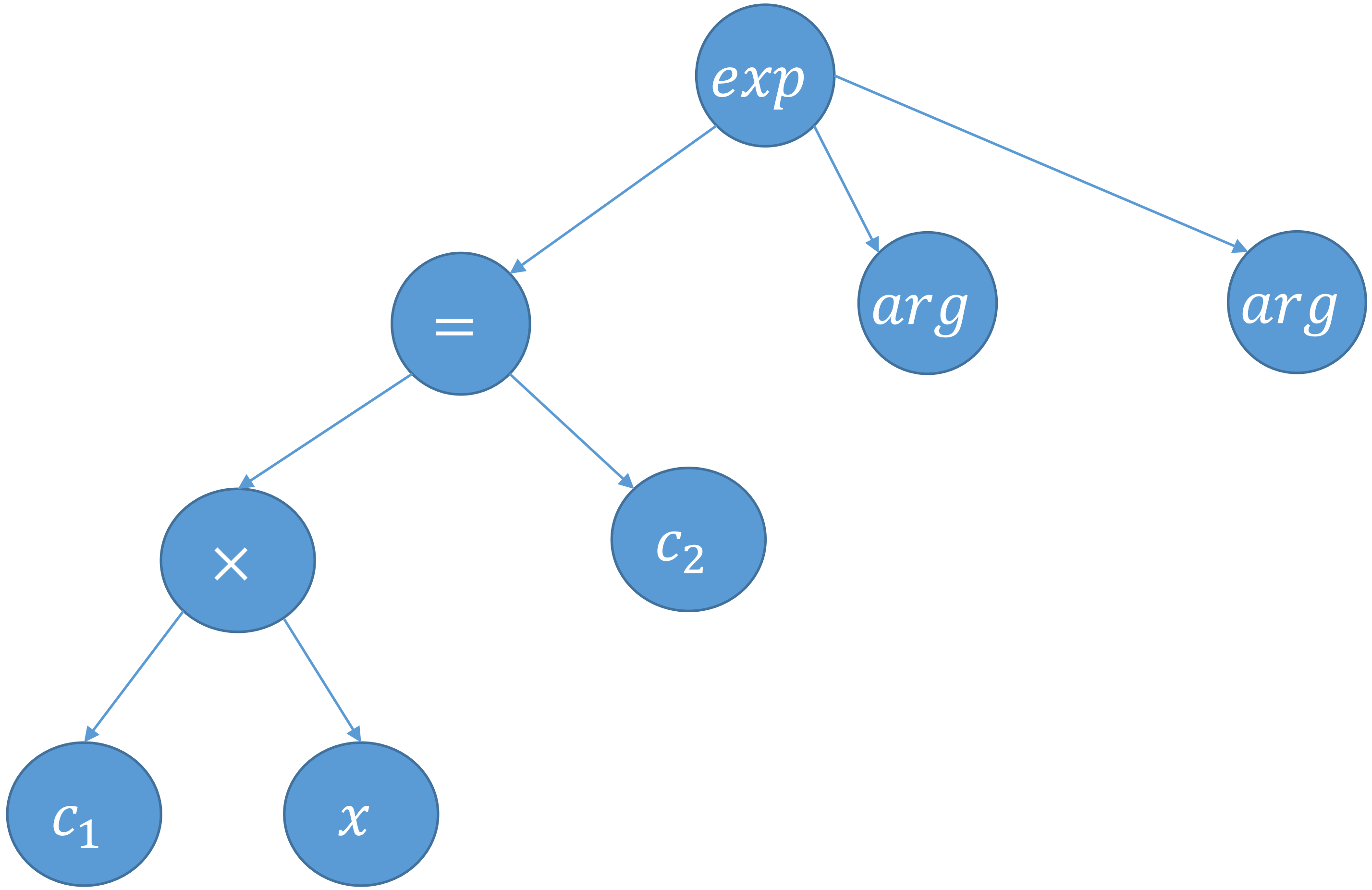
formulas with expensive operators (e.g. multipliers) are often very hard to solve

