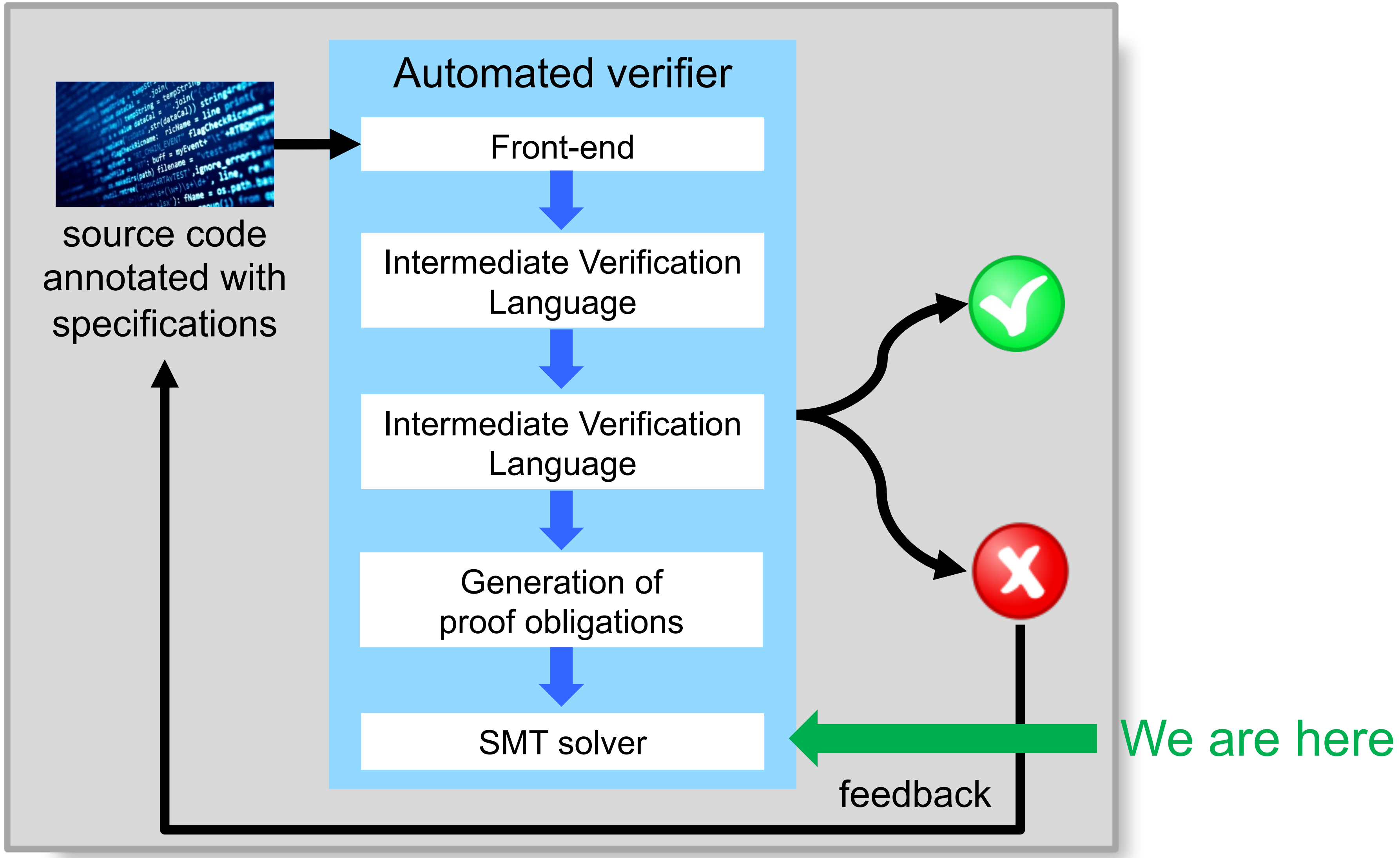
INTRODUCTION TO SMT

(slides developed in cooperation with Christoph Matheja)

Last week

- Introduction to SAT
- DNF, CNF, Tseitin’s algorithm
- minisat demo
- Examples of problems that can be encoded as SAT

This week: foundations of Dafny



Automating program verification

Main steps of a tool for automatically verifying $\models \{\, \mathbf{A} \,\}\ \mathtt{S}\ \{\, \mathbf{B} \,\}$

1. Compute *weakest preconditions* for B under S: $wp\,[\![\,\mathtt{S}\,]\!]\,(\mathbf{B})$

2. Decide $\mathbf{A} \Rightarrow wp\,[\![\,\mathtt{S}\,]\!]\,(\mathbf{B})$ $\rightarrow$ We employ an SMT solver

SMT solvers

1. Propositional logic and satisfiability solvers

2. Using Z3 as a SAT solver

3. First-order logic and SMT solvers

4. Using Z3 as an SMT solver

Propositional logic

Syntax of propositional logic

taken from a set **Var** of variables

$\mathbf{F} ::= x \mid \texttt{true} \mid \texttt{false} \mid \neg\mathbf{F} \mid \mathbf{F} \wedge \mathbf{F} \mid \mathbf{F} \vee \mathbf{F} \mid \mathbf{F} \Rightarrow \mathbf{F} \mid \mathbf{F} \Leftrightarrow \mathbf{F}$

binds stronger

Syntactic sugar:

