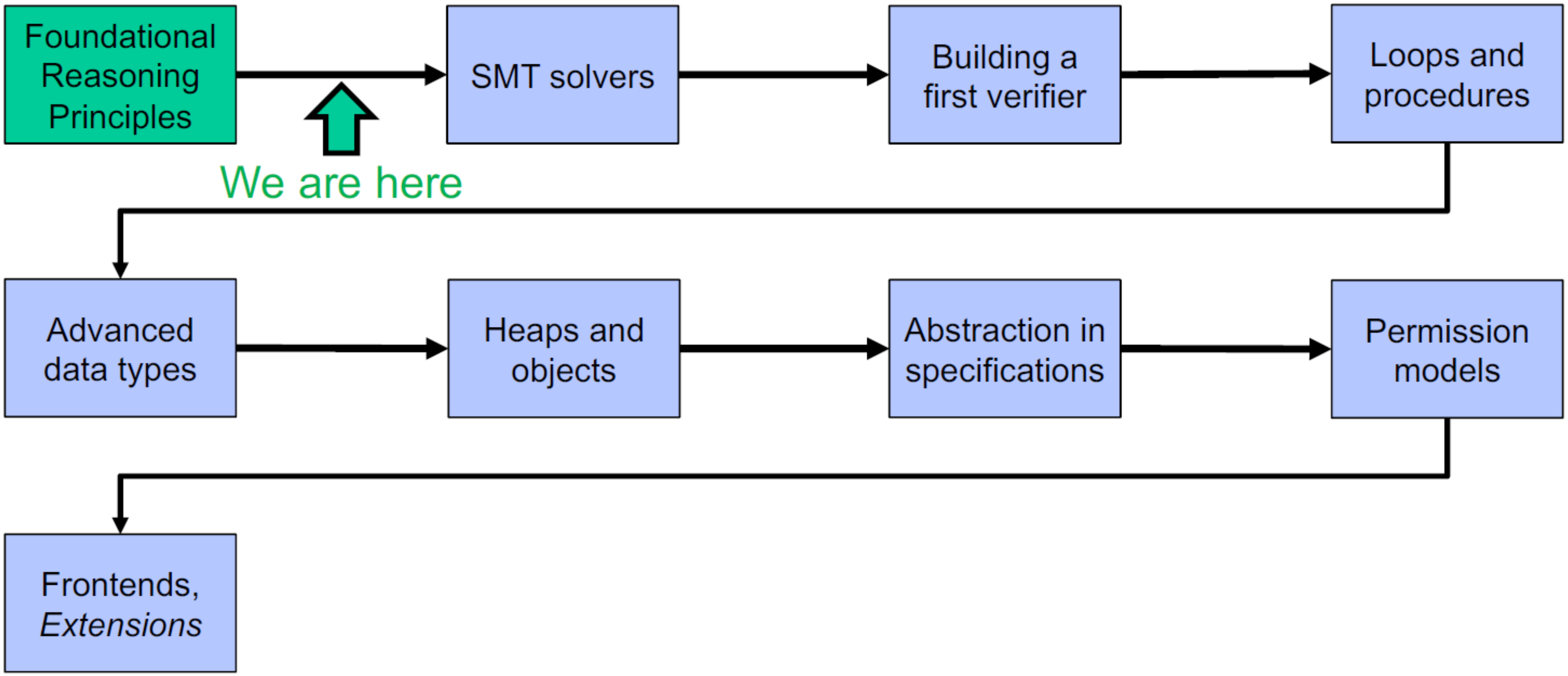
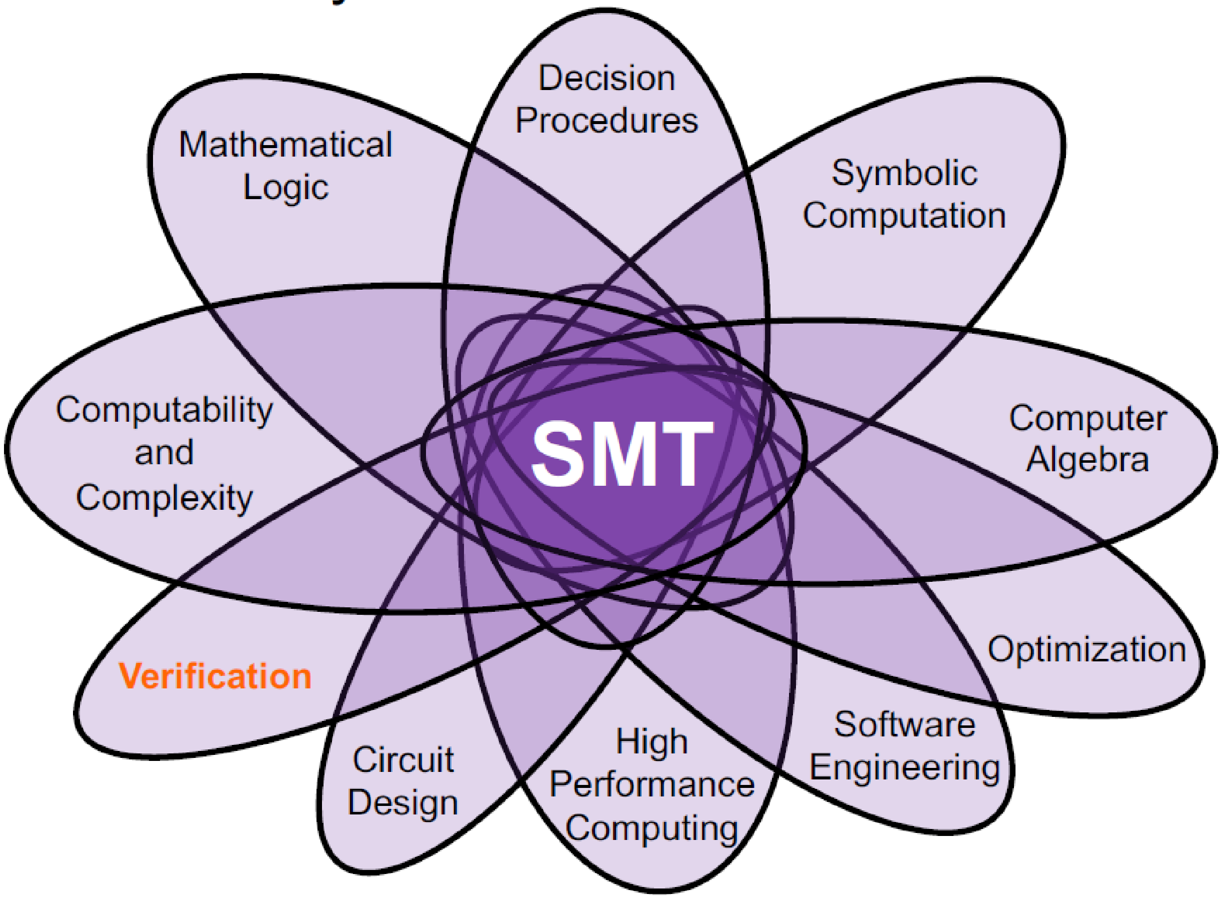
Methodologies for Software Processes

