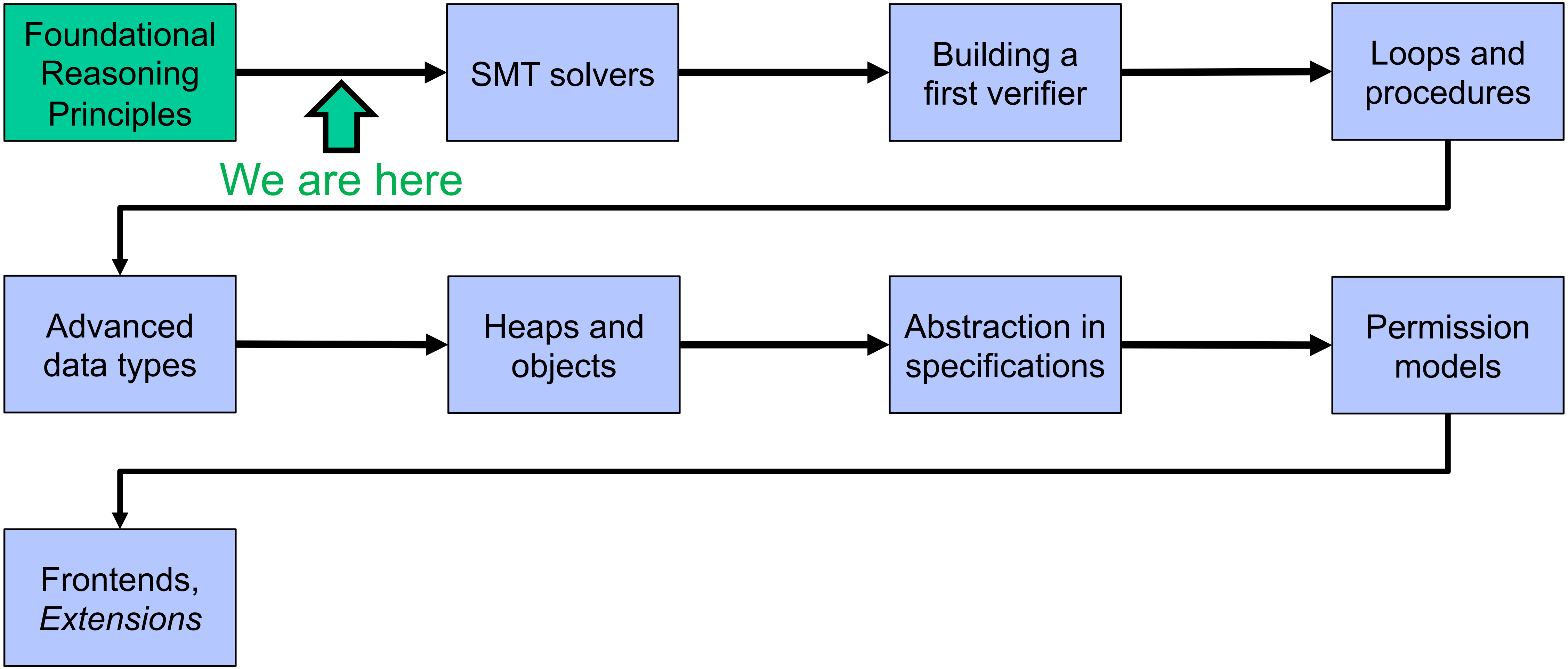
02245 – Lecture 2

FOUNDATIONS & SMT SOLVERS

Tentative course outline



Foundational Reasoning Principles

SMT solvers

Building a first verifier

Loops and procedures

Advanced data types

Heaps and objects

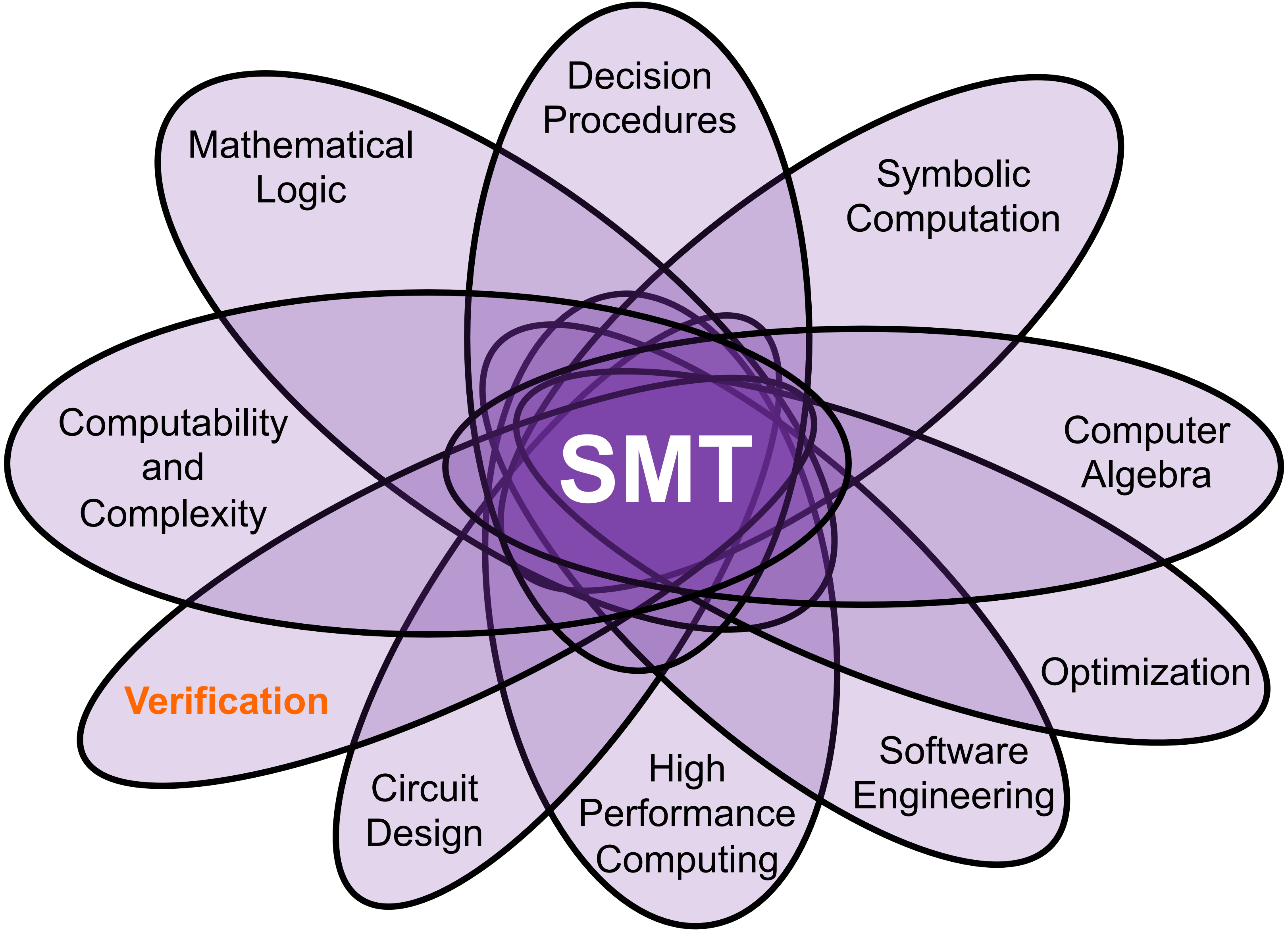
Abstraction in specifications

Permission models

Frontends, *Extensions*

We are here

Satisfiability Modulo Theories Solvers

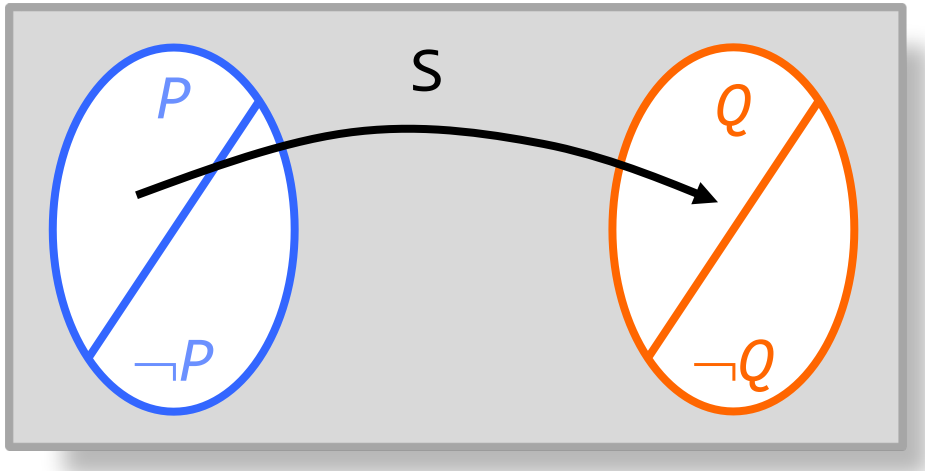