$\texttt{false} ::= x \wedge \neg x \qquad \texttt{true} ::= \neg\texttt{false}$

$\mathbf{A} \vee \mathbf{B} ::= \neg(\neg\mathbf{A} \wedge \neg\mathbf{B}) \qquad \mathbf{A} \Rightarrow \mathbf{B} ::= \neg\mathbf{A} \vee \mathbf{B}$

$\mathbf{A} \Leftrightarrow \mathbf{B} ::= (\mathbf{A} \Rightarrow \mathbf{B}) \wedge (\mathbf{B} \Rightarrow \mathbf{A})$

Interpretation: $\mu\colon \mathbf{Var} \to \{\texttt{true}, \texttt{false}\}$

μ is a model of **F** iff μ satisfies **F**

Satisfaction relation $\models$

$\mu \models x$	iff	$\mu(x) = \texttt{true}$
$\mu \models \neg\mathbf{A}$	iff	$\mu \not\models \mathbf{A}$
$\mu \models \mathbf{A} \wedge \mathbf{B}$	iff	$\mu \models \mathbf{A}$ and $\mu \models \mathbf{B}$

$\mu = [x = \texttt{false}, y = \texttt{true}]$

$\mu \models x \wedge (x \Rightarrow y) \Rightarrow y$

$\mu \not\models x \wedge (x \Rightarrow y) \Leftrightarrow y$

$\mu \models x \wedge (x \Rightarrow y) \Leftrightarrow x \wedge y$

Satisfiability and validity

- **F** is satisfiable iff **F** has some model $(x \Rightarrow y) \Rightarrow y$
- **F** is unsatisfiable iff **F** has no model $x \wedge \neg y \wedge (x \Rightarrow y)$
- **F** is valid iff every interpretation is a model of **F** ($\neg$**F** is unsatisfiable) $x \wedge (x \Rightarrow y) \Rightarrow y$
- **F** is not valid iff some interpretation is not a model of **F** ($\neg$**F** is satisfiable) $x \wedge (x \Rightarrow y) \Leftrightarrow y$

The satisfiability problem

- A formula is satisfiable if it has a model

$$
\begin{aligned}
& (x_1 \vee x_2 \vee \neg x_3) \\
\wedge\ & (x_5 \vee \neg x_2) \\
\wedge\ & (\neg x_1 \vee \neg x_3 \vee x_4 \vee \neg x_5)
\end{aligned}
$$

- Satisfiability (SAT) problem:
  Given a propositional logic formula,
  decide whether it is satisfiable

- If yes, ideally also provide a witness

$$
\begin{aligned}
\mu \;=\; [\, & x_1 = \texttt{true}, x_2 = \texttt{true}, \\
& x_3 = \texttt{true}, x_4 = \texttt{true}, x_5 = \texttt{true} \,]
\end{aligned}
$$

Complexity of SAT

- For formulas in conjunctive normal form (CNF), SAT is the classical NP-complete problem
  - Many difficult problems can be efficiently encoded
  - Every known algorithm is exponential in the formula's size

$$\bigwedge_i \bigvee_j C_{i,j} \text{ where } C_{i,j} \in \{x_{i,j}, \neg x_{i,j}\}$$

- Modern SAT solvers are extremely efficient in practice
  - Scale to formulas with millions of variables
  - May still perform poorly on certain formulas

Exercise: placement of wedding guests

Model the following problem as a SAT problem:

Consider three chairs in a row: left, middle, right. Can we assign chairs to Alice, Bob, and Charlie such that:

- Alice does not sit next to Charlie,
- Alice does not sit on the leftmost chair, and
- Bob does not sit to the right of Charlie?