Lecture 3

Tentative course outline



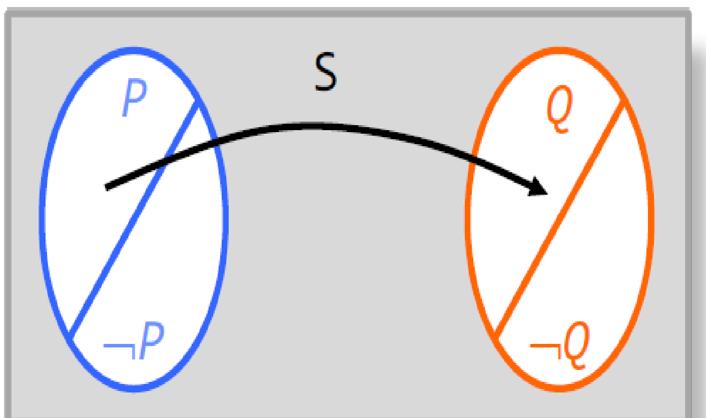
Satisfiability Modulo Theories Solvers



- A foundational topic in theoretical and applied computer science
- Our focus: effectively applying SMT technology to program verification

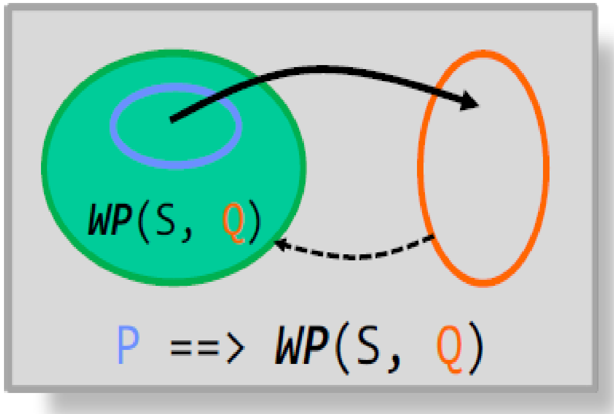
But first: Recap

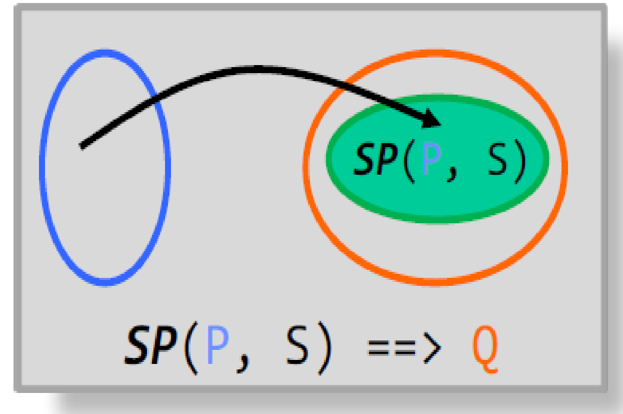
The **Floyd-Hoare triple** { P } S { Q } is **valid** if and only if
every execution of S that starts in a state satisfying precondition P
terminates without an error in a state satisfying postcondition Q.




```
method foo(x: Int)
  returns (r: Int)
  requires x > 0
  ensures r > y
{
  // S
  var y: Int := 7
  r := x + y
}
```

Recap: Weakest Pre & Strongest Post





S	*WP*(S, Q) (total correctness)	SP(P, S) (partial correctness: accepts errors/divergence)
var x	**forall** x :: Q	**exists** x :: Q
x := a	Q[x / a]	**exists** x0 :: P[x / x0] && x == a[x / x0]
assert R	R && Q	P && R
assume R	R ==> Q	P && R
S1; S2	*WP*(S1, *WP*(S2, Q))	*SP*(*SP*(P, S1), S2)
S1 [] S2	*WP*(S1, Q) && *WP*(S2, Q)	*SP*(P, S1) || *SP*(P, S2)

Automating Program Verification

Mains steps of a tool for checking that `{ P } S { Q }` is valid:

1. Compute *WP*(S, Q) → last lecture

2. Check whether *P* ==> *WP*(S, Q) is valid → delegate to SMT solver

Alternative approach

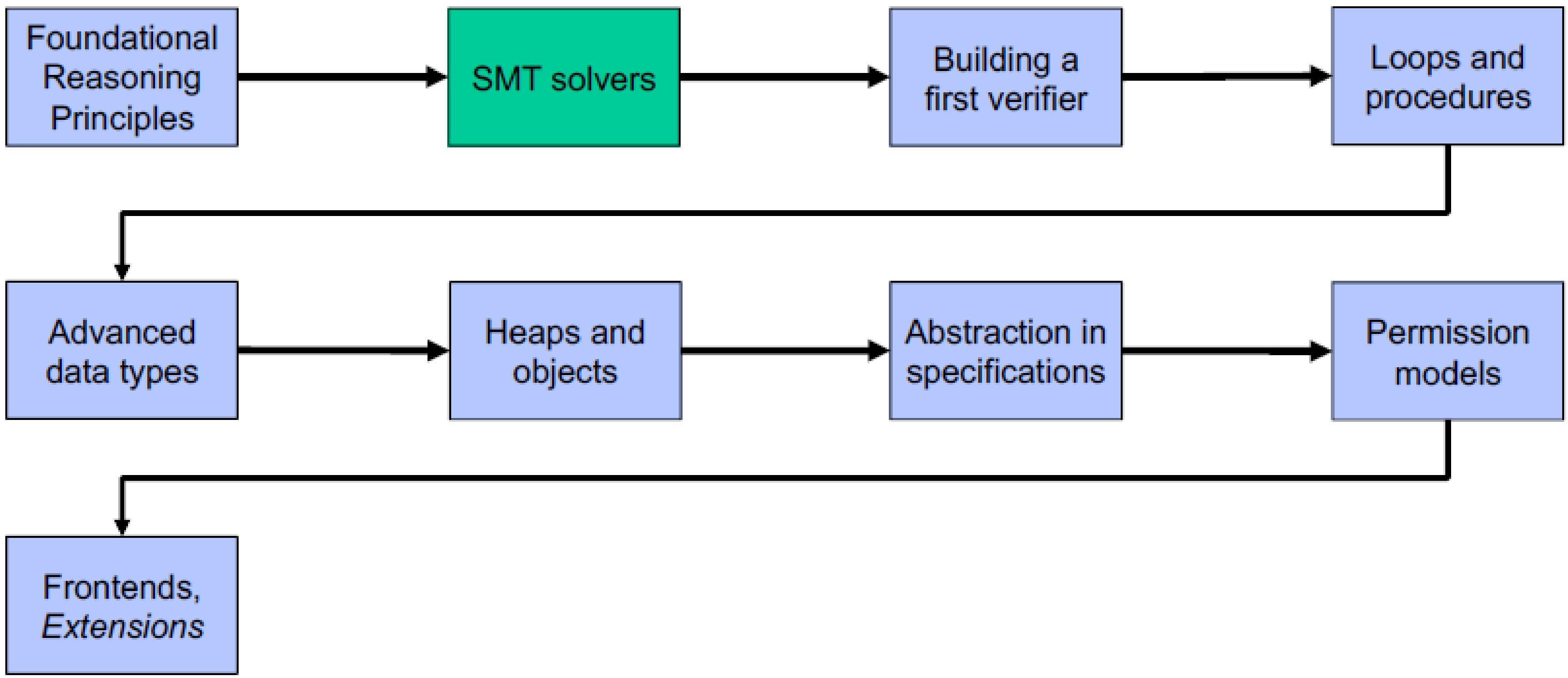
Mains steps of a tool for checking that `{ P } S { Q }` is valid:

1. Compute ***SP***`(P, S)` and ***SAFE***`(P, S)`

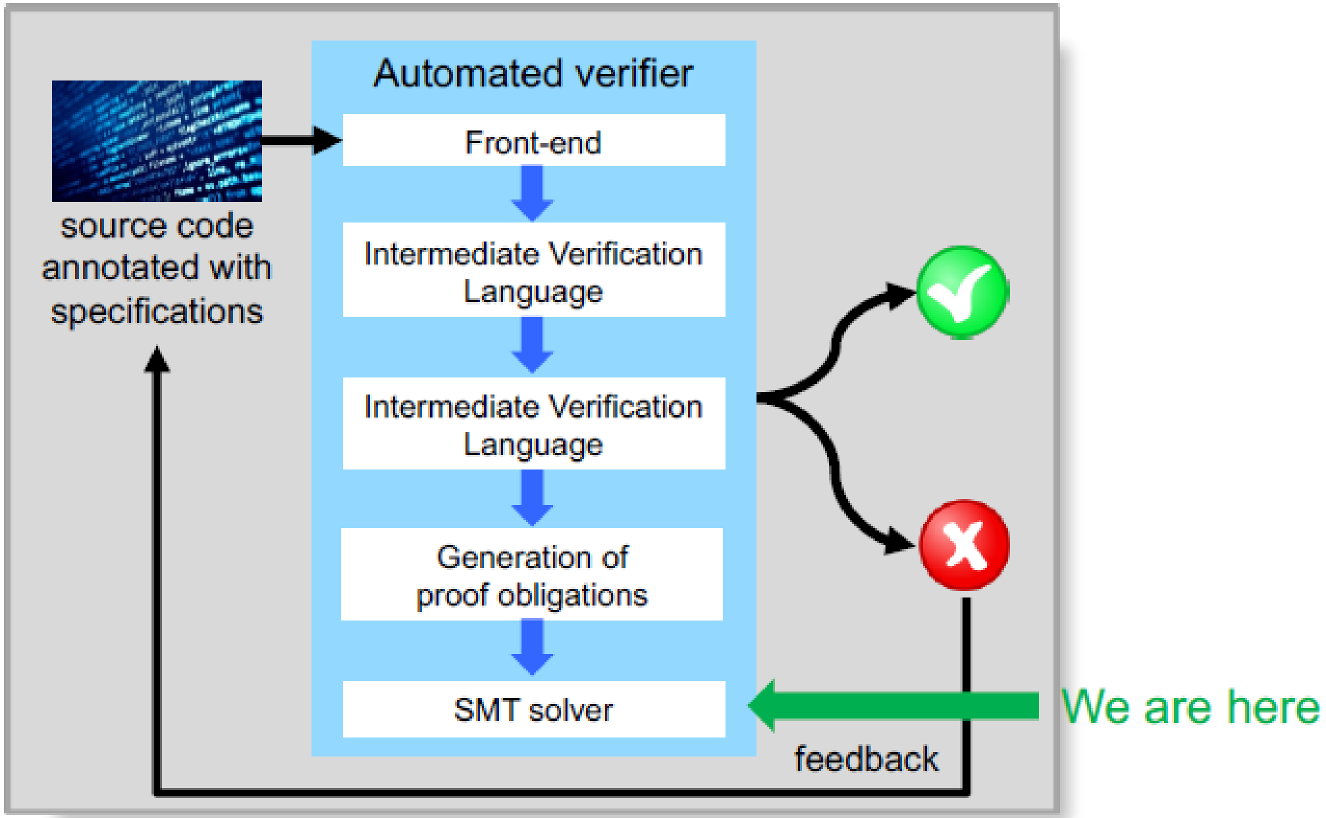
2. Check whether ***SP***`(P, S) ==> Q` is valid

   and ***SAFE***`(P, S)` is valid

Tentative course outline



Roadmap


Automated verifier
Front-end
Intermediate Verification Language
Intermediate Verification Language
Generation of proof obligations
SMT solver
source code annotated with specifications
feedback
We are here

Overview

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

Propositional Logic

X: Boolean variable in **Var**

Syntax

$\mathtt{F} ::= \mathtt{false} \mid \mathtt{true} \mid \mathtt{X} \mid \neg \mathtt{F} \mid \mathtt{F} \wedge \mathtt{F} \mid \mathtt{F} \vee \mathtt{F} \mid \mathtt{F} \Rightarrow \mathtt{F} \mid \mathtt{F} \Leftrightarrow \mathtt{F}$

Interpretation $\mathfrak{I}$: $\mathbf{Var} \to \{ \text{true}, \text{false} \}$

Satisfaction relation

$\mathfrak{I} \models \mathtt{true}$	iff	always
$\mathfrak{I} \models \mathtt{X}$	iff	$\mathfrak{I}(X) = \text{true}$
$\mathfrak{I} \models \neg\mathtt{F}$	iff	not $\mathfrak{I} \models \mathtt{F}$
$\mathfrak{I} \models \mathtt{F} \wedge \mathtt{G}$	iff	$\mathfrak{I} \models \mathtt{F}$ and $\mathfrak{I} \models \mathtt{G}$

$\mathfrak{I}$ is a **model** of F iff $\mathfrak{I} \models \mathtt{F}$

$\mathfrak{I} ::= [X = \text{false}, Y = \text{true}]$

$\mathfrak{I} \models \neg X \vee Y$

$\mathfrak{I} \models X \Rightarrow Y$

$\mathfrak{I} \models (\neg X \vee Y) \Leftrightarrow (X \Rightarrow Y)$

Satisfiability & Validity

- **F** is satisfiable iff **F** has some model

$(X \Rightarrow Y) \Rightarrow Y$

Models: $[X = \text{true}, Y = \text{true}]$, $[X = \text{false}, Y = \text{true}]$, $[X = \text{true}, Y = \text{false}]$

- **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)

$X \wedge (X \Rightarrow Y) \Leftrightarrow Y$

Model of $\neg$**F**: $[X = \text{false}, Y = \text{true}]$

The Satisfiability Problem

- A formula is satisfiable if it has a model

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

- If yes, provide a model as a witness

$$
\begin{aligned}
& (X \lor Y \lor \neg Z) \\
\land\ & (U \lor \neg Y) \\
\land\ & (\neg X \lor \neg Z \lor U \lor V)
\end{aligned}
$$

$$
\begin{aligned}
\mathfrak{I} \ ::= \ [\\
& U = \mathrm{false} \\
& V = \mathrm{false} \\
& X = \mathrm{true} \\
& Y = \mathrm{false} \\
& Z = \mathrm{false} \\
]
\end{aligned}
$$

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

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

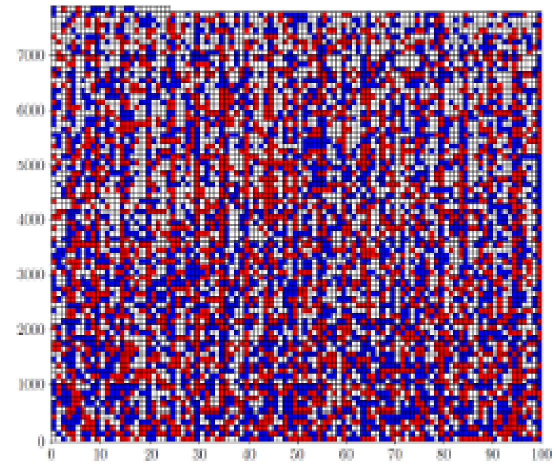
- Modern SAT solvers are extremely efficient in practice
  - Scale to formulas with millions of variables
  - May still perform poorly on certain formulas

Example: Boolean Pythagorean Triples

- BPT: a triple of natural numbers $1 \leq a \leq b \leq c$ with $a^2 + b^2 = c^2$
- Question: Can we color all natural numbers with just two colors such that no BPT is monochromatic?
- Answer: No! The set {1, ..., 7825} always contains a monochromatic BPT

- This was first proven using a SAT solver
  - number of combinations: 2^{7825}
  - "the largest math proof ever" (ca. 200 TB)

- **Modern SAT solvers are efficient in practice**



credits: Marijn J.H. Heule, "Everything's Bigger in Texas - The Largest Math Proof Ever", GCAI 2017

Overview

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

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, Rust, OCaml, ...

