Synthesis

Reading Reflection

Discuss in groups

- How would the different synthesis approaches described in the reading affect the user interaction model?
- How would the approaches described in the reading apply or not apply to the various synthesis project ideas you brainstormed last Tuesday?
- Please also take a couple minutes to discuss what you learned from the Rosette assignment!

Reading Key Takeaways

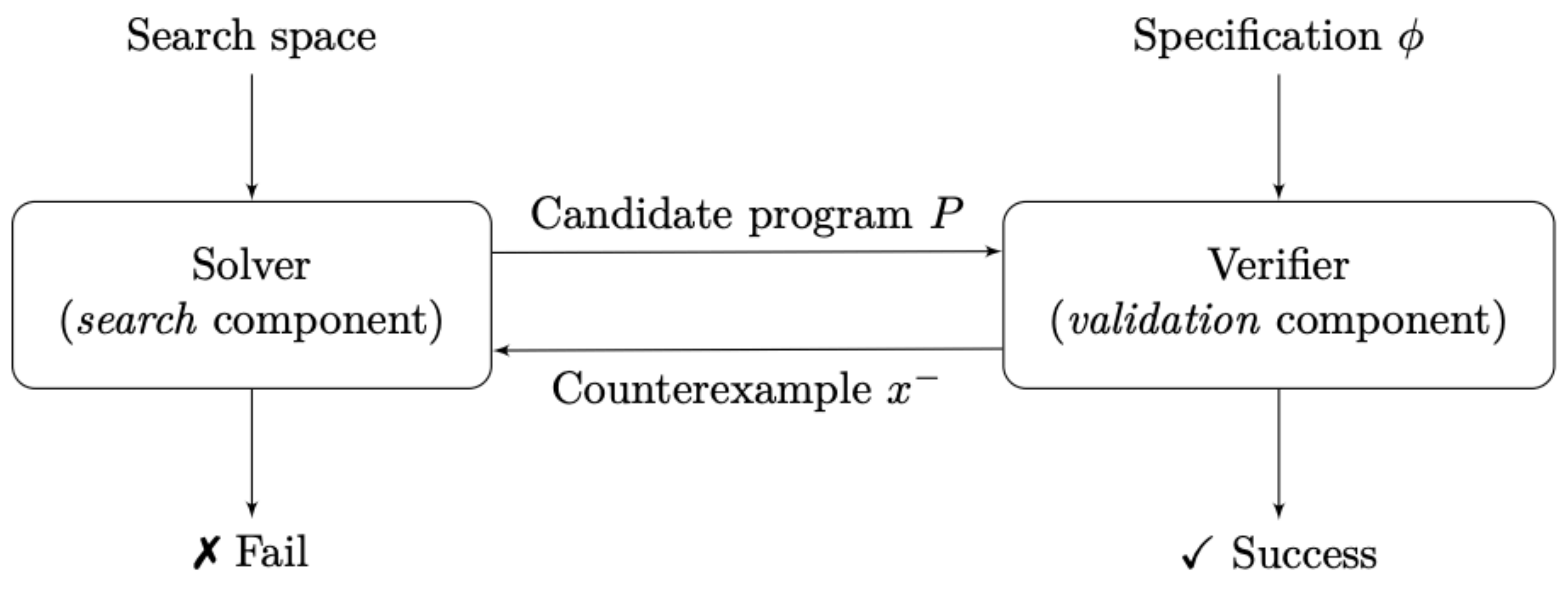


Figure 3.2: Counterexample-guided inductive synthesis.

- CEGIS!

- Distinguishing inputs—2 programs match our spec. How will we find the one we want? Ask the user what we should do on this next input, for which the programs produce different outputs.
- Syntactic bias—as we've already discussed, language shapes the search space
- SyGuS—SyGuS solvers can be a really useful starting point for a new synthesis project! See Fig 3.10 for how nice the programs are.

SyGuS string example

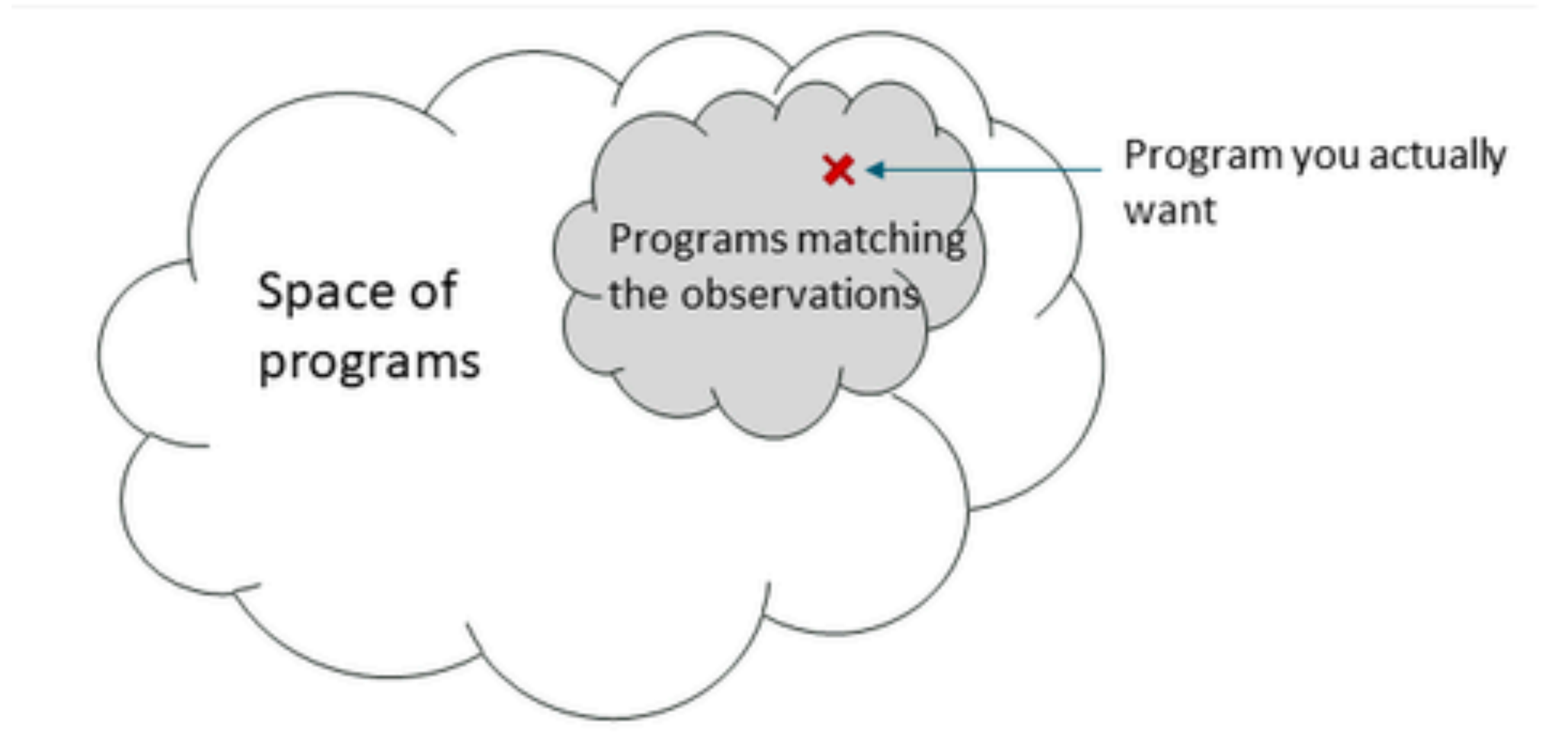
```
(set-logic SLIA)

(synth-fun f ((name String)) String
    ((Start String (ntString))
    (ntString String (name " " "." "Dr." (str.++ ntString ntString)
        (str.replace ntString ntString ntString) (str.at ntString ntInt) (int.to.str ntInt)
        (str.substr ntString ntInt ntInt)))
    (ntInt Int (0 1 2 (+ ntInt ntInt) (- ntInt ntInt) (str.len ntString)
        (str.to.int ntString) (str.indexof ntString ntString ntInt)))
    (ntBool Bool (true false (str.prefixof ntString ntString)
        (str.suffixof ntString ntString) (str.contains ntString ntString)))))

(declare-var name String)
(constraint (= (f "Nancy FreeHafer") "Dr. Nancy"))
(constraint (= (f "Andrew Cencici") "Dr. Andrew"))
(constraint (= (f "Jan Kotas") "Dr. Jan"))
(constraint (= (f "Mariya Sergienko") "Dr. Mariya"))

(check-synth)
```

What did you learn from the Rosette assignment?