Solution: placement of wedding guests

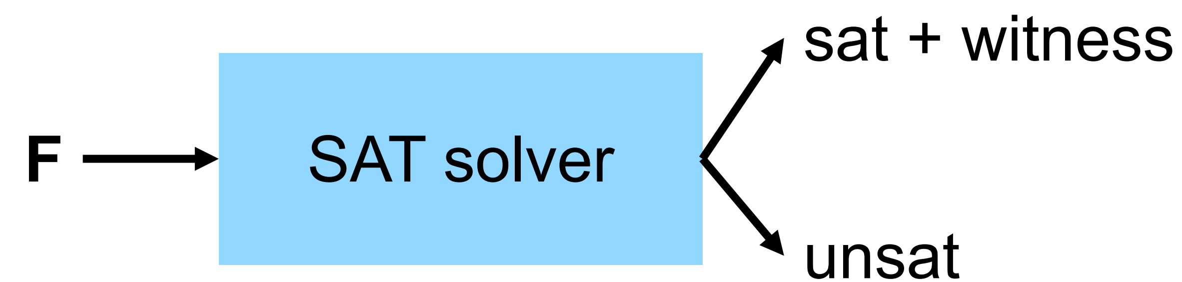
- Model assignment via nine boolean variables $x_{p,c}$: "person p sits in chair c"
- Alice does not sit next to Charlie $(x_{A,l} \vee x_{A,r} \Rightarrow \neg x_{C,m}) \wedge (x_{A,m} \Rightarrow \neg x_{C,l} \wedge \neg x_{C,r})$
- Alice does not sit on the leftmost chair $\neg x_{A,l}$
- Bob does not sit to the right of Charlie $(x_{C,l} \Rightarrow \neg x_{B,m}) \wedge (x_{C,m} \Rightarrow \neg x_{B,r})$
- Each person gets a chair $\bigwedge_{1 \leq p \leq 3} \bigvee_{1 \leq c \leq 3} x_{p,c}$
- Every person gets at most one chair $\bigwedge_{1 \leq p \leq 3} \bigwedge_{1 \leq c,d \leq 3, c \neq d} (\neg x_{p,c} \vee \neg x_{p,d})$
- Every chair gets at most one person $\bigwedge_{1 \leq p,q \leq 3, p \neq q} \bigwedge_{1 \leq c \leq 3} (\neg x_{p,c} \vee \neg x_{q,c})$

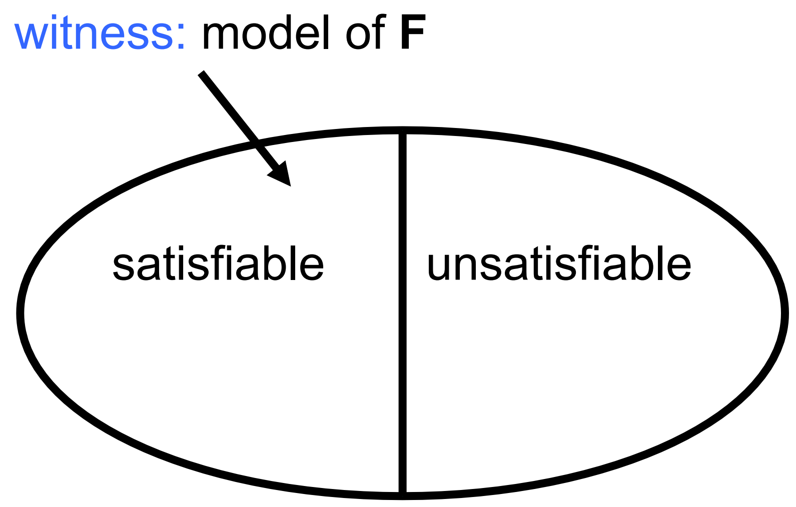
SMT solvers

1. Propositional logic and satisfiability solvers
2. Using Z3 as a SAT solver
3. First-order logic and SMT solvers
4. Using Z3 as an SMT solver

