Propositional Encoding
- Decision Procedure

Daniel Kroening, Ofer Strichman

Presented by Changki Hong

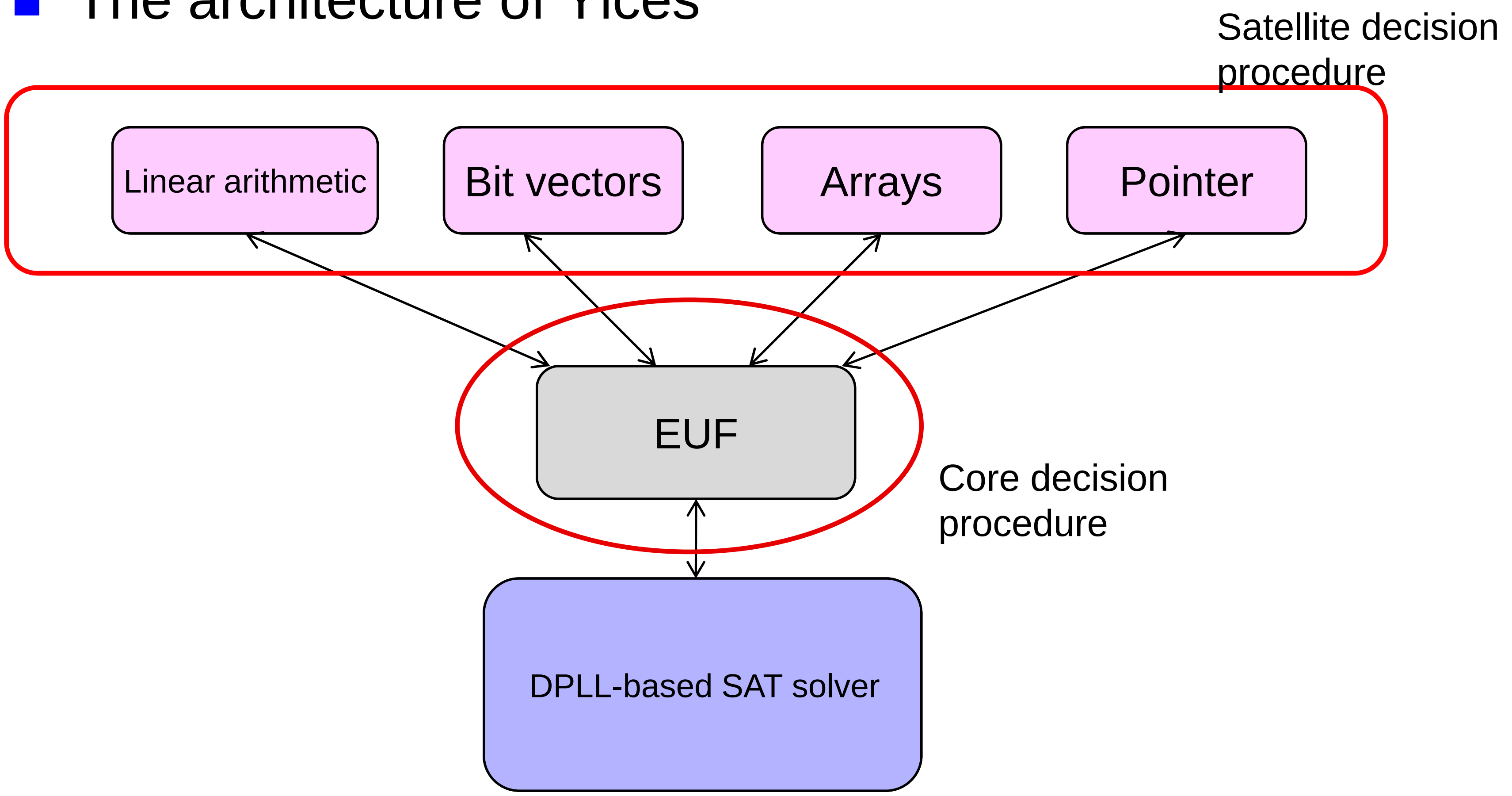
Decision procedures so far..