A few learning goals

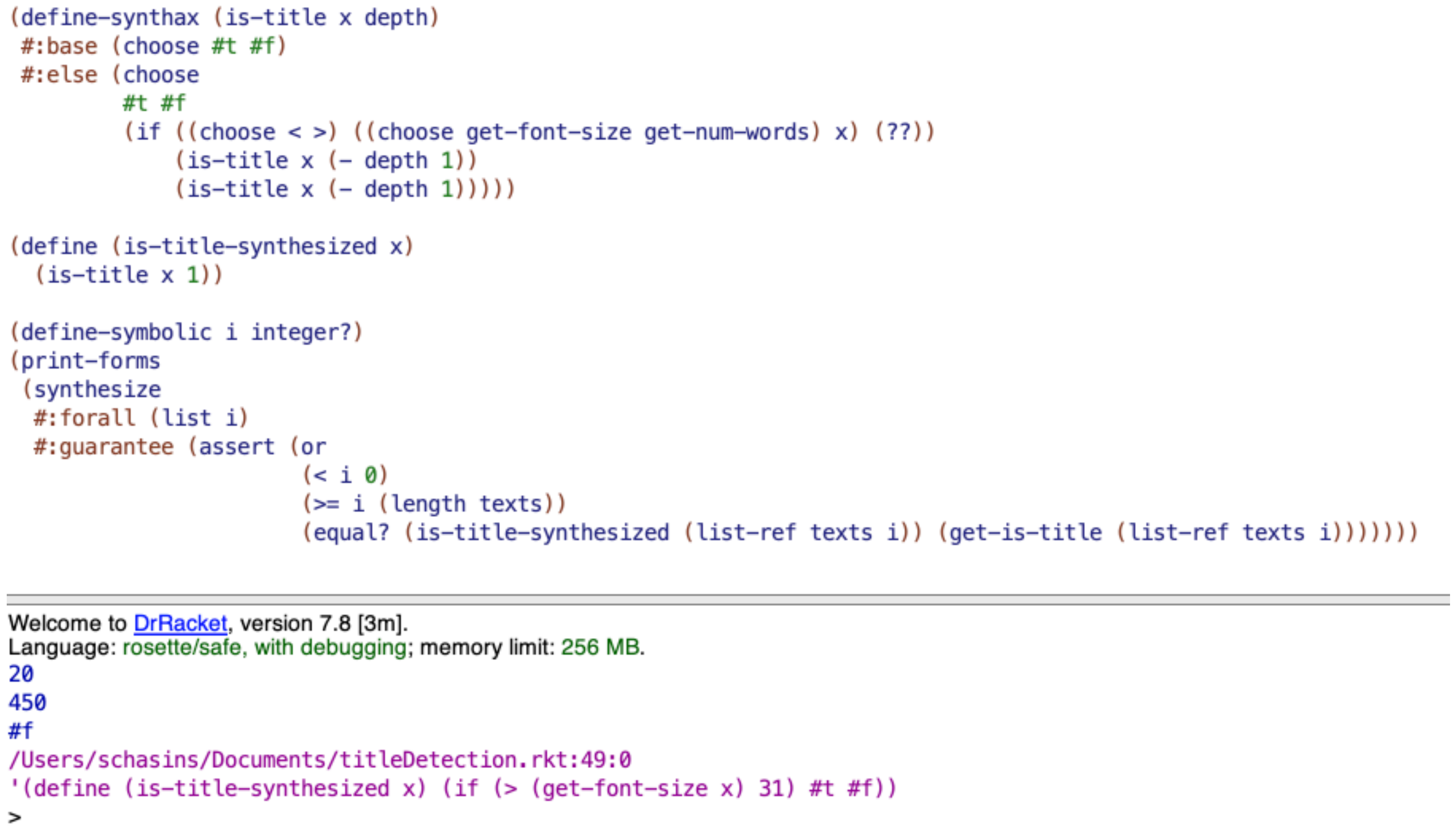
You might have learned…

- That you can write a synthesizer!
- That there are many possible ways of designing the grammar, many possible ways of designing the spec
- A visceral understanding of the difference between finding a program that meets your spec and the program you actually want. :) Especially in example-based specs.
- The limits of what you can control in Rosette.



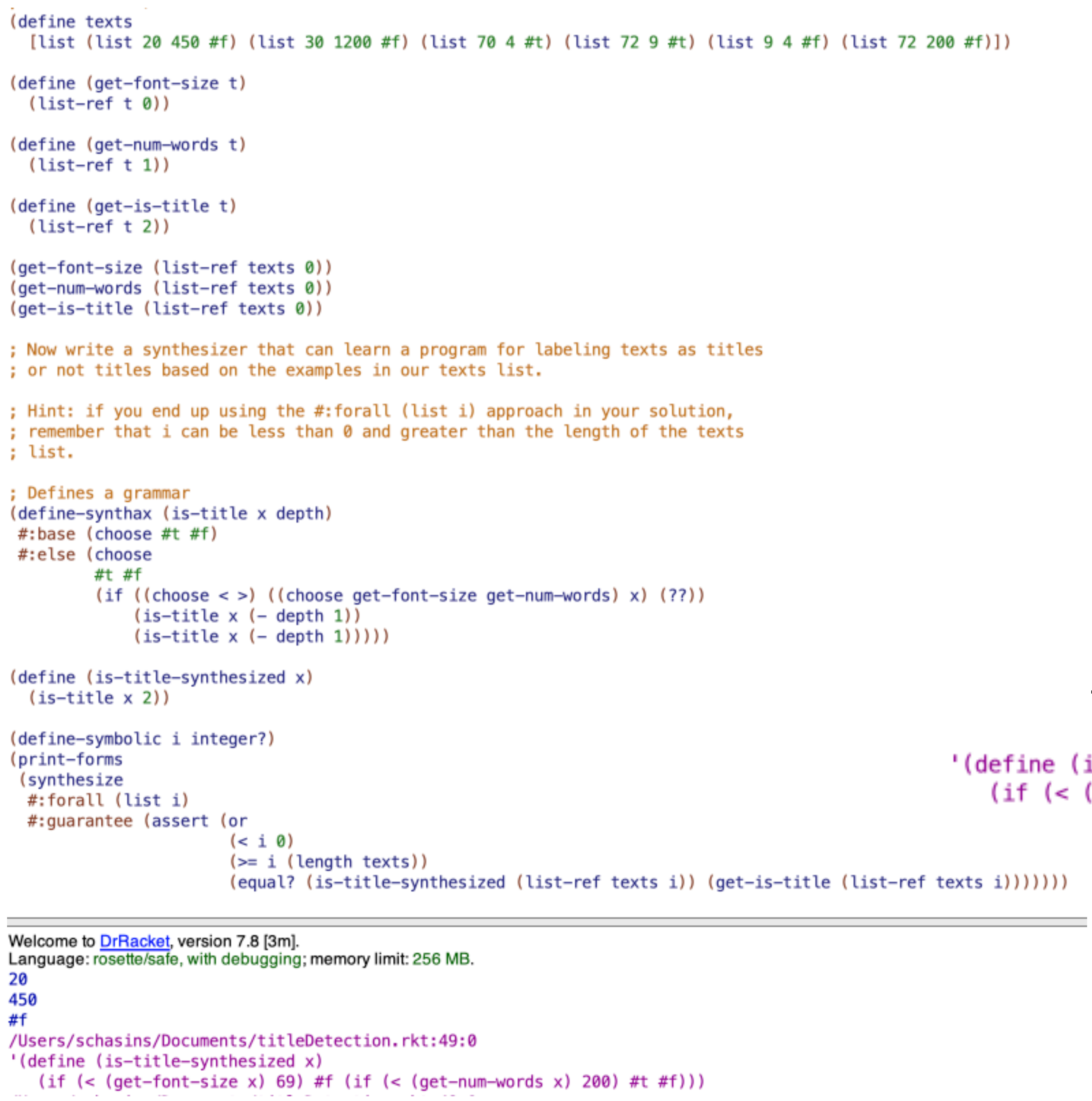
https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture2.htm
Armando Solar-Lezama

One (of many) solutions



```
(define-synthax (is-title x depth)
 #:base (choose #t #f)
 #:else (choose
         #t #f
         (if ((choose < >) ((choose get-font-size get-num-words) x) (??))
             (is-title x (- depth 1))
             (is-title x (- depth 1)))))

(define (is-title-synthesized x)
  (is-title x 1))

(define-symbolic i integer?)
(print-forms
 (synthesize
  #:forall (list i)
  #:guarantee (assert (or
                       (< i 0)
                       (>= i (length texts))
                       (equal? (is-title-synthesized (list-ref texts i)) (get-is-title (list-ref texts i)))))))
```

```
Welcome to DrRacket, version 7.8 [3m].
Language: rosette/safe, with debugging; memory limit: 256 MB.
20
450
#f
/Users/schasins/Documents/titleDetection.rkt:49:0
'(define (is-title-synthesized x) (if (> (get-font-size x) 31) #t #f))
>
```



```
(define texts
  [list (list 20 450 #f) (list 30 1200 #f) (list 70 4 #t) (list 72 9 #t) (list 9 4 #f) (list 72 200 #f)])

(define (get-font-size t)
  (list-ref t 0))

(define (get-num-words t)
  (list-ref t 1))

(define (get-is-title t)
  (list-ref t 2))

(get-font-size (list-ref texts 0))
(get-num-words (list-ref texts 0))
(get-is-title (list-ref texts 0))

; Now write a synthesizer that can learn a program for labeling texts as titles
; or not titles based on the examples in our texts list.

; Hint: if you end up using the #:forall (list i) approach in your solution,
; remember that i can be less than 0 and greater than the length of the texts
; list.

; Defines a grammar
(define-synthax (is-title x depth)
 #:base (choose #t #f)
 #:else (choose
         #t #f
         (if ((choose < >) ((choose get-font-size get-num-words) x) (??))
             (is-title x (- depth 1))
             (is-title x (- depth 1)))))

(define (is-title-synthesized x)
  (is-title x 2))

(define-symbolic i integer?)
(print-forms
 (synthesize
  #:forall (list i)
  #:guarantee (assert (or
                         (< i 0)
                         (>= i (length texts))
                         (equal? (is-title-synthesized (list-ref texts i)) (get-is-title (list-ref texts i)))))))
```

```
Welcome to DrRacket, version 7.8 [3m].
Language: rosette/safe, with debugging; memory limit: 256 MB.
20
450
#f
/Users/schasins/Documents/titleDetection.rkt:49:0
'(define (is-title-synthesized x)
   (if (< (get-font-size x) 69) #f (if (< (get-num-words x) 200) #t #f)))
```

And this is adaptable as we get more complicated inputs from our user…

Original input-output pairs

```
(list 20 450 #f) (list 30 1200 #f) (list 70 4 #t)
```

