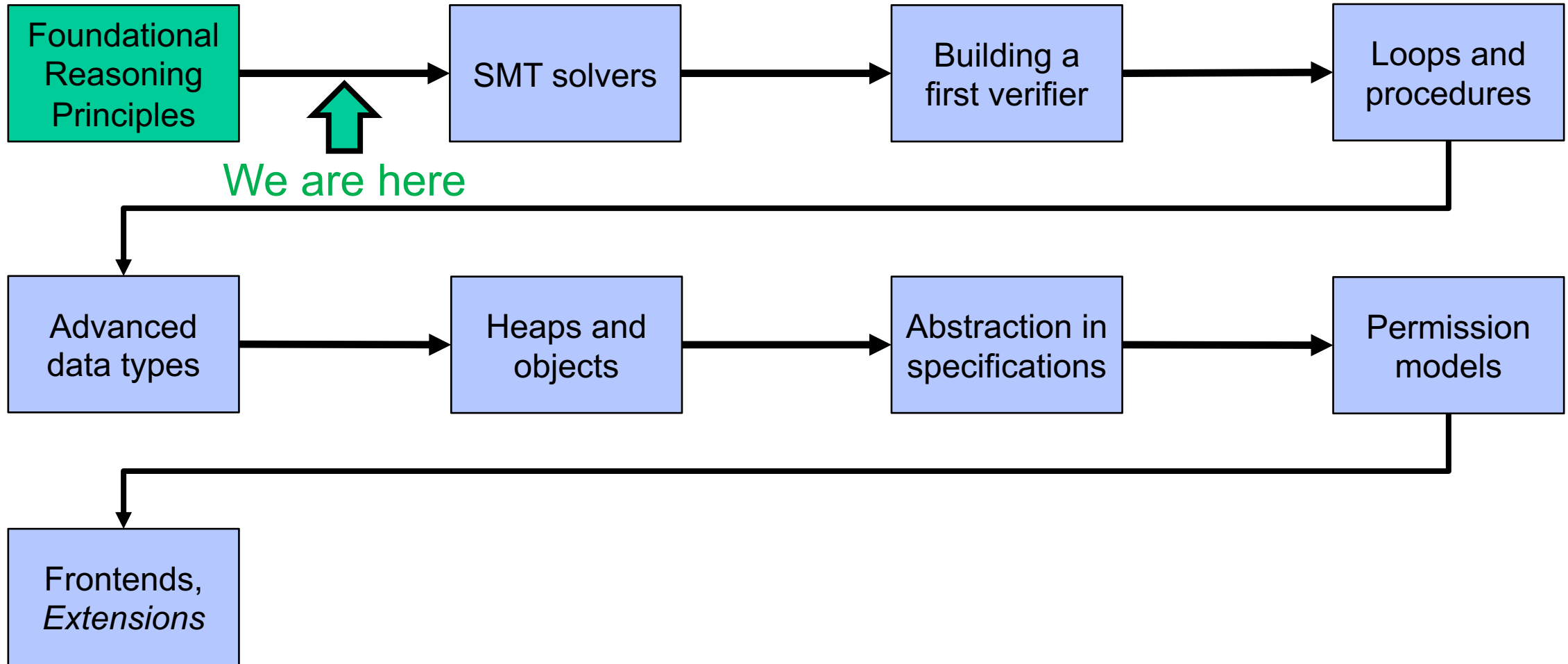


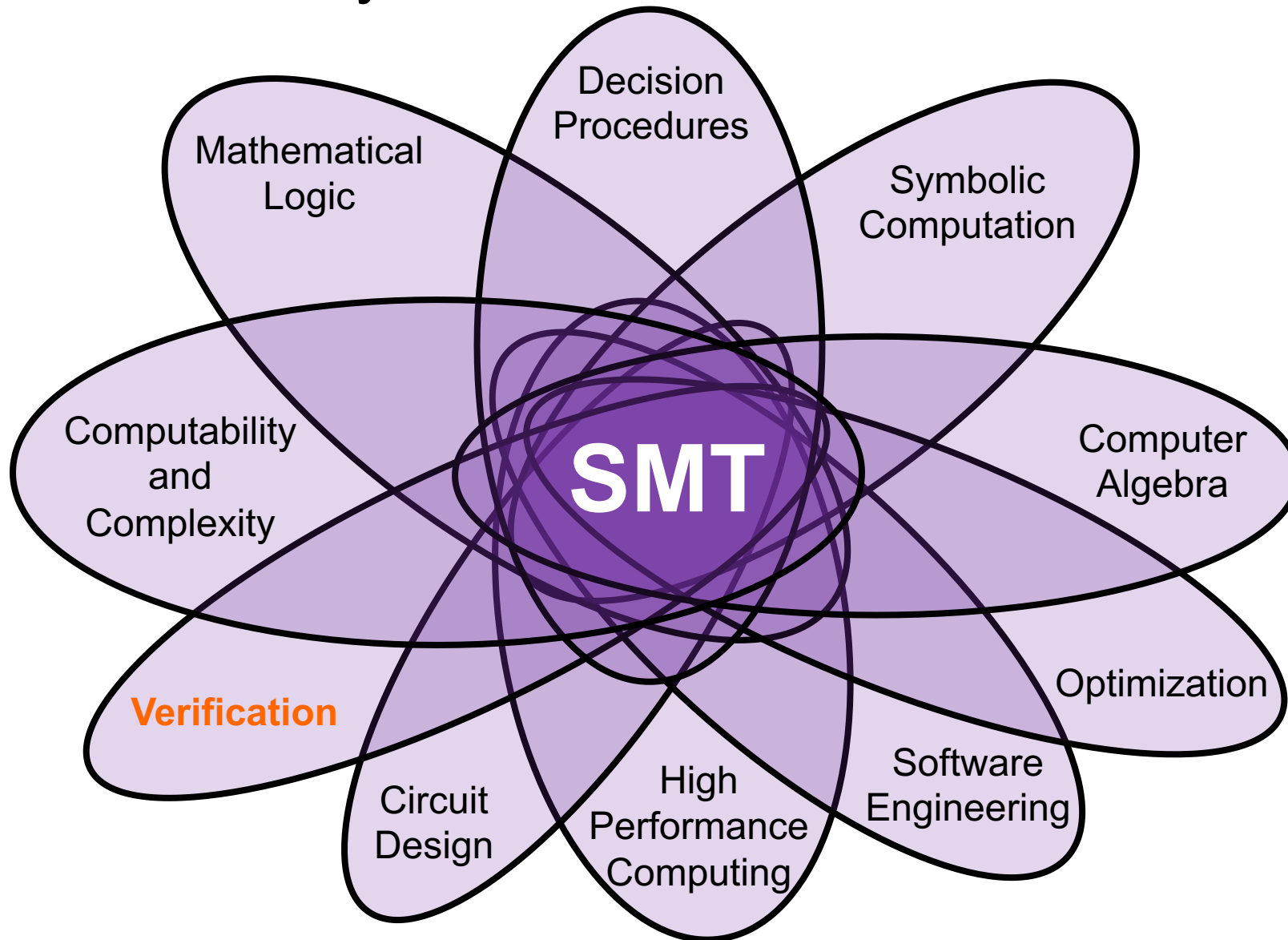
02245 – Lecture 2

FOUNDATIONS & SMT SOLVERS

Tentative course outline



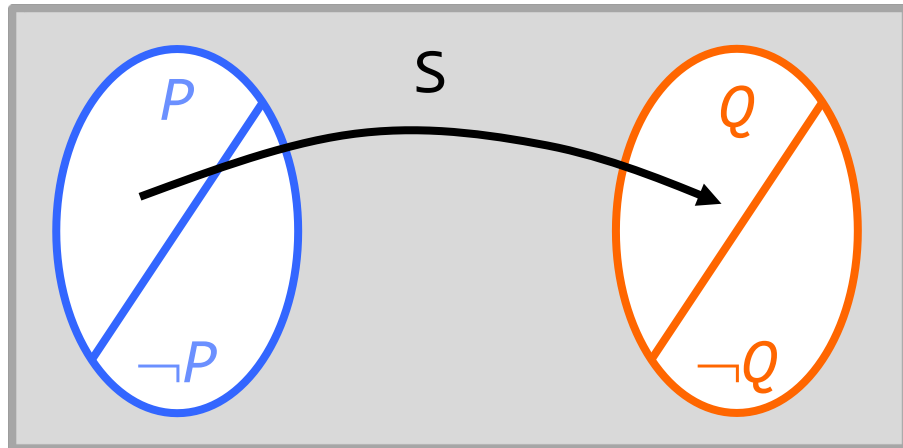
Satisfiability Modulo Theories Solvers



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

But first: Recap

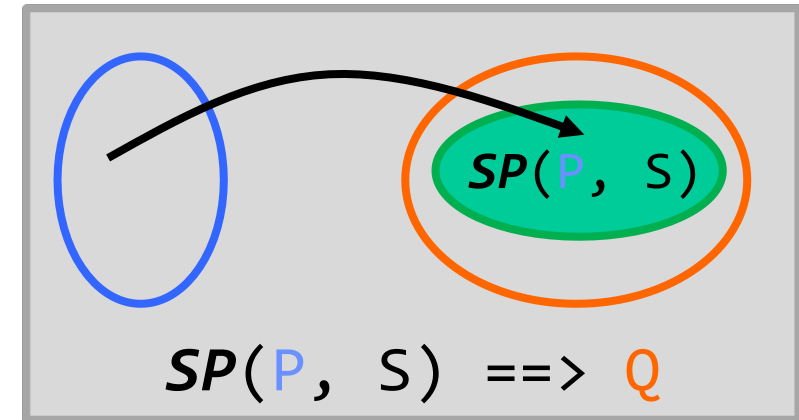
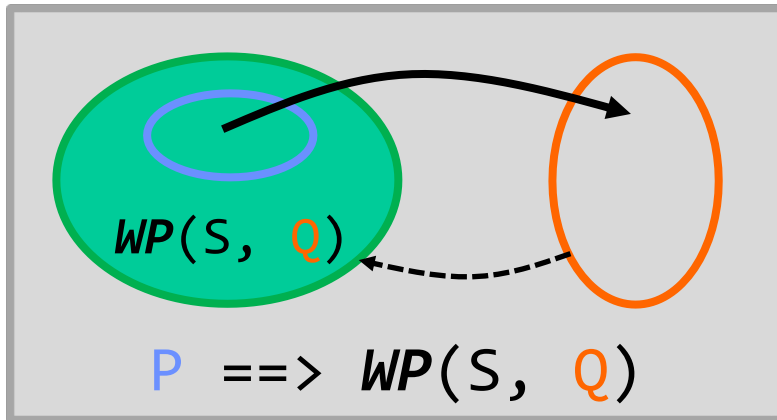
The **Floyd-Hoare triple** $\{ P \} S \{ Q \}$ is **valid** if and only if every execution of S that starts in a state satisfying precondition P terminates without an error in a state satisfying postcondition Q .



```
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 $x_0 :: P[x / x_0] \ \&\& \ x == a[x / x_0]$
assert R	$R \ \&\& \ Q$	$P \ \&\& \ R$
assume R	$R \implies Q$	$P \ \&\& \ R$
$S1; S2$	$WP(S1, WP(S2, Q))$	$SP(SP(P, S1), S2)$
$S1 \ [] \ S2$	$WP(S1, Q) \ \&\& \ WP(S2, Q)$	$SP(P, S1) \ \ SP(P, S2)$

Automating Program Verification

Main steps of a tool for checking that $\{ P \} S \{ Q \}$ is valid:

1. Compute $WP(S, Q)$ → last lecture
2. Check whether $P \implies WP(S, Q)$ is valid → delegate to SMT solver

Alternative approach

Main steps of a tool for checking that $\{ P \} S \{ Q \}$ is valid:

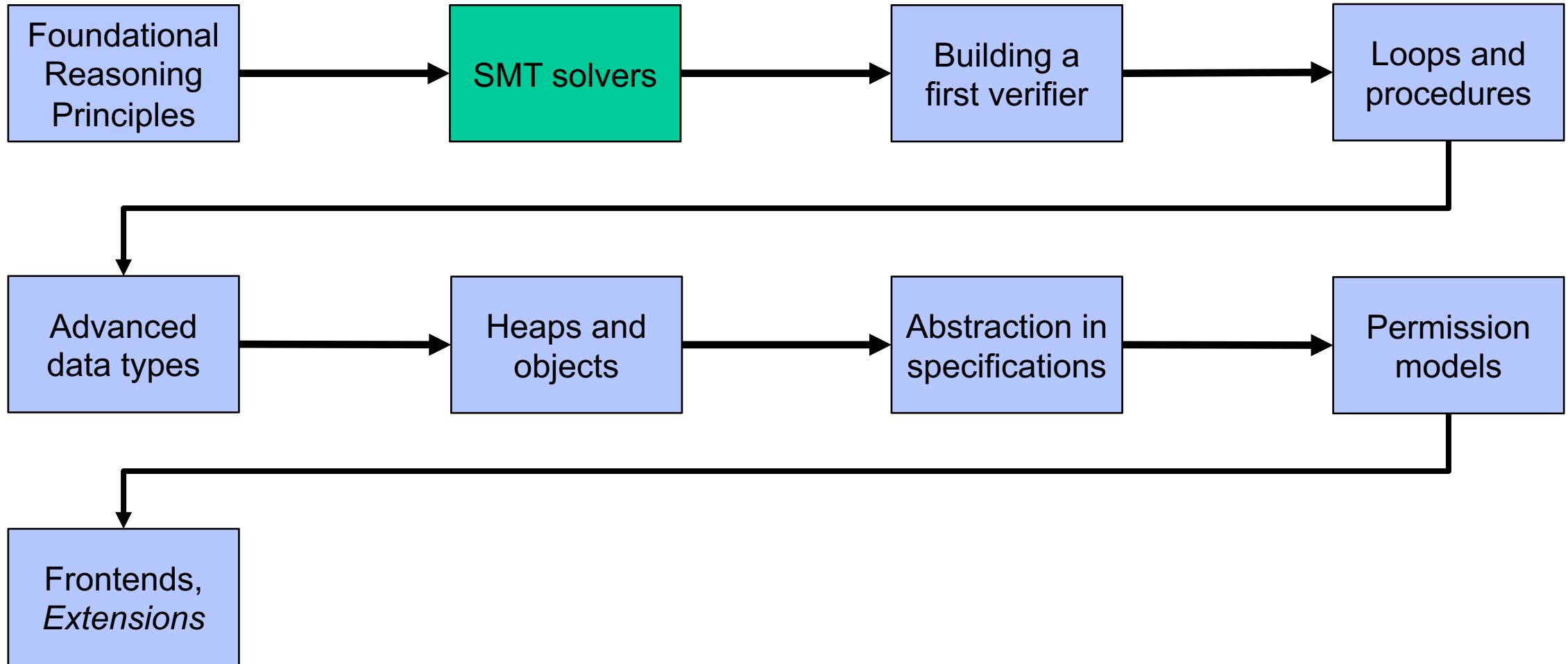
1. Compute $SP(P, S)$ and $SAFE(P, S)$
2. Check whether $SP(P, S) \implies Q$ is valid
and $SAFE(P, S)$ is valid

→ Homework

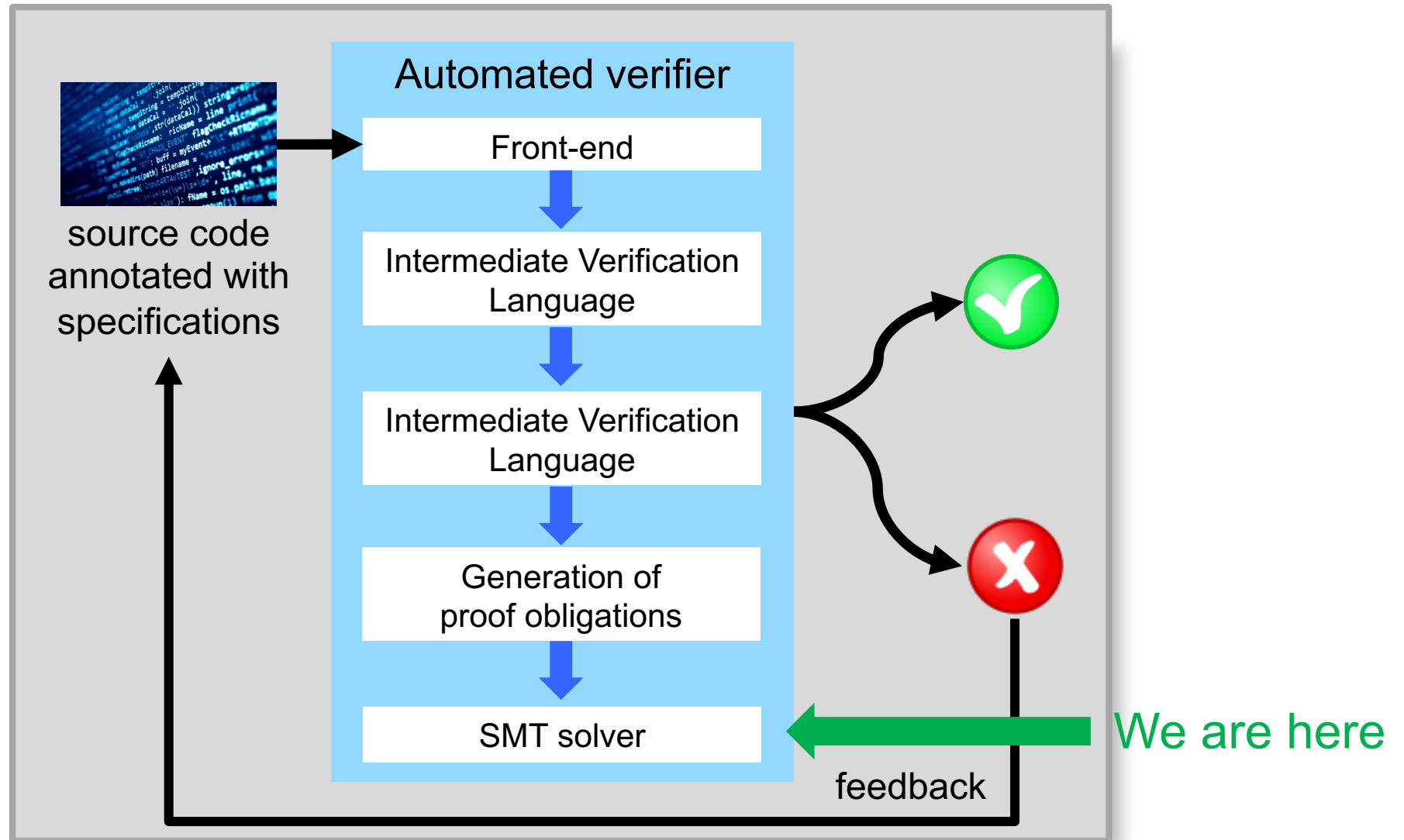
< Homework W1 >

Solutions will be published on course page

Tentative course outline



Roadmap



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