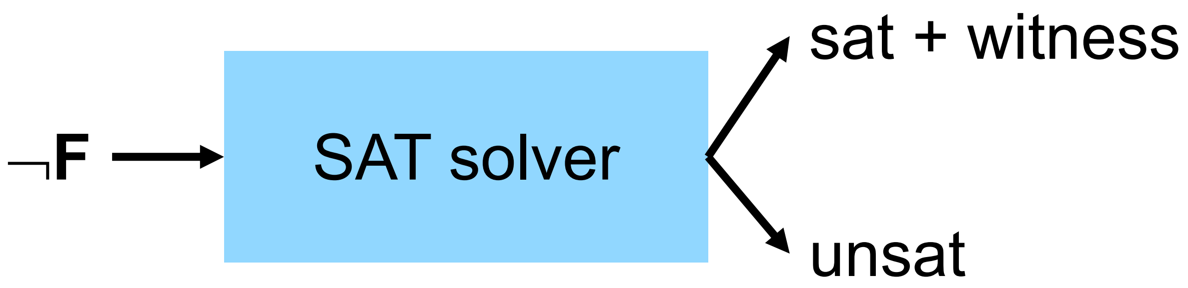
Using a SAT solver

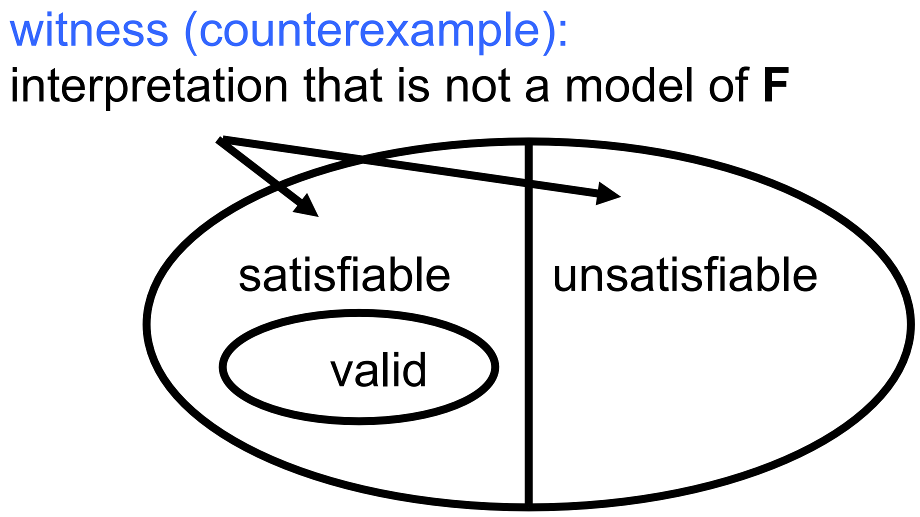
- Is **F** satisfiable?





- Is **F** valid?





The Z3 Satisfiability Modulo Theories solver

- Developed by Microsoft (under MIT license)
- Building block of many verification tools including Viper
- Various input formats and APIs
  - Z3, SMTLIB-2, C, C++, Python, Java, OCaml, ...
- For now: Use Z3 as a SAT solver
- Tutorial: https://ericpony.github.io/z3py-tutorial/guide-examples.htm

A first example in Z3

```
from z3 import *

# declare variables
x = Bool('x')
y = Bool('y')

# define formula: x ⇒ y
F = Implies(x, y)

# print the formula
print(F)

# find a model for F
solve(F)

# find a counterexample for F
solve(Not(F))
```

F is satisfiable, this is a model

```
Implies(x, y)
[y = False, x = False]
[y = False, x = True]
```

F is not valid, this is a counterexample

A valid formula example in Z3

```
from z3 import *

# declare variables
x = Bool('x')
y = Bool('y')

# define formula: ¬(x ∧ y) ⇔ ¬x ∨ ¬y
F = Not(And(x, y)) == Or(Not(x), Not(y))

# print the formula
print(F)

# find a model for F
solve(F)

# find a counterexample for F
solve(Not(F))
```

F is satisfiable, all interpretations are models

```
Not(And(x, y)) == Or(Not(x), Not(y))
[]
no solution
```

F is valid, no interpretation is a counterexample

A more complex example in Z3

```
from z3 import *

# declare multiple variables
x, y = Bools('x y’)

# create a solver instance
s = Solver()

# add conjuncts
s.add( Implies(x, y) )
s.add( Implies(y, x) )

# check satisfiability
print( s.check() )
print( s.model() )

s.add( x )
s.add( Not(y) )

# check satisfiability
print( s.check() )
```

The first two conjuncts are satisfiable, we get a model

```
sat
[y = False, x = False]
unsat
```

All four conjuncts together are unsatisfiable

Exercise: placement of wedding guests in Z3

Encode the placement of wedding guests in Z3.

- Model assignment via nine boolean variables $x_{p,c}$: "person p sits in chair c"
- Alice does not sit next to Charlie $(x_{A,l} \vee x_{A,r} \Rightarrow \neg x_{C,m}) \wedge (x_{A,m} \Rightarrow \neg x_{C,l} \wedge \neg x_{C,r})$
- Alice does not sit on the leftmost chair $\neg x_{A,l}$
- Bob does not sit to the right of Charlie $(x_{C,l} \Rightarrow \neg x_{B,m}) \wedge (x_{C,m} \Rightarrow \neg x_{B,r})$
- Each person gets a chair $\bigwedge_{1 \leq p \leq 3} \bigvee_{1 \leq c \leq 3} x_{p,c}$
- Every person gets at most one chair $\bigwedge_{1 \leq p \leq 3} \bigwedge_{1 \leq c,d \leq 3, c \neq d} (\neg x_{p,c} \vee \neg x_{p,d})$
- Every chair gets at most one person $\bigwedge_{1 \leq p,q \leq 3, p \neq q} \bigwedge_{1 \leq c \leq 3} (\neg x_{p,c} \vee \neg x_{q,c})$

Using a SAT solver to verify a program

```
{ true }
// Check that this entailment is valid (negation is unsatisfiable)
{ (a ⇒ (b ⇒ (true ⇔ (a ⇒ b))) ∧ (¬b ⇒ (false ⇔ (a ⇒ b)))) ∨ (¬a ⇒ (true ⇔ (a ⇒ b))  }
if (a) {
{ (b ⇒ (true ⇔ (a ⇒ b))) ∧ (¬b ⇒ (false ⇔ (a ⇒ b))) }
  if (b) {
{ true ⇔ (a ⇒ b) }
    res := true
{ res ⇔ (a ⇒ b) }
  } else {
{ false ⇔ (a ⇒ b) }
    res := false
{ res ⇔ (a ⇒ b) }
  }
{ res ⇔ (a ⇒ b) }
} else {
{ true ⇔ (a ⇒ b) }
  res := true
{ res ⇔ (a ⇒ b) }
}
{ res ⇔ (a ⇒ b) }
```

Propositional logic is not enough!