Here we add 3 more

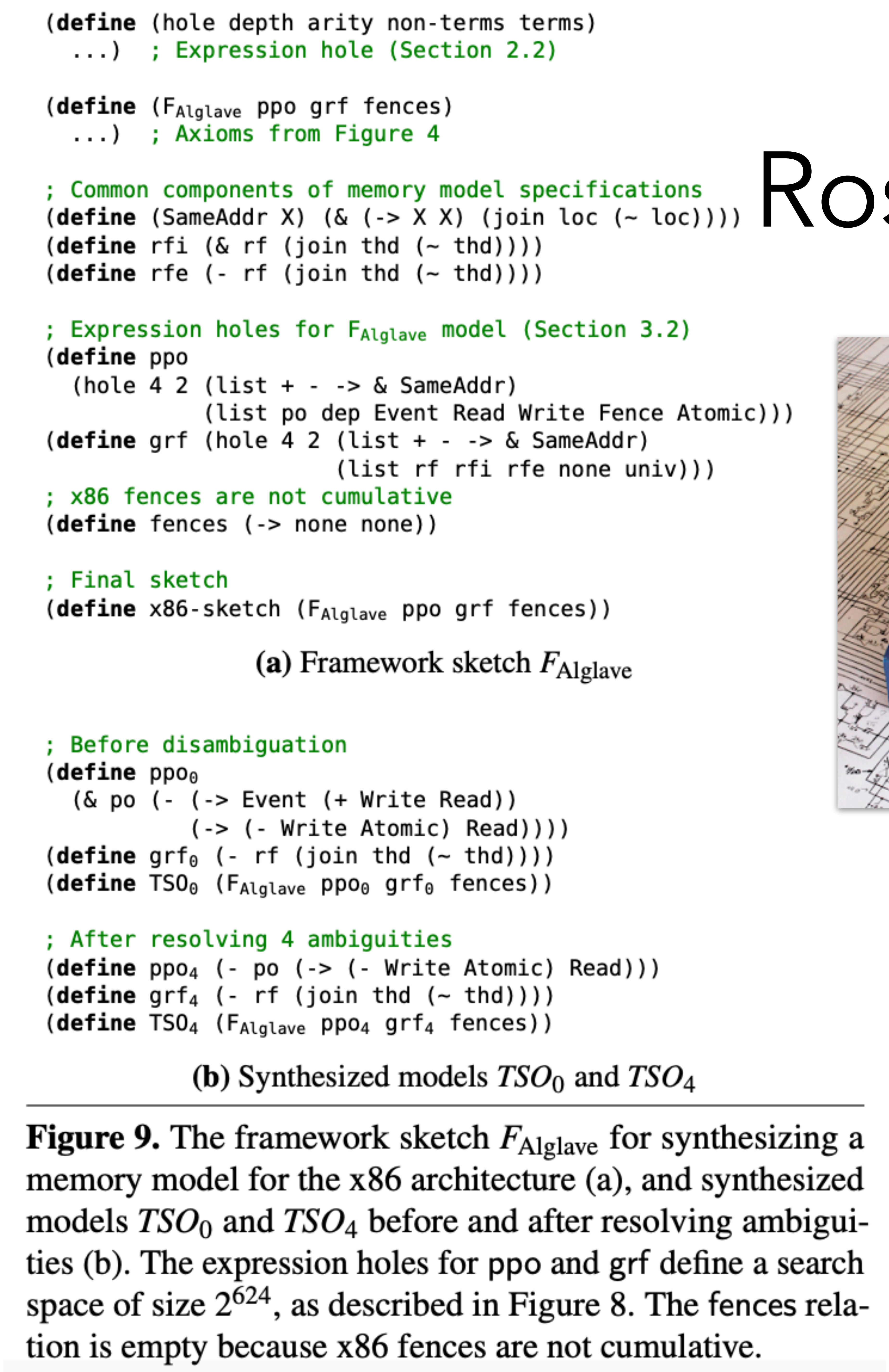
```
(list 72 9 #t) (list 9 4 #f) (list 72 200 #f)])
```

mavbe a footnote maybe a pull-out quote

The same synthesizer now produces:

```
'(define (is-title-synthesized x)
   (if (< (get-font-size x) 69) #f (if (< (get-num-words x) 200) #t #f)))
```

Rosette for more realistic tasks…



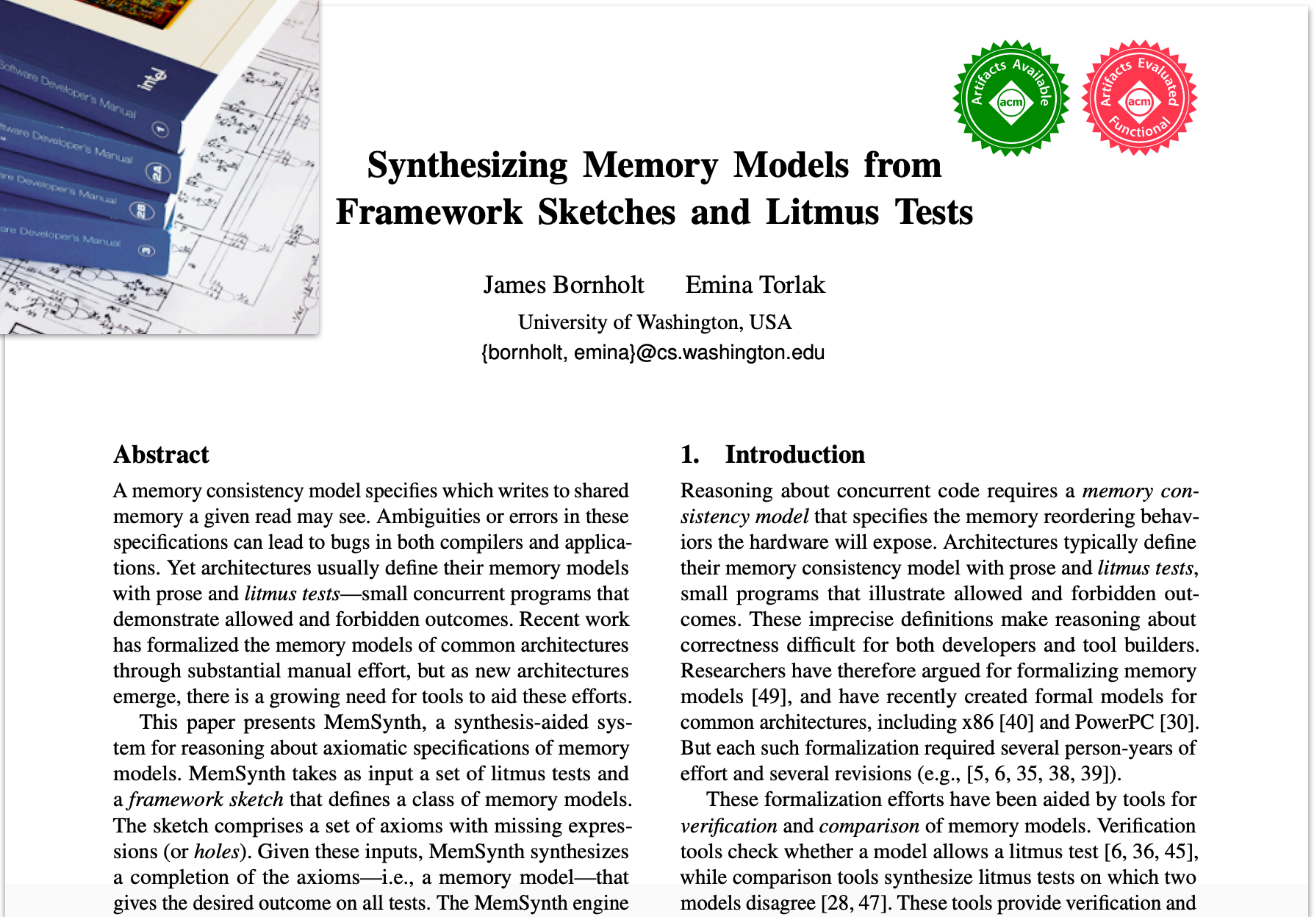
```
(define (hole depth arity non-terms terms)
  ...)  ; Expression hole (Section 2.2)

(define (FAlglave ppo grf fences)
  ...)  ; Axioms from Figure 4

; Common components of memory model specifications
(define (SameAddr X) (& (-> X X) (join loc (~ loc))))
(define rfi (& rf (join thd (~ thd))))
(define rfe (- rf (join thd (~ thd))))

; Expression holes for FAlglave model (Section 3.2)
(define ppo
  (hole 4 2 (list + - -> & SameAddr)
            (list po dep Event Read Write Fence Atomic)))
(define grf (hole 4 2 (list + - -> & SameAddr)
                      (list rf rfi rfe none univ)))
; x86 fences are not cumulative
(define fences (-> none none))

; Final sketch
(define x86-sketch (FAlglave ppo grf fences))
```

(a) Framework sketch F_{Alglave}

```
; Before disambiguation
(define ppo0
  (& po (- (-> Event (+ Write Read))
           (-> (- Write Atomic) Read))))
(define grf0 (- rf (join thd (~ thd))))
(define TSO0 (FAlglave ppo0 grf0 fences))

; After resolving 4 ambiguities
(define ppo4 (- po (-> (- Write Atomic) Read)))
(define grf4 (- rf (join thd (~ thd))))
(define TSO4 (FAlglave ppo4 grf4 fences))
```

(b) Synthesized models TSO_0 and TSO_4

Figure 9. The framework sketch F_{Alglave} for synthesizing a memory model for the x86 architecture (a), and synthesized models TSO_0 and TSO_4 before and after resolving ambiguities (b). The expression holes for ppo and grf define a search space of size 2^{624}, as described in Figure 8. The fences relation is empty because x86 fences are not cumulative.



Synthesizing Memory Models from Framework Sketches and Litmus Tests

James Bornholt Emina Torlak
University of Washington, USA
{bornholt, emina}@cs.washington.edu

Abstract

A memory consistency model specifies which writes to shared memory a given read may see. Ambiguities or errors in these specifications can lead to bugs in both compilers and applications. Yet architectures usually define their memory models with prose and *litmus tests*—small concurrent programs that demonstrate allowed and forbidden outcomes. Recent work has formalized the memory models of common architectures through substantial manual effort, but as new architectures emerge, there is a growing need for tools to aid these efforts.