- A foundational topic in theoretical and applied computer science

- Our focus: effectively applying SMT technology to program verification

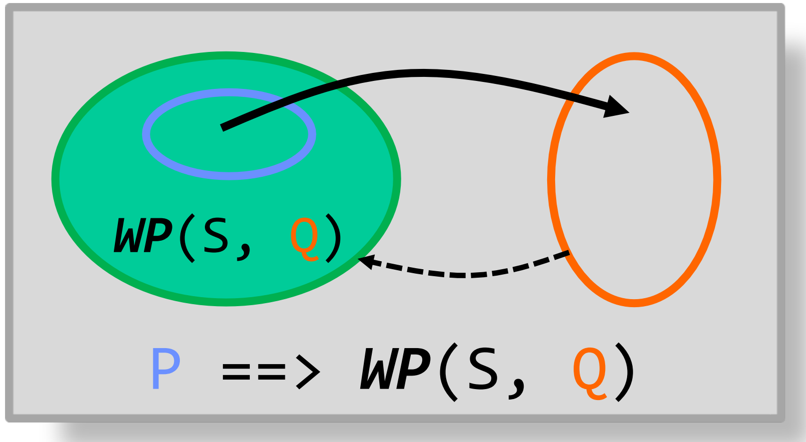
But first: Recap

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*.

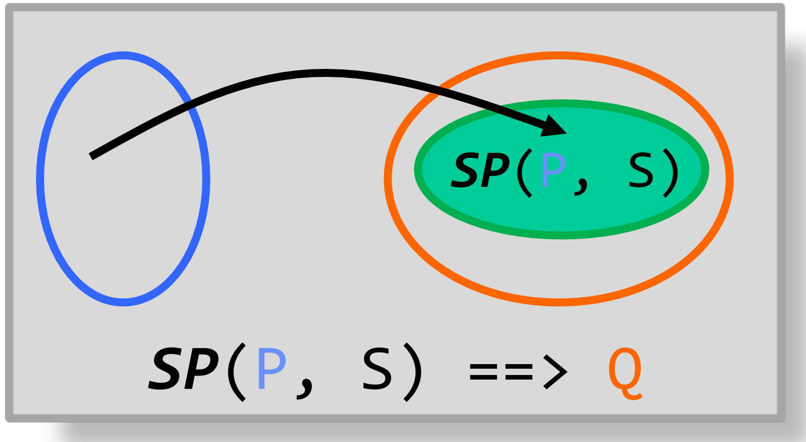


```
method foo(x: Int)
  returns (r: Int)
  requires x > 0
  ensures r > y
{
  // S
  var y: Int := 7
  r := x + y
}
```

Recap: Weakest Pre & Strongest Post



P ==> *WP*(S, Q)