- What about this entailment?

```
{ a = 1 ∧ 0 ≤ b*b – 4*c }
// Check that this entailment is valid
{ b*b – 4*a*c < 0 ∧ false ∨
  ¬(b*b – 4*a*c < 0) ∧ a*((-b + √(b*b – 4*a*c)) / 2)² + b*((-b + √(b*b – 4*a*c)) / 2) + c = 0 }
```

- Entailment is not in propositional logic
  - Real-valued variables (a, b, c) and numeric constants
  - Arithmetic operations (+, -, *, /, ^{2}, √) and comparisons (=, <, ≤)
- Logic must support at least the expressions appearing in programs
  - It is also useful to support quantifiers (e.g., for array algorithms)
- General framework: first-order predicate logic (FO) over suitable theories

SMT solvers

1. Propositional logic and satisfiability solvers
2. Using Z3 as a SAT solver
3. First-order logic and SMT solvers
4. Using Z3 as an SMT solver

First-order (FO) predicate logic

FO logic is a framework with three syntactical ingredients:

1. Logical symbols — $\wedge,\ \vee,\ \neg,\ \Rightarrow,\ \Leftrightarrow,\ \exists,\ \forall\ \ldots$
2. Theory symbols
   variables, constant symbols, function symbols — $x,\ y,\ z,\ldots\quad 0,\ 1,\ldots\quad +,\ -,\ *,\ldots$
3. Predicate symbols — $<, =, \ldots$
   bridge from theories to logic

Special case: a sort identifies a non-empty set S with a unary predicate symbol interpreted as membership in S — $\texttt{Bool}, \texttt{Int}, \texttt{Real}, \ldots$

Terms are constructed from theory symbols — $x, \quad 0, \quad 0 + x - y + 1$

Constraints lift terms to the logical level via predicates — $x + y < 1 + z - 0, \quad \texttt{Int}(1 + x)$

A signature Σ collects all constants, functions, and predicates — $\Sigma \ = \ \{\texttt{Int}, 0, 1, +, *, <\}$
assumption: Σ contains at least one sort

A Σ-formula is a logical formula over constraints — $\forall x\, \exists y\, (y * y = x * x + (1 + 1) * x + 1)$

Exercise: satisfiability of FO formulas

Is $\forall x \exists y \; (y = x + 1 \land y * y = x * x + (1 + 1) * x + 1)$ satisfiable?

Solution: satisfiability of FO formulas

Is $\forall x \exists y\ (y = x + 1 \land y * y = x * x + (1+1) * x + 1)$ satisfiable?

Yes, if

- the theory symbols $1,\ +,\ ,\ *, =$ have the usual interpretation

No, if

- 1 actually means 2, or
- addition is interpreted as maximum

Satisfiability of FO formulas depends on the $\underbrace{\text{admissible}}_{\text{determined by “theories”}}$ $\underbrace{\text{interpretations of theory symbols}}_{\text{determined by “structures”}}$

Semantics of FO

Let D denote the union of the sets of all sorts in signature Σ

$\Sigma = \{\texttt{Int}, 0, 1, +, =\}$ $D = \texttt{Int}$

A Σ-structure μ interprets the theory symbols in Σ by mapping:

- each free variable (those not bound by a quantifier) to an element in D
- each constant to an element in D
- each n-*ary* function symbol to a function of type $D^n \to D$
- each n-ary predicate symbol to a predicate of type $D^n \to \{\texttt{true}, \texttt{false}\}$

$\mu(0) = 0 \quad \mu(1) = 1$

$\mu(+)\colon \texttt{Int} \times \texttt{Int} \to \texttt{Int}$

$(a, b) \mapsto a + b$

$\mu(=)\colon \texttt{Int} \times \texttt{Int} \to \{\texttt{true}, \texttt{false}\}$

$(a, b) \mapsto a = b$

Satisfaction relation for Σ-formulas

$\mu \models \mathit{pred}(t_1, \ldots, t_n)$	iff	$\mu(\mathit{pred})(\mu(t_1), \ldots, \mu(t_1)) = \texttt{true}$
$\mu \models \exists x \mathbf{A}$	iff	for some $d \in D$, $\mu[x := d] \models \mathbf{A}$
$\mu \models \forall x \mathbf{A}$	iff	for every $d \in D$, $\mu[x := d] \models \mathbf{A}$
$\mu \models \mathbf{A} \land \mathbf{B}$	iff	$\mu \models \mathbf{A}$ and $\mu \models \mathbf{B}$
	$\vdots$	

$$\mu \models \forall x\, \exists y\, (\\ y * y = x * x + (1 + 1) * x + 1 \\ )$$

Satisfiability Modulo Theories

- A sentence is a formula without free variables
- An axiomatic system **AX** is a set of Σ-sentences
- The Σ-theory $\mathcal{T}$ given by **AX** is the set of all Σ-sentences inferable from **AX**

A Σ-formula **F** is $\mathcal{T}$-satisfiable iff
there exists a Σ-structure μ such that
- $\mu \models$ **F**, and
- $\mu \models$ **A** holds for every sentence **A** $\in \mathcal{T}$.