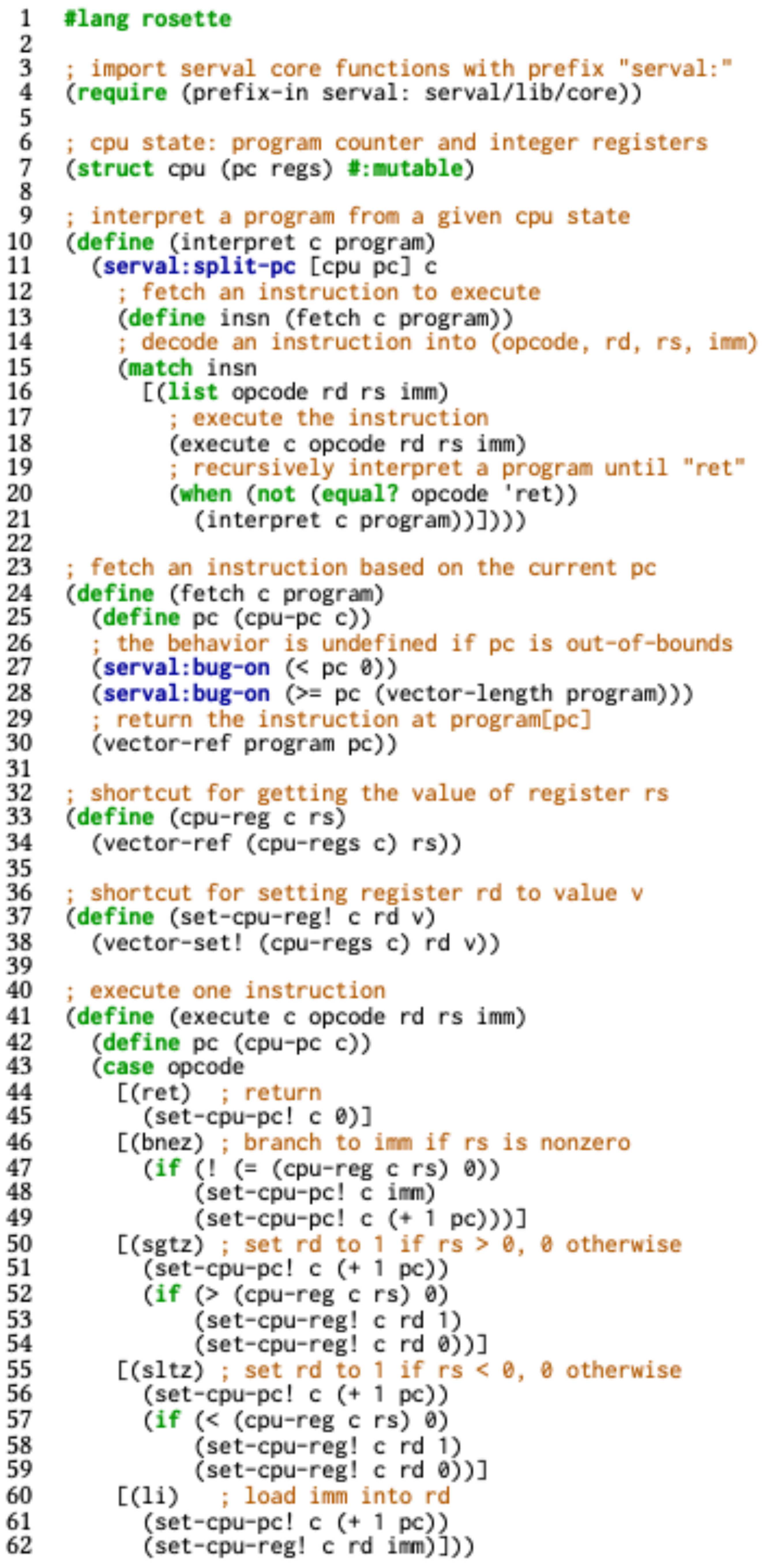
This paper presents MemSynth, a synthesis-aided system for reasoning about axiomatic specifications of memory models. MemSynth takes as input a set of litmus tests and a *framework sketch* that defines a class of memory models. The sketch comprises a set of axioms with missing expressions (or *holes*). Given these inputs, MemSynth synthesizes a completion of the axioms—i.e., a memory model—that gives the desired outcome on all tests. The MemSynth engine

1. Introduction

Reasoning about concurrent code requires a *memory consistency model* that specifies the memory reordering behaviors the hardware will expose. Architectures typically define their memory consistency model with prose and *litmus tests*, small programs that illustrate allowed and forbidden outcomes. These imprecise definitions make reasoning about correctness difficult for both developers and tool builders. Researchers have therefore argued for formalizing memory models [49], and have recently created formal models for common architectures, including x86 [40] and PowerPC [30]. But each such formalization required several person-years of effort and several revisions (e.g., [5, 6, 35, 38, 39]).

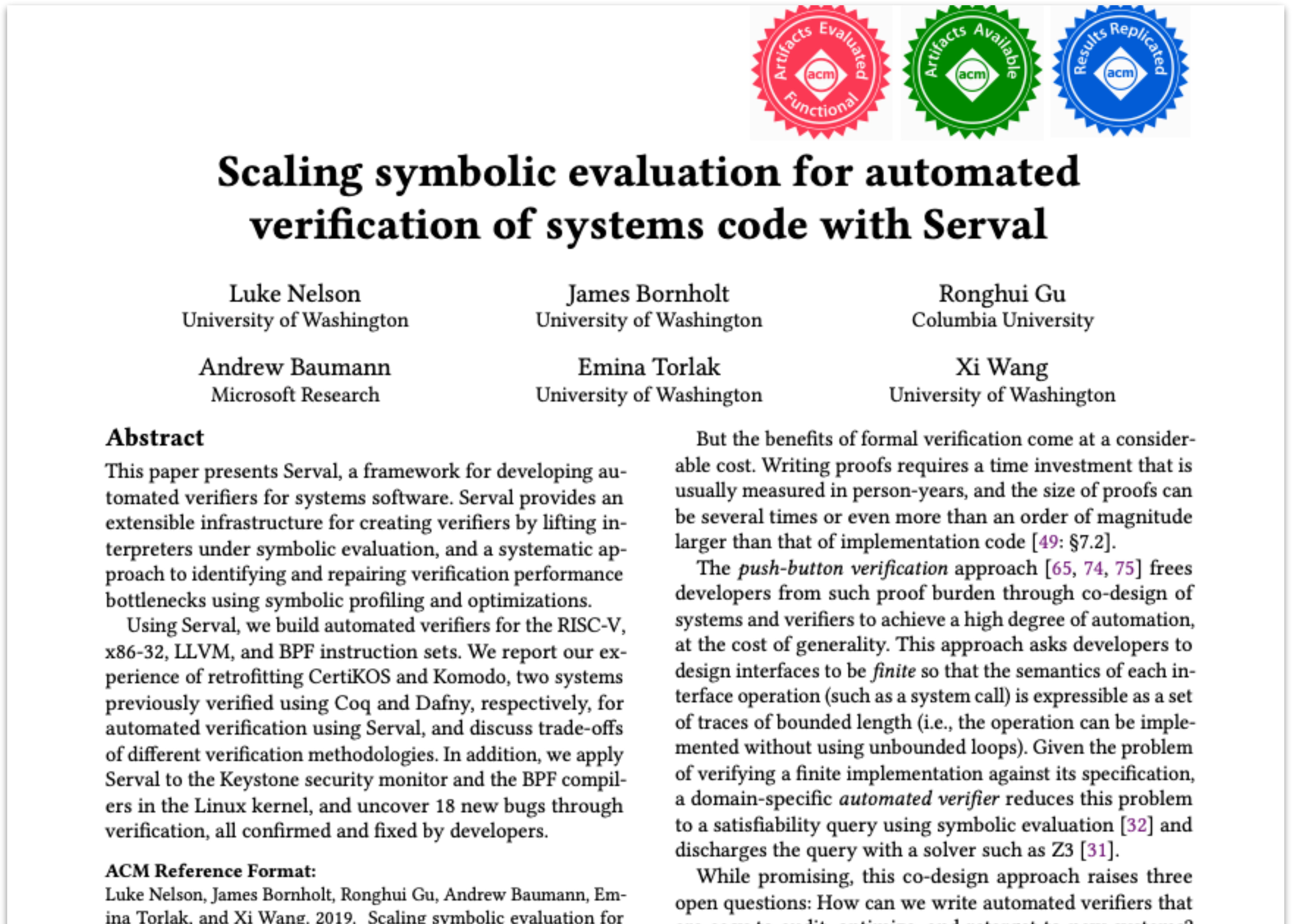
These formalization efforts have been aided by tools for *verification* and *comparison* of memory models. Verification tools check whether a model allows a litmus test [6, 36, 45], while comparison tools synthesize litmus tests on which two models disagree [28, 47]. These tools provide verification and

Rosette for more realistic tasks…



```
#lang rosette

; import serval core functions with prefix "serval:"
(require (prefix-in serval: serval/lib/core))

; cpu state: program counter and integer registers
(struct cpu (pc regs) #:mutable)

; interpret a program from a given cpu state
(define (interpret c program)
  (serval:split-pc [cpu pc] c
    ; fetch an instruction to execute
    (define insn (fetch c program))
    ; decode an instruction into (opcode, rd, rs, imm)
    (match insn
      [(list opcode rd rs imm)
        ; execute the instruction
        (execute c opcode rd rs imm)
        ; recursively interpret a program until "ret"
        (when (not (equal? opcode 'ret))
          (interpret c program))])))

; fetch an instruction based on the current pc
(define (fetch c program)
  (define pc (cpu-pc c))
  ; the behavior is undefined if pc is out-of-bounds
  (serval:bug-on (< pc 0))
  (serval:bug-on (>= pc (vector-length program)))
  ; return the instruction at program[pc]
  (vector-ref program pc))

; shortcut for getting the value of register rs
(define (cpu-reg c rs)
  (vector-ref (cpu-regs c) rs))

; shortcut for setting register rd to value v
(define (set-cpu-reg! c rd v)
  (vector-set! (cpu-regs c) rd v))

; execute one instruction
(define (execute c opcode rd rs imm)
  (define pc (cpu-pc c))
  (case opcode
    [(ret)  ; return
      (set-cpu-pc! c 0)]
    [(bnez) ; branch to imm if rs is nonzero
      (if (! (= (cpu-reg c rs) 0))
          (set-cpu-pc! c imm)
          (set-cpu-pc! c (+ 1 pc)))]
    [(sgtz) ; set rd to 1 if rs > 0, 0 otherwise
      (set-cpu-pc! c (+ 1 pc))
      (if (> (cpu-reg c rs) 0)
          (set-cpu-reg! c rd 1)
          (set-cpu-reg! c rd 0))]
    [(sltz) ; set rd to 1 if rs < 0, 0 otherwise
      (set-cpu-pc! c (+ 1 pc))
      (if (< (cpu-reg c rs) 0)
          (set-cpu-reg! c rd 1)
          (set-cpu-reg! c rd 0))]
    [(li)   ; load imm into rd
      (set-cpu-pc! c (+ 1 pc))
      (set-cpu-reg! c rd imm)]))
```

Figure 4. A ToyRISC interpreter using Serval (in Rosette).



Scaling symbolic evaluation for automated verification of systems code with Serval

Luke Nelson, University of Washington
James Bornholt, University of Washington
Ronghui Gu, Columbia University
Andrew Baumann, Microsoft Research
Emina Torlak, University of Washington
Xi Wang, University of Washington

Abstract

This paper presents Serval, a framework for developing automated verifiers for systems software. Serval provides an extensible infrastructure for creating verifiers by lifting interpreters under symbolic evaluation, and a systematic approach to identifying and repairing verification performance bottlenecks using symbolic profiling and optimizations.