- For now: Use Z3 as a SAT solver

A first example (SMTLIB-2)

```
; declare variables
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)

; define formula (X ⇒ Y ⇒ Z) ∧ X
(assert (=> X Y Z))
(assert X)

(check-sat)

(get-model) ; fails if unsat
```

```
$ z3 01-example.smt2

sat
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    true)
  (define-fun Y () Bool
    false)
)
```

A first example (Z3Py)

```
from z3 import *

# declare variables
X = Bool('X')
Y = Bool('Y')
Z = Bool('Z')

# define formula F
F = And( Implies(X, Implies(Y, Z)), X)

solve(F) # find a model for F

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

F is satisfiable, this is a model

```
$ python .\02-example.py

[Z = False, X = True, Y = False]

[Z = False, X = False, Y = True]
```

$\neg$**F** is satisfiable, this is a model

Proofs with Z3

```
(declare-const x Bool)
(declare-const y Bool)

(echo "De Morgan's law: !(x && y) == (!x || !y)")
(assert
    (=
        (not (and x y) )
        (or (not x) (not y))
    )
)
(check-sat) ; result: sat
```

What does this tell us about De Morgan's law?

Proofs with Z3

```
(declare-const x Bool)
(declare-const y Bool)

(echo "De Morgan's law: !(x && y) == (!x || !y)")
(assert
    (=
        (not (and x y) )
        (or (not x) (not y))
    )
)
(check-sat) ; result: sat
```

there is an example for which the law is true

What does this tell us about De Morgan's law?

Proofs with Z3

```
(declare-const x Bool)
(declare-const y Bool)

(echo "De Morgan's law: !(x && y) == (!x || !y)")
(assert
  (not
    (=
        (not (and x y) )
        (or (not x) (not y))
    )
  )
)
(check-sat) ; result: unsat
```

What does this tell us about De Morgan's law?

Proofs with Z3

```
(declare-const x Bool)
(declare-const y Bool)

(echo "De Morgan's law: !(x && y) == (!x || !y)")
(assert
  (not
    (=
        (not (and x y) )
        (or (not x) (not y))
    )
  )
)
(check-sat) ; result: unsat
```

There is no counterexample

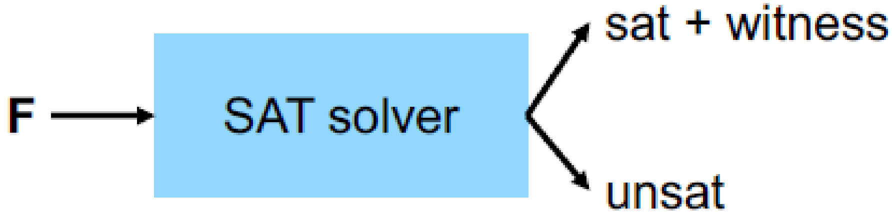
➔ the formula is valid

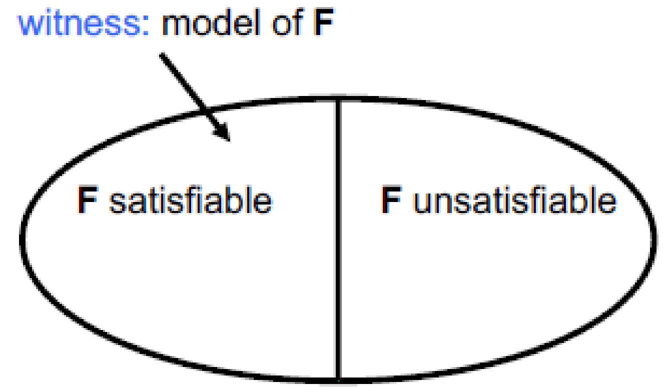
➔ De Morgan's law holds

What does this tell us about De Morgan's law?

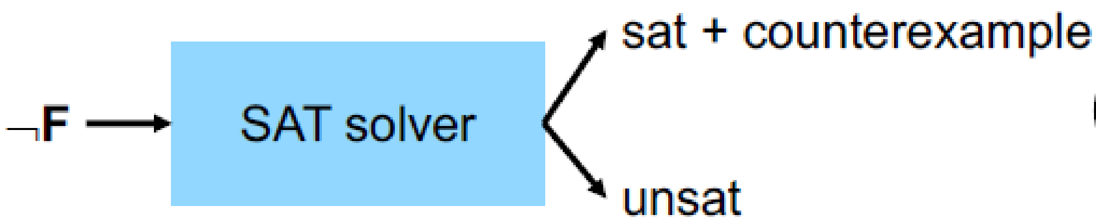
Using a SAT solver

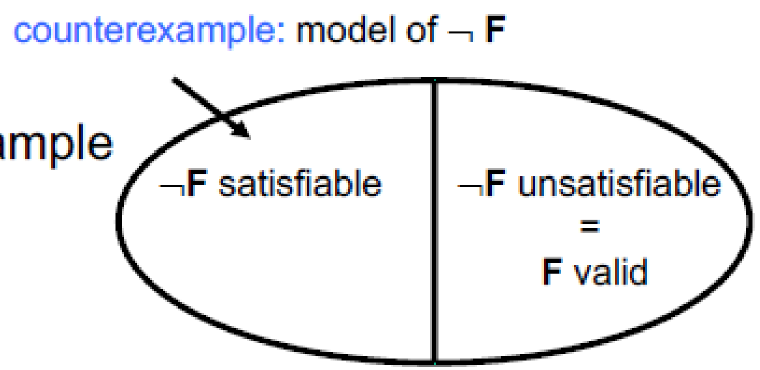
- Is **F** satisfiable?





- Is **F** valid?





Using a SAT Solver for Program Verification

Mains steps of a tool for checking that `{ P } S { Q }` is valid:

1. Compute `WP(S, Q)` ➔ last lecture

2. Check whether entailment `P ==> WP(S, Q)` is valid ➔ ask SAT solver
   - Check satisfiability of negation: `P && !WP(S, Q)`
   - `unsat` ➔ ✓
   - `sat` ➔ ✗ model explains why `{ P } S { Q }` is not valid

Using a SAT Solver for Program Verification

```
{ true }
// check that validity of true ⇒ a ∧ ...
{ 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

```
{ x == X && y == Y }
{ y == Y && y - Y == 0 }
// ... swap X and Y
{ x == Y && y == X }
```

Entailment to check:

```
(x == X && y == Y) ==> y == Y && y - Y == 0
```

- Entailment is not in propositional logic
  - Integer-valued variables (`x,X,y,Y`) and numeric constants (0)
  - Arithmetic operations (-) 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 (FOL)

Overview

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

Ingredients of Many-sorted First-order logic (FOL)

1. Sorts — `Bool, Int, Real, T`
    - specifies possible types
2. Typed Variables — `x, y, z, …`
3. Typed Function symbols — `0, 1.5` `+, *, _?_:_`
    - building blocks of terms — `0` `x ? y - 17 : z*z + 1`
4. Typed Relational symbols — `<` `prime` `R`
    - turn terms into logical propositions — `x < 0` `prime(y+4)` `R(x,y,z)`
5. Logical symbols — $\wedge$ $\vee$ $\neg$ $\Rightarrow$ $\Leftrightarrow$ $\exists$ $\forall$ …
6. An equality symbol — $=$

FOL Formulas

- A **signature Σ** is a set of
  - at least one sort
  - function symbols
  - relational symbols (= does not count)

$0, 1 : \mathbf{Int}$ (type: constant)

$\leq : \mathbf{Int} \times \mathbf{Int}$ (type: binary relation)

$$\Sigma = \{\, \mathbf{Int}, 0, 1, +, *, \leq \}$$

$+ : \mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int}$
(type: function with 2 arguments)

- A **Σ-formula** is a logical formula over propositions built from symbols in Σ

$$\forall x{:}\, \mathbf{Int}\ \exists y{:}\, \mathbf{Int}\ (y = x + 1 \wedge y * y \leq x * x + (1 + 1) * x + 1)$$

Is this Σ-formula satisfiable?

