Program verification

Nelma Moreira

Decidable first order theories and SMT Solvers
Lecture 21

Decision algorithm DP_T: quantifier-free theories

The aim is to solve combinations such as

$$(x_1 = x_2 \vee x_1 = x_3) \wedge (x_1 = x_2 \vee x_2 = x - 4) \wedge x_1 \neq x_3 \wedge x_1 \neq x_4$$
$$(x_1 + 2x_3 < 5) \vee \neg(x_3 \leq 1) \wedge (x_2 \geq 3)$$
$$(i = j \wedge a[j] = 1) \wedge \neg(a[i] = 1)$$

We consider quantifier-free theories, T, for which there exists a decision algorithm DP_T for the conjunction of atomic formulae.

Example:Equality Logic

- Corresponds to the equality theory $\mathcal{T}_E$ only with variables (and constants that can be eliminated) and quantifers-free

$$\begin{aligned} \varphi &:= \varphi \wedge \varphi \mid (\varphi) \mid \neg\varphi \mid t = t \\ t &:= x \mid c \end{aligned}$$

- has the same expressivity and complexity of propositional logic.

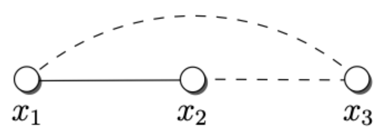
Exerc. 21.1. *Describe an algorithm no eliminate constants from a formula with equalities.* ◇

Decition procedure for theory of equality (conjunctions), DP_T

- Seja φ a conunction of equalities and inequalities
- Build a graph $G = (N, E_=, E_{\neq})$ where
- N are variables of φ,
- $E_=$, edges (x_i, x_j) correspond to equalities $x_i = x_j \in \varphi$ (dashes)

- $E_{\neq}$, edges (x_i, x_j) correspond to inequalities $x_i \neq x_j \in \varphi$ (filled)
- φ is not satisfiable if and only if there exists an edge $(v_1, v_2) \in E_{\neq}$ such that v_2 is reachable from v_1 by edges of $E_=$.

For $x_2 = x_3 \wedge x_1 = x_3 \wedge x_1 \neq x_2$, we conclude that is not satisfiable



Using SAT solvers for SMT

There are two approaches for the Boolean combination of atomic formulas

- *eager*
  - translate to an equisatisfiable propositional formula
  - that is solved by a SAT solver
- *lazy*
  - incrementally encode the formula in a proposicional formula
  - use DPLL SAT solver
  - use a solver for the theory (DP_T) to refine the formula and guide the SAT solver
- the lazy approach seems to work better

Lazy approach

Mainly in the case that φ contains other connectives besides conjunction is better to integrate D_T in a SAT solver.

- Suppose φ in (NNF)
- $at(\varphi)$ set of atomic formulae over Σ in φ; $at_i(\varphi)$ i-th atomic formula
- To each atomic formula $a \in at(\varphi)$ associate $e(a)$ a proposicional variable, called the *encoder*
- Extend the encoding e to φ, and let $e(\varphi)$ be the formula resulting from substituting each Σ-literal by its encoder.
- For example if $\varphi := (x = y \vee x = z)$ then $e(\varphi) := e(x = y) \vee e(x = z)$

Example

Let

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

We have

$$e(\varphi) := e(x = y) \wedge ((e(y = z) \wedge \neg(e(x = z))) \vee e(x = z)) := \mathcal{B}$$

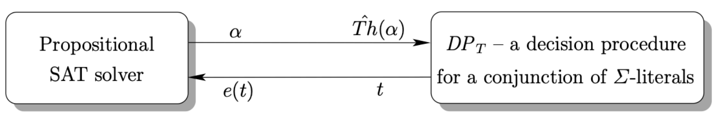
Using a SAT solver we obtain an assignment for $\mathcal{B}$:

$$\alpha := \{e(x = y) \mapsto \mathsf{true}, e(y = z) \mapsto \mathsf{true}, e(x = z) \mapsto \mathsf{false}\}$$

The procedure DP_T checks if the conjunction of literals correspondent to α is satisfiable, i. e.,

$$\hat{Th}(\alpha) = (x = y) \wedge (y = z) \wedge x \neq z$$

This formula is not satisfiable, thus $\neg\hat{Th}(\alpha)$ is a tautology. We can make the conjunction $e(\neg\hat{Th}(\alpha)) \wedge \mathcal{B}$ and call again the SAT solver but α will be blocked as it will not satisfy $e(\neg\hat{Th}(\alpha))$ (*blocking clause*).



Let α' be a new assignment

$$\alpha' := \{e(x = y) \to \mathsf{true}, e(y = z) \to \mathsf{true}, e(x = z) \to \mathsf{true}\}$$

that corresponds to

$$\hat{Th}(\alpha') := (x = y) \wedge (y = z) \wedge x = z$$

which is satisfiable, proving that the original formula φ is satisfiable.