Using Serval, we build automated verifiers for the RISC-V, x86-32, LLVM, and BPF instruction sets. We report our experience of retrofitting CertiKOS and Komodo, two systems previously verified using Coq and Dafny, respectively, for automated verification using Serval, and discuss trade-offs of different verification methodologies. In addition, we apply Serval to the Keystone security monitor and the BPF compilers in the Linux kernel, and uncover 18 new bugs through verification, all confirmed and fixed by developers.

ACM Reference Format:
Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for

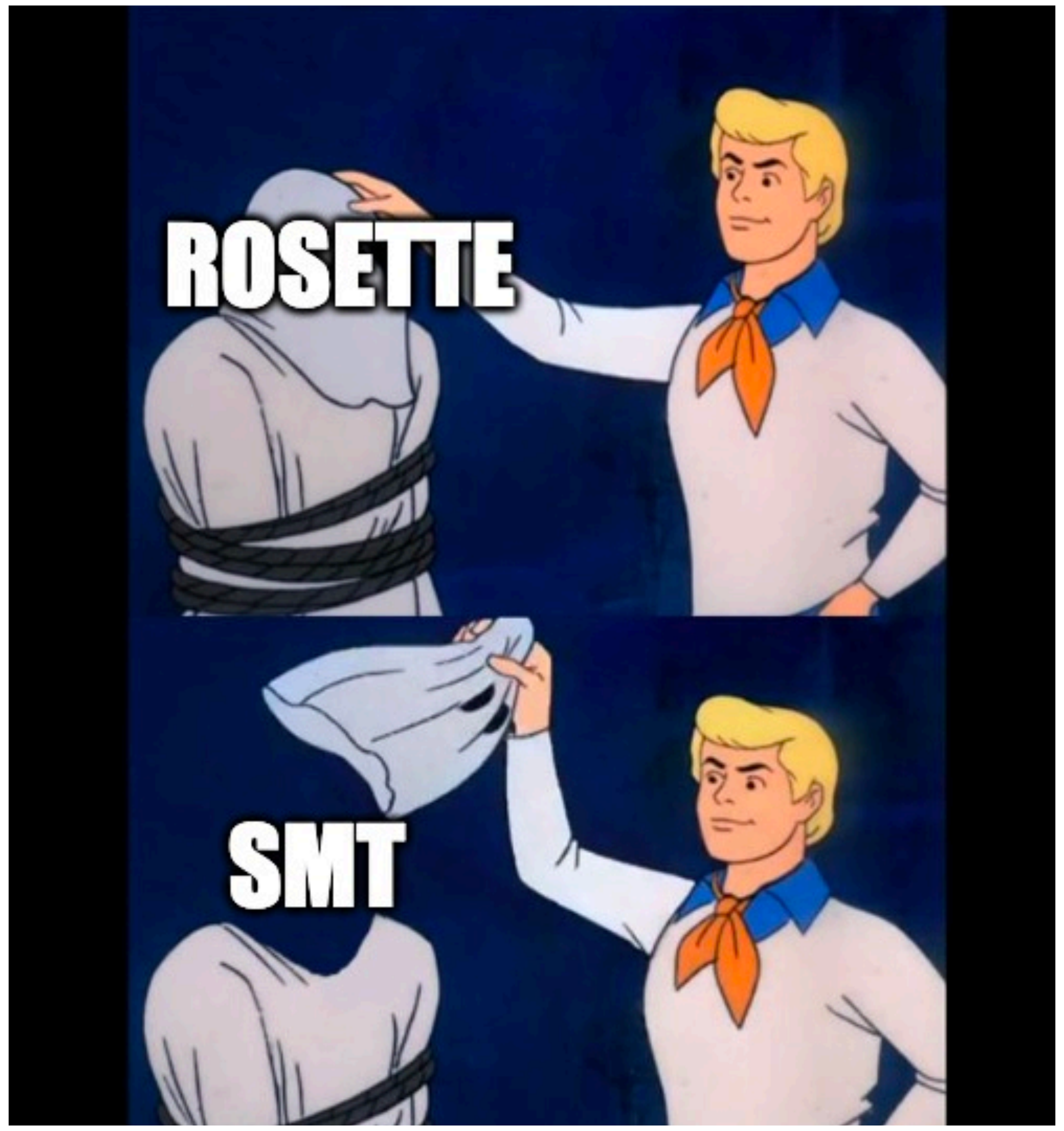
But the benefits of formal verification come at a considerable cost. Writing proofs requires a time investment that is usually measured in person-years, and the size of proofs can be several times or even more than an order of magnitude larger than that of implementation code [49: §7.2].

The *push-button verification* approach [65, 74, 75] frees developers from such proof burden through co-design of systems and verifiers to achieve a high degree of automation, at the cost of generality. This approach asks developers to design interfaces to be *finite* so that the semantics of each interface operation (such as a system call) is expressible as a set of traces of bounded length (i.e., the operation can be implemented without using unbounded loops). Given the problem of verifying a finite implementation against its specification, a domain-specific *automated verifier* reduces this problem to a satisfiability query using symbolic evaluation [32] and discharges the query with a solver such as Z3 [31].

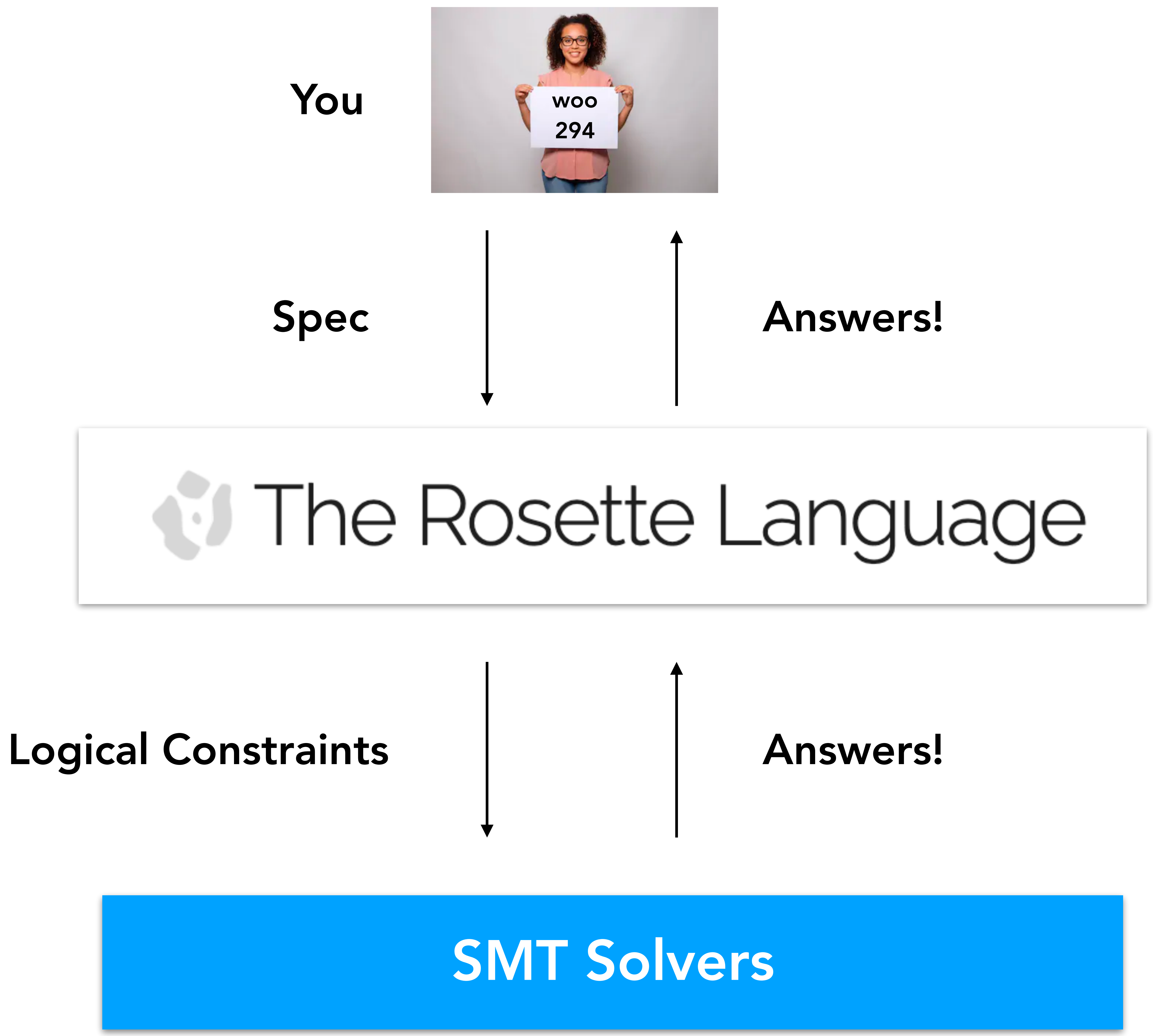
While promising, this co-design approach raises three open questions: How can we write automated verifiers that

Reflections on Rosette

- Concise program -> quite complex and sophisticated synthesizers
- Opacity
- Control


ROSETTE
SMT

Today's topic: SMT



OK, what's SMT?