FOL Formulas

Is this Σ-formula satisfiable?

$\Sigma = \{ \mathbf{Int}, 0, 1, +, *, \leq \}$

$$\forall x{:}\, \mathbf{Int}\ \exists y{:}\, \mathbf{Int}\ (y = x + 1 \wedge y * y \leq x * x + (1 + 1) * x + 1)$$

Yes, if the symbols $+, *, =$ have the canonical meaning

No, if

- 1 actually means 2, or
- + actually means maximum

Satisfiability of Σ-formulas depends on the admissible interpretations of symbols in Σ

determined by Σ-theories

FOL Σ-Interpretations

$\Sigma = \{\ \mathbf{Int}, one, plus, leq\ \}$

- A **Σ-structure $\mathfrak{A}$** assigns
  - a non-empty domain (set) $\mathbf{U}^{\mathfrak{A}}$ to each sort $\mathbf{U}$ in Σ
  - a function $f^{\mathfrak{A}}$ over domains (respecting types) to each function symbol f in Σ
  - a relation $R^{\mathfrak{A}}$ over domains (respecting types) to each relational symbol R in Σ

$\mathfrak{A} ::= (\mathbf{Int}^{\mathfrak{A}}, one^{\mathfrak{A}}, plus^{\mathfrak{A}}, leq^{\mathfrak{A}})$

$\mathbf{Int}^{\mathfrak{A}} ::= \mathbb{Z}$

$one^{\mathfrak{A}} ::= 1$

$plus^{\mathfrak{A}} : \mathbf{Int}^{\mathfrak{A}} \times \mathbf{Int}^{\mathfrak{A}} \to \mathbf{Int}^{\mathfrak{A}}, (a, b) \mapsto a + b$

$leq^{\mathfrak{A}} ::= \{\ (a, b) \in \mathbf{Int}^{\mathfrak{A}} \times \mathbf{Int}^{\mathfrak{A}} \mid a \leq b\ \}$

- A **Σ-assignment β** maps variables x of sort $\mathbf{U}$ to domain elements in $\mathbf{U}^{\mathfrak{A}}$

$\beta : \mathbf{Var} \to \mathbf{Int}^{\mathfrak{A}}$

- A **Σ-interpretation** is a pair $\mathfrak{I} = (\mathfrak{A}, \beta)$

- $\mathfrak{I}(t)$ denotes the domain element obtained by evaluating term t in $\mathfrak{I}$

$\mathfrak{I}(plus(plus(one, one), x))$
$= \mathrm{plus}^{\mathfrak{A}}(\mathrm{plus}^{\mathfrak{A}}(\mathrm{one}^{\mathfrak{A}}, \mathrm{one}^{\mathfrak{A}}), \beta(x))$
$= (1+1) + \beta(x)$

FOL Semantics

$\mathfrak{I}$ is a model of **F** iff $\mathfrak{I} \vDash$ **F**

we can always express equality between terms

FOL formula F (excerpt)	$\mathfrak{I} = (\mathfrak{A}, \beta) \vDash$ F if and only if
$t_1 = t_2$	$\mathfrak{I}(t_1) = \mathfrak{I}(t_2)$
$R(t_1, \dots, t_n)$	$\big(\mathfrak{I}(t_1), \dots, \mathfrak{I}(t_n)\big) \in R^{\mathfrak{A}}$
$\mathbf{G} \wedge \mathbf{H}$	$\mathfrak{I} \vDash \mathbf{G}$ and $\mathfrak{I} \vDash \mathbf{H}$
$\mathbf{G} \Rightarrow \mathbf{H}$	If $\mathfrak{I} \vDash \mathbf{G}$, then $\mathfrak{I} \vDash \mathbf{H}$
$\exists x\colon \mathrm{T}\ (\mathbf{G})$	For some $\mathrm{v} \in \mathrm{T}^{\mathfrak{A}}$, $\mathfrak{I}[x := \mathrm{v}] \vDash \mathbf{G}$
$\forall x\colon \mathrm{T}\ (\mathbf{G})$	For all $\mathrm{v} \in \mathrm{T}^{\mathfrak{A}}$, $\mathfrak{I}[x := \mathrm{v}] \vDash \mathbf{G}$

F is satisfiable iff **F** has some model

Issues with FOL Satisfiability

$\Sigma = \{\, \mathbf{Nat}, zero, one, plus, leq \,\}$

- All symbols are uninterpreted
- The meaning of functions and relations is determined in the chosen model
- Many formulas are satisfiable if we can choose Σ-structures that defy the intended meaning of functions and relations

➔ Filter out unwanted Σ-structures

$\mathbf{F} ::= \exists x{:}\, \mathbf{Nat}(\ x\ plus\ one\ leq\ zero)$

infix notation for $leq(plus(x, one), zero)$

sat: $\mathbf{Nat} = \mathbb{N}, one^{\mathfrak{A}} ::= 0, \mathrm{leq} ::= \leq$
$zero^{\mathfrak{A}} ::= 1, plus^{\mathfrak{A}} ::= +$

sat: $\mathbf{Nat} = \mathbb{N}, one^{\mathfrak{A}} ::= 1, \mathrm{leq} ::= \leq$
$zero^{\mathfrak{A}} ::= 0, plus^{\mathfrak{A}} ::= -$

Satisfiability Modulo Theories

- A Σ-sentence is a formula without free variables
- An axiomatic system **AX** is a set of Σ-sentences
- The Σ-theory **Th** given by **AX** is the set of all Σ-sentences implied by **AX**

A Σ-formula **F** is satisfiable modulo **Th** iff there exists a Σ-interpretation $\Im$ such that
- $\Im \models$ **F** , and
- $\Im \models$ **G** for every sentence **G** in **Th**.

A Σ-formula **F** is valid modulo **Th** iff $\neg$**F** is *not* satisfiable modulo **Th**.

Some important theories

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

- EUF: Equality logic with Uninterpreted Functions **decidable**
  - $\Sigma = \{ \mathbf{U}, =, f, g, h, ... \}$
  - arbitrary non-empty domain **U**
  - axioms ensure that $=$ is an equivalence relation
  - arbitrary number of uninterpreted function symbols of any arity
  - axioms do *not* constrain function symbols

- We typically need a combination of multiple theories
  - Program verification: theories for modeling different data types

Overview

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

Using Theories (SMTLIB-2)

- Sorts
  - `Bool`, `Int`, `Real`, `BitVec(precision)`
  - `DeclareSort(name)` (uninterpreted)
- Uninterpreted functions are declared with parameter and return types
- Variables are uninterpreted functions of arity 0
  - `Const(name, sort)`

```
(declare-sort Pair)

(declare-fun cons (Int Int) Pair)
(declare-fun first (Pair) Int)

(declare-const null Pair)

; first axiom
(assert (= null (cons 0 0)))
; second axiom
(assert (forall ((x Int) (y Int))
          (= x (first (cons x y)))
))

; formula (negated for validity check)
(assert (not (= (first null) 0)))

(check-sat)
```

Using Theories (Z3Py)

- Sorts
  - `Bool`, `Int`, `Real`, `BitVec(precision)`
  - `DeclareSort(name)` (uninterpreted)

- Uninterpreted functions are declared with parameter and return types

- Variables are uninterpreted functions of arity 0
  - `Const(name, sort)`

```
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() )
```

Custom theories for user-defined data types and operations

- Encoding via
  - uninterpreted sorts
  - constants
  - uninterpreted functions
  - axioms enforcing the data type's properties
- We call such an encoding an axiomatization
- Week 5: advanced data types
  - sets, sequences, trees
  - accessors, mutators
  - recursive functions

```
(declare-sort Pair)

(declare-fun cons (Int Int) Pair)
(declare-fun first (Pair) Int)

(declare-const null Pair)

; first axiom
(assert (= null (cons 0 0)))
; second axiom
(assert (forall ((x Int) (y Int))
           (= x (first (cons x y)))
))

; formula (negated for validity check)
(assert (not (= (first null) 0)))

(check-sat)
```

Incorporating custom theories

Original verification condition: `P ==> WP(S, Q)` valid

A Σ-formula **F** is valid modulo **Th** iff **!F** is *not* satisfiable modulo **Th**.

A Σ-formula **F** is satisfiable modulo **Th** iff there exists a Σ-interpretation $\Im$ such that

- $\Im \models$ **F** , and
- $\Im \models$ **G** for every sentence **G** in **Th**.

Enriched verification condition:

```
    P ==> WP(S, Q)  valid modulo custom theory
iff BP && P && !WP(S, Q)  unsat
iff BP ==> P ==> WP(S, Q)  valid
```

Background **P**redicate: conjunction of all our axioms defining our theory

Automating Program Verification

Mains steps of a tool for checking that `{ P } S { Q }` is valid:

0. Determine axioms of underlying theory ➔ background predicate BP
   - *Reusable:* once for every type or function ➔ Week 5
   - Z3 has built-in theories for common theories (e.g. arithmetic) ➔ **Today**

1. Compute `WP(S, Q)` ➔ Week 1 & 2

2. Check whether `BP ==> P ==> WP(S, Q)` is valid ➔ SMT solver
   - Check satisfiability of negation: `BP && P && !WP(S, Q)`
   - `unsat` ➔ ✔
   - `sat` ➔ ✘ model explains why `{ P } S { Q }` is not valid

Axiomatize with Care

- Axiomatizations are part of the trusted codebase

- Inconsistent axioms invalidate verification results

  `false ==> P ==> WP(S, Q)` is always valid

- Axioms do not show up in verification problems

  `{ x == null } S { y > null }`

- Axiomatizations require separate validation
  - proofs, testing, profiling, ...

```
(declare-const null Int)

; inconsistent axioms
(assert (= null 0)
(assert (= null 17)

(assert (not
  (> 42 23) ; wrong statement
))

(check-sat) ; unsat
```

Theory applications

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

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

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

Fixed-size bitvector arithmetic

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

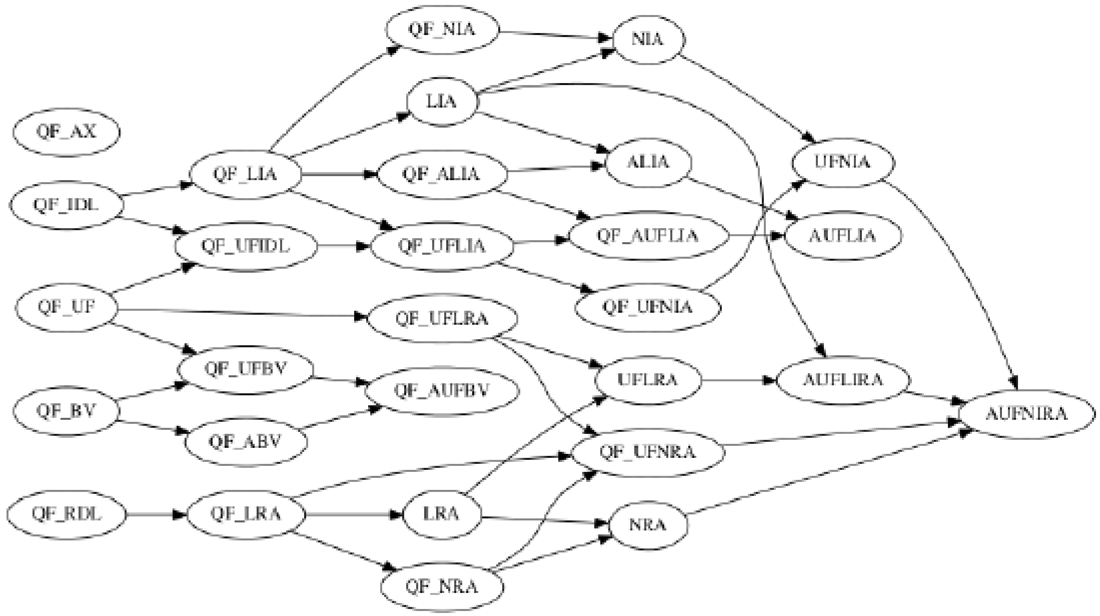
- To encode bit-level operations
- To perform bit-precise reasoning, e.g., floats

Array theory

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

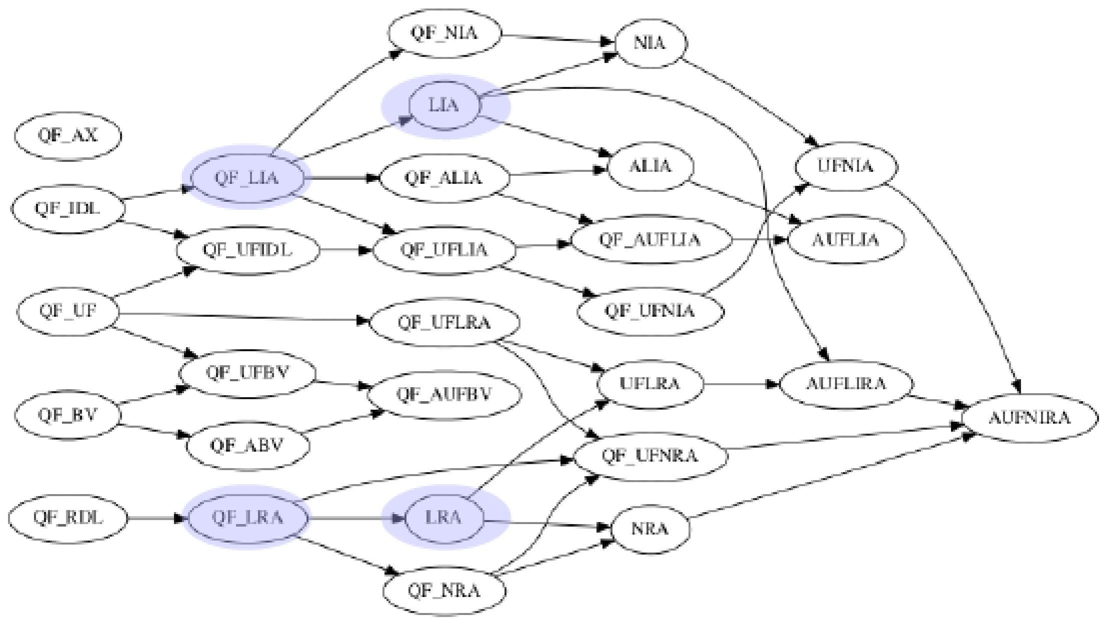
- To encode data types such as arrays

Z3 built-in theories



➔ no explicit background predicate needed

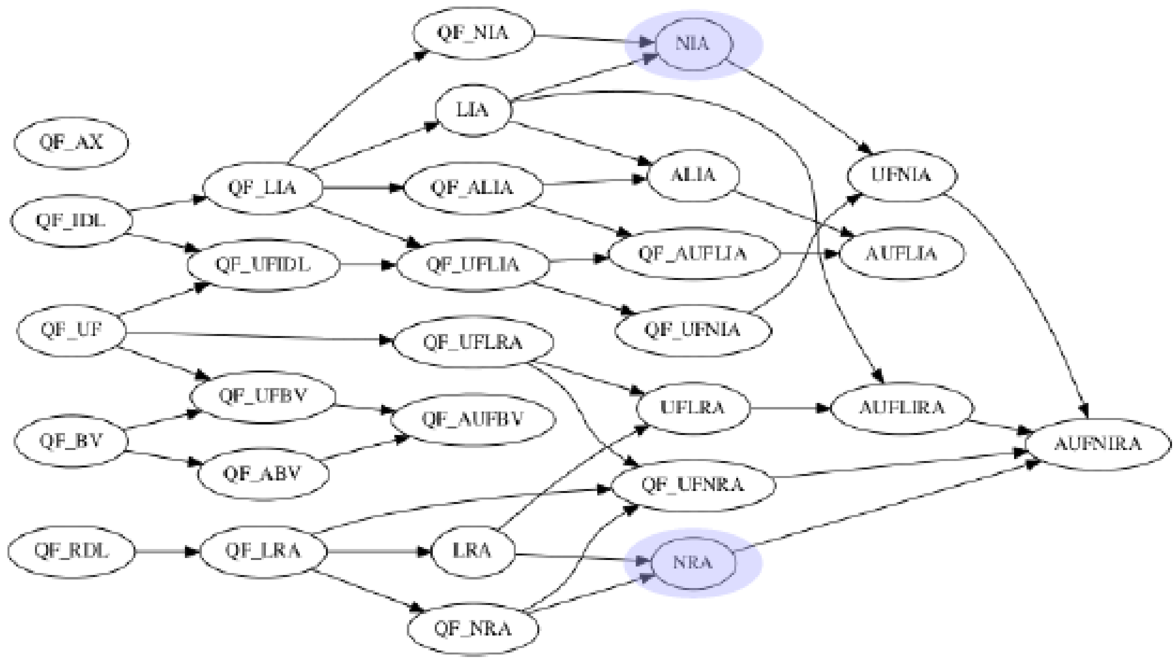
Z3 built-in theories



(Quantifier-free) Linear Integer/Real Arithmetic

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

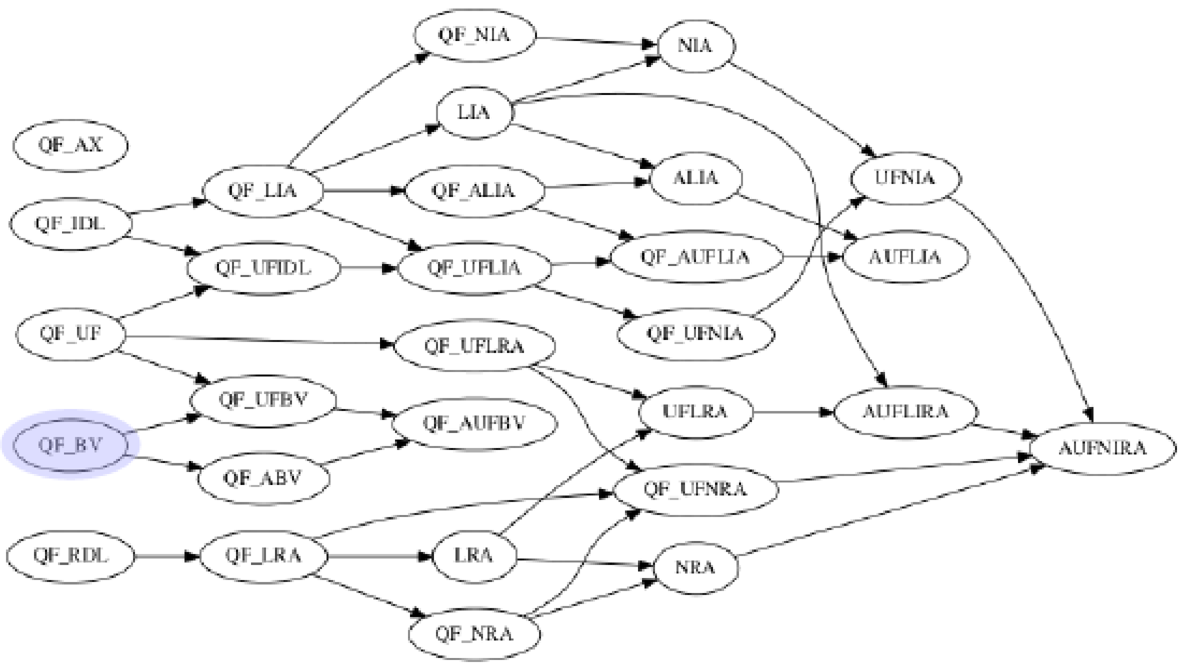
Z3 built-in theories



***Non*-Linear Integer/Real Arithmetic**

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

Z3 built-in theories



Quantifier-free fixed-size **bitvector** arithmetic

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

Using Z3 to verify a program

```
{ a = 1 ∧ 0 ≤ b*b – 4*c }
discriminant := b*b – 4*a*c;
if (discriminant < 0) {
  assert false
} else {
  x := (b + √discriminant) / 2
}
{ a*x² + b*x + c = 0 }
```

Step 1: use ***WP*** to determine the verification condition

Using Z3 to verify a program

```
{ a = 1 ∧ 0 ≤ b*b - 4*c }
// ==>
{ 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 }
discriminant := b*b - 4*a*c;
{ discriminant < 0 ∧ false ∨
  ¬discriminant < 0 ∧ a*((-b + √discriminant) / 2)² + b*((-b + √discriminant) / 2) + c = 0 }
if (discriminant < 0) {
{ false }
  assert false
{ a*x² + b*x + c = 0 }
} else {
{ a*((-b + √discriminant) / 2)² + b*((-b + √discriminant) / 2) + c = 0 }
  x := (-b + √discriminant) / 2
{ a*x² + b*x + c = 0 }
}
{ a*x² + b*x + c = 0 }
```

Using Z3 to verify a program

- Step 1: use ***WP*** to determine the verification condition

```
{ a = 1 ∧ 0 ≤ b*b - 4*c }
// ==>
{ 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 }
```

- Step 2: check whether the verification condition is valid
  - Check satisfiability of negation: `Pre && !WP(S, Post)`

```
; declarations … (full example available online)

; precondition
(assert (and (= a 1) (<= 0 (- (* b b) (* 4 c)))))

; negated weakest precondition
(assert (not <complicated expression here>))

(check-sat) ; want: unsat
```

Z3's 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 decidable, e.g., quantifier-free linear arithmetic
  - SMT solver will terminate and report either "sat" or "unsat"

- Some theories are undecidable, 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

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
5. Quantifiers

Quantifiers

- Program specifications will often require quantifiers to express complex properties
  - "Array *a* is sorted": $\forall i, j. 0 \leq i < j < |a| \Rightarrow a[i] \leq a[j]$
  - "All entries in *a* are non-null": $\forall i. 0 \leq i < |a| \Rightarrow a[i] \neq null$

- Quantifiers are also useful to model additional theories and data types
  - Function definitions: $\forall x. f(x) = e$
  - We will see an example of this soon

- Addition of quantifiers makes many problems undecidable
  - Quantifier-free linear arithmetic is decidable
  - Linear arithmetic with quantifiers is not

Existential quantifiers

- Existential quantifiers in positive positions can be eliminated via skolemization
  - Positive position means non-negated.
  - Quantifier in $(\exists x.\, P) \wedge Q$ is in a positive position, in $(\exists x.\, P) \Rightarrow Q$ it is not.