$$t \times (s \ll (s+t)) \Leftrightarrow s \times (t \ll (s+t))$$

32bits. 10^5 variables.
Can't be solved by CaDiCal within 2 hour

Rewrite before Bit-Blasting

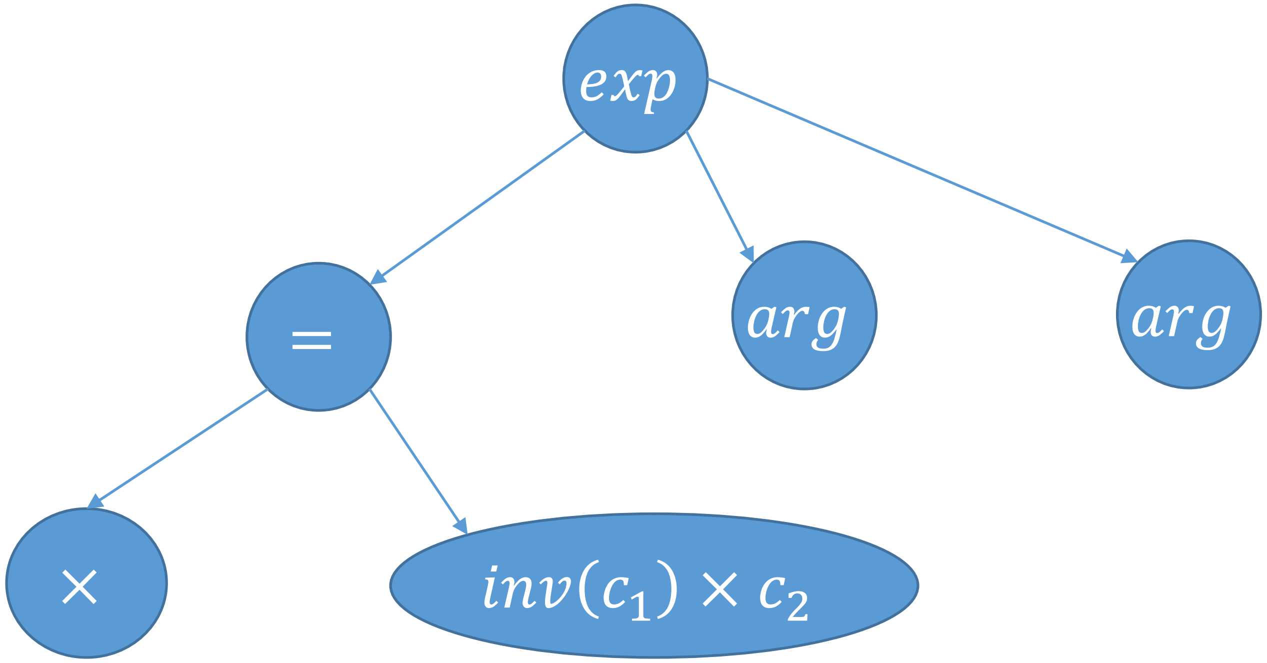
$$c_1 \times x = c_2$$



Rewrite before Bit-Blasting

$$c_1 \times x = c_2$$

$$x = inv(c_1) \times c_2$$



reduce one multiplier

Deep first order travelling

Theory rewrite rules

- bit2bool (c is 0 or 1)
  - $(ite\ x\ y\ z) = c \rightarrow (ite\ x\ (y = c)\ (z = c))$
  - $(not\ x) = c \rightarrow x = (1 - c)$
- mul_eq
  - $cx = c' \rightarrow x = c_{inv} \times c'$
  - $cx = c'x_2 \rightarrow x = (c_{inv} \times c')\ x_2$
  - …
- mul
  - $cx + c'x \rightarrow (c + c')x$
  - …
- add
  - $(x + (y \ll x)) \rightarrow (x|(y \ll x))$
  - $(x + y \times x) \rightarrow x \times (y + 1)$
- …