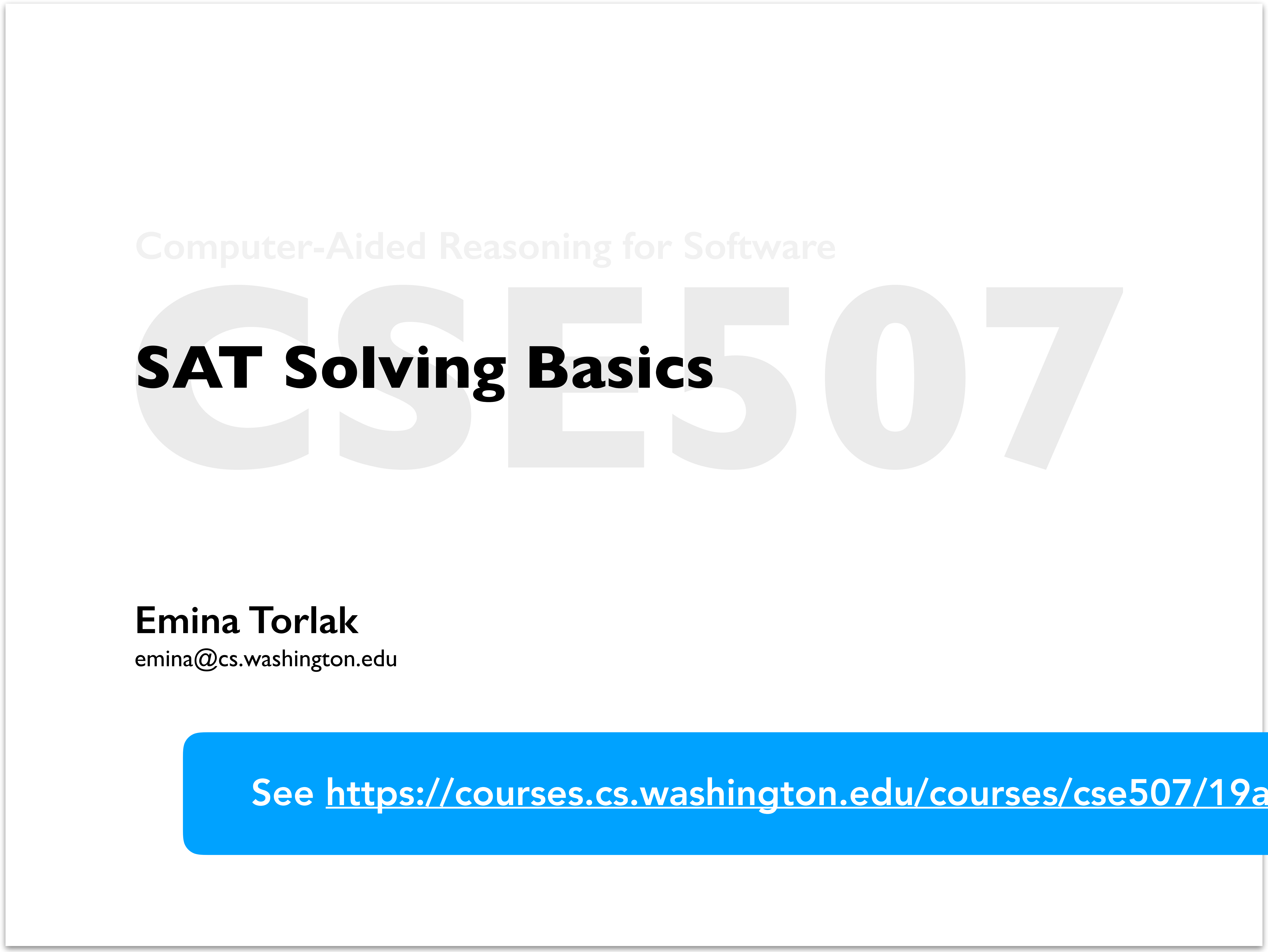
Satisfiability Modulo Theories

ok, and what's satisfiability??

Let’s back up

- **SAT: Boolean satisfiability problem**; also sometimes called SATISFIABILITY
  - Given a Boolean formula, is there an interpretation of the formula that satisfies it? **Can we replace the variables of the Boolean formula with the values TRUE or FALSE such that the formula evaluates to TRUE?**
    - If yes, the formula is satisfiable
    - If no assignment out of all possible assignments makes the formula TRUE, it’s unsatisfiable
- Examples:
  - $p \wedge q$ is satisfiable; (p=TRUE, q=TRUE)
  - $p \wedge \neg p$ is unsatisfiable
- SAT is NP-complete
  - …but that hasn’t stopped folks from building some seriously efficient SAT solvers and using them to solve real problems

Next few slides shamelessly lifted from…


Computer-Aided Reasoning for Software

CSE507

SAT Solving Basics

Emina Torlak
emina@cs.washington.edu

See https://courses.cs.washington.edu/courses/cse507/19au/calendar.html for more

Syntax of propositional logic

$$(\neg \boldsymbol{p} \wedge \top) \vee (\boldsymbol{q} \rightarrow \bot)$$

Syntax of propositional logic

$(\neg \boldsymbol{p} \wedge \top) \vee (\boldsymbol{q} \rightarrow \bot)$

Atom	**truth symbols**: $\top$ ("true"), $\bot$ ("false")
	propositional variables: $p, q, r, \ldots$

Syntax of propositional logic

$(\neg \boldsymbol{p} \land \top) \lor (\boldsymbol{q} \to \bot)$

Atom	**truth symbols**: $\top$ ("true"), $\bot$ ("false") **propositional variables**: $p, q, r, \ldots$
Literal	an atom α or its negation $\neg\alpha$

Syntax of propositional logic

$$(\neg \boldsymbol{p} \wedge \top) \vee (\boldsymbol{q} \rightarrow \bot)$$

Atom	**truth symbols**: $\top$ ("true"), $\bot$ ("false") **propositional variables**: $p, q, r, \ldots$
Literal	an atom α or its negation $\neg\alpha$
Formula	an atom or the application of a **logical connective** to formulas F_1, F_2:

$\neg F_1$	"not"	(negation)
$F_1 \wedge F_2$	"and"	(conjunction)
$F_1 \vee F_2$	"or"	(disjunction)
$F_1 \rightarrow F_2$	"implies"	(implication)
$F_1 \leftrightarrow F_2$	"if and only if"	(iff)

Semantics of propositional logic: interpretations

An **interpretation** *I* for a propositional formula *F* maps every variable in *F* to a truth value:

$$I : \{\, p \mapsto \text{true},\ q \mapsto \text{false},\ \ldots \}$$

Semantics of propositional logic: interpretations

An **interpretation** I for a propositional formula F maps every variable in F to a truth value:

$$I : \{ p \mapsto \text{true}, q \mapsto \text{false}, \ldots\}$$

I is a **satisfying interpretation** of F, written as $I \vDash F$, if F evaluates to true under I.

I is a **falsifying interpretation** of F, written as $I \nvDash F$, if F evaluates to false under I.

Semantics of propositional logic: interpretations

An **interpretation** *I* for a propositional formula *F* maps every variable in *F* to a truth value:

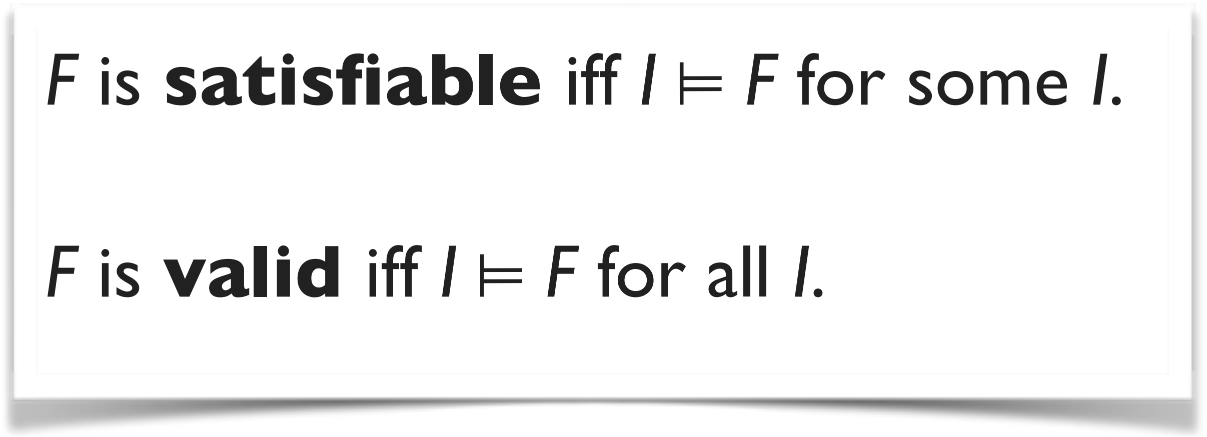
$$I : \{\, p \mapsto \text{true}, q \mapsto \text{false}, \ldots\}$$

I is a **satisfying interpretation** of *F*, written as $I \vDash F$, if *F* evaluates to true under *I*.

I is a **falsifying interpretation** of *F*, written as $I \nvDash F$, if *F* evaluates to false under *I*.

A satisfying interpretation is also called a **model**.

Satisfiability & validity of propositional formulas



F is **satisfiable** iff $I \models F$ for some *I*.

F is **valid** iff $I \models F$ for all *I*.

Satisfiability & validity of propositional formulas

F is **satisfiable** iff $I \models F$ for some *I*.

F is **valid** iff $I \models F$ for all *I*.

Duality of satisfiability and validity:

F is valid iff $\neg F$ is unsatisfiable.

Satisfiability & validity of propositional formulas

F is **satisfiable** iff $I \models F$ for some I.