A Σ-formula **F** is $\mathcal{T}$-valid iff
for all Σ-structures μ,
(for all **A** $\in \mathcal{T}$, $\mu \models$ **A**)
implies $\mu \models$ **F** .

Exercise: satisfiability and validity

$$\Sigma = \{\texttt{Nat},\ zero,\ one,\ \oplus,\ \equiv\}$$

$\mathcal{T}$ is given by the axioms:

$$\forall x\ (x \equiv x) \qquad \forall x\ \forall y\ (x \oplus y \equiv y \oplus x)$$

$$\textbf{F} ::= \exists x\ (x \oplus zero \equiv one)$$

Is **F** $\mathcal{T}$-satisfiable?

Is **F** $\mathcal{T}$-valid?

Solution: satisfiability and validity

$$\Sigma = \{\,\mathtt{Nat},\ \mathit{zero},\ \mathit{one},\ \oplus,\ \equiv\,\}$$

$\mathcal{T}$ is given by the axioms:

$$\forall x\ (x \equiv x) \qquad \forall x\ \forall y\ (x \oplus y \equiv y \oplus x)$$

$$\mathbf{F} \ ::= \ \exists x\ (x \oplus \mathit{zero} \equiv \mathit{one})$$

Is **F** $\mathcal{T}$-satisfiable?

Is **F** $\mathcal{T}$-valid?

✔ $\mu(x) = 1$

$\mu(\mathit{zero}) = 0 \qquad \mu(\mathit{one}) = 1$

$\mu(\oplus)$: addition

$\mu(\equiv)$: equality

✘ $\mu(\mathit{zero}) = 1$ and $\mu(\mathit{one}) = 0$

✔ after adding an axiom

$$\forall x\ (x \oplus \mathit{zero} \equiv x)$$

Some important theories

- Arithmetic (with canonical axioms)
  - Presburger arithmetic: $\Sigma = \{\texttt{Int}, 0, 1, +, <\}$ decidable
  - Peano arithmetic: $\Sigma = \{\texttt{Int}, 0, 1, +, *, <\}$ undecidable
  - Real arithmetic: $\Sigma = \{\texttt{Real}, 0, 1, +, *, <\}$ decidable

- Equality logic with uninterpreted functions (EUF) decidable
  - $\Sigma = \{\mathsf{U}, =, f_1, f_2, \dots\}$
  - arbitrary universe U (no specific sort)
  - axioms ensure that $=$ is an equivalence relation (reflexive, symmetric, transitive)
  - arbitrary number of uninterpreted function symbols of any arity

- We typically need a combination of multiple theories
  - Example: Presburger arithmetic + uninterpreted functions
  - Program verification: theories for modeling different data types

SMT solvers

1. Propositional logic and satisfiability solvers

2. Using Z3 as a SAT solver

3. First-order logic and SMT solvers

4. Using Z3 as an SMT solver

Using theories

- Sorts (beyond Bool)
  - `Int`, `Real`, `BitVec(precision)`
  - `DeclareSort(name)` (uninterpreted)
- Variables are syntactic sugar for uninterpreted constants
  - `Const(name, sort)`
- Uninterpreted functions are declared with parameter and result types
- We will discuss quantifiers later

```
from z3 import *

Pair = DeclareSort('Pair')

null = Const('null', Pair)

cons = Function('cons', IntSort(), IntSort(), Pair)
first = Function('first', Pair, IntSort())

ax1 = (null == cons(0, 0))
x, y = Ints('x y')
ax2 = ForAll([x, y], first(cons(x, y)) == x)

s = Solver()
s.add(ax1)
s.add(ax2)

F = first(null) == 0

# check validity
s.add(Not(F))
print( s.check() )
```

Using an SMT solver to verify a program

```
{ a = 1 ∧ 0 ≤ b*b – 4*c }
// Check that this entailment is valid (its negation is unsatisfiable)
{ b*b – 4*a*c < 0 ∧ false ∨
  ¬(b*b – 4*a*c < 0) ∧ a*((-b + √b*b – 4*a*c) / 2)² + b*((-b + √b*b – 4*a*c) / 2) + c = 0 }
```

```
from z3 import *

a, b, c = Reals('a b c')
d = b*b - 4*a*c

PO = Implies(
       And(a == 1, 0 <= b*b - 4*c),
       Or( And(d < 0, False),
           And(Not(d < 0),
               a*((-b + Sqrt(d))/2)*((-b + Sqrt(d))/2) + b*((-b + Sqrt(d))/2) + c == 0
      )))

# check validity
s = Solver()
s.add(Not(PO)); print( s.check() )
```

