4. SMT Technology

Prof. Roberto Amadini

Department of Computer Science and Engineering, University of Bologna, Italy

Combinatorial Decision Making and Optimization

2nd cycle degree programme in Artificial Intelligence
University of Bologna, Academic Year 202/23

SMT Solvers

- Given a theory $\mathcal{T}$, a $\mathcal{T}$-solver is a procedure for deciding whether a conjunction of $\mathcal{T}$-literals is satisfiable
  - $\mathcal{T} =$ EUF, arithmetic, arrays, bit-vectors,...
- We can define a SMT solver as a collection of $\mathcal{T}_i$-solvers for different theories $\mathcal{T}_i$
  - Maybe combinations of these theories
- SMT solvers can handle formulas involving variables of different sort
  - sort $\equiv$ type (Integer, Real, Array, String, ...)
- The user interacts with SMT solver through queries
  - e.g., to check the satisfiability of a formula or add new formulas

SMT Solvers

- Nowadays plenty of SMT solvers available
  - Especially used for software analysis applications
- Some of them are "special-purpose"
  - E.g., only for solving bit-vectors or non-linear arithmetic formulas
- Some others more "general-purpose"
  - They can handle disparate theories
  - We shall see 2 of them: Z3 and CVC5

Z3

- Z3 is a well-known SMT solver with specialized algorithms for efficiently tackling several theories
  - First paper describing Z3: De Moura, L., and N. Bjørner. "*Z3: An efficient SMT solver*" TACAS 2008
- Z3 is open source: `https://github.com/z3prover/z3`
- It provides APIs for common programming languages like Python, Java, C++, .Net
  - We'll see some examples of Z3Py
  - `https://ericpony.github.io/z3py-tutorial/guide-examples.htm`
- Let's see some of the Z3 main features
  - From `https://theory.stanford.edu/~nikolaj/programmingz3.html`

Z3Py Example

```
from z3 import *
Tie, Shirt = Bools('Tie Shirt')
s = Solver()
s.add(
  Or(Tie, Shirt),
  Or(Not(Tie), Shirt),
  Or(Not(Tie), Not(Shirt))
)
print(s.check())
print(s.model())
```

- The solver check if $(Tie \vee Shirt) \wedge (\neg Tie \vee Shirt) \wedge (\neg Tie \vee \neg Shirt)$ is satisfiable, and in case prints a model
  - Is this formula satisfiable?

tie_shirt.py

Z3Py Example

```python
from z3 import *
Tie, Shirt = Bools('Tie Shirt')
s = Solver()
s.add(
  Or(Tie, Shirt),
  Or(Not(Tie), Shirt),
  Or(Not(Tie), Not(Shirt))
)
print(s.check())
print(s.model())
```

- The solver check if $(Tie \lor Shirt) \land (\neg Tie \lor Shirt) \land (\neg Tie \lor \neg Shirt)$ is satisfiable, and in case prints a model
  - Is this formula satisfiable?

```
sat
[Tie = False, Shirt = True]
```

Z3 Sorts

- Z3 handles different sorts apart from built-in `Bool`, e.g.
  - `Int`
  - `Real`
  - `BitVec`
  - `Array`
  - `String`

- Formulas are terms of `Bool` sort. They may include (un-)interpreted functions and constants. E.g., in:

```
B = BoolSort() ; Z = IntSort()
f = Function('f', B, Z) ; g = Function('g', Z, B)
a = Bool('a')
solve(g(1+f(a)))
```

  we have $f : \mathbb{B} \to \mathbb{Z}, g : \mathbb{Z} \to \mathbb{B}, a \in \mathbb{B}$, with $\mathbb{B} = \{true, false\}$ and we ask if $g(1 + f(a))$ is satisfiable

Example

```
Z = IntSort()
f = Function('f', Z, Z)
x, y, z = Ints('x y z')
A = Array('A', Z, Z)
fml = Implies(x + 2 == y,
              f(Store(A, x, 3)[y-2]) == f(y-x+1))
solve(Not(fml))
```