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

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

Satisfaction relation

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

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

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

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

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

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

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**
($\neg\mathbf{F}$ is unsatisfiable)

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

- **F** is **not valid** iff **some interpretation is not a model** of **F**
($\neg\mathbf{F}$ is satisfiable)

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

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

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**

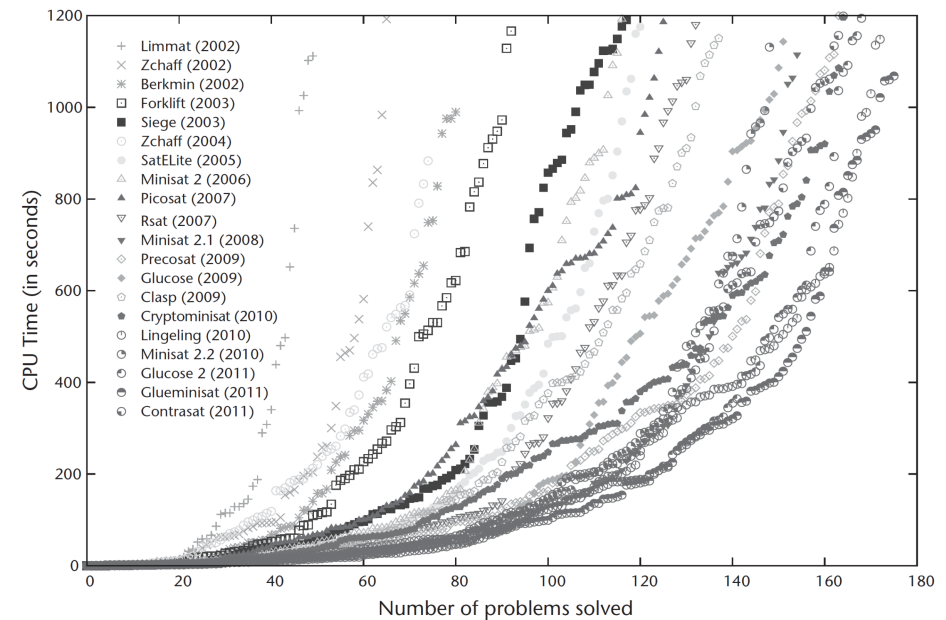
$$\begin{aligned} & (X \vee Y \vee \neg Z) \\ \wedge & (U \vee \neg Y) \\ \wedge & (\neg X \vee \neg Z \vee U \vee V) \end{aligned}$$

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

Complexity of SAT

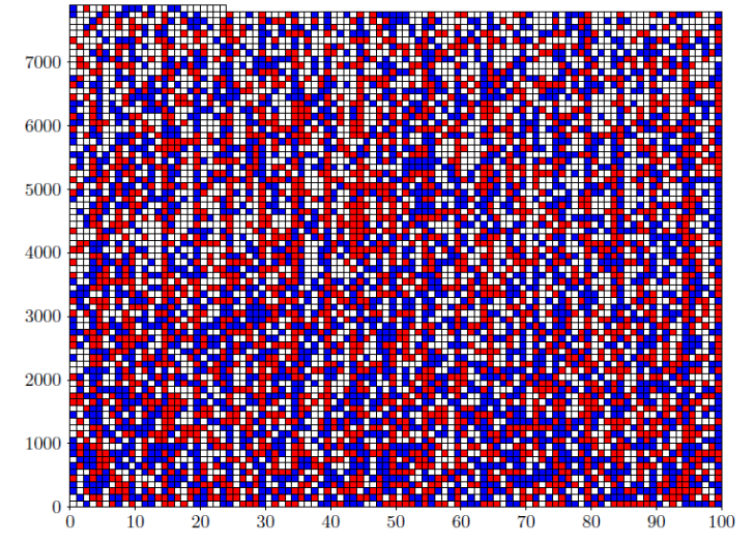
- For formulas in conjunctive normal form (CNF), SAT is **the classical NP-complete problem**
- Many difficult problems can be efficiently encoded
- Every known algorithm is exponential in the formula's size

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



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 $\{1, \dots, 7825\}$ 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