Some important theories

Linear integer/real arithmetic

$$19 * x + 2 * y = 42$$

- (Unbounded) arithmetic is often used to approximate int and float
- Multiplication by constants is supported

Non-linear integer/real arithmetic

$$x * y + 2 * x * y + 1 = (x + y) * (x + y)$$

- Useful for programs that perform multiplication and division, e.g., crypto libraries

Equality logic with uninterpreted functions

$$(x = y \land u = v) \Rightarrow f(x, u) = f(y, v)$$

- Universal mechanism to encode operations not natively supported by a theory

Fixed-size bitvector arithmetic

$$x \,\&\, y \quad \leq \quad x \mid y$$

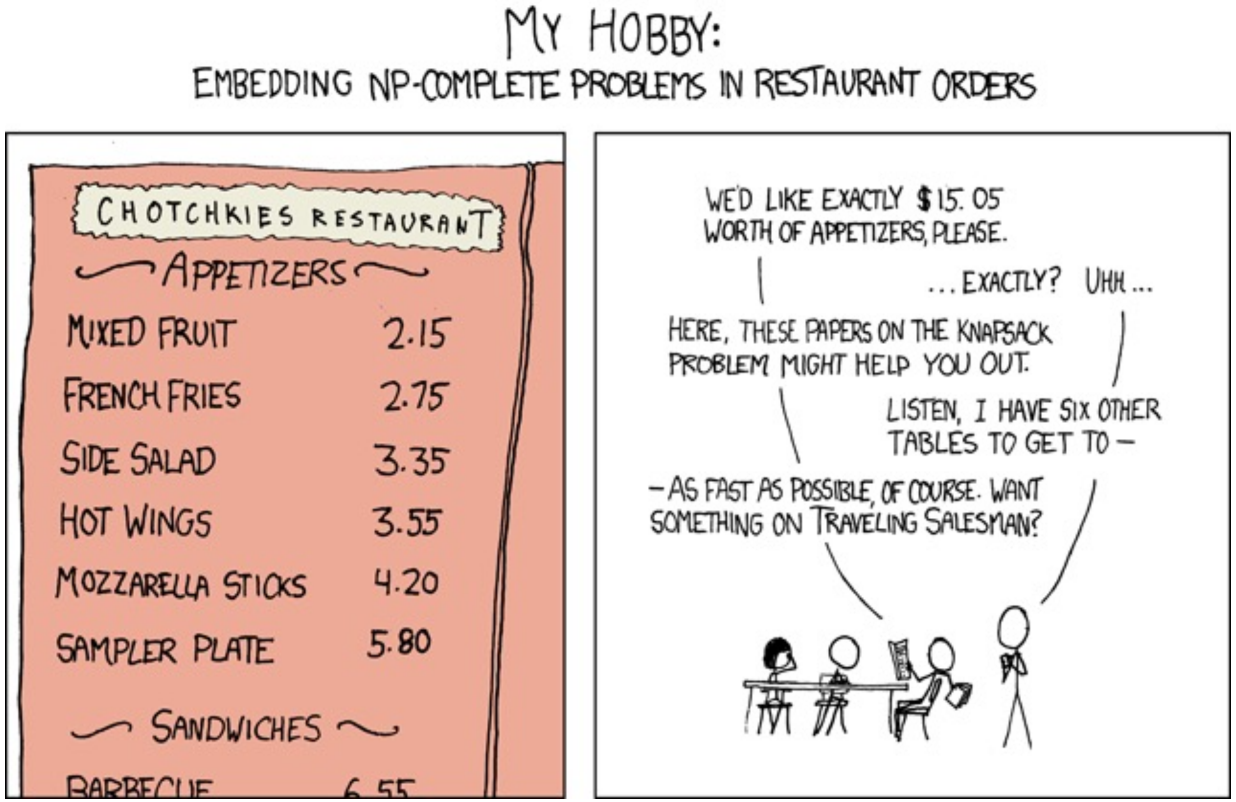
- To encode bit-level operations
- To perform bit-precise reasoning, e.g., floats

Array theory

$$\textit{read}(\textit{write}(a, i, v), i) = v$$

- To encode data types such as arrays

Example: encoding hard problems to SMT



https://xkcd.com/287/

How do we model this as an SMT query?