Here `fml` corresponds to formula:

$$x + 2 = y \implies f(write(A, x, 3)[y - 2]) = f(y - x + 1)$$

so `Not(fml)` is:

$$x + 2 = y \;\wedge\; f(write(A, x, 3)[y - 2]) \neq f(y - x + 1)$$

Is `Not(fml)` satisfiable?

not_fml.smt2

EUF theory

- Z3 supports several well-known theories. The "baseline" is the EUF theory. E.g., we can write something like:

```
S = DeclareSort('S')
a, b, c, d, e, s, t = Consts('a b c d e s t', S)
f = Function('f', S, S, S) ; g = Function('g', S, S)
solve([a == b, b == c, d == e, b == s, d == t,
       f(a, g(d)) != f(g(e), b)])
```

Is this formula satisfiable?

euf.py

EUF theory

- Z3 supports several well-known theories. The “baseline” is the EUF theory. E.g., we can write something like:

```
S = DeclareSort('S')
a, b, c, d, e, s, t = Consts('a b c d e s t', S)
f = Function('f', S, S, S) ; g = Function('g', S, S)
solve([a == b, b == c, d == e, b == s, d == t,
       f(a, g(d)) != f(g(e), b)])
```

Is this formula satisfiable? Yes. The returned model is:

```
s = S!val!0, b = S!val!0, a = S!val!0, c = S!val!0,
d = S!val!1, e = S!val!1, t = S!val!1,
f = [(S!val!2, S!val!0) -> S!val!4,
                      else -> S!val!3],
g = [else -> S!val!2]]
```

This is computed with a standard congruence closure procedure

Arithmetic theory

- Arithmetical constraints are clearly fundamental. Z3 has different procedures according to which fragment of arithmetic is used:

Logic	Description	Solver	Example
LRA	Linear Real Arithmetic	Dual Simplex [28]	$x + \frac{1}{2}y \leq 3$
LIA	Linear Integer Arithmetic	Cuts + Branch	$a + 3b \leq 3$
LIRA	Mixed Real/Integer	[7, 12, 14, 26, 28]	$x + a \geq 4$
IDL	Integer Difference Logic	Floyd-Warshall	$a - b \leq 4$
RDL	Real Difference Logic	Bellman-Ford	$x - y \leq 4$
UTVPI	Unit two-variable / inequality	Bellman-Ford	$x + y \leq 4$
NRA	Polynomial Real Arithmetic	Model based CAD [42]	$x^2 + y^2 < 1$
NIA	Non-linear Integer Arithmetic	CAD + Branch [41] Linearization [15]	$a^2 = 2$

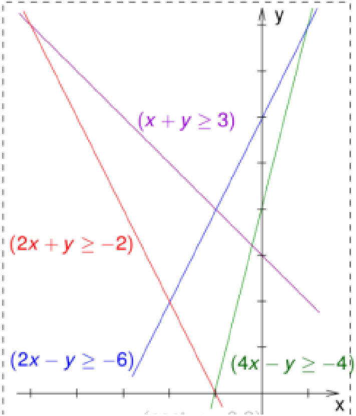
CAD = Cylindrical Algebraic Decomposition

- If we need to precisely model finite precision arithmetic, then using fixed-width bit-vectors is probably a better choice
  - Z3 handles bit-vectors with eager SAT encoding (bit-blasting)

Example

$$
\begin{aligned}
\varphi \stackrel{\text{def}}{=} \quad & (\neg A_1 \vee (2x + y \geq -2)) \\
\wedge \quad & (\ \ A_1 \vee (x + y \geq 3)) \\
\wedge \quad & (\neg A_2 \vee (4x - y \geq -4)) \\
\wedge \quad & (\ \ A_2 \vee (2x - y \geq -6))
\end{aligned}
$$

```
from z3 import *
x, y = Ints('x y')
A1, A2 = Bools('A1 A2')
s = Solver()
s.add(Or(Not(A1), 2*x + y >= -2))
s.add(Or(A1, x + y >=  3))
s.add(Or(Not(A2), 4*x - y >= -4))
s.add(Or(A2, 2*x - y >= -6))
print(s.check())
print(s.model())
```