- The decision procedures so far focus on one specific theory
  - We know how to
    - Decide Equality logic with Uninterpreted Functions (EUF) :
      - $(x_1 = x_2) \wedge (f(x_2) = x_3) \wedge \ldots$
    - Decide linear arithmetic :
      - $3x_1 + 5x_2 \geq 2x_3 \wedge x_3 \leq x_5$

- How about a combined formula?
  - A combination of linear arithmetic and EUF:
    - $(x_2 \geq x_1) \wedge (x_1 - x_3 \geq x_2) \wedge f(f(x_1) - f(x_2)) \neq f(x_3)$
  - A combination of bit-vectors and uninterpreted functions:
    - $f(a[32], b[1]) = f(b[32], a[1]) \wedge a[32] = b[32]$

Example

- The architecture of Yices


Satellite decision procedure

Linear arithmetic

Bit vectors

Arrays

Pointer

EUF

Core decision procedure

DPLL-based SAT solver

from tool paper describing Yices

Combination of theories

- Approach 1 : Combine decision procedures of the individual theories.
  - Nelson-Oppen method

- Approach 2 : Reduce all theories to a common logic if possible (e.g. Propositional logic)
  - Combine decision procedure for individual theories with a propositional SAT solver.

Approach 2 In detail

- Two encoding schemes in the category of the approach 2
  - Eager encoding
    - SAT solver is invoked only once with no further interaction with decision procedure of each theories.
  - Lazy encoding
    - Keep interacting between SAT solver and decision procedures of each theories.
    - Almost every tool that participated in the SMT competitions in 2005-2007 belongs to this category of solvers.

Contents

- Motivation
- Preliminaries
- Eager encoding
- Lazy encoding
- Conclusion

Preliminaries

- Basic architecture of DPLL SAT solver

```
While (true)
{
        if (Decide() == FALSE) return (SAT);
        while (BCP() == “conflict”) {
                backtrack-level = Analyze_Conflict();
                if (backtrack-level < 0) return (UNSAT);
                else BackTrack(backtrack-level);
        }
}
```

Choose the next variable and value.
Return False if all variables are assigned

Apply repeatedly the *unit clause rule*.
Return “conflict” if reached a conflict

Backtrack until no conflict.
Return -1 if impossible

Eager encoding

- Eager encoding
  - Perform a full reduction from the problem of deciding Σ-formulas to one of deciding propositional formulas.
  - All the necessary clauses are added to the propositional skeleton.
  - SAT solver is invoked only once, with no further interaction with decision procedure of each theories.
  - Example
    - Equality logic and Uninterpreted Functions
      - Substitute equality literals into Boolean variables and add constraints
    - Array logic
      - Substitute array read operation into UF

Eager encoding

- Algorithm 4. Eager-encoding

```
Input: A formula $\phi$
Output: "Satisfiable" if $\phi$ is satisfiable and "Unsatisfiable" otherwise

function Eager-Encoding($\phi$)
    $e(P) := Deduction(lit(\phi))$;
    $\phi_E := e(\phi) \wedge e(P)$;
    $<\alpha, res> := SAT\text{-}Solver(\phi_E)$;
    if res ="Unsatisfiable" then return "Unsatisfiable";
    else return "Satisfiable";
```

Lazy encoding

- Two main engines
  - SAT solver : assigns truth values to literals in order to satisfy the Boolean structure of the formula
  - Decision procedure of the individual theories : checks whether this assignment is consistent in theory.

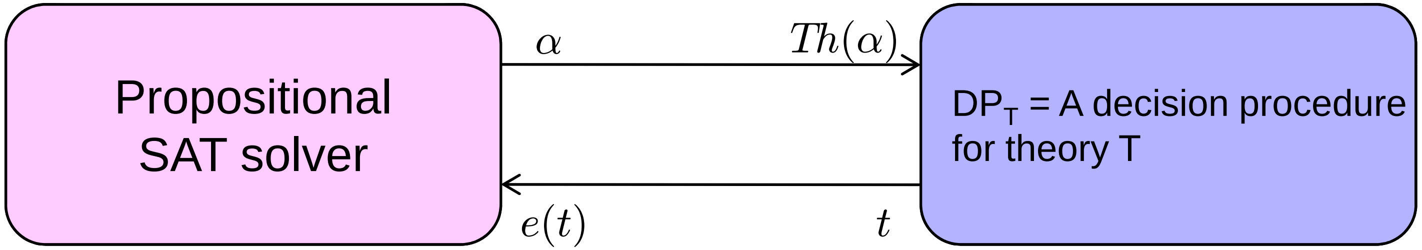
- Definition 1. (*Boolean encoder*)
  - Given a Σ-literal l, we associate with it a unique Boolean variable $e(l)$, which we call the *Boolean encoder* of this literal.
  - Given a Σ-formula t, $e(t)$ denotes the *Boolean formula* resulting from substituting each Σ-literal in t with its *Boolean encoder*. We also call it as *propositional skeleton*.

Overview of lazy encoding

- Example
  - Let theory T be equality logic.
    - $\phi := \; x = y \wedge ((y = z \wedge x \neq z) \vee x = z)$

1. Compute propositional skeleton of the given formula
   - $\phi := \; e(x = y) \wedge ((e(y = z) \wedge e(x \neq z)) \vee e(x = z))$
   - $Let \; B := e(\phi)$
2. Pass B to a SAT solver
   - $\alpha := \{e(x = y) \mapsto \mathrm{TRUE}, e(y = z) \mapsto \mathrm{TRUE}, e(x \neq z) \mapsto \mathrm{TRUE}, \; e(x = z) \mapsto \mathrm{FALSE}\}$
3. Decision procedure decides whether the conjunction of the literals corresponding to this assignment $(\mathrm{Th}(\alpha))$ is satisfiable
   - $Th(\alpha) := \; x = y \wedge y = z \wedge x \neq z \wedge \neg(x = z)$
   - blocking clause : $e(\neg Th(\alpha)) := \neg e(x = y) \vee \neg e(y = z) \vee \neg e(x \neq z) \vee e(x = z)$
4. Pass $B \wedge e(\neg Th(\alpha))$ to a SAT solver.
   - $\alpha := \{e(x = y) \mapsto \mathrm{TRUE}, e(y = z) \mapsto \mathrm{TRUE}, e(x \neq z) \mapsto \mathrm{FALSE}, \; e(x = z) \mapsto \mathrm{TRUE}\}$

Overview of lazy encoding



Propositional SAT solver

DP_T = A decision procedure for theory T

- α - current assignment returned by SAT solver

- $Th(\alpha)$ - conjunction of the literal corresponding to current assignment and we define each literal, denoted $Th(lit_i, \alpha)$, as follows:

$$Th(lit_i, \alpha) \doteq \begin{cases} lit_i & \alpha(lit_i) = \text{TRUE} \\ \neg lit_i & \alpha(lit_i) = \text{FALSE} \end{cases}$$

- t - t is returned by DP_T and it is called blocking clause or lemma. This clause contradicts the current assignment, and hence blocks it from being repeated.

- $e(t)$ - Boolean formula of the blocking clause.

Lazy algorithm

- Algorithm 1. Lazy-basic

```
Input: A formula $\phi$
Output: "Satisfiable" if $\phi$ is satisfiable, and "Unsatisfiable" otherwise

function $Lazy\ Basic\ (\phi)$
    $B := e(\phi)$;
    while (true) do
        <$\alpha$, $res$> := SAT-Solver($B$);
        if $res$ ="Unsatisfiable" then return "Unsatisfiable";
        else
            <$t$, $res$> := Deduction$(Th(\alpha))$ ;
            if $res$ ="Satisfiable" then return "Satisfiable";
            $B := B \wedge e(t)$;
```

- Deduction
  - input - conjunction of the literal corresponding to current assignment
  - output - a tuple of the form <blocking clause, result> where the result is one of {"Satisfiable", "Unsatisfiable"}