> $\exists x.\, P$ is $\mathcal{T}$-satisfiable iff $P[x/x']$, where x' is fresh, is $\mathcal{T}$-satisfiable.

- Quantifier alternation requires introducing fresh functions
  - Example: $\forall x.\, \exists y.\, P$ is satisfiable iff $\forall x.\, P[y/f(x)]$ is satisfiable, where *f* is fresh

- Quantifiers in negative positions can be rewritten:

$$\neg \exists x.\, P \Leftrightarrow \forall x.\, \neg P$$
$$\neg \forall x.\, P \Leftrightarrow \exists x.\, \neg P$$

Universal quantifiers

- Universal quantifiers cannot be eliminated in general
  - Checking whether some formula is satisfiable for all values in an infinite sort is a hard problem

- Recall: Our goal is to prove validity of P, i.e., to prove $\neg P$ unsatisfiable
  - Usually, to prove unsatisfiability, only specific instantiations are needed

Example: To prove unsatisfiability of

$$\mathbf{F} \; ::= \; (f(0) = 1) \wedge (\forall x.\, f(x) = x)$$

we only need to instantiate x with 0 to get

$$(f(0) = 1) \wedge (f(0) = 0)$$

Unsatisfiable

Universal quantifier instantiation

- Idea: Let users provide hints about relevant quantifier instantiations

- Treat quantified constraints separately from non-quantified constraints
  - Do not attempt to satisfy quantifier in general
  - Quantifier only generates specific instantiations, solver satisfies all generated instantiations
  - Resulting formulas can be solved by decision procedures for quantifier-free problems

- New format of universal quantifiers is $\forall x.\, \{e(x)\} P$
  - $e(x)$ is the pattern or trigger, must contain all quantified variables
  - Quantifier body *P* is instantiated for any term with the shape $e(x)$ occurring in the current constraints or model
  - Trigger can consist of multiple terms, quantifier can have multiple triggers
  - Most SMT solvers impose syntactic restrictions on possible trigger terms

Quantifier instantiation using triggers

- A ground term is a term in the formula containing no quantified variables

> A quantified constraint $\forall x.\ \{c(x)\}\ P$ in the current formula leads to an instantiation $P[x/v]$ iff the current formula/model contains a ground term $c(v)$

- Example: $(\forall x.\ \{f(x)\} f(x) > 0) \wedge f(2) < 2$ instantiates $f(2) > 0$

- Trigger matching is performed modulo equality
  - Example: $g(1) = 2 \wedge h(2) = 3 \wedge (\forall x.\ \{h(g(x))\}\ h(g(x)) < x)$ instantiates $h(g(1)) < 1$
  - This instantiation technique is called e-matching for this reason

Problematic triggers

- Too strict or wrong triggering leads to incompleteness
  - Example: $f(0) = 1 \wedge (\forall x.\ \{g(f(x))\}\ f(x) = x)$ Unknown
- Generally, SMT solvers using e-matching cannot prove satisfiability
  - Quantifiers might not be satisfied for values for which they were not triggered
  - Only report unsat or unknown

- Too liberal triggering leads to many or infinite instantiations
  - Cause of most performance issues in automated verifiers
  - Always write triggers yourself! Otherwise, the solver will infer them
- Matching loop: Quantifier instantiation (transitively) triggers same quantifier again, ad infinitum
  - Example: $\forall i.\ \{a[i]\}\ 0 \leq i < |a| - 1 \Rightarrow a[i] \leq a[i+1]$
  - In practice, tools eventually stop after some limit is reached

Mathematical data types

- Our language so far has a very restricted set of types

  Types
  T ::= `Bool` | `Int` | `Rational` | `Real`

- Mathematical data types are useful to encode languages that have them
  - Many functional languages offer lists, tuples, and abstract data types (ADTs)

- Mathematical data types will also be important to specify imperative code
  - *"Array sort leaves the multiset of elements unchanged"*
  - *"All implementations of Java's List interface store a sequence of elements"*
  - We will discuss this kind of specification in Lecture 7

Encoding of custom data types

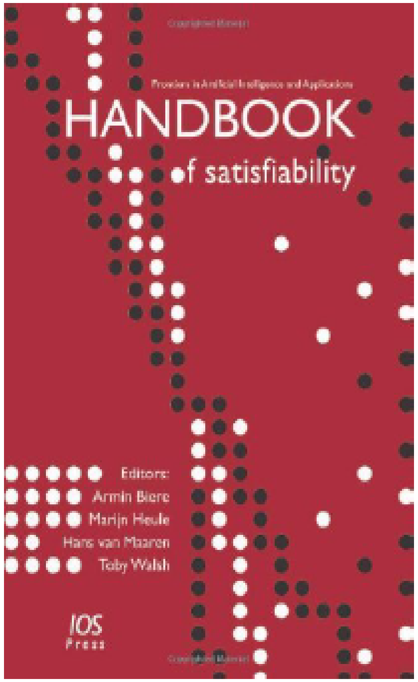
- We encode custom data types into SMT using
  - uninterpreted types
  - constants
  - uninterpreted functions
  - axioms that express properties of the constants and functions

```
from z3 import *
Set = DeclareSort('Set')
empty = Const('empty', Set)
card = Function('card', Set, IntSort())
…
ax1 = (card(empty) == 0)
…
```

- We call such an encoding an axiomatization
- Generics can be handled via monomorphization, that is, by generating a separate axiomatization for each instantiation of a generic type

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/

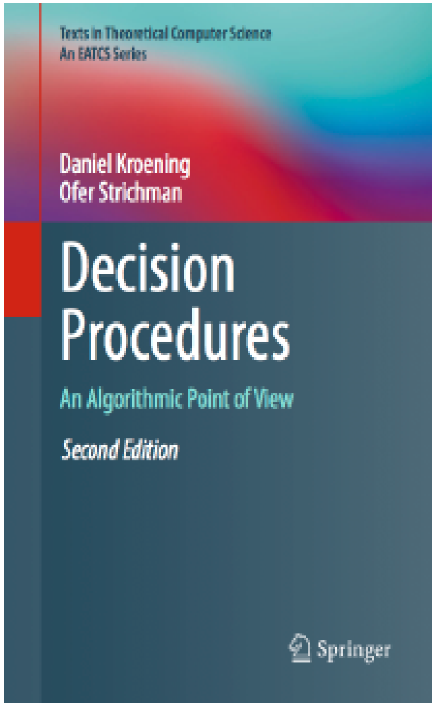


More background on SMT solvers

- 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

- Efficient E-Matching for SMT Solvers. Leonardo de Moura, Nikolaj Bjørner, 2007



Wrap-up

Mains steps of a tool for checking that `{ P } S { Q }` is valid:

0. Determine axioms of underlying theory → background predicate BP
   - *Reusable:* once for every type or function → Week 5
   - Z3 has built-in theories for common theories (e.g. arithmetic)

1. Compute `WP(S, Q)` → Week 1 & 2

2. Check whether `BP ==> P ==> WP(S, Q)` is valid → SMT solver
   - Check satisfiability of negation: `BP && P && !WP(S, Q)`
   - `unsat` → ✔
   - `sat` → ✘ model explains why `{ P } S { Q }` is not valid
   - `unknown` → decidability issues, strengthen theory, hacks → future classes