SP(P, S) ==> Q

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

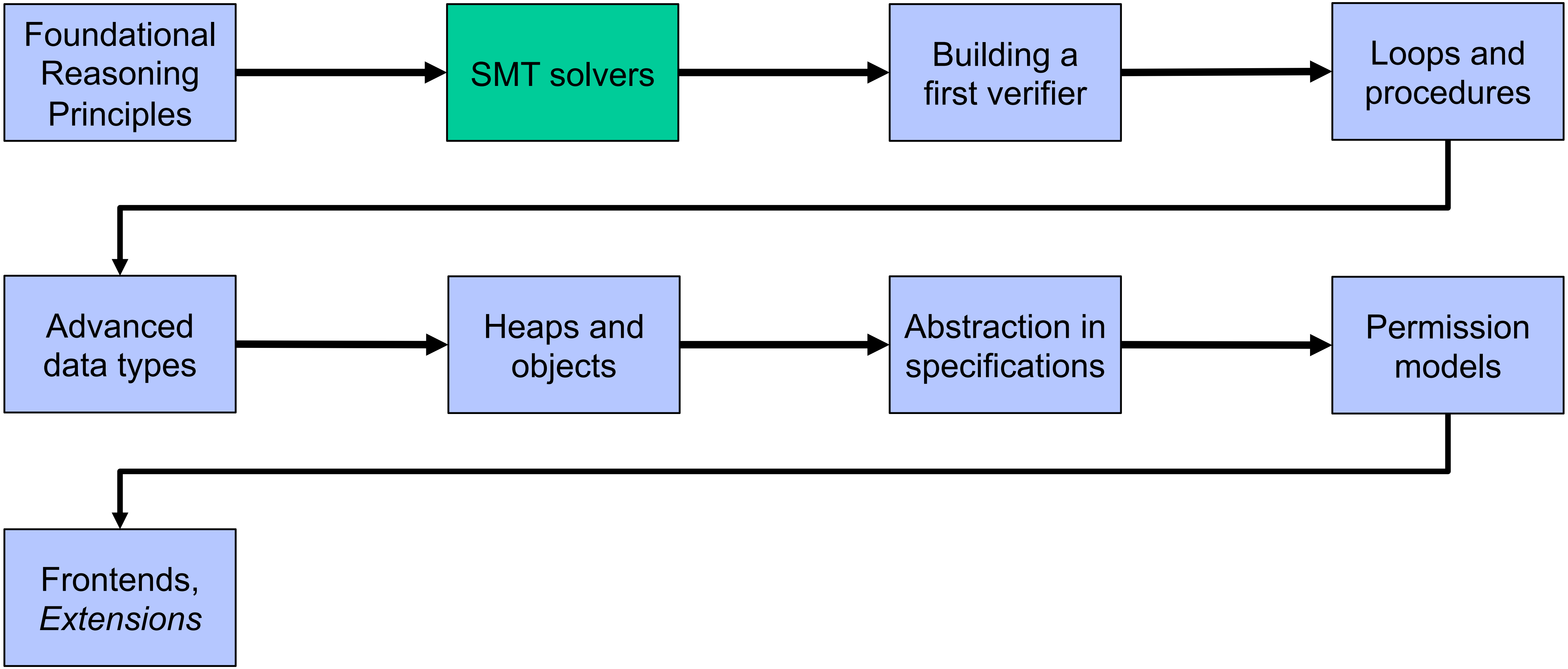
   and ***SAFE***`(P, S)` is valid

➔ Homework

< Homework W1 >

Solutions will be published on course page

Tentative course outline



Foundational Reasoning Principles

SMT solvers

Building a first verifier

Loops and procedures

Advanced data types

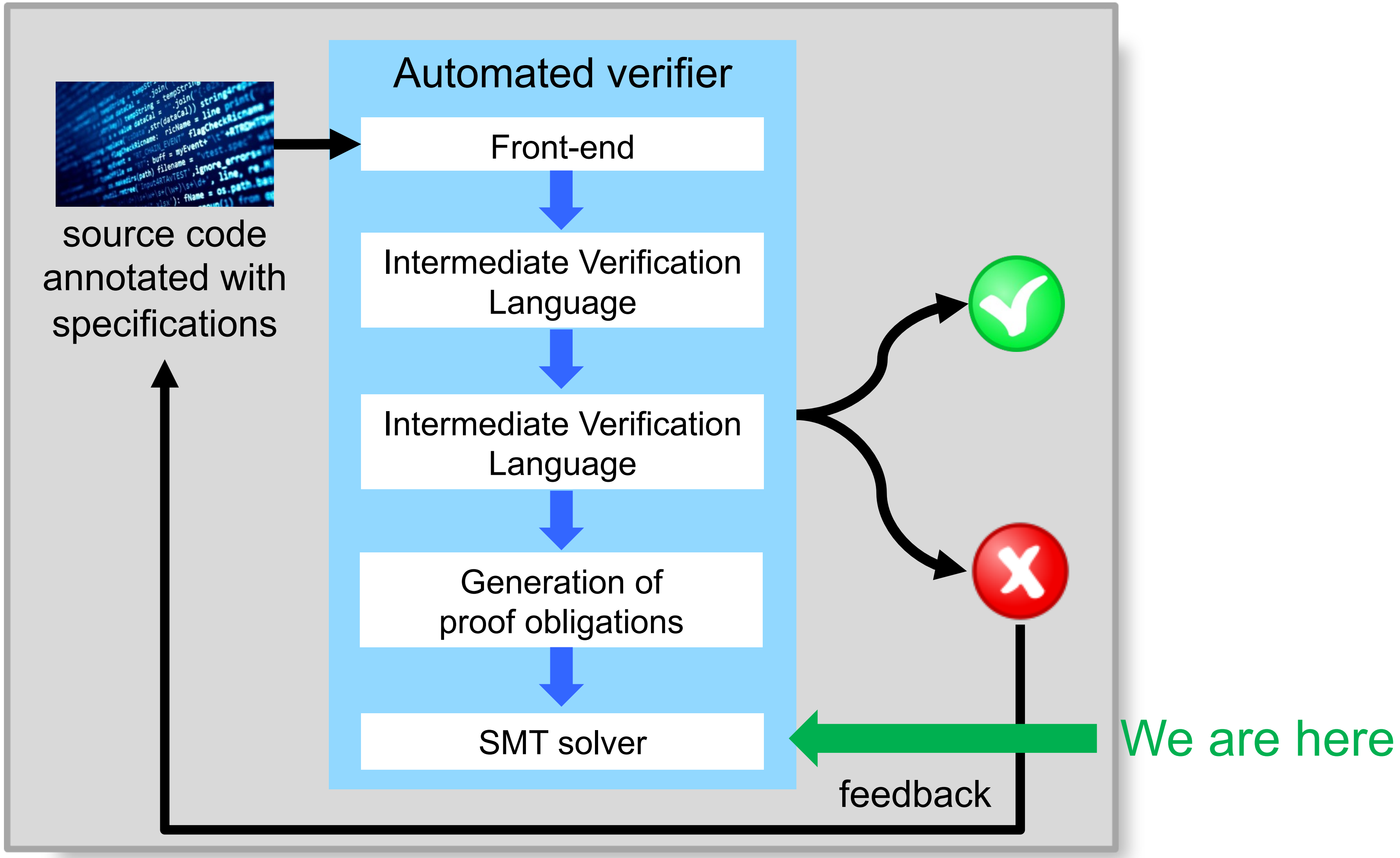
Heaps and objects

Abstraction in specifications

Permission models

Frontends, *Extensions*

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

$\mathtt{F} ::= \mathtt{false} \mid \mathtt{true} \mid \mathtt{X} \mid \neg\mathtt{F} \mid \mathtt{F} \land \mathtt{F} \mid \mathtt{F} \lor \mathtt{F} \mid \mathtt{F} \Rightarrow \mathtt{F} \mid \mathtt{F} \Leftrightarrow \mathtt{F}$

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 $\mathfrak{I}(\mathtt{X}) = \text{true}$

$\mathfrak{I} \models \neg\mathtt{F}$ iff not $\mathfrak{I} \models \mathtt{F}$

$\mathfrak{I} \models \mathtt{F} \land \mathtt{G}$ iff $\mathfrak{I} \models \mathtt{F}$ and $\mathfrak{I} \models \mathtt{G}$

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

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

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

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

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

Satisfiability & Validity

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

$(X \Rightarrow Y) \Rightarrow Y$

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

- **F** is unsatisfiable iff **F** has no model

$X \land \neg Y \land (X \Rightarrow Y)$

- **F** is valid iff every interpretation is a model of **F** (¬**F** is unsatisfiable)

$X \land (X \Rightarrow Y) \Rightarrow Y$

- **F** is not valid iff some interpretation is not a model of **F** (¬**F** is satisfiable)