Reduce the number of operator

Expensive operator → cheap operator

Propagate const values

- Given an equality $t = c$, when c is constant, then replaces t everywhere with c

$t = c$

Replace $F(t)$ with $F(c)$

$formula[0]$, $formula[1]$, $formula[2]$, … , $formula[n]$

cyclical scan till fixed

Variable elimination does not always help

$$x = y + z + w$$
$$\dots(x + z)\dots$$
$$\dots(x + 2z)\dots$$
$$\dots(x + 3z)\dots$$
$$\dots(x + 4z)\dots$$

$$\Rightarrow$$

$$\dots(y + 2z + w)\dots$$
$$\dots(y + 3z + w)\dots$$
$$\dots(y + 4z + w)\dots$$
$$\dots(y + 5z + w)\dots$$

6 adder

8 adder

How to avoid increasing the number of adder and multipliers?

only eliminate variables that occur at most twice

Eliminate unconstrained variables

- a bit-vector function f can be replaced by a fresh bit-vector variable if
  - at least one of its operands is an unconstrained variable v (free variable)
  - f can be "inverted" with respect to v

$$v3 + t = v1 \,\&\, v2$$

$$\Downarrow$$

$$v3 + t = v4$$

$$\Downarrow$$

$$v5 = v4$$

$$\Downarrow$$

$$v6$$

If $v1$ and $v2$ are unconstrained variables then no matter what's the value of LHS, it is satisfiable.

If $v3$ is unconstrained variables then no matter what's the value of $v4$ and t, it is satisfiable.

bv_size_reduction

- Reduce bv size using upper bound and lower bound

$$1 \leq \; x \leq \; 4 \;\; (x \text{ has 8 bits})$$

$$\downarrow$$

Replace x with $(concat\ 00000\ x')$

x' is new variable of 3-bits

Local contextual simplification

- bool rewrite

$$\left(or\ args[0]\ \ldots\ args\left[num_{args}-1\right]\right)$$

replace $args[i]$ by $false$ in the other arguments

$$(x\ != \ 0\ or\ \ y\ \ =\ \ x+1)\ \ {-}{>}\ \ (x\ != \ 0\ or\ y\ \ =\ \ 1)$$

Hoist, max sharing

- Reduce the number of adder and multiplier using distribution and association

2 multiplier + 1 adder → 1 multiplier + 1 adder

Hoist: $a * b \ + \ a * c \ \rightarrow \ (b + c) * a$

Max Sharing: $a \ + \ (b \ + \ c), a \ + \ (b \ + \ d) \rightarrow \ (a + b) + c, (a + b) + d$

$(a + b)$ *only need to calculte once*

AIG

AIGs can be used to represent arbitrary boolean formulas and circuits

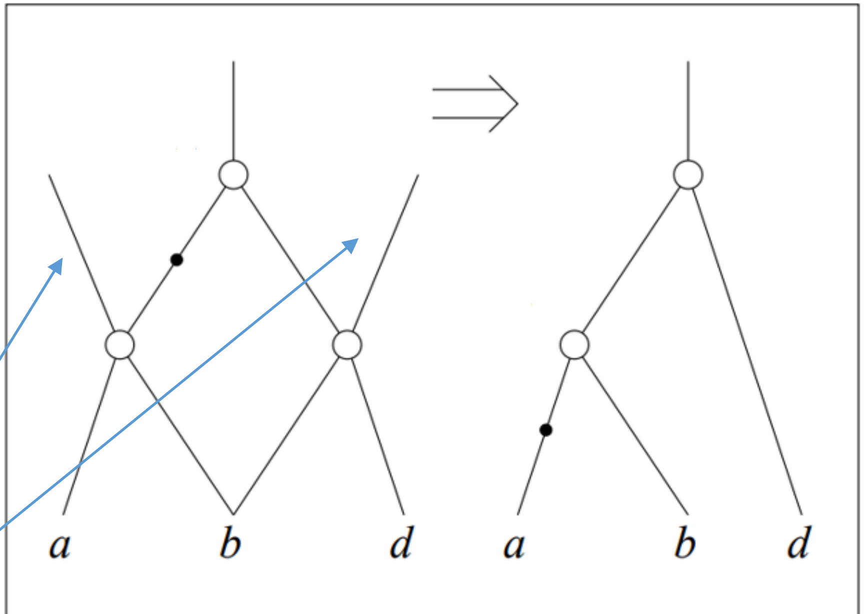
Automatic structure sharing and the simplicity of AIGs make them a compact, simple, easy to use, and scalable representation.