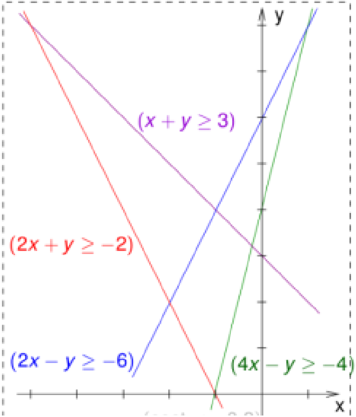


ex_lia.py

Example

$$\varphi \stackrel{\text{def}}{=} (\neg A_1 \vee (2x + y \geq -2)) \wedge (A_1 \vee (x + y \geq 3)) \wedge (\neg A_2 \vee (4x - y \geq -4)) \wedge (A_2 \vee (2x - y \geq -6))$$

```
from z3 import *
x, y = Ints('x y')
A1, A2 = Bools('A1 A2')
s = Solver()
s.add(Or(Not(A1), 2*x + y >= -2))
s.add(Or(A1, x + y >=  3))
s.add(Or(Not(A2), 4*x - y >= -4))
s.add(Or(A2, 2*x - y >= -6))
print(s.check())
print(s.model())
```




```
[A1 = True, x = -1, A2 = False, y = 1]
```

(note we are not optimizing here)

Other theories

- Z3 offers a number of other theories:
  - Arrays
    - Via reduction to EUF
  - Floating points
    - Via reduction to Bit-vectors
  - Algebraic Datatypes
    - Captures the theory of finite trees
  - String and Sequences
    - Theory of free monoids + specific operations (length, replace, . . .)

Incremental solving

- Z3 allows incremental solving via push and pop operations
  - This creates local scopes: assertions added within a “push” are retracted on the matching “pop”
  - CP solvers don’t have (yet) this capability!

```
p, q, r = Bools('p q r')
s = Solver()
s.add(Implies(p,q))
s.add(Not(q))
print(s.check())  # sat:   p->q /\ !q
s.push()
s.add(p)          # unsat: p->q /\ !q /\ p
print(s.check())
s.pop()           # sat:   p->q /\ !q
print(s.check())
```

Z3 and optimization

- Z3 enables OMT via the `Optimize` module in 2 ways:
  - By specifying an objective function
  - Via soft constraints
- The objective function is either a linear arithmetical term or a bit-vector term
- Soft constraints are assertions that the solver can ignore. The goal is maximizing the satisfied soft constraints
  - MaxSMT
- Soft constraints might have an optional weight. In this case the goal is minimizing the sum of the weights of unsatisfied constraints

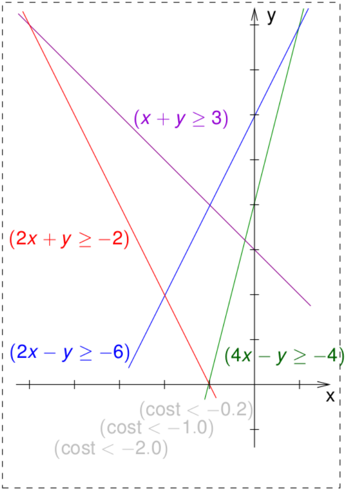
Example

[w. pure-literal filt. $\Longrightarrow$ partial assignments]

- OMT($\mathcal{LRA}$) problem:

$$
\begin{array}{rl}
\varphi \stackrel{\text{def}}{=} & (\neg A_1 \vee (2x + y \geq -2)) \\
\wedge & (\ \ A_1 \vee (x + y \geq 3)) \\
\wedge & (\neg A_2 \vee (4x - y \geq -4)) \\
\wedge & (\ \ A_2 \vee (2x - y \geq -6)) \\
\wedge & (\text{cost} < -0.2) \\
\wedge & (\text{cost} < -1.0) \\
\wedge & (\text{cost} < -2.0) \\
\text{cost} \stackrel{\text{def}}{=} & x
\end{array}
$$