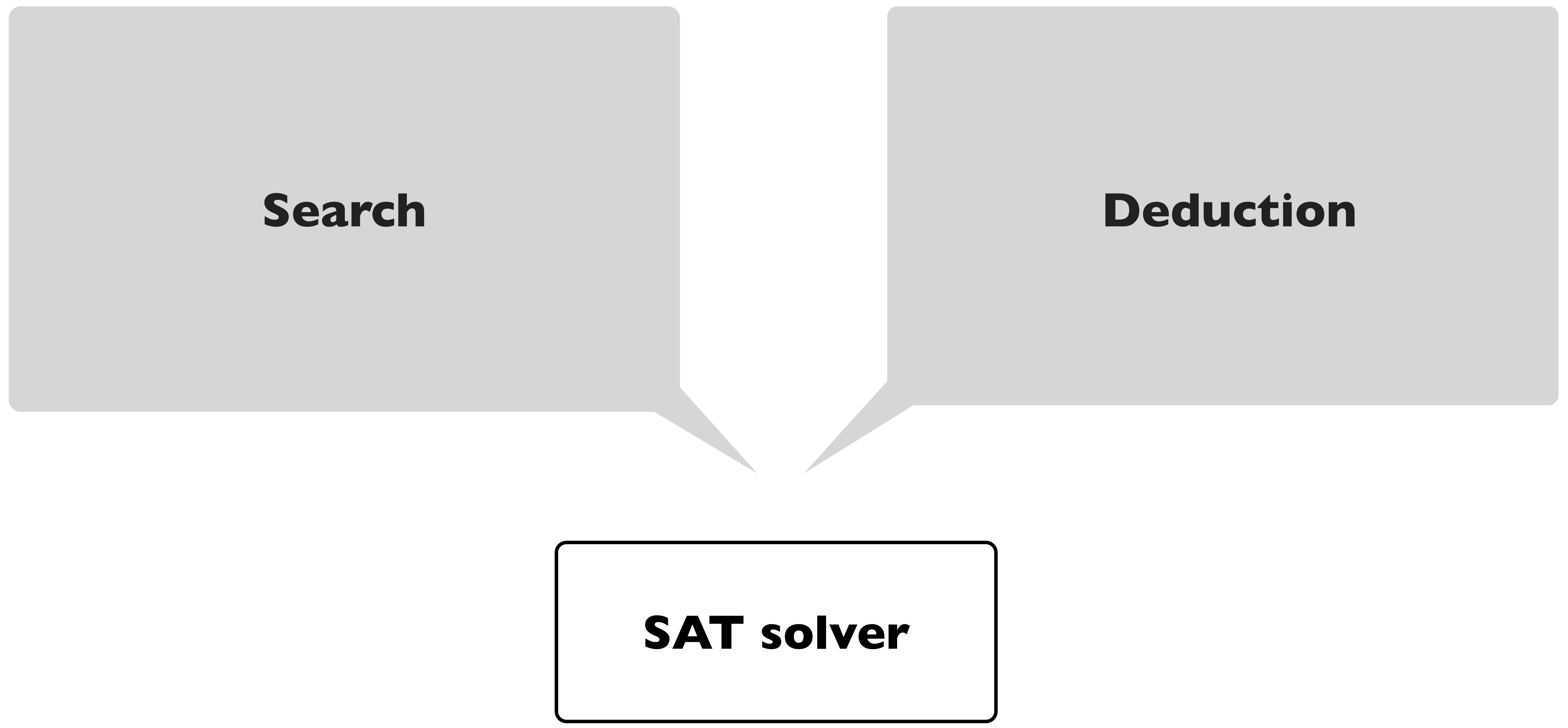
F is **valid** iff $I \models F$ for all I.

Duality of satisfiability and validity:

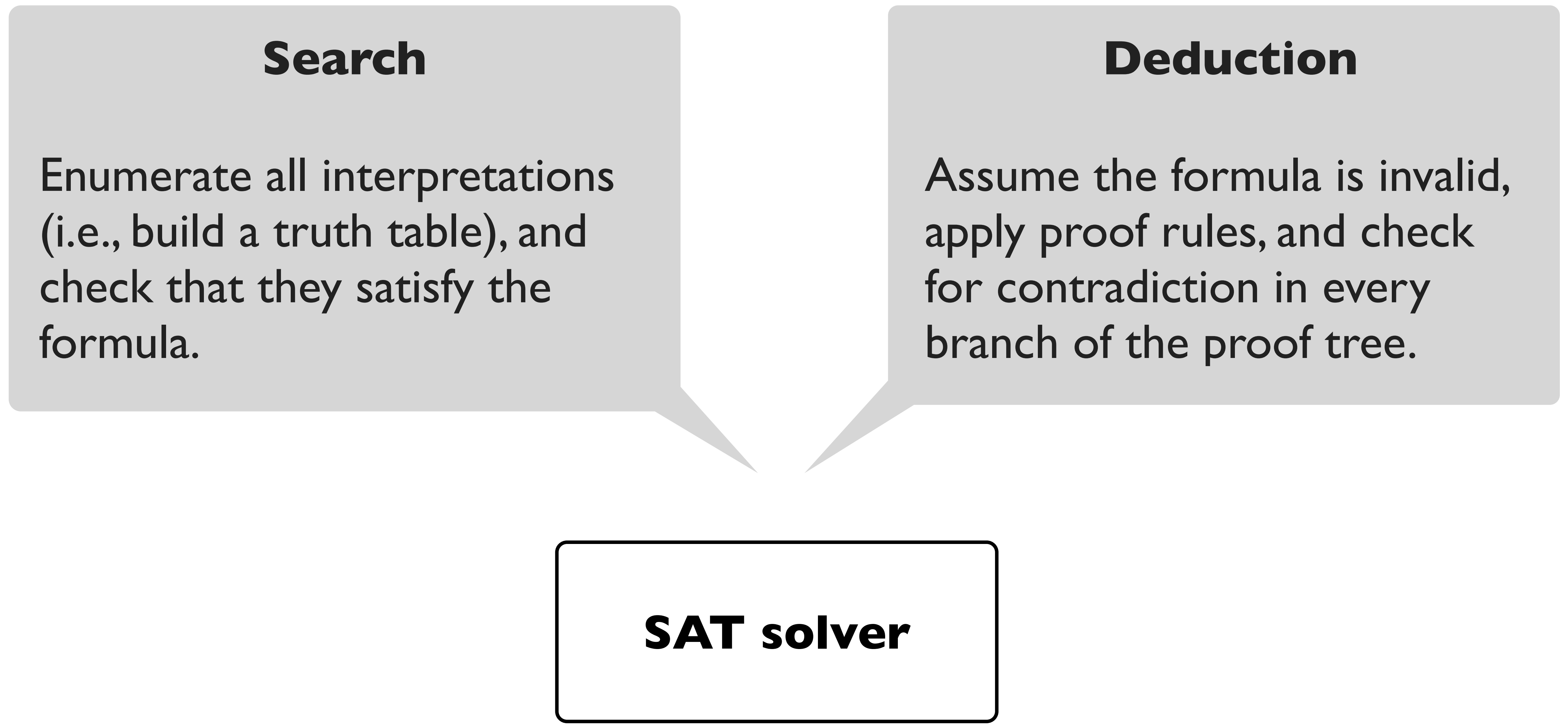
F is valid iff $\neg F$ is unsatisfiable.

If we have a procedure for checking satisfiability, we can also check validity of propositional formulas, and vice versa.

Techniques for deciding satisfiability & validity



Techniques for deciding satisfiability & validity



Search

Enumerate all interpretations (i.e., build a truth table), and check that they satisfy the formula.

Deduction

Assume the formula is invalid, apply proof rules, and check for contradiction in every branch of the proof tree.

Proof by search: enumerating interpretations

F: $(p \wedge q) \rightarrow (p \vee \neg q)$

p	q	$p \wedge q$	$\neg q$	$p \vee \neg q$	F
0	0	0	1	1	1
0	1	0	0	0	1
1	0	0	1	1	1
1	1	1	0	1	1

Valid.

Questions?

5 minute breakout chat

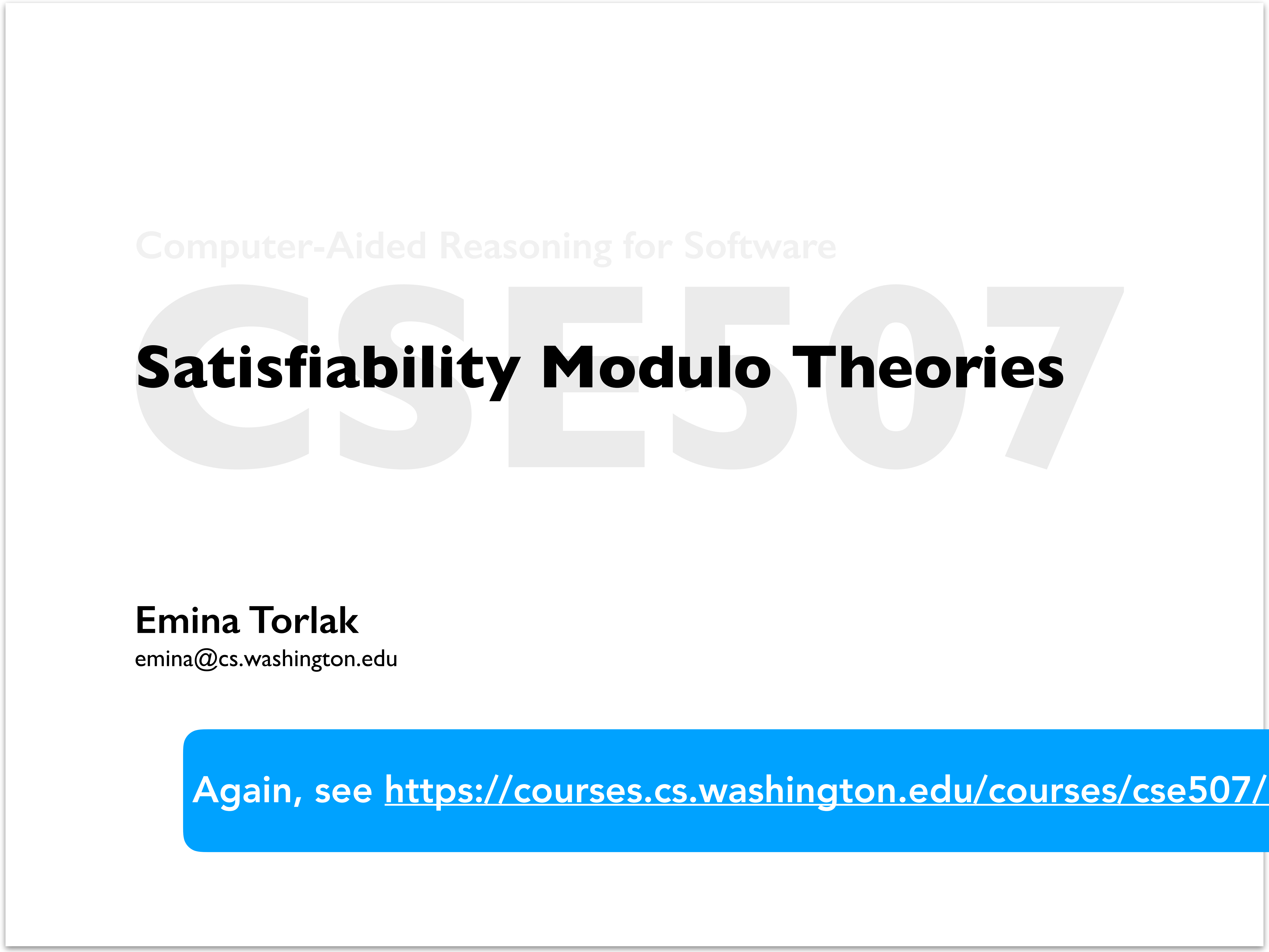
5 minute whole-group discussion

10 minute break

Now that we know about SAT…

- Ok seriously, what's SMT?
- Satisfiability (SAT) **Modulo Theories**

Next few slides shamelessly lifted from…



Computer-Aided Reasoning for Software

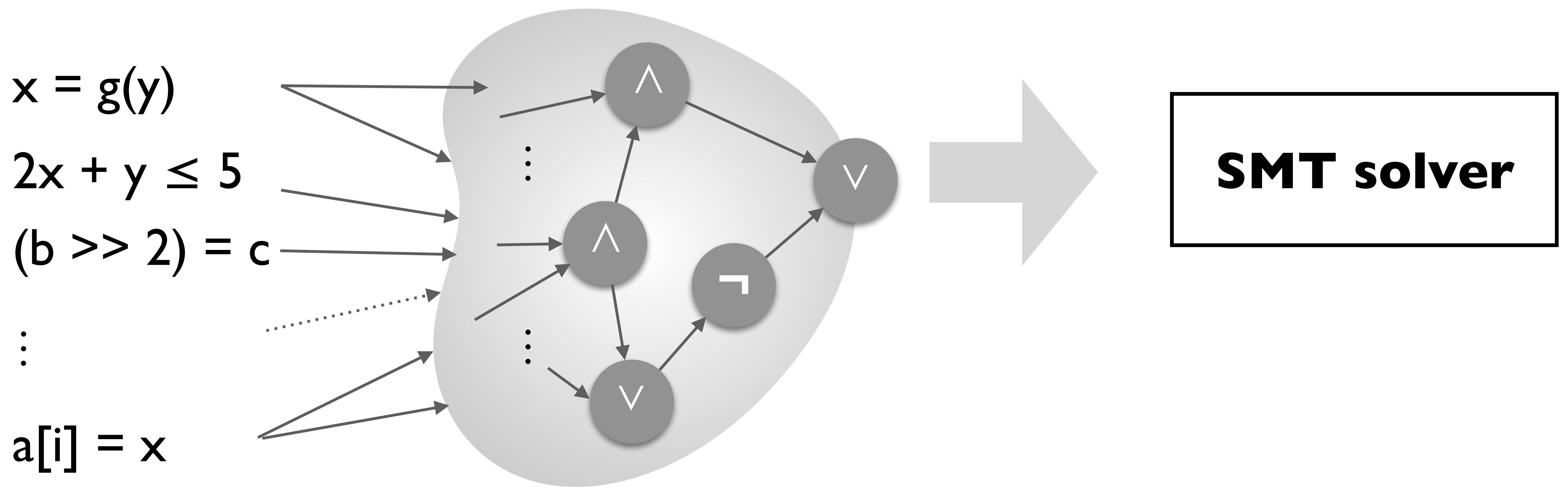
CSE507

Satisfiability Modulo Theories

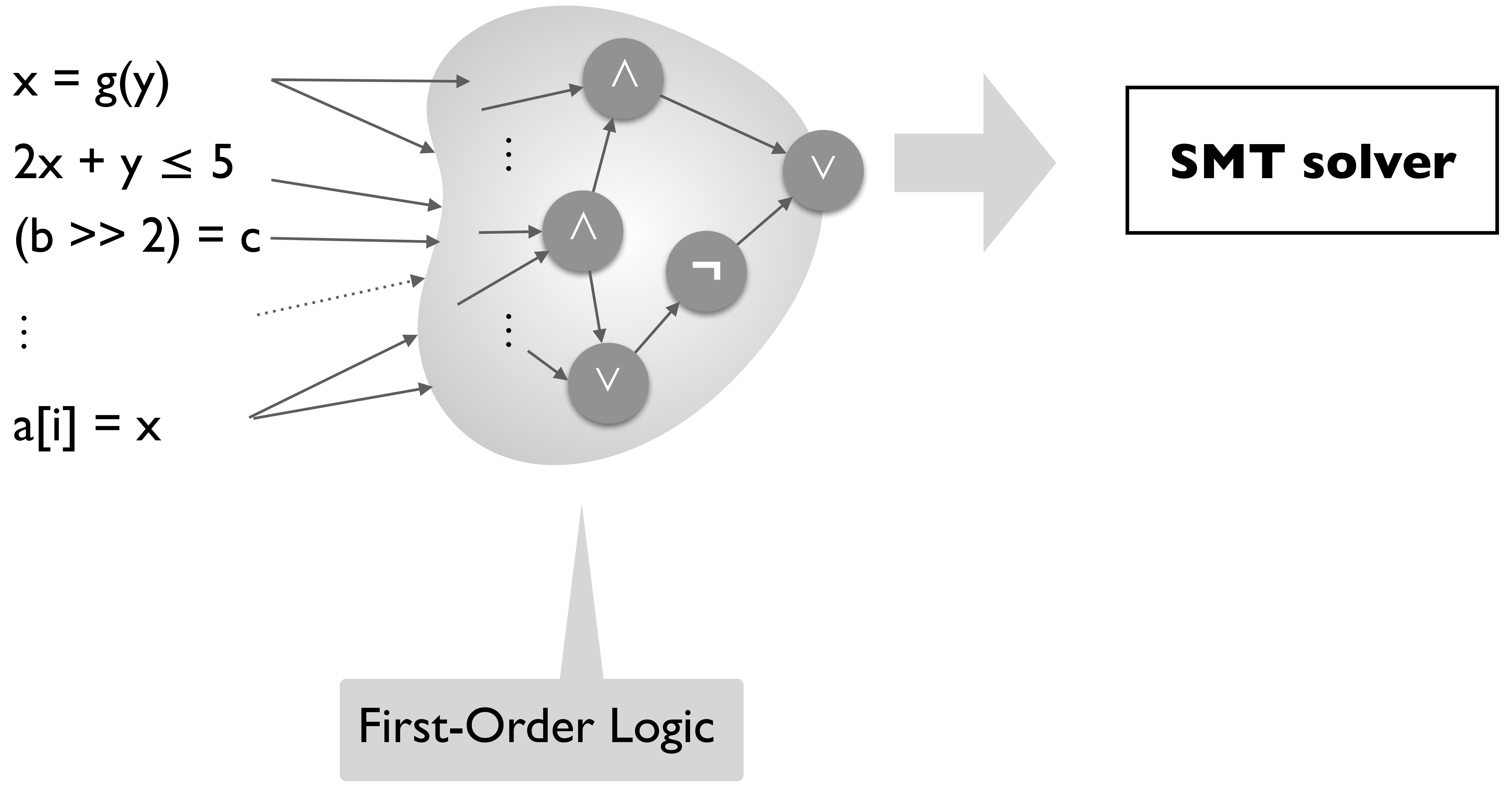
Emina Torlak
emina@cs.washington.edu

Again, see https://courses.cs.washington.edu/courses/cse507/19au/calendar.html for more

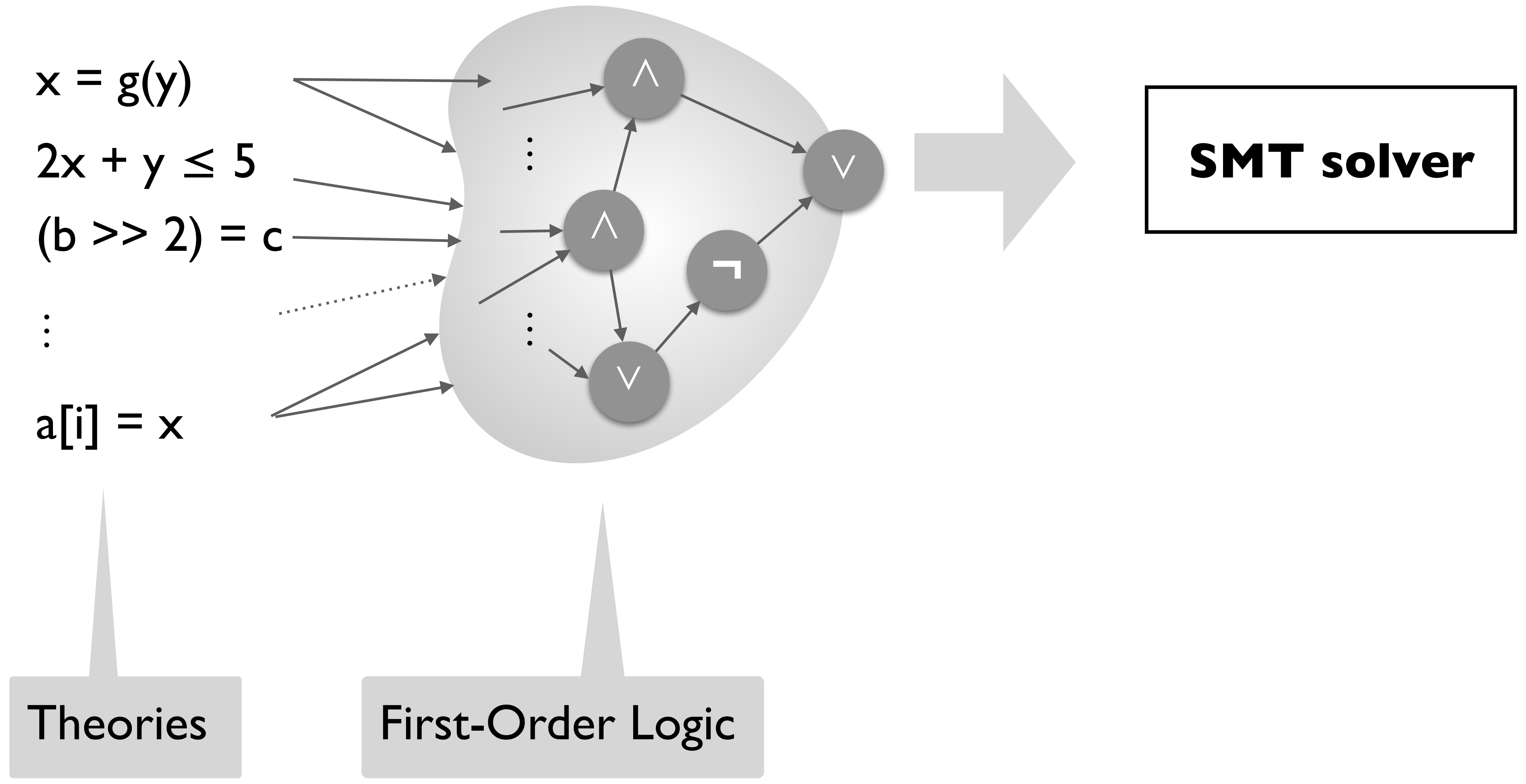
Satisfiability Modulo Theories (SMT)