Exercise: Seating of Wedding Guests



Exercise

- Model the following problem as an instance of the SAT problem.
- There are 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

- Model assignment via nine variables
- Alice does not sit next to Charlie
- Alice does not sit on the leftmost chair
- Bob does not sit to the right of Charlie
- Each person gets a chair
- Every person gets at most one chair
- Every chair gets at most one person

$x_{p,c}$: “person p sits in chair c ”

$$(x_{A,l} \vee x_{A,r} \Rightarrow \neg x_{C,m}) \wedge (x_{A,m} \Rightarrow \neg x_{C,l} \wedge \neg x_{C,r})$$

$$\neg x_{A,l}$$

$$(x_{C,l} \Rightarrow \neg x_{B,m}) \wedge (x_{C,m} \Rightarrow \neg x_{B,r})$$

$$\bigwedge_{1 \leq p \leq 3} \bigvee_{1 \leq c \leq 3} x_{p,c}$$

$$\bigwedge_{1 \leq p \leq 3} \bigwedge_{1 \leq c, d \leq 3, c \neq d} (\neg x_{p,c} \vee \neg x_{p,d})$$

$$\bigwedge_{1 \leq p, q \leq 3, p \neq q} \bigwedge_{1 \leq c \leq 3} (\neg x_{p,c} \vee \neg x_{q,c})$$

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

Example: Course Selection

- You have to take CS Modeling, Physics, or Chemistry
- For CS Modeling, you also need Discrete Math
- For Verification, you need CS Modeling
- For Physics and Chemistry, you need Statistics
- Statistics and Discrete Math are at the same time
- CS Modeling and Physics are at the same time
- Verification and Chemistry are at the same time

Is it possible to take Verification and all preliminaries?

Is it possible to take Physics and Discrete Math?

Solution: Course Selection