- $$\mu = \left\{ \begin{array}{l} A_1, \neg A_1, \quad A_2, \neg A_2, \\ (4x - y \geq -4), \\ (x + y \geq 3), \\ (2x + y \geq -2), \\ (2x - y \geq -6) \\ (\text{cost} < -0.2) \\ (\text{cost} < -1.0) \\ (\text{cost} < -2.0) \end{array} \right\}$$



Example

```
from z3 import *
x, y = Reals('x y')
A1, A2 = Bools('A1 A2')
s = Optimize()
s.add(Or(Not(A1), 2*x + y >= -2))
s.add(Or(A1, x + y >= 3))
s.add(Or(Not(A2), 4*x - y >= -4))
s.add(Or(A2, 2*x - y >= -6))
z = s.minimize(x)
print(s.check())
print(s.model())
print(z.value())
```

omt.py

Example

```
from z3 import *
x, y = Reals('x y')
A1, A2 = Bools('A1 A2')
s = Optimize()
s.add(Or(Not(A1), 2*x + y >= -2))
s.add(Or(A1, x + y >= 3))
s.add(Or(Not(A2), 4*x - y >= -4))
s.add(Or(A2, 2*x - y >= -6))
z = s.minimize(x)
print(s.check())
print(s.model())
print(z.value())

sat
[A1 = True, y = 2, x = -2, A2 = False]
-2
```

CVC

- Many other SMT solvers exist apart from Z3
  - https://smt-comp.github.io/2023/
- A well-known family of SMT solvers is CVC* (*Cooperating Validity Checker*), a saga originated at Stanford University
  - CVC (2002)
  - CVC Lite (2004)
  - CVC3 (2007)
  - CVC4 (2011)
  - CVC5 (2022): Barbosa, H., et al. "*cvc5: A Versatile and Industrial-Strength SMT Solver.*" International Conference on Tools and Algorithms for the Construction and Analysis of Systems 2022.

CVC5

- CVC5 is a major improvement of CVC4 1.8 (final CVC4 version)
  - New APIs, theories, solvers and procedures
- It provides strong performance on industrial use cases
- It is open-source: `https://github.com/cvc5/cvc5`
  - Releases: `https://github.com/cvc5/cvc5/releases`
- It can be programmed via APIs (C++,Java,Python) or executed in interactive mode
  - Documentation: `https://cvc5.github.io/docs/cvc5-0.0.11`
  - Optimization not supported, but they're working on that: `https://arxiv.org/abs/2404.16122`

SMT-LIB

- What solver should we use? Z3? CVC5? Both? None of them?
  - Yices2, Vampire, Bitwuzla, (Opti)MathSAT, . . .
- Clearly each SOTA solvers has strengths and weaknesses, selecting the best of them for an unforeseen SMT formula is not trivial
  - Algorithm Selection problem
- Surely we shouldn't write $n > 1$ programs for solving the same formula with n solvers
  - *Model once, solve everywhere*
- Need for standardization

SMT-LIB

- SMT-LIB initiative started in 2003 to:
  - Provide rigorous descriptions of SMT theories
  - Develop and promote common languages for SMT solvers.
  - Connect developers, researchers and users of the SMT community
  - Establish and make available benchmarks for SMT solvers.
  - Collect and promote software tools useful to the SMT community
- SMT-LIB website: `https://smtlib.cs.uiowa.edu/`
- SMT-LIB 2.6 specifications: `https://link.springer.com/content/pdf/bbm%3A978-3-662-50497-0%2F1.pdf`

SMT-LIB language

- SMT-LIB uses a parenthesized prefix notation (similar to LISP)
  - Designed to be machine-readable rather than human-readable
  - E.g. `(= (+ a b) c)` or `(< (f x) (g y z))`
- 3 main components: theory declarations, logic declarations, scripts
- SMT-LIB theories are defined by sorts and functions
  - Predicates $\equiv$ `Bool`-valued functions
  - E.g., theory of Ints, Reals, ...
- SMT-LIB logic $=$ Theory declarations $+$ restrictions on formulas
  - E.g., `QF_IDL` logic is based on theory of Ints and restricts (in)equalities to be of the form $x - y \bowtie k$ with $\bowtie \in \{=, \neq, <, \leq, \geq, >\}$