Name	Function	Representation by two-input AND and inversion
Inversion	$\neg x$	$\neg x$
Conjunction	$x \wedge y$	$x \wedge y$
Disjunction	$x \vee y$	$\neg(\neg x \wedge \neg y)$
Implication	$x \rightarrow y$	$\neg(x \wedge \neg y)$
Equivalence	$x \leftrightarrow y$	$\neg(x \wedge \neg y) \wedge \neg(\neg x \wedge y)$
Xor	$x \oplus y$	$\neg(\neg(x \wedge \neg y) \wedge \neg(\neg x \wedge y))$

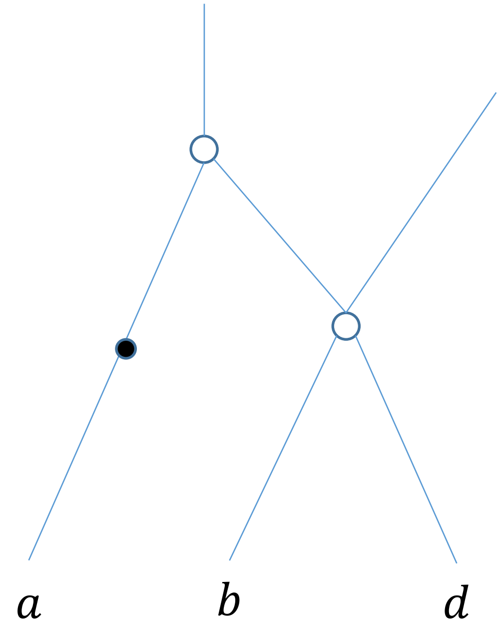
Table 1. Basic logic operations with two-input AND gates and negation.