```
(declare-const Verification Bool)
; ...
```

```
(assert
  (and
    (or ComputerScienceModelling Physics Chemistry)
    (=> ComputerScienceModelling DiscreteMath)
    (=> Verification ComputerScienceModelling )
    (=> (or Physics Chemistry) Statistics)
    (xor Statistics DiscreteMath)
    (xor ComputerScienceModelling Physics)
    (xor Verification Chemistry)
  )
)
```

```
(assert Verification)
(check-sat)
(get-model)
```

Exercise: Seating of Wedding Guests

- Use Z3 to check whether we can assign suitable seats to all wedding guests
- There are three chairs in a row: left, middle, right.
- We want to 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.

Solutions in example files

04-wedding.smt2 / .py

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 Z3 for Homework

- Here is an excerpt from a proof in the first homework assignment:

```
valid: P ==> WP(assert R, Q)
iff
valid: P ==> (R && Q)
iff
valid: P ==> R
and valid: (P && R) ==> Q

iff
valid: SAFE(p, assert R)
and valid: SP(P, assert R) ==> Q
```

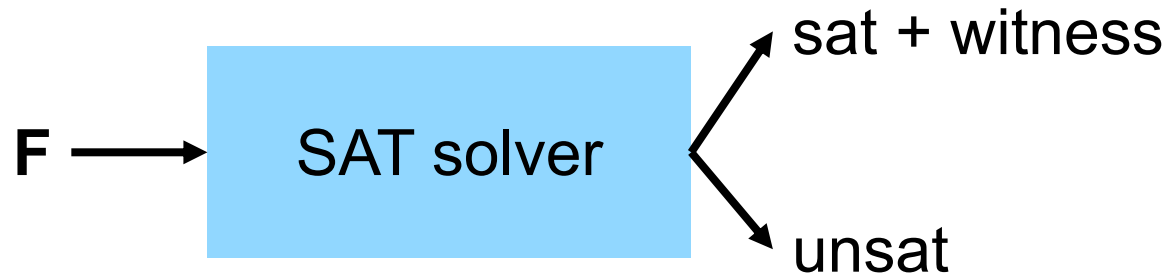
- Use Z3 to prove the blue equivalence.

Solution

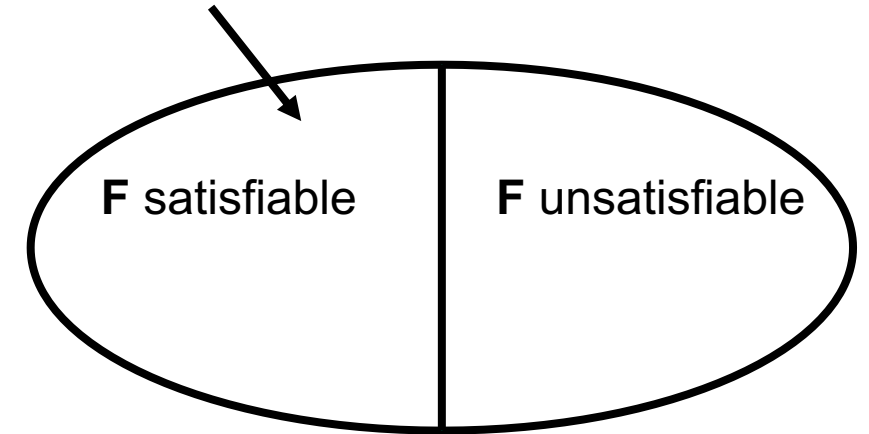
see code examples

Using a SAT solver

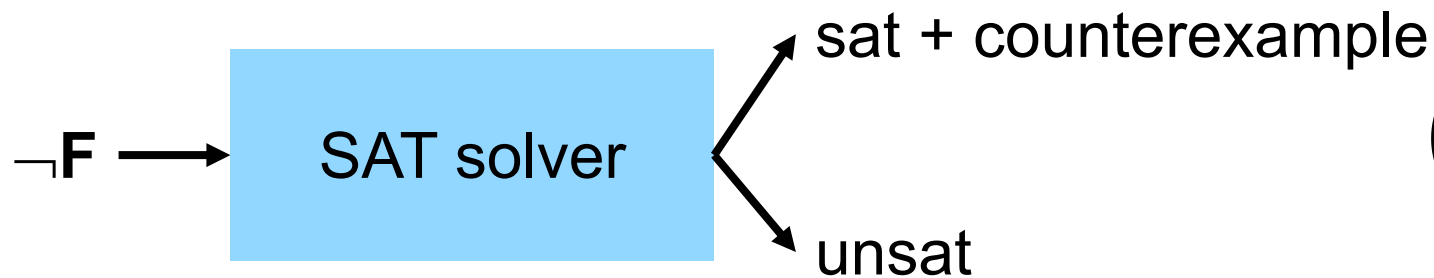
- Is F satisfiable?



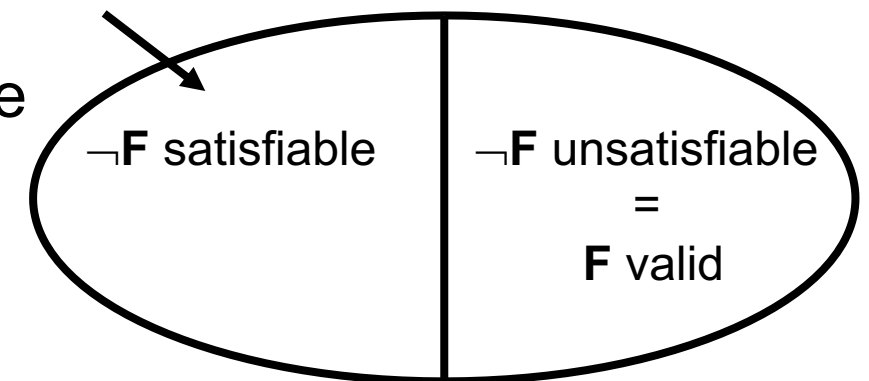
witness: model of F



- Is F valid?



counterexample: model of $\neg F$



Using a SAT Solver for Program Verification

Main steps of a tool for checking that $\{ P \} S \{ Q \}$ is valid:

1. Compute $WP(S, Q)$

→ last lecture

2. Check whether entailment $P \implies WP(S, Q)$ 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  $\Rightarrow a \wedge \dots$ 
{ a  $\wedge$  (b  $\wedge$  (true  $\Leftrightarrow$  (a  $\Rightarrow$  b))  $\vee$   $\neg$ b  $\wedge$  (false  $\Leftrightarrow$  (a  $\Rightarrow$  b)))  $\vee$   $\neg$ a  $\wedge$  (true  $\Leftrightarrow$  (a  $\Rightarrow$  b)) }
if (a) {
  { b  $\wedge$  (true  $\Leftrightarrow$  (a  $\Rightarrow$  b))  $\vee$   $\neg$ b  $\wedge$  (false  $\Leftrightarrow$  (a  $\Rightarrow$  b)) }
  if (b) {
    { true  $\Leftrightarrow$  (a  $\Rightarrow$  b) }
    res := true
    { res  $\Leftrightarrow$  (a  $\Rightarrow$  b) }
  } else {
    { false  $\Leftrightarrow$  (a  $\Rightarrow$  b) }
    res := false
    { res  $\Leftrightarrow$  (a  $\Rightarrow$  b) }
  }
} else {
  { true  $\Leftrightarrow$  (a  $\Rightarrow$  b) }
  res := true
  { res  $\Leftrightarrow$  (a  $\Rightarrow$  b) }
}
{ res  $\Leftrightarrow$  (a  $\Rightarrow$  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:

$$(x == X \ \&\& \ y == Y) \implies y == Y \ \&\& \ y - Y == 0$$

- 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

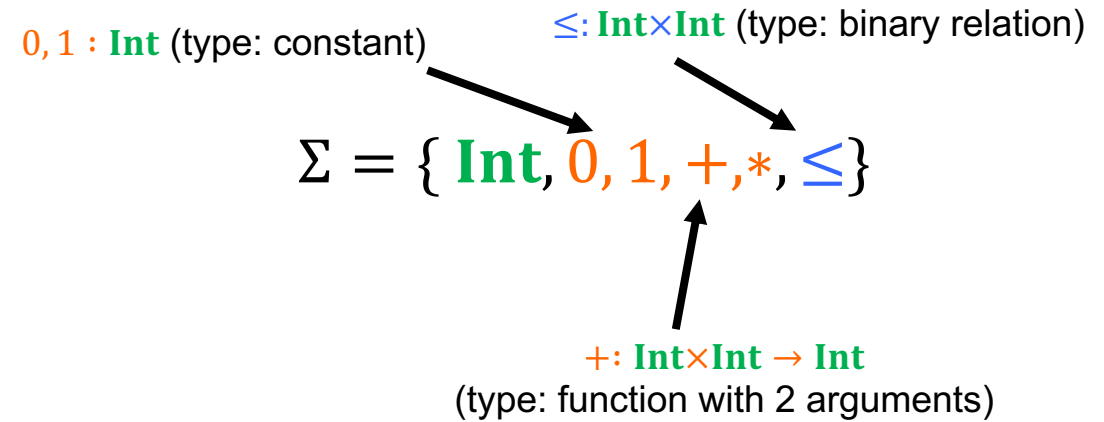
\wedge \vee \neg \Rightarrow \Leftrightarrow \exists \forall ...

6. An equality symbol

=

FOL Formulas

- A **signature** Σ is a set of
 - at least one sort
 - function symbols
 - relational symbols (= does not count)
- A Σ -**formula** is a **logical formula** over **propositions** built from **symbols** in Σ



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

Is this Σ -formula satisfiable?

FOL Formulas

Is this Σ -formula satisfiable?

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned} & \mathfrak{I}(\mathit{plus}(\mathit{plus}(\mathit{one}, \mathit{one}), x)) \\ &= \mathit{plus}^{\mathfrak{A}}(\mathit{plus}^{\mathfrak{A}}(\mathit{one}^{\mathfrak{A}}, \mathit{one}^{\mathfrak{A}}), \beta(x)) \\ &= (1+1) + \beta(x) \end{aligned}$$