Theory of integers

```
(theory Ints

 :smt-lib-version 2.6
 :smt-lib-release "2017-11-24"
 :written-by "Cesare Tinelli"
 :date "2010-04-17"
 :last-updated "2015-04-25"
 :update-history
 "Note: history only accounts for content changes, not release changes.
  2015-04-25 Updated to Version 2.5.
 "

 :sorts ((Int 0))

 :funs ((NUMERAL Int)
        (- Int Int)                  ; negation
        (- Int Int Int :left-assoc) ; subtraction
        (+ Int Int Int :left-assoc)
        (* Int Int Int :left-assoc)
        (div Int Int Int :left-assoc)
        (mod Int Int Int)
        (abs Int Int)
        (<= Int Int Bool :chainable)
        (<  Int Int Bool :chainable)
        (>= Int Int Bool :chainable)
        (>  Int Int Bool :chainable)
       )
```

Integer difference logic

```
(logic QF_IDL

 :smt-lib-version 2.6
 :smt-lib-release "2017-11-24"
 :written-by "Cesare Tinelli"
 :date "2010-04-30"
 :last-updated "2015-04-25"
 :update-history
 "Note: history only accounts for content changes, not release changes.
  2015-04-25 Updated to Version 2.5.
 "

 :theories ( Ints )

 :language
 "Closed quantifier-free formulas with atoms of the form:
  - q
  - (op (- x y) n),
  - (op (- x y) (- n)), or
  - (op x y)
  where
    - q is a variable or free constant symbol of sort Bool,
    - op is <, <=, >, >=, =, or distinct,
    - x, y are free constant symbols of sort Int,
    - n is a numeral.
 "
)
```

SMT-LIB syntax

- `set-logic` specifies the logic
  - E.g. `(set-logic QF_IDL)` or `(set-logic QF_LRA)`
- `declare-fun` introduces a new function symbol, so it can be used to declare variables too (variables $\equiv$ uninterpreted constants)
  - E.g., function `(declare-fun f (Int Int) Bool)` or variable `(declare-fun x () Real)`
  - Command `(declare-const x () Real)` is also available
- `assert` specifies formulas, and `check-sat` checks the satisfiability of all the specified formulas
  - E.g.,`(assert (= (+ a b) c))`
- Others: `(set-option :produce-models true)`, `(get-model)`, `(get-unsat-core (x))`, ...

SMT-LIB script

- This is an example of SMT-LIB script, i.e., a sequence of commands

```
(set-logic QF_LRA)
(set-option :produce-models true)
(declare-fun x () Real)
(declare-fun y () Real)
(declare-fun A1 () Bool)
(declare-fun A2 () Bool)
(assert (or (not A1) (>= (+ (* 2 x) y) (- 2))))
(assert (or A1 (>= (+ x y) 3)))
(assert (or (not A2) (>= (- (* 4 x) y) (- 4))))
(assert (or A2 (>= (- (* 2 x) y) (- 6))))
(check-sat)
(get-model)
```

- It encodes $A1 \Rightarrow 2x + y \geq -2 \;\wedge\; \neg A1 \Rightarrow x + y \geq 3 \;\wedge\; A2 \Rightarrow 4x - y \geq -4 \;\wedge\; \neg A2 \Rightarrow 2x - y \geq -6$

ex_lra.smt2

SMT-LIB script

```
$ z3 ex_lra.smt2
sat
(
  (define-fun A1 () Bool true)
  (define-fun y  () Real (/ 13.0 5.0))
  (define-fun x  () Real (- (/ 3.0 5.0)))
  (define-fun A2 () Bool false)
)

$ cvc5 ex_lra.smt2
sat
(
  (define-fun x  () Real (/ 4 3))
  (define-fun y  () Real (/ 28 3))
  (define-fun A1 () Bool false)
  (define-fun A2 () Bool true)
)
```

Assertion stack

- SMT solvers react to commands by modifying an assertion stack
- Each stack element is called level and consists of a set of assertions
  - formulas + declarations/definitions of sorts and functions
- By default a new assertion always belongs to the current level
  - In the example above, only 1 level
- Levels can be added and removed with `push` and `pop` commands
  - pop removes all level assertions, including declarations/definitions