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

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

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

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

$\mathfrak{I}\ ::=\ [$

$U = \text{false}$

$V = \text{false}$

$X = \text{true}$

$Y = \text{false}$

$Z = \text{false}$

$]$

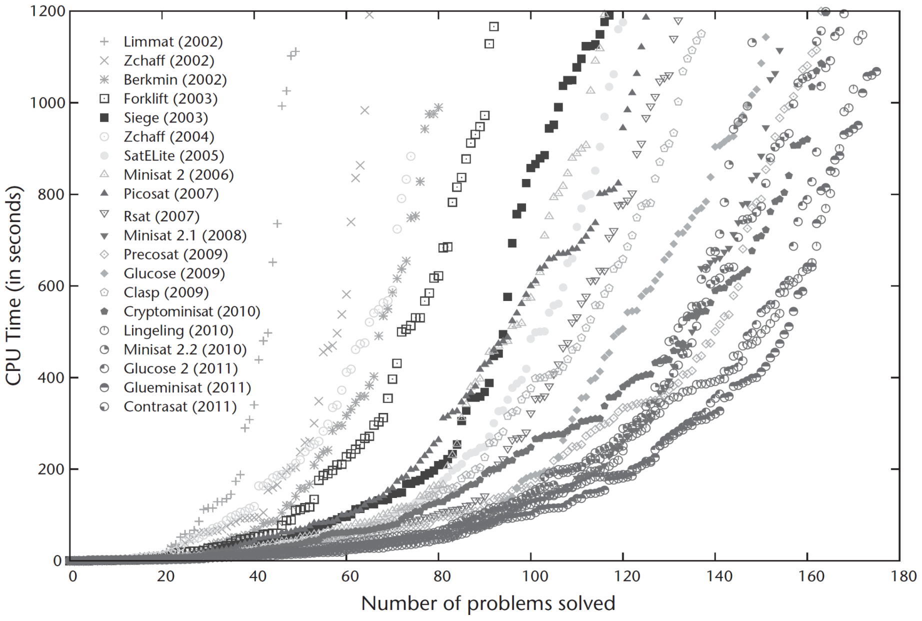
Complexity of SAT

- For formulas in conjunctive normal form (CNF), SAT is the classical NP-complete problem

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

- Many difficult problems can be efficiently encoded

- Every known algorithm is exponential in the formula's size



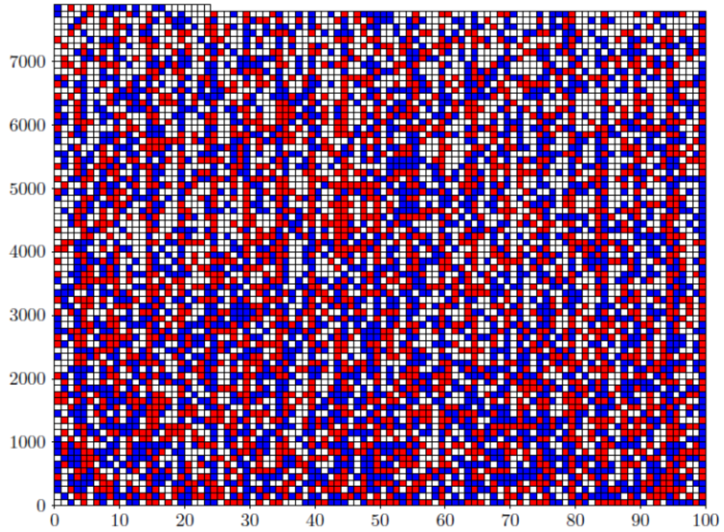
Example: Boolean Pythagorean Triples

- 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

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?

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)

```python
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]
```

¬**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?

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.

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?

Using Z3 for Homework

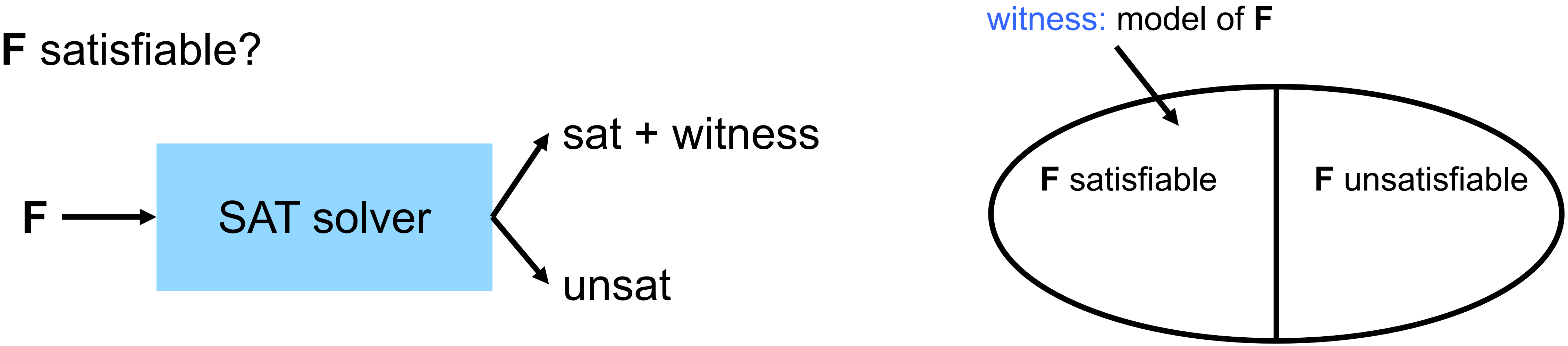
- Here is an excerpt from a proof in the first homework assignment:

```
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.

Using a SAT solver

- Is **F** satisfiable?



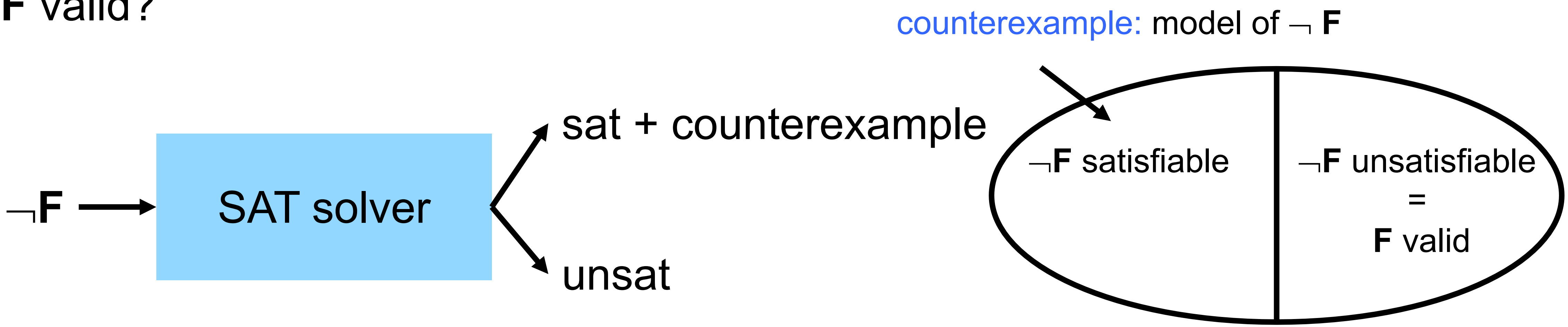
witness: model of **F**

F → SAT solver → sat + witness / unsat

F satisfiable

F unsatisfiable

- Is **F** valid?



counterexample: model of ¬ **F**

¬**F** → SAT solver → sat + counterexample / unsat

¬**F** satisfiable

¬**F** unsatisfiable
=
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` ➔ `{ P } S { Q }` is valid
   - `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	
▪ we assume a corresponding set of values	`Bool = {true, false}`
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	∧ ∨ ¬ ⇒ ⇔ ∃ ∀ …

FOL Formulas

- A signature Σ is a set of
  - at least one sort
  - function symbols
  - relational symbols (including $=$)

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

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

$$\forall \mathrm{x}: \mathbf{Int}\ \exists \mathrm{y}: \mathbf{Int}\ (\mathrm{y} = \mathrm{x} + 1 \land \mathrm{y} * \mathrm{y} = \mathrm{x} * \mathrm{x} + (1 + 1) * \mathrm{x} + 1)$$

Is this Σ-formula satisfiable?

FOL Σ-Interpretations

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

- A **Σ-structure $\mathfrak{A}$** assigns
  - a non-empty domain (set) $\mathbf{U}^{\mathfrak{A}}$ to each sort **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

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

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

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

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

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

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

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

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

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

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

FOL Semantics

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

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

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

Issues with FOL Satisfiability

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

- All symbols are uninterpreted

- The meaning of functions and relations is determined in the chosen model

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

- Many formulas are satisfiable if we can choose Σ-structures that defy the intended meaning of functions and relations

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

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

➔ Filter out unwanted Σ-structures

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

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

Exercise

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

$$\forall \mathrm{x}{:}\ \mathbf{Nat}\ (\mathrm{x}\ \mathit{eq}\ \mathrm{x}) \qquad \forall \mathrm{x}{:}\ \mathbf{Nat}\ \forall \mathrm{x}{:}\ \mathbf{Nat}\ \ (\mathrm{x}\ \mathit{plus}\ \mathrm{y}\ \ \mathit{eq}\ \ \mathrm{y}\ \mathit{plus}\ \mathrm{x})$$

- and the formula **F** $::= \exists x{:}\, \mathbf{Nat}(\, x\ \mathit{plus}\ \mathit{one}\ \mathit{eq}\ \mathit{zero})$.

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

b) Propose an axiom such that **F** is also valid 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**
  - $\Sigma = \{\, \mathbf{U}, =, f, g, h, \ldots \}$
  - 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)`

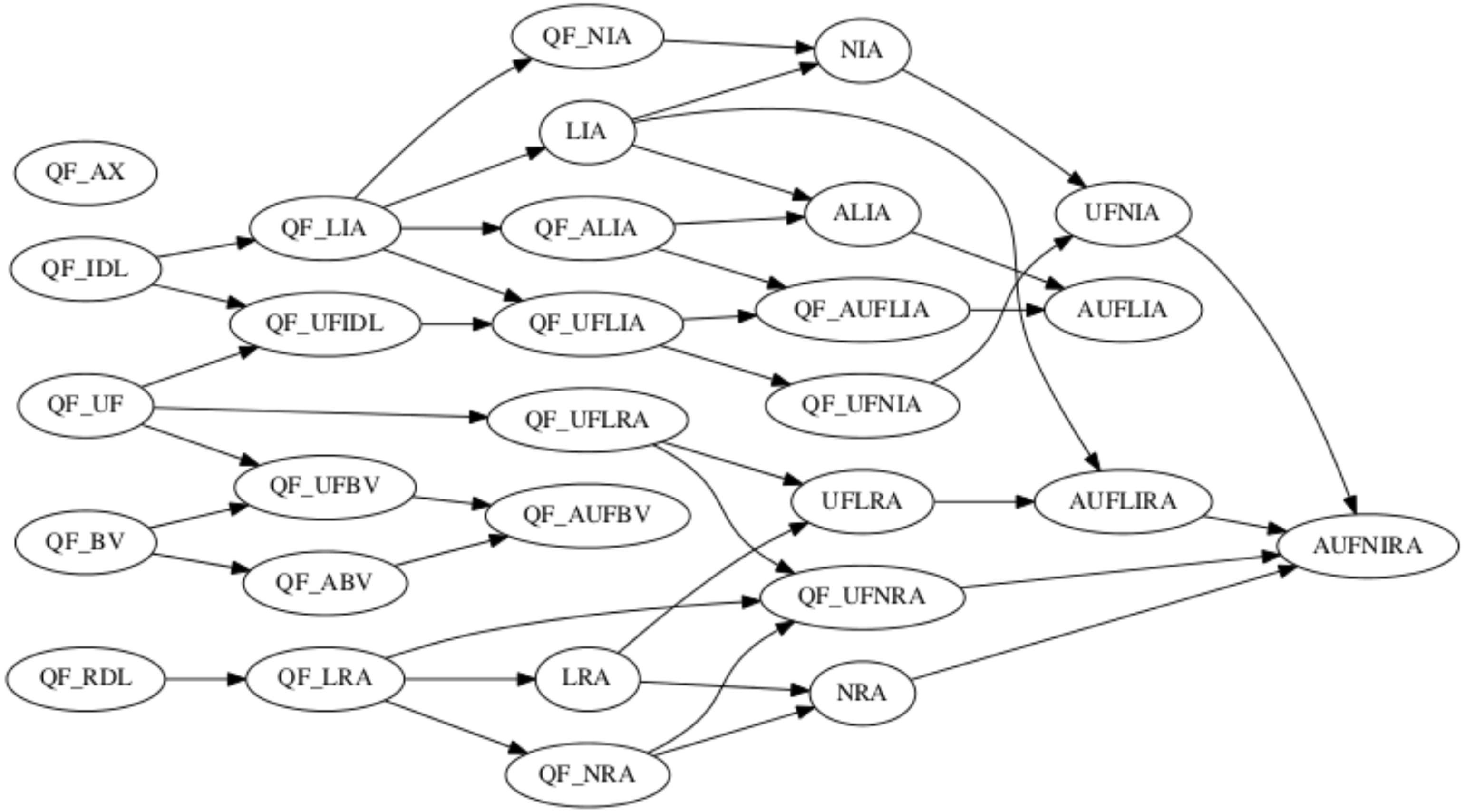
```
(declare-sort Pair)

(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)

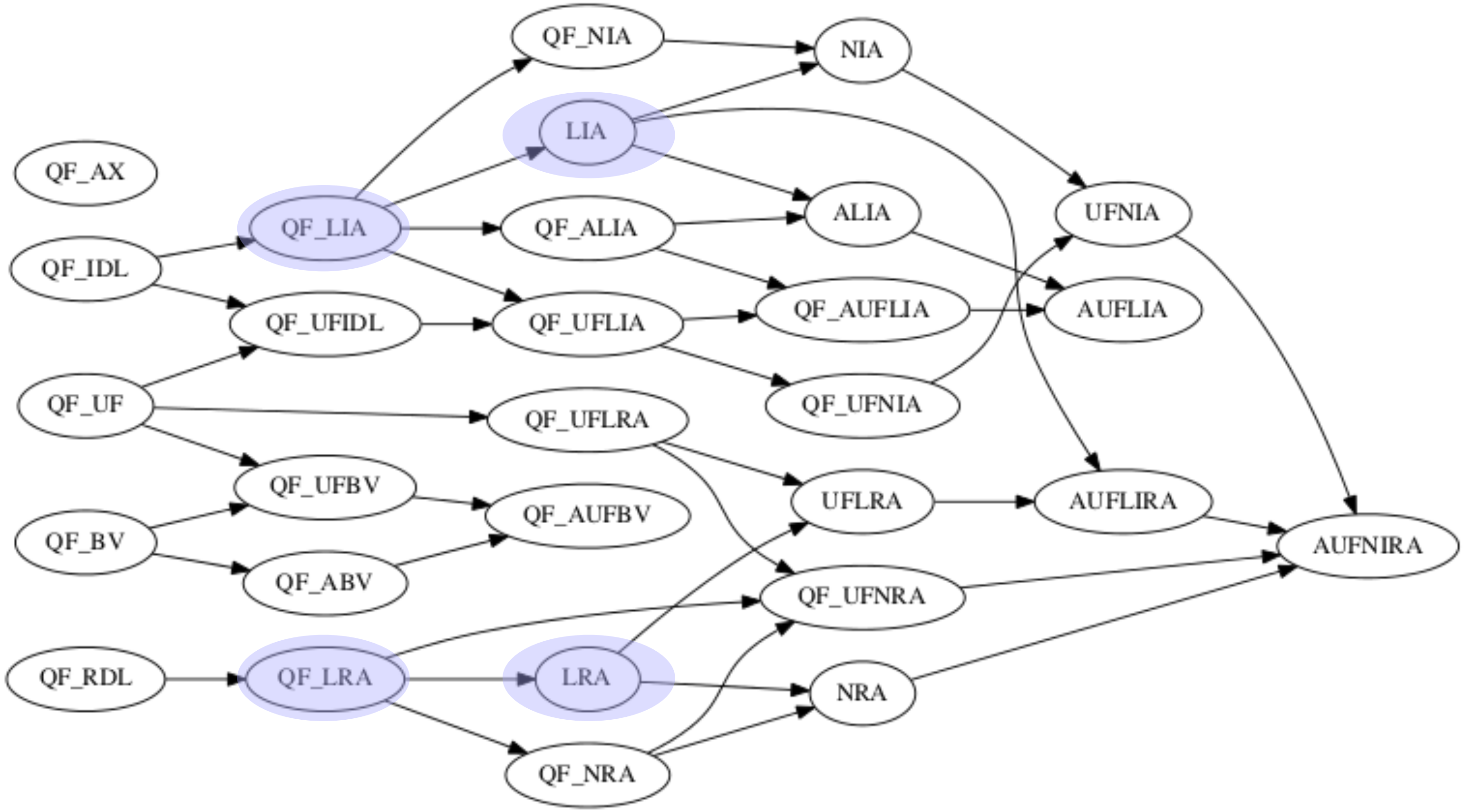
- Sorts
  - `Bool`, `Int`, `Real`, `BitVec(precision)`
  - `DeclareSort(name)` (uninterpreted)
- Uninterpreted functions are declared with parameter and return types
- Variables are uninterpreted functions of arity 0
  - `Const(name, sort)`

```
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() )
```