FOL Semantics

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

we can always express equality between terms

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

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

Issues with FOL Satisfiability

- 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

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

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

infix notation for $\mathit{leq}(\mathit{plus}(x, \mathit{one}), \mathit{zero})$

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

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

Satisfiability Modulo Theories

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

A Σ -formula F is **satisfiable modulo \mathbf{Th}** iff there exists a Σ -interpretation \mathfrak{I} such that

- $\mathfrak{I} \models F$, and
- $\mathfrak{I} \models \mathbf{G}$ for every sentence \mathbf{G} in \mathbf{Th} .

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

Exercise

- Consider the signature $\Sigma = \{ \mathbf{Nat}, zero, one, plus, leq \}$,
 $zero: \mathbf{Nat}, one: \mathbf{Nat}, plus: \mathbf{Nat} \times \mathbf{Nat} \rightarrow \mathbf{Nat}, eq: \mathbf{Nat} \times \mathbf{Nat}$
- the theory **Th** given by the axioms

$$\forall x: \mathbf{Nat} (x \text{ leq } x) \quad \forall x: \mathbf{Nat} \forall y: \mathbf{Nat} (x \text{ plus } y \text{ leq } y \text{ plus } x)$$

- and the formula $F ::= \exists x: \mathbf{Nat} (x \text{ plus } zero \text{ leq } one)$.

a) Give a model witnessing that **F** is satisfiable modulo **Th**.

b) Propose an axiom such that **F** is also valid modulo **Th**.

Solution


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

$\forall x: \mathbf{Nat} (x \text{ leq } x)$

$\forall x: \mathbf{Nat} \forall y: \mathbf{Nat} (x \text{ plus } y \text{ leq } y \text{ plus } x)$


a)

$\mathfrak{I}(x) = 1$
 $\mathbf{Nat} = \mathbb{N},$
 $one^{\mathfrak{A}} ::= 1,$
 $zero^{\mathfrak{A}} ::= 0,$
 $plus^{\mathfrak{A}} ::= +,$
 $leq ::= =$



b)

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



Th-valid after adding axiom

$\forall x: \mathbf{Nat} (x \text{ plus zero leq } x)$

Some important theories

- Arithmetic (with canonical axioms)

- Presburger arithmetic: $\Sigma = \{ \text{Int}, <, 0, 1, + \}$ **decidable**
- Peano arithmetic: $\Sigma = \{ \text{Int}, <, 0, 1, +, * \}$ **undecidable**
- Real arithmetic: $\Sigma = \{ \text{Real}, <, 0, 1, +, * \}$ **decidable**

- EUF: Equality logic with Uninterpreted Functions **decidable**

- $\Sigma = \{ \mathbf{U}, =, f, g, h, \dots \}$
- 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: $P \implies WP(S, Q)$ valid

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

A Σ -formula F is **satisfiable modulo Th** iff there exists a Σ -interpretation \mathfrak{I} such that

- $\mathfrak{I} \models F$, and
- $\mathfrak{I} \models G$ for every sentence G in **Th** .

Enriched verification condition:

$P \implies WP(S, Q)$ valid modulo custom theory
iff **$BP \ \&\& \ P \ \&\& \ \neg WP(S, Q)$ unsat**
iff **$BP \implies P \implies WP(S, Q)$ valid**

Background Predicate:
conjunction of all our axioms
defining our theory

Automating Program Verification

Main 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 $BP \implies P \implies WP(S, Q)$ 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
```