Incremental solving

```
(declare-fun x () Real)
(declare-fun y () Real)

(push 1)
(declare-fun A () Bool)
(assert (or (not A) (>= (+ (* 2 x) y) (- 2))))
(assert (or A (>= (- (* 4 x) y) (- 4))))
(check-sat)
(get-model)
(pop 1)

(declare-fun A () Int)
(assert (or (< A 0) (>= (- (* 4 x) y) (- 4))))
(assert (or (>= A 0) (< (- (* 2 x) y) (- 6))))
(check-sat)
(get-model)
```

ex_lra2.smt2

Incremental solving

- In the above example we have 2 levels
- After `(push 1)`, one decision level including declarations for Boolean variable `A` and $\{A \Rightarrow 2x + y \geq -2, A \Rightarrow 4x - y \geq -4\}$ is pushed on assertion stack
  - Then we check for satisfiability and ask for a model
- After `(pop 1)`, the last decision level is removed from the stack, so `A` can be declared again, this time with a different sort (`Int`)
  - Then we check again for satisfiability and ask for a model

Incremental solving

```
$ cvc5 -i --produce-models ex_lra2.smt2

sat
(
(define-fun x () Real (/ (- 9) 8))
(define-fun y () Real (/ (- 1) 2))
(define-fun A () Bool false)
)
sat
(
(define-fun x () Real 0.0)
(define-fun y () Real 8.0)
(define-fun A () Int (- 1))
)
```

SMT-LIB and optimization

- Standard SMT-LIB does not have an explicit support to optimization
- One possible workaround is to implement an offline OMT approach
  - SMT solvers as black-boxes
- SMT solver should be in incremental mode avoiding to restart each time a new bound is found
- In binary search mode, one should use push/pop primitives

Offline OMT($\mathcal{LRA}$)

Algorithm 1 Offline OMT($\mathcal{LA}(\mathbb{Q})$) Procedure based on Mixed Linear/Binary Search.

```
Require: ⟨φ, cost, lb, ub⟩ {ub can be +∞, lb can be −∞}
l ← lb; u ← ub; PIV ← ⊤; M ← ∅
φ ← φ ∪ {¬(cost < l), (cost < u)}
while (l < u ) do
    if (BinSearchMode()) then {Binary-search Mode}
        pivot ← ComputePivot(l, u)
        PIV ← (cost < pivot)
        φ ← φ ∪ {PIV}
        ⟨res, μ⟩ ← SMT.IncrementalSolve(φ)
        η ← SMT.ExtractUnsatCore(φ)
    else {Linear-search Mode}
        ⟨res, μ⟩ ← SMT.IncrementalSolve(φ)
        η ← ∅
    end if
    if (res = SAT) then
        ⟨M, u⟩ ← Minimize(cost, μ)          u = current best bound
        φ ← φ ∪ {(cost < u)}
    else {res = UNSAT }
        if (PIV ∉ η) then
            l ← u                           Linear search completed
        else
            l ← pivot
            φ ← φ \ {PIV}                   Updating binary search pivot
            φ ← φ ∪ {¬PIV}
        end if
    end if
end while
return ⟨M, u⟩
```

From R. Sebastiani, S. Tomasi: *Optimization Modulo Theories with Linear Rational Costs*. ACM Trans. Comput. Log. 16(2): 12:1-12:43 (2015)

SMT-LIB $\Longleftrightarrow$ MiniZinc

- One may think to SMT-LIB as the SMT equivalent of MiniZinc language for CP problems
- SMT-LIB is actually "*lower-level*", more similar to FlatZinc language
  - MiniZinc models, together with optional data and solver-specific redefinitions, are compiled into FlatZinc instances
- Translating SMT-LIB $\rightarrow$ MiniZinc is quite straightforward, except that MiniZinc does not support all the standard SMT-LIB theories
  - E.g., theory of arrays and strings not officially supported by MiniZinc
  - Global constraints likely lost
- One can translate SMT-LIB $\rightarrow$ FlatZinc, if target solver is known or simply ignored