Integration into DPLL

- Algorithm 2. Lazy-DPLL

```
Input: A formula $\phi$
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

function Lazy-DPLL
    AddClauses($e(\phi)$);
    if BCP() = "conflict" then return "Unsatisfiable";
    while (true) do
        if $\neg$Decide() then
            <t, res>:=Deduction($Th(\alpha)$);
            if res="Satisfiable" then return "Satisfiable";
            AddClauses(e(t));
            while (BCP() = "conflict") do
                backtrack-level := Analyze-Conflict();
                if backtrack-level < 0 then return "Unsatisfiable";
                else BackTrack(backtrack-level);
        else
            while (BCP() = "conflict") do
                backtrack-level := Analyze-Conflict();
                if backtrack-level < 0 then return "Unsatisfiable";
                else BackTrack(backtrack-level);
```

If there is no more assignment to do

Improvement

- Algorithm 2 does not call Deduction() until a full satisfying assignment is found.
  - Example
    - Assume that the Decide() procedure assigns $e(x_1 \geq 10) \mapsto \text{TRUE}$ and $e(x_1 < 0) \mapsto \text{TRUE}$.
    - Deduction() results in a contradiction.
    - Time taken to complete the assignment is wasted.
- Algorithm 2 can be improved by running Deduction before a full assignment to the Boolean encoder is available.
  - Contradictory partial assignment are ruled out early.
  - Implications of literals that are still unassigned can be communicated back to the SAT solver.
    - ex) once $e(x_1 \geq 10)$ has been assigned TRUE, we can infer that $e(x_1 < 0)$ must be FALSE and avoid conflict.