Z3 built-in theories



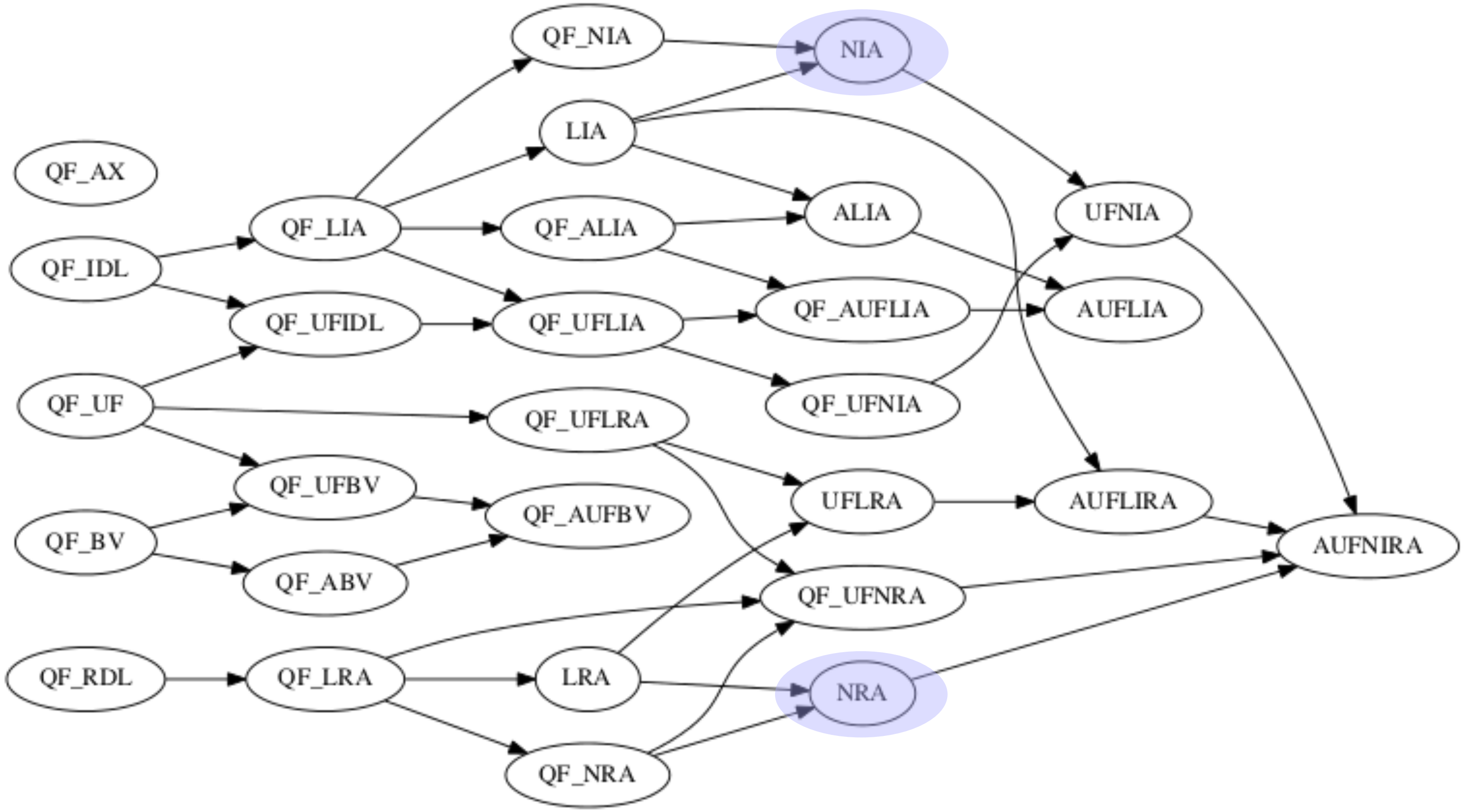
Z3 built-in theories



(Quantifier-free) Linear Integer/Real Arithmetic

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

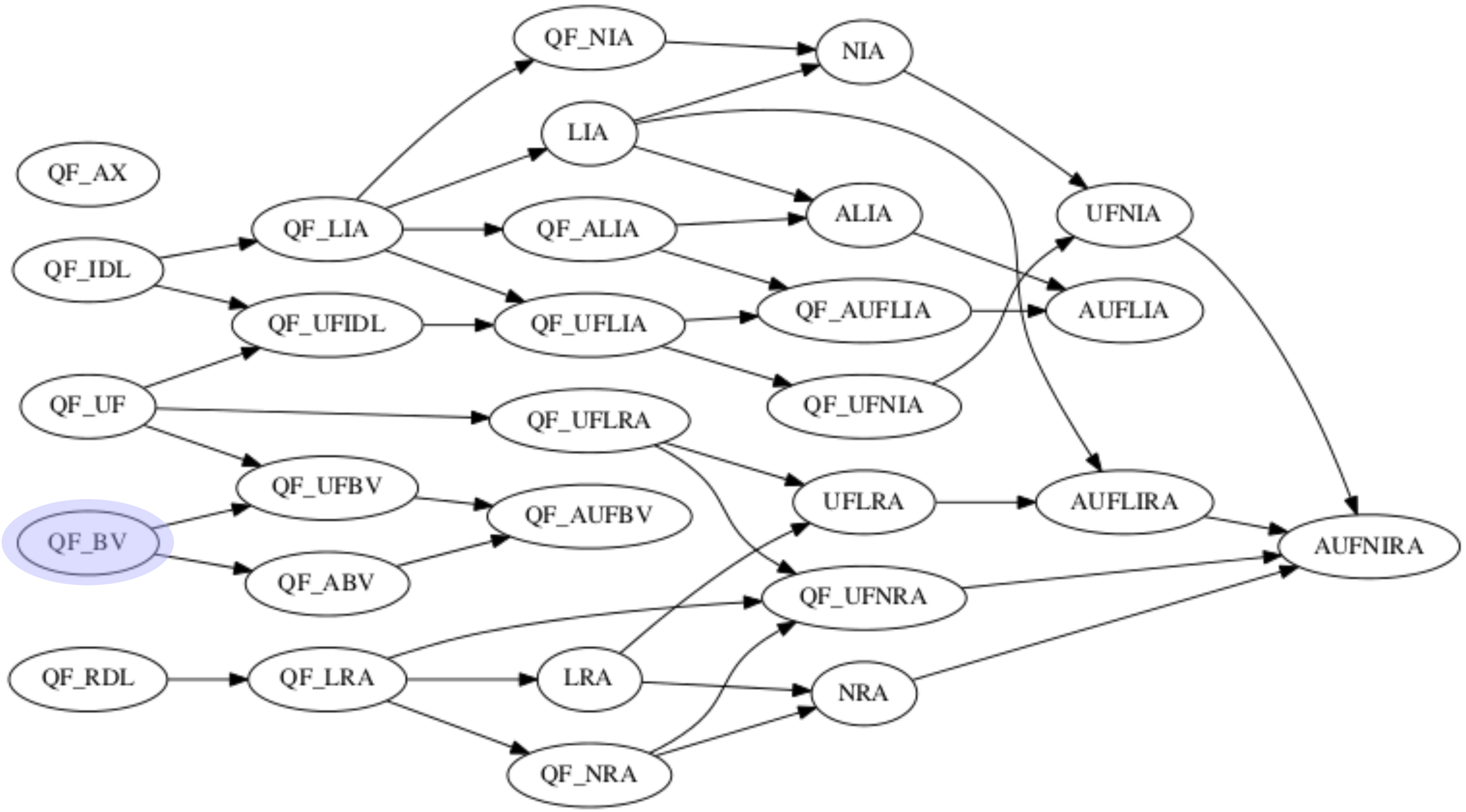
Z3 built-in theories



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

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

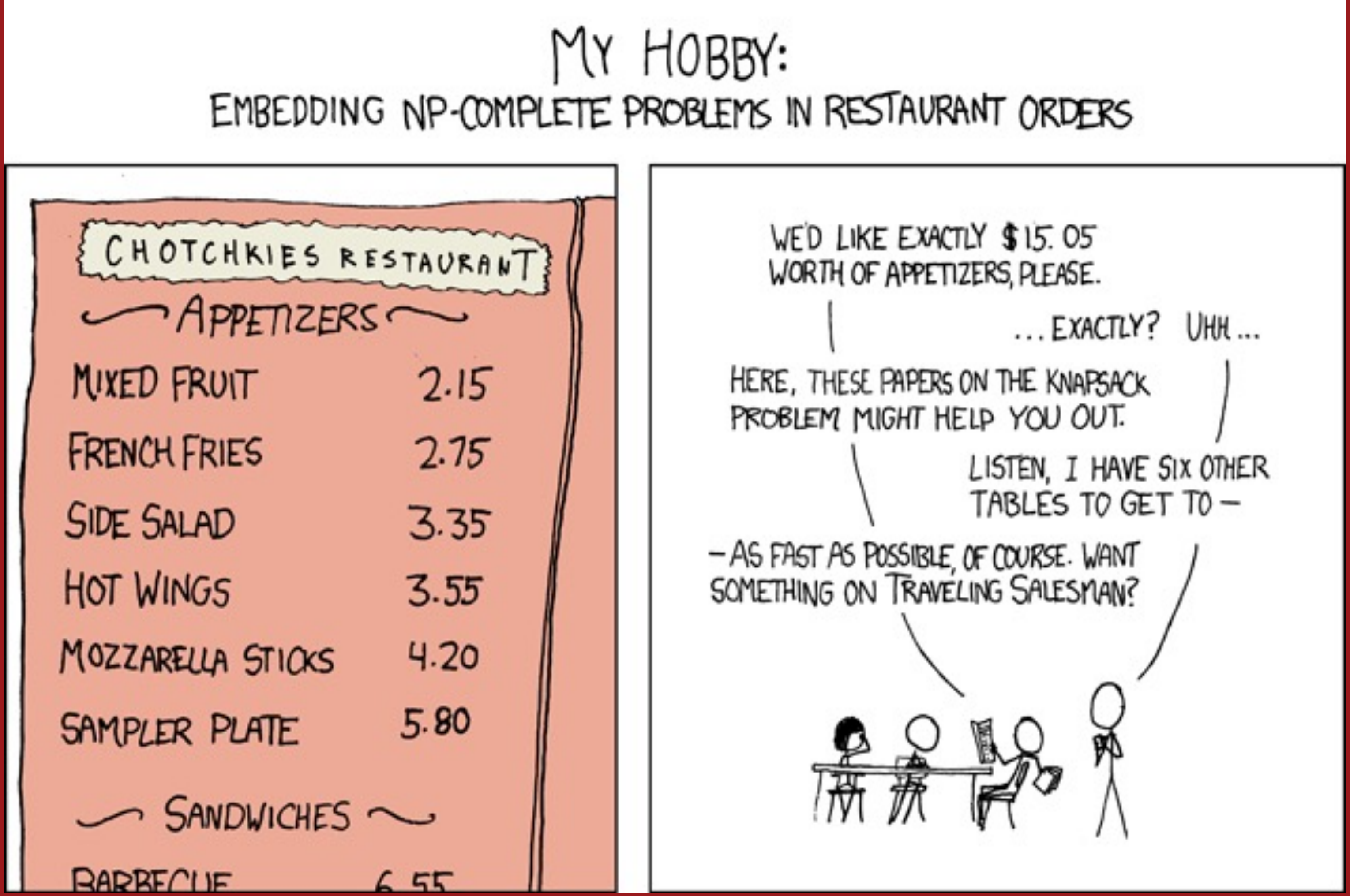
Z3 built-in theories



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

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

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



https://xkcd.com/287/

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 }
  abort
{ 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

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)
```

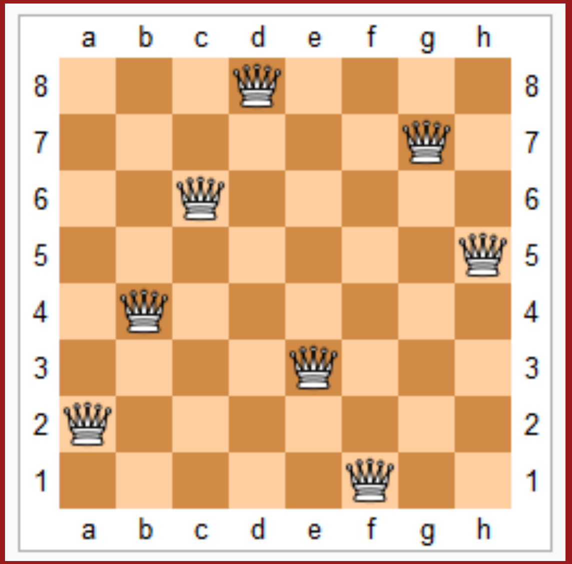
```
$ z3 …
unknown
```

Exercise