Formally, given a encoding $e(\varphi)$ and an assignment α, for each encoder $e(at_i)$ we have

$$Th(at_i, \alpha) = \begin{cases} at_i & \alpha(e(at_i)) = \mathsf{true} \\ \neg at_i & \alpha(e(at_i)) = \mathsf{false} \end{cases}$$

and let the set of literals be

$$Th(\alpha) = \{Th(at_i, \alpha) \mid at_i \in \varphi\}$$

then $\hat{Th}(\alpha)$ is the conjunction of literals in $Th(\alpha)$.

Let DEDUCTION be the procedure DP_T with the possible generation of a blocking clause , $t = \neg\hat{Th}(\alpha)$.

Algorithm 3.3.1: LAZY-BASIC

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

```
function LAZY-BASIC(φ)
    B := e(φ);
    while (TRUE) do
        ⟨α, res⟩ := SAT-SOLVER(B);
        if res = "Unsatisfiable" then return "Unsatisfiable";
        else
            ⟨t, res⟩ := DEDUCTION(T̂h(α));
            if res = "Satisfiable" then return "Satisfiable";
            B := B ∧ e(t);
```

Consider the following three requirements on the formula t that is returned by Deduction:

1. t is valid in $\mathcal{T}$.
2. The atoms in t are restricted to those appearing in φ
3. The encoding of t contradicts α, i.e., $e(t)$ is a blocking clause

The first requirement 1. ensures soundness. The second and third requirements 2. e 3.

are sufficient to guaranteeing termination.

Two can be weakened:

- It is enough that t implies φ
- In t can occur other atomic formulas

Beside considering an incremental SAT (that keeps the $\mathcal{B}$ from previous calls, it is more efficient to integrate the procedure DEDUCTION in the CDCL algorithm.

CDCL(T): integrar DP_T em CDCL-SAT

Algorithm 3.3.2: LAZY-CDCL

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

```
function Lazy-CDCL
    AddClauses(cnf(e(φ)));
    while (true) do
        while (BCP() = "conflict") do
            backtrack-level := Analyze-Conflict();
            if backtrack-level < 0 then return "Unsatisfiable";
            else BackTrack(backtrack-level);
        if ¬Decide() then                              ▷ Full assignment
            ⟨t, res⟩:=Deduction(T̂h(α));               ▷ α is the assignment
            if res="Satisfiable" then return "Satisfiable";
            AddClauses(e(t));
```

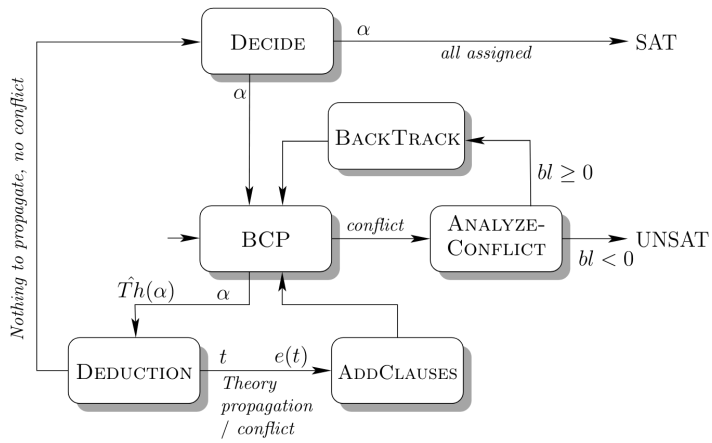
This algorithm uses a procedure ADDCLAUSES, which adds new clauses to the current set of clauses at run time.

Theory propagation

Suppose that φ has an integer variable x_1 and the literals $x_1 < 0$ and $x_1 > 10$. If $e(x_1 > 10) \mapsto \mathsf{true}$ and $e(x_1 < 0) \mapsto \mathsf{true}$ ther will be a contradiction but that is only detected after being obtained a full assignment. However that can be improved, if the call to DEDUCTION is made earlier. That allows to

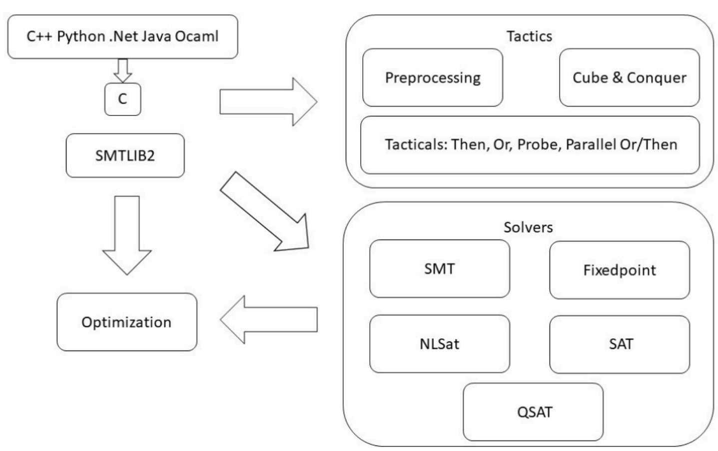
- Contradictory partial assignments are ruled early
- Implications of literals that are still unassigned can be communicated back to the Sat solver. We call this technique *theory propagation*.
- For example, if $e(x_1 > 10) \leftarrow \mathsf{true}$ we can infer that $e(x_1 < 0) \leftarrow \mathsf{false}$ and and thus avoid the conflict altogether.