Exercise: Color Axioms

- Axiomatize a custom type for colors
 - `Color = Black | White | Red | Green | Blue | Yellow`
- Your type should support two operations:
 1. A function `isInDanishFlag` to determine whether a color appears in the Danish flag
 2. A function `mix` that takes two colors and returns the color obtained from adding them (see right); it does not matter what the function returns when the resulting color is not supported
- Verify the program on the right with Z3



```
{ x == Red }  
var g: Color := Green  
var b: Color := Blue  
r := mix(b, mix(x, g))  
{ isInDanishFlag(r) }
```


Solution

```
// BP ==>
{ x == Red } // ==>
{ isInDanishFlag(mix(Blue, mix(x, Green)) ) }
var g: Color := Green
{ isInDanishFlag(mix(Blue, mix(x, g)) ) }
var b: Color := Blue
{ isInDanishFlag(mix(b, mix(x, g)) ) }
r := mix(b, mix(x, g))
{ isInDanishFlag(r) }
```

```
; declarations
(declare-sort Color)
...

; background predicate
(assert (forall ...))

; precondition
(assert (= x Red))

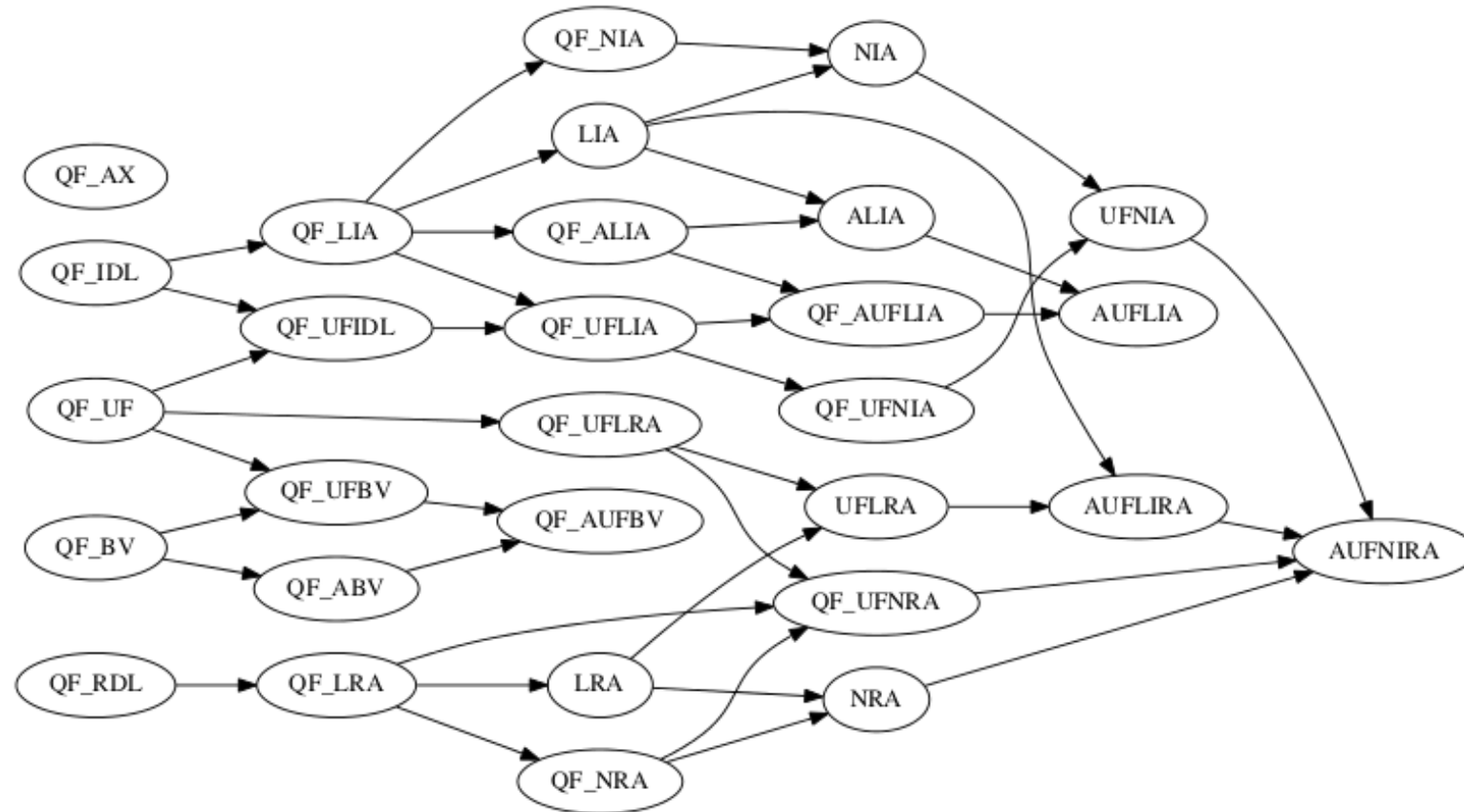
; !WP(S, Q)
(assert (not
  (isInDanishFlag ...)
))

(check-sat)
```



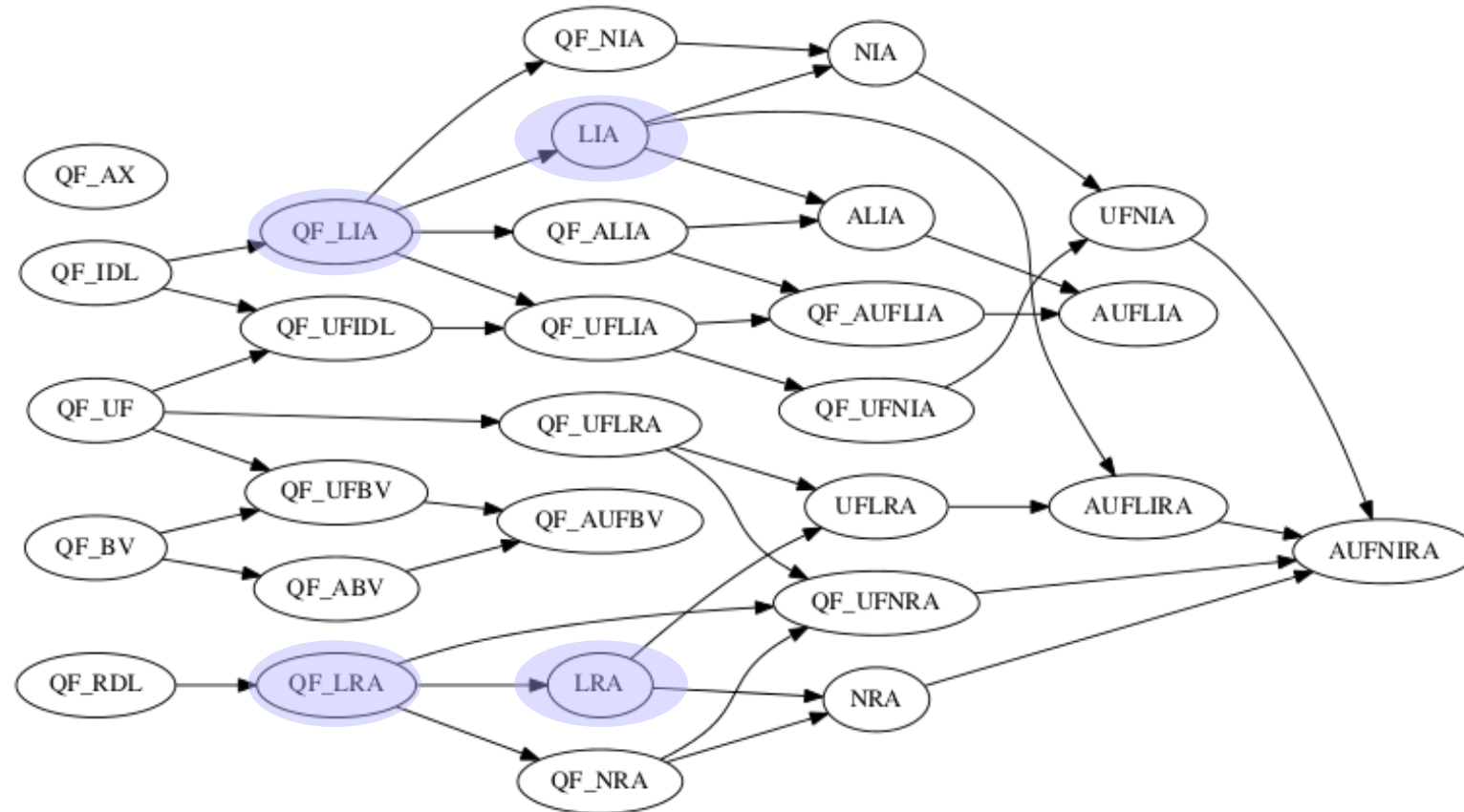
Full solution: 02-smt/09b-colors-verification.smt