Local 2-level AIG rewrite

2-level:
Consider children and grand-children



harmful substitution variant



Referenced by other nodes

Locally size decreasing, global non increasing

$$\neg(a \wedge b) \wedge (b \wedge d) \Rightarrow (\neg a \wedge b) \wedge d$$

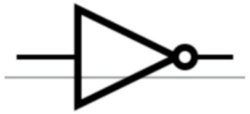
Local 2-level AIG rewrite

Name	LHS	RHS	O	S	Condition
Neutrality	$a \wedge \top$	a	1	S	
Boundedness	$a \wedge \bot$	$\bot$	1	S	
Idempotence	$a \wedge b$	a	1	S	$a = b$
Contradiction	$a \wedge b$	$\bot$	1	S	$a \neq b$
Contradiction	$(a \wedge b) \wedge c$	$\bot$	2	A	$(a \neq c) \vee (b \neq c)$
Contradiction	$(a \wedge b) \wedge (c \wedge d)$	$\bot$	2	S	$(a \neq c) \vee (a \neq d) \vee (b \neq c) \vee (b \neq d)$
Subsumption	$\neg(a \wedge b) \wedge c$	c	2	A	$(a \neq c) \vee (b \neq c)$
Subsumption	$\neg(a \wedge b) \wedge (c \wedge d)$	$c \wedge d$	2	S	$(a \neq c) \vee (a \neq d) \vee (b \neq c) \vee (b \neq d)$
Idempotence	$(a \wedge b) \wedge c$	$a \wedge b$	2	A	$(a = c) \vee (b = c)$
Resolution	$\neg(a \wedge b) \wedge \neg(c \wedge d)$	$\neg a$	2	S	$(a = d) \wedge (b \neq c)$
Substitution	$\neg(a \wedge b) \wedge c$	$\neg a \wedge b$	3	A	$b = c$
Substitution	$\neg(a \wedge b) \wedge (c \wedge d)$	$\neg a \wedge (c \wedge d)$	3	S	$b = c$
Idempotence	$(a \wedge b) \wedge (c \wedge d)$	$(a \wedge b) \wedge d$	4	S	$(a = c) \vee (b = c)$
Idempotence	$(a \wedge b) \wedge (c \wedge d)$	$a \wedge (c \wedge d)$	4	S	$(b = c) \vee (b = d)$
Idempotence	$(a \wedge b) \wedge (c \wedge d)$	$(a \wedge b) \wedge c$	4	S	$(a = d) \vee (b = d)$
Idempotence	$(a \wedge b) \wedge (c \wedge d)$	$b \wedge (c \wedge d)$	4	S	$(a = c) \vee (a = d)$

Table 2. All locally size decreasing, globally non increasing, two-level optimization rules. "O" is the optimization level, "S" the type of symmetry. Subsumption is also known as "Absorption". The condition $a \neq b$ is a short hand for $a = \neg b$ or $b = \neg a$.

Circuit to CNF

Tseitin Transformation

Type	Operation	CNF Sub-expression
 AND	$C = A \cdot B$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$
 NAND	$C = \overline{A \cdot B}$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee C) \wedge (B \vee C)$
 OR	$C = A + B$	$(A \vee B \vee \overline{C}) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee C)$
 NOR	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\overline{A} \vee \overline{C}) \wedge (\overline{B} \vee \overline{C})$
 NOT	$C = \overline{A}$	$(\overline{A} \vee \overline{C}) \wedge (A \vee C)$
 XOR	$C = A \oplus B$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee B \vee C)$

→ SAT solver

Pseudo-Boolean to BV

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \geq c$$

$$\Downarrow$$

$$lhs \geq rhs$$

$$a_ix_i \Leftrightarrow ite(x_i, bv(a_i), bv(0))$$
$$lhs = bvadd\ (ite_1, ite_2, \dots, ite_n)$$
$$rhs = bv(c)$$

other relation operators (e.g. LT, GT, EQ) can be represent by GE

LIA/NIA to BV

$foreach$ variable x:

1. collect low bound low and upper bound up

2. BV size

If ($low \leq x \leq up$)

$$bits = \log_2(1 + |up - low|)$$

Otherwise

$$bits = num_{bits}$$

num_{bits} = bit_size of abs(Largest constant) + 1

Under approximate
unbound → bound
satisfiability is not preserving

3. BitVector

If (has low)

$$x \Leftrightarrow x_{bv} + low$$

else if (has up)

$$x \Leftrightarrow up - x_{bv}$$

else

$$x \Leftrightarrow x - 2^{bits-1}$$

$(-2^{\mathrm{bits}-1})$ is the *lower bound* of signed int of size $bits$

LIA/NIA to BV

$x \; op \; y$

1. Align BV size of x and y

2. Extend BV size of x and y according to op

$x_{3bits} + \mathrm{y}_{4\mathrm{bits}}$

$\Downarrow$

$x_{4bits} + y_{4bits}$ $\qquad\qquad$ $x_{4bits} \times y_{4bits}$

$\Downarrow$ $\qquad\qquad$ $\Downarrow$

$x_{5bits} + y_{5bits}$ $\qquad\qquad$ $x_{8bits} \times y_{8bits}$

Thank you!