- 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:
  - It is ok to give a solution using just Z3 for a fixed instance but 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. X2 = 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]
```

Wrap-Up

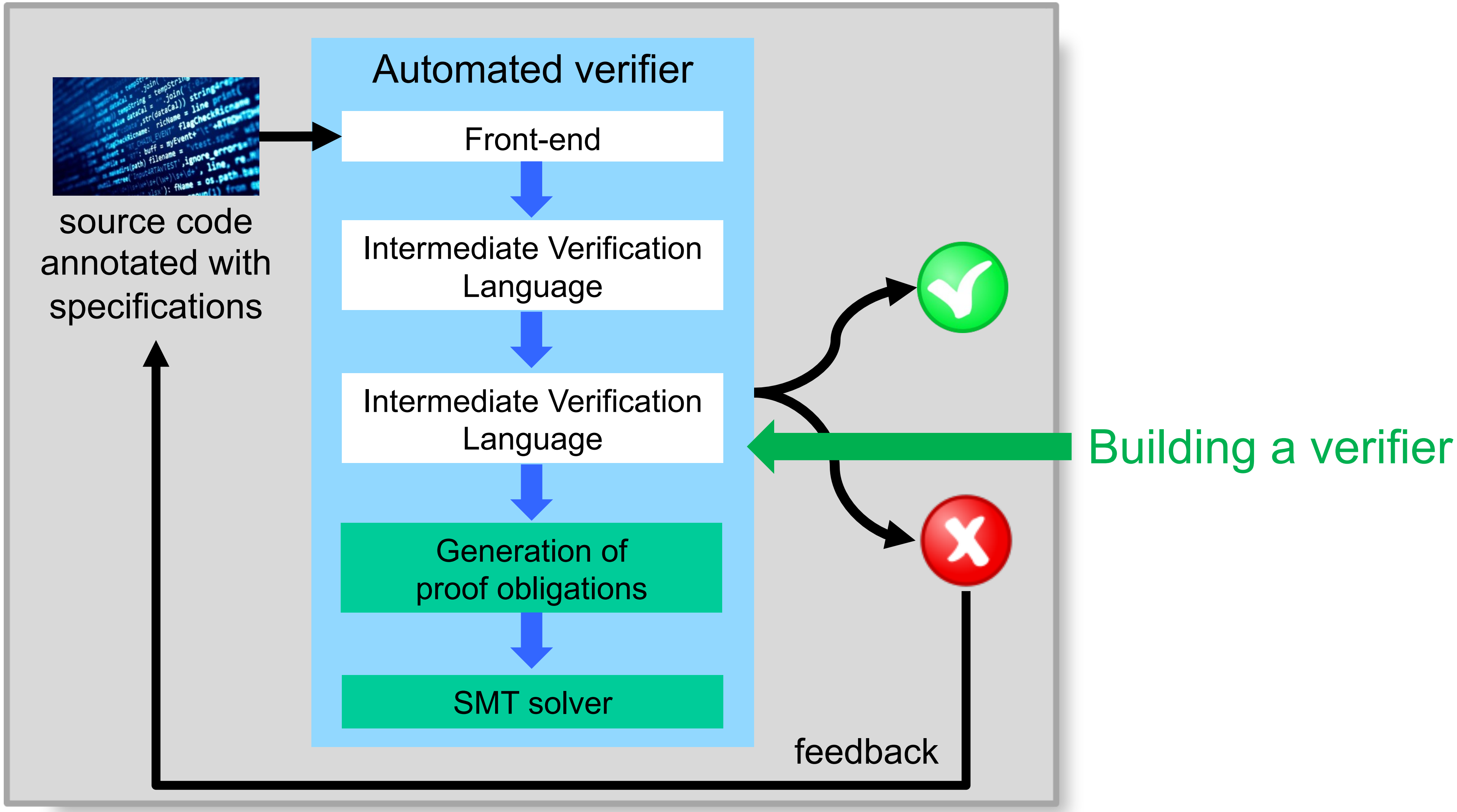
Mains steps of a tool for checking that `{ P } S { Q }` is valid:

1. Compute `wp(S, Q)` → last lecture

2. Check whether entailment `P ==> WP(S, Q)` is valid → ask SMT solver

- Check satisfiability of negation: `P && !WP(S, Q)`
- `unsat` → `{ P } S { Q }` is valid
- `sat` → model explains why `{ P } S { Q }` is not valid
- `unknown` → decidability issues, strengthen theory, hacks → future lectures

What next?



source code annotated with specifications

Automated verifier

Front-end

Intermediate Verification Language

Intermediate Verification Language

Generation of proof obligations

SMT solver

feedback

Building a verifier