Z3 built-in theories



➔ no explicit background predicate needed

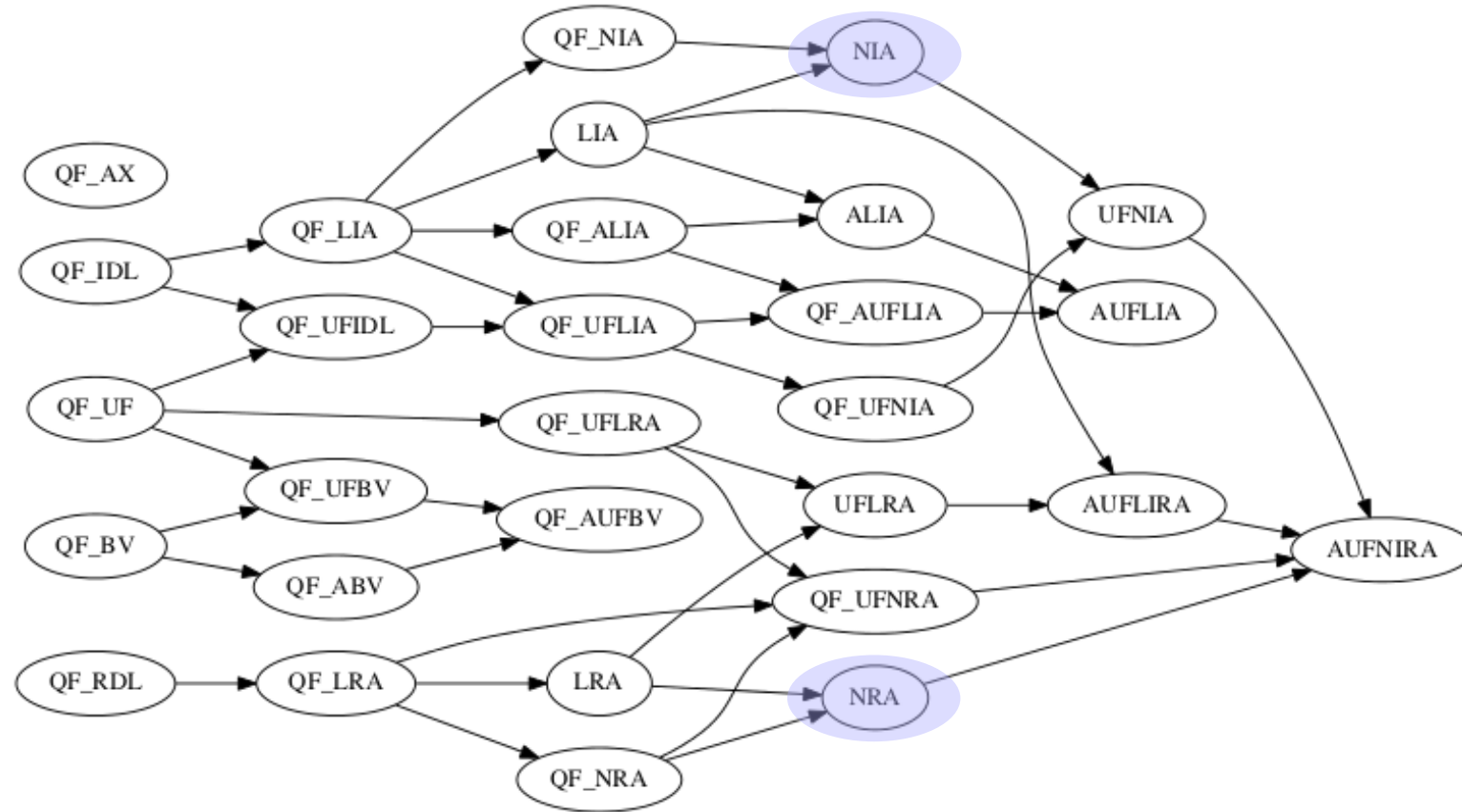
Z3 built-in theories



(Quantifier-free) Linear Integer/Real Arithmetic

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

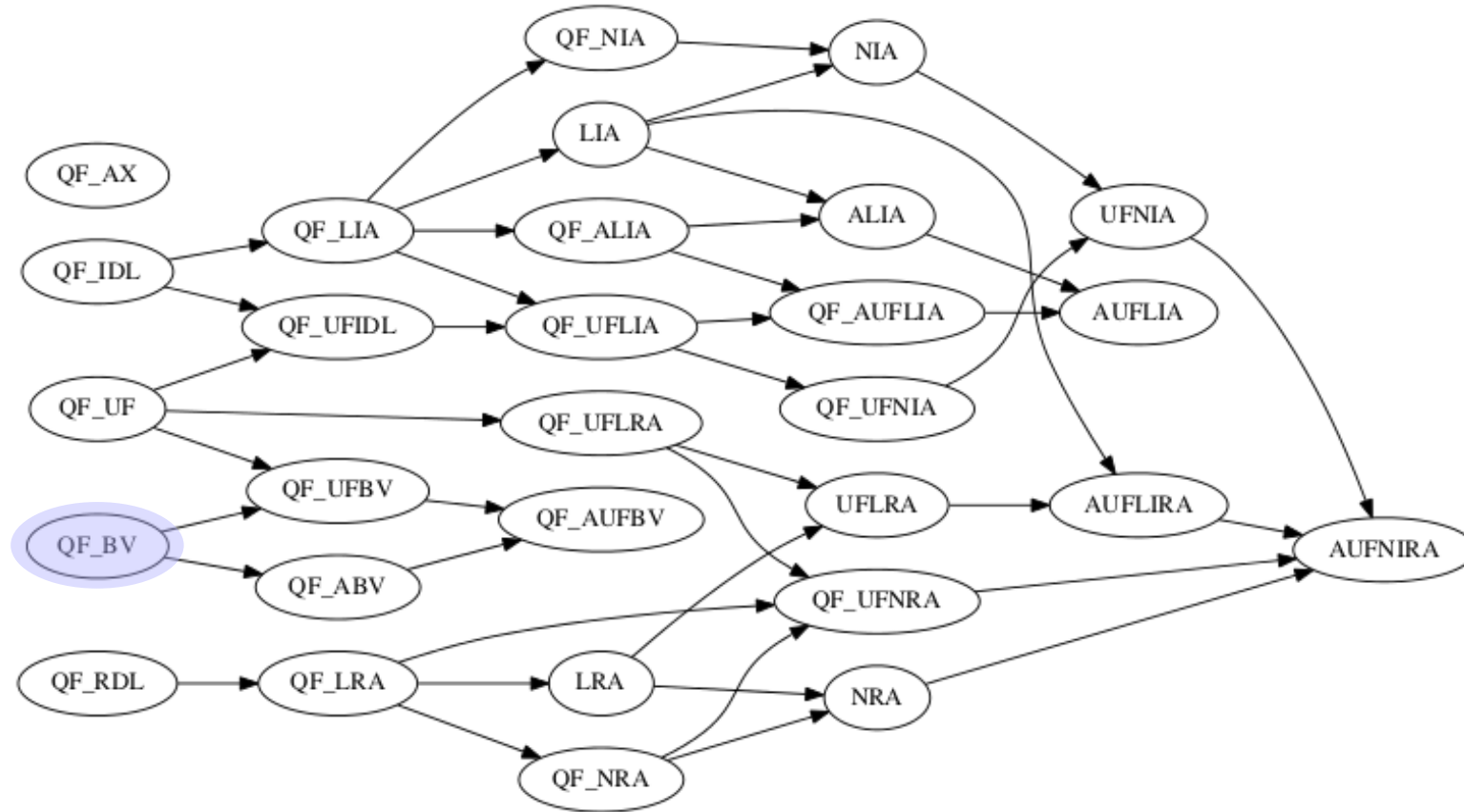
Z3 built-in theories



Non-Linear Integer/Real Arithmetic

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

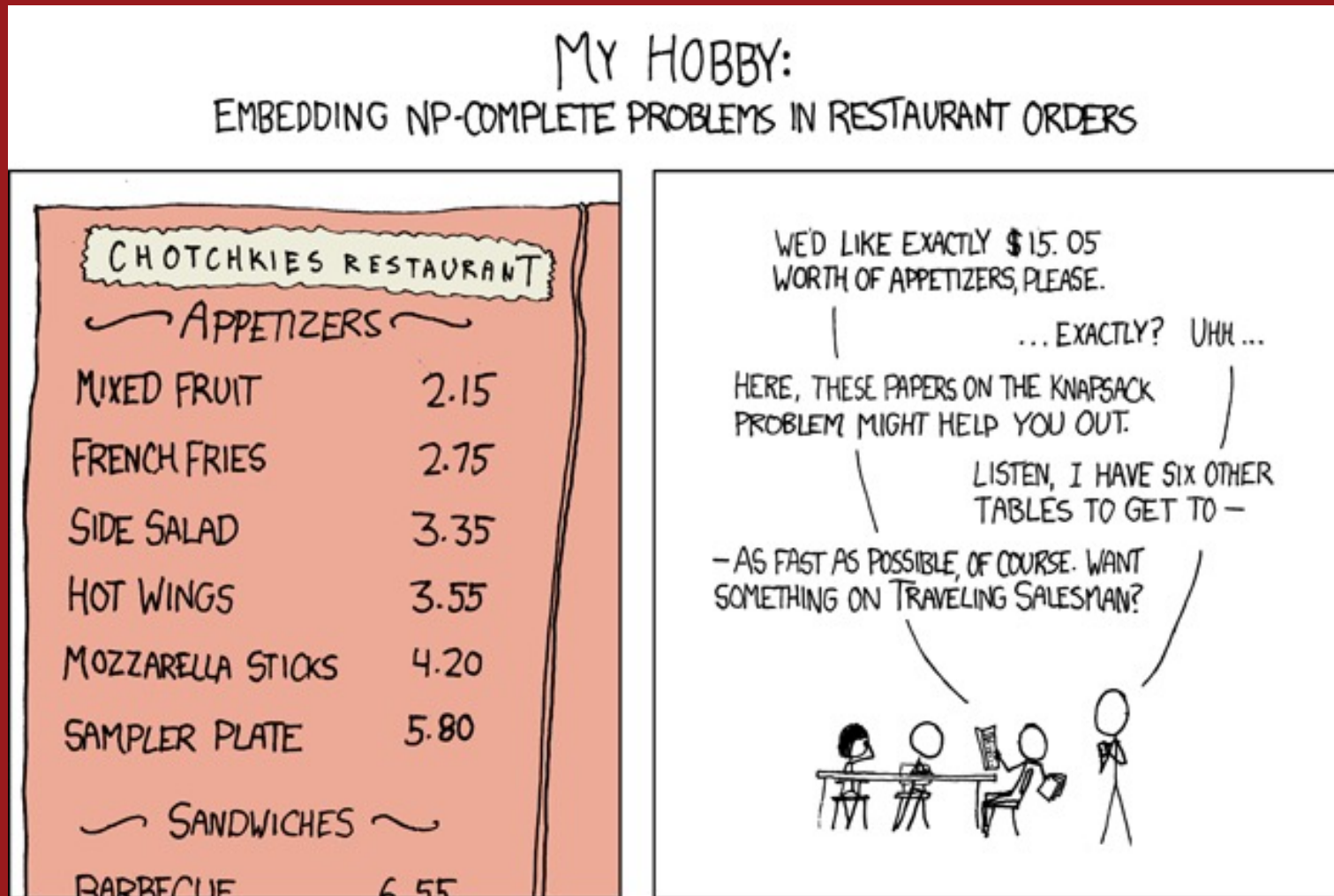
Z3 built-in theories



Quantifier-free fixed-size bitvector arithmetic

$$x \& y \leq x | y$$

Exercise: use Z3 to find *all* suitable restaurant orders



<https://xkcd.com/287/>

Solution

see code examples

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: $\text{Pre} \ \&\& \ !\text{WP}(S, \text{Post})$

```
; 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

Working with quantifiers is non-trivial

```
(assert
  (exists ((x Int))
    (forall ((y Real))
      (=> (> y x) (> (* y y) 1))
    )
  )
)

(check-sat)
```

```
$ z3 ...
sat
```

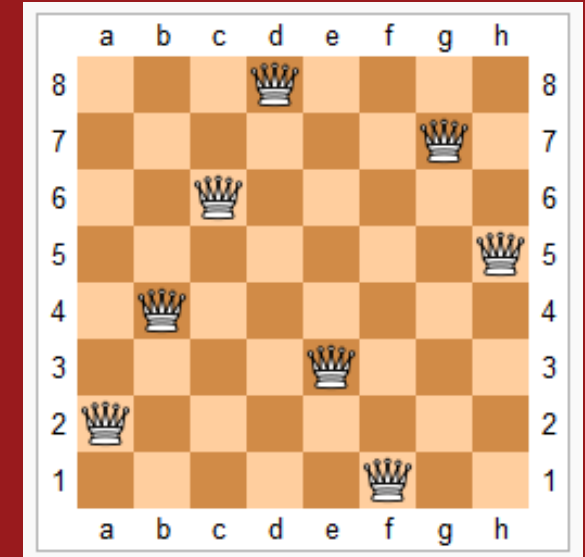
```
(assert
  (forall ((x Real))
    (exists ((y Real))
      (= x (* y y))
    )
  )
)

(check-sat)
```

```
$ z3 ...
unknown
```