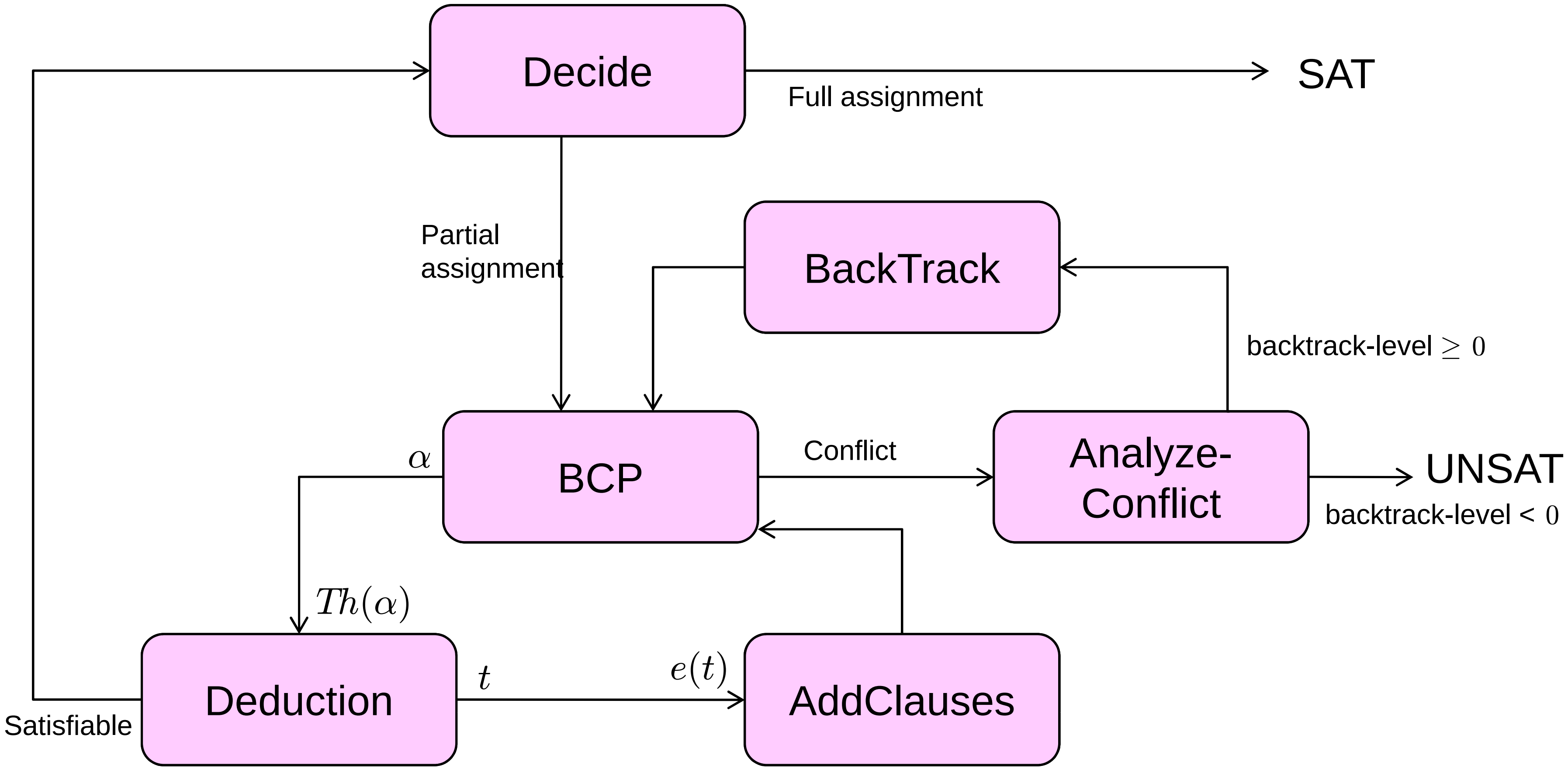
Improved Lazy-DPLL

- Algorithm 3. DPLL(T)

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

```
function DPLL(T)
        AddClauses(e(φ));
        if BCP() = "conflict" then return "Unsatisfiable";
        while (true) do
                if ¬Decide() then return "Satisfiable";      Full assignment
                repeat
                    while (BCP() = "conflict") do
                            backtrack-level := Analyze-Conflict();
                            if backtrack-level < 0 then return "Unsatisfiable";
                            else BackTrack(backtrack-level);
                    <t, res>:=Deduction(Th(α));              Partial assignment
                    AddClauses(e(t));
                until res = Satisfiable
```

DPLL (T)


Decide
SAT
Full assignment
Partial
assignment
BackTrack
backtrack-level ≥ 0
α
BCP
Conflict
Analyze-
Conflict
UNSAT
backtrack-level < 0
$Th(\alpha)$
t
$e(t)$
Deduction
AddClauses
Satisfiable

Implementation details of DPLL(T)

- Deduction
  - Returning blocking clause
    - If *S* is the set of literals that serve as the premises in the proof of unsatisfiability, then the blocking clause is

$$t := \left(\bigvee_{l \in S} \neg l\right)$$

    - Example
      - $Th(\alpha) := x = y \wedge y = z \wedge x \neq z \wedge \neg(x = z)$
      - blocking clause - $t := \neg(x = y) \vee \neg(y = z) \vee \neg(x \neq z) \vee x = z$

Implementation details of DPLL(T)

- Deduction
  - Returning implied assignment instead of blocking clauses
    - $Th(\alpha)$ implies a literal lit_i, then

$$t := \ (lit_i \vee \neg \hat{Th}(\alpha))$$

    - The encoded clause e(t) is of the form

$$\left(e(lit_i) \vee \bigvee_{lit_j \in Th(\alpha)} \neg e(lit_j)\right)$$

    - Example
      - Let $e(x_1 \geq 10) \mapsto \mathrm{TRUE}$, $e(x_1 < 0)$ is unassigned yet.
      - Deduction detects that $\neg(x_1 < 0)$ is implied.
      - $t := \neg(x_1 \geq 10) \vee \neg(x_1 < 0)$
      - $e(t) := (\neg(x_1 \geq 10) \vee \neg(x_1 < 0))$

Conclusions

- Two encoding schemes
  - Eager encoding
    - SAT solver is invoked only once with no further interaction with decision procedure of each theories.
  - Lazy encoding
    - Keep interacting between SAT solver and decision procedure of each theories.
    - Almost every tool that participated in the SMT competitions in 2005-2007 belongs to this category of solvers.