DPLL(T)

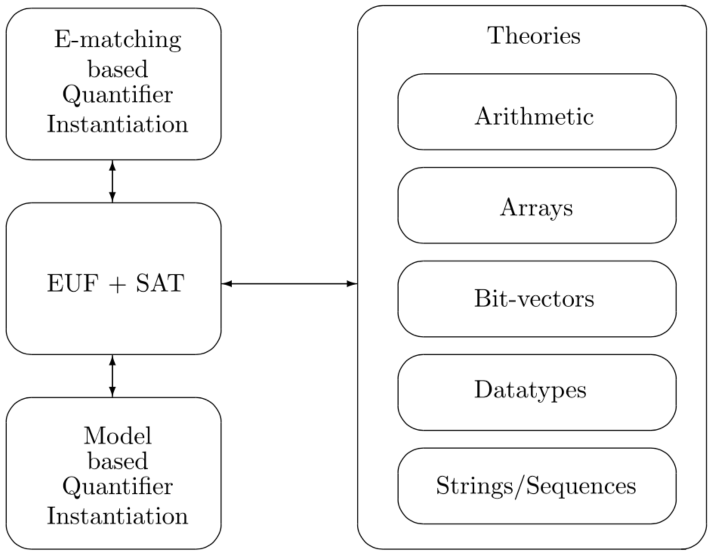


Z3

- Z3 https://github.com/Z3Prover/z3
- Z3 https://z3prover.github.io/papers/programmingz3.html
- https://z3prover.github.io/papers/z3internals.html
- Python : pip install z3-solver
- Tutorial: https://ericpony.github.io/z3py-tutorial/guide-examples.htm



Z3 Architecture of a SMT Solver



pyZ3

```
x = Real('x')
y = Real('y')
z = Real('z')
s = Solver()
s.add(3*x + 2*y - z == 1)
s.add(2*x - 2*y + 4*z == -2)
s.add(-x + 0.5*y - z == 0)
print(s.check())
print(s.model())
```

pyZ3

- Logical variables are created indicating their `Sort`: `Real`, `Bool`, `Int`, or any new declared type:

```
S = DeclareSort('S')
f = Function('f', S, S)
x = Const('x', S)
y = Const('y', S)
z  = Const('z', S)
s = Solver()
s.add(Or(x!=y,Or(f(x)==f(y),f(x)!=f(z))))
print(s.check())
print(s.model())
```

```
solve(Or(x!=y,Or(f(x)==f(y),f(x)!=f(z)))
```

- `solve()` creates a `Solver`, adds a formula and checks if it is satisfiable returning a solution (`model`).
- `Const` and `Function` define zero or more variables, respectively

SMT-LIB

- a standard language for SMT is the SMT-LIB (similar to LISP), but we can use the Python interface

```
x, y = Ints('x y')
s = Solver()
s.add((x % 4) + 3 * (y / 2) > x - y)
print(s.sexpr())
```

- outputs

```
(declare-fun y () Int)
(declare-fun x () Int)
(assert (> (+ (mod x 4) (* 3 (div y 2))) (- x y)))
```

- Quantifiers: `ForAll`, `Exists`

```
solve([y == x + 1, ForAll([y], Implies(y <= 0, x < y))])
```

The first occurence of `y` is free, the second is bounded.

Example SMT-LIB 2

```
(set-logic QF UFLIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (distinct x y z))
(assert (> (+ x y) (* 2 z)))
(assert (>= x 0))
(assert (>= y 0))
(assert (>= z 0))
(check-sat)
(get-model)
(get-value (x y z))
```

Usando `% z3 exemplo1.smt2`

```
sat
(
  (define-fun x () Int
    3)
  (define-fun z () Int
    1)
  (define-fun y () Int
    0)
)
((x 3)
 (y 0)
 (z 1))
```

pyz3: `s.from_file("exemplo1.smt2")`

Z3 API

- help(class) or help(function)
- *describe_tactics*.
-

References

[BdM15] Nikolai Bjorner and Leonardo de Moura. *Z3 Theorem Prover*. Rise, Microsft, 2015.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Verlag, 2007.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures:An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.