Theory reasoning

- Z3 selects theories based on the features appearing in formulas
  - Most verification problems require a combination of many theories

> Quantifier-free linear integer arithmetic with uninterpreted functions
>
> $$17 * x + 23 * f(y) > x + y + 42$$

- Some theories are decideable, e.g., quantifier-free linear arithmetic
  - SMT solver will terminate and report either “sat” or “unsat”

- Some theories are undecideable, e.g., nonlinear integer arithmetic
  - Especially in combination with quantifiers
  - SMT solver uses heuristics and may not terminate or return “unknown”
  - Results can be flaky, e.g., depend on order of declarations or random seeds

Working with quantifiers is non-trivial

```
from z3 import *
s = Solver()

x = Real('x')
f = Function('f', RealSort(), RealSort())

s.add(
  ForAll(x, Implies(x >= 0, f(x) * f(x) == x))
)

s.add(x > 0)
s.add( Sqrt(x) == f(x))

print(s.check())
```

```
$ python ...
unknown
```

Exercise: the N-queens problem

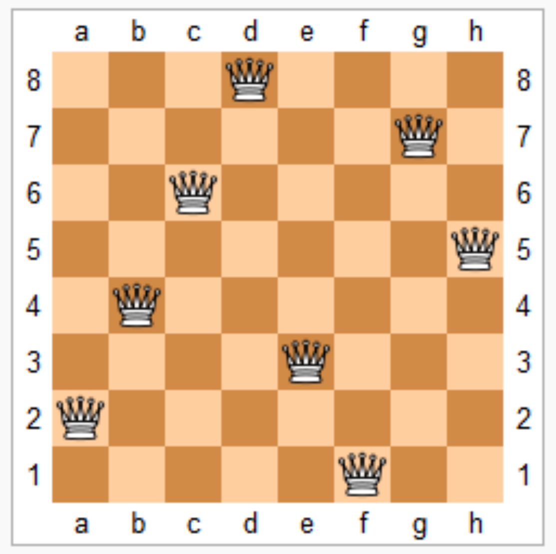
The N-queens problem is to place N-queens on an N x N chess board such that no two queens threaten each other.

Let’s use Z3 to compute a solution to the N-queens problem for any given N.

Hints:

- Represent the board as a list of N integers: `IntVector(‘board’, N)`. `board[i]` gives the row of the queen in column `i`.
- `Distinct(l)` is a Z3-constraint that expresses that all elements in list `l` are disjoint.
- You can easily check the diagonals by shifting the queens vertically and then checking the rows.

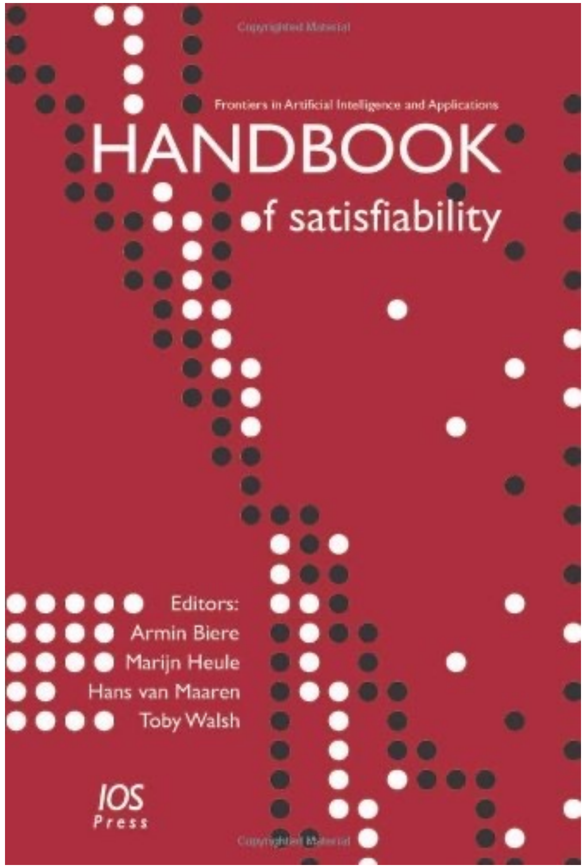
Extend your encoding to find all solutions. How many are there?



`[2, 4, 6, 8, 3, 1, 7, 5]`

More background on SAT solvers

- DPLL: Davis-Putnam-Logemann-Loveland Algorithm
  - A machine program for theorem-proving. Martin Davis, George Logemann, and Donald Loveland. 1962.
- CDCL: Conflict-Driven Clause Learning Algorithm
  - GRASP – A New Search Algorithm for Satisfiability. João P. Marques Silva and Karem A. Sakallah. 1996.
- Further developments
  - Chaff: engineering an efficient SAT solver. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001.
  - SAT-solving in practice. Koen Claessen, Niklas Een, Mary Sheeran, Niklas Sörensson. 2008.
- Annual SAT competition:
  - http://www.satcompetition.org/



More background on SMT solvers

- http://www.decision-procedures.org/ (website of book)

- Programming Z3, Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, Christoph M. Wintersteiger, 2018

- SMT-LIB standard

- Other teaching material
  - SMT solvers: Theory and Implementation. Leonardo de Moura
  - SMT Solvers: Theory and Practice. Clark Barrett
  - Satisfiability Checking, Erika Ábrahám