Exercise (not covered in class)

- The N-queens problem is to place N-queens on an N x N chess board such that no two queens threaten each other.
- Use Z3 to compute a solution to the N-queens problem for any given N.
- Hints:
 - We recommend using Z3Py or another Z3 API such that you can write programs around your Z3 queries.
 - Represent the board using multiple integer variables, e.g. $X_2 = 5$ means the queen is in row 5 in column 2.
 - `distinct(l)` is a shortcut for stating all elements of list `l` are pairwise disjoint.
 - You can easily check the diagonals by shifting the queens vertically and then checking the rows.



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

Solution

see code examples

Wrap-up

Main 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 $BP \implies P \implies WP(S, Q)$ is valid

→ SMT solver

- Check satisfiability of negation: $BP \ \&\& \ P \ \&\& \ !WP(S, Q)$

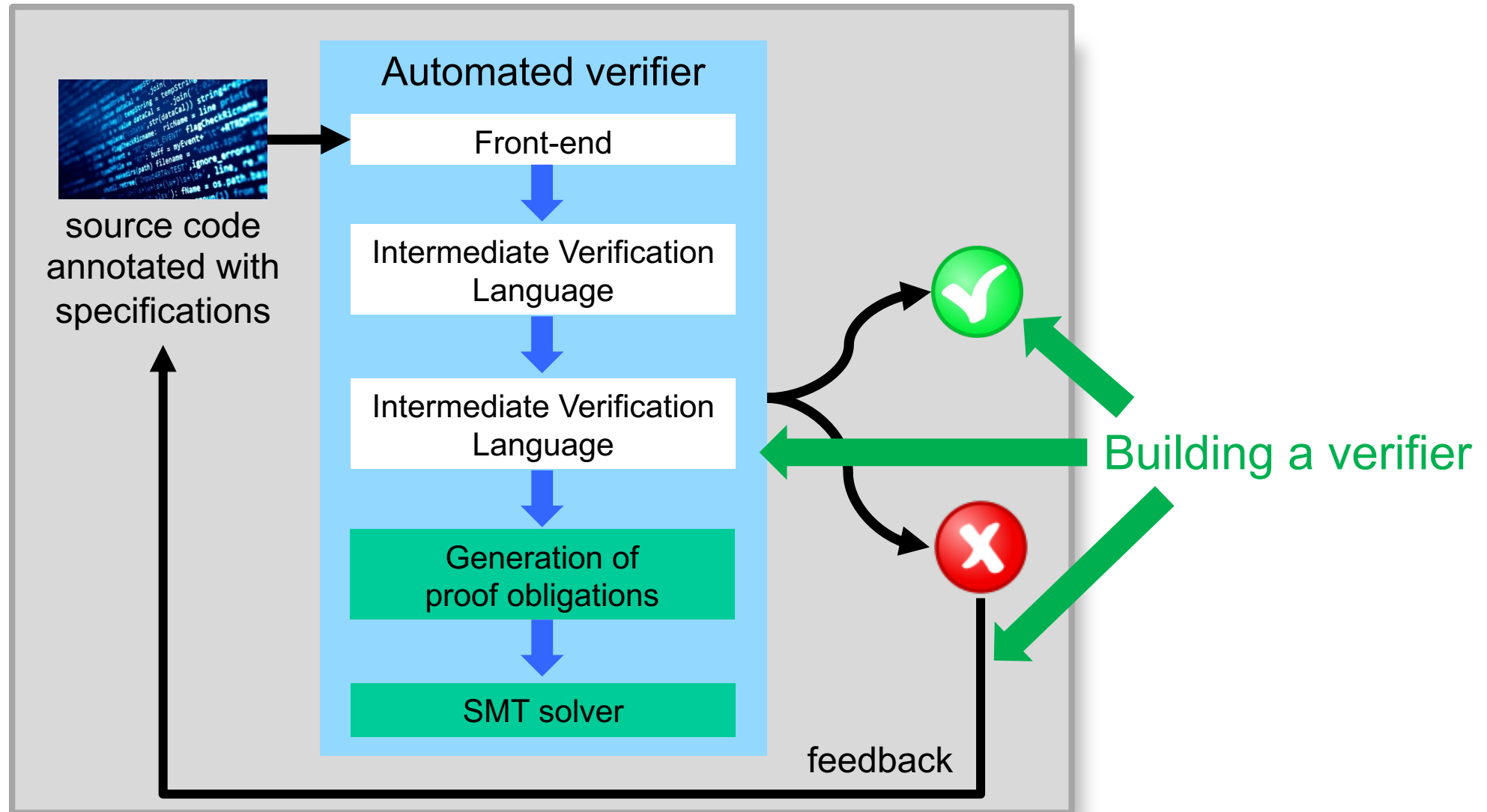
- unsat → 

- sat →  model explains why $\{ P \} S \{ Q \}$ is not valid

- unknown → decidability issues, strengthen theory, hacks

→ future classes

What next?



Questions, murky points, feedback



<https://forms.gle/Nds2CwBtEdUmR4qQ8>