SMT-LIB $\Longleftrightarrow$ MiniZinc

- An early proposal to convert FlatZinc $\rightarrow$ SMT-LIB by Bofill et al. is fzn2smt, used by Yices SMT solver in MiniZinc Challenges 2010–2013
  - Based on obsolete MiniZinc versions and no longer maintained
- A prototypical converter SMT-LIB $\rightarrow$ MiniZinc called smt2mzn-str was developed by G. Gange for solving string constraints
  - String support in MiniZinc is still experimental
- Contaldo et al. proposed 2 compilers STM-LIB $\leftrightarrow$ FlatZinc called fzn2omt and omt2fzn
  - Contaldo, F. et al. "*From MiniZinc to Optimization Modulo Theories, and Back*". CPAIOR 2020.
  - They use the default MiniZinc $\rightarrow$ FlatZinc decomposition for global constraints and SMT-LIB with optimization extensions

Writing SMT-LIB

- From SMT-LIB standard: *"...Preferring ease of parsing over human readability is reasonable in this context not only because SMT-LIB benchmarks are meant to be read by solvers but also because they are produced in the first place by automated tools like verification condition generators or translators..."*

- One may write a SMT-LIB instance manually...
  - Tricky prefix notation, impractical for large instances
- ...or define an ad hoc script producing a SMT-LIB instance
  - E.g., Bash or Python script
- ...or define the instance with Z3 and then use Z3's API to generate a corresponding SMT-LIB instance
  - E.g., Z3Py + `to_smt2()` method of `Solver` class
  - Easier writing, no need to define ad hoc script
  - Translation may introduce additional variables

Z3Py Example

```python
from z3 import *
Tie, Shirt = Bools('Tie Shirt')
s = Solver()
s.add(
  Or(Tie, Shirt),
  Or(Not(Tie), Shirt),
  Or(Not(Tie), Not(Shirt))
)
print(s.to_smt2())
print(s.dimacs())
```

- `to_smt2` returns the SMT-LIB instance for solver's assertions
- `dimacs` returns the DIMACS instance for solver's assertions

tie_shirt.py

SMT-LIB output

```
; benchmark generated from python API
(set-info :status unknown)
(declare-fun Shirt () Bool)
(declare-fun Tie () Bool)
(assert
  (or Tie Shirt))
(assert
  (let (($x8 (not Tie)))
  (or $x8 Shirt)))
(assert
  (let (($x8 (not Tie)))
  (or $x8 (not Shirt))))
(check-sat)
```

- Note the introduction of $x8, a local variable defined through the let binder

DIMACS output

```
p cnf 2 3
1 2 0
-1 2 0
-1 -2 0
c 1 Tie
c 2 Shirt
```

- Header `p cnf 2 3` means CNF formula with 2 variables and 3 clauses
- Follows one clause per line, terminated with `0`
  - `1 2 0` is the clause $x_1 \vee x_2$
  - `-1 2 0` is the clause $\neg x_1 \vee x_2$
  - `-1 -2 0` is the clause $\neg x_1 \vee \neg x_2$
- Each line that begins with `c` is a comment

Exercises

- CVC5 + IDL Theory project: write C++ code
  `https://github.com/cvc5/cvc5/blob/idl-lab/project.md`
- Define SMT-LIB specifications for well-known NP-Hard problems (subset-sum, knapsack, TSP, ...) and solve with different SMT solvers
- Take some MiniZinc models and manually translate to SMT-LIB
  `https://github.com/MiniZinc/minizinc-benchmarks`
  - compare with fzn2omt translation
    `https://github.com/PatrickTrentin88/fzn2omt`

Resources

- Z3 solver
  - *Programming Z3*. N. Bjørner, L. de Moura, L. Nachmanson, and C. Wintersteiger
    https://theory.stanford.edu/~nikolaj/programmingz3.html
  - https://ericpony.github.io/z3py-tutorial/guide-examples.htm
- CVC5 solver
  - https://cvc5.github.io/
  - https://github.com/cvc5/cvc5
- SMT-LIB initiative
  - https://smtlib.cs.uiowa.edu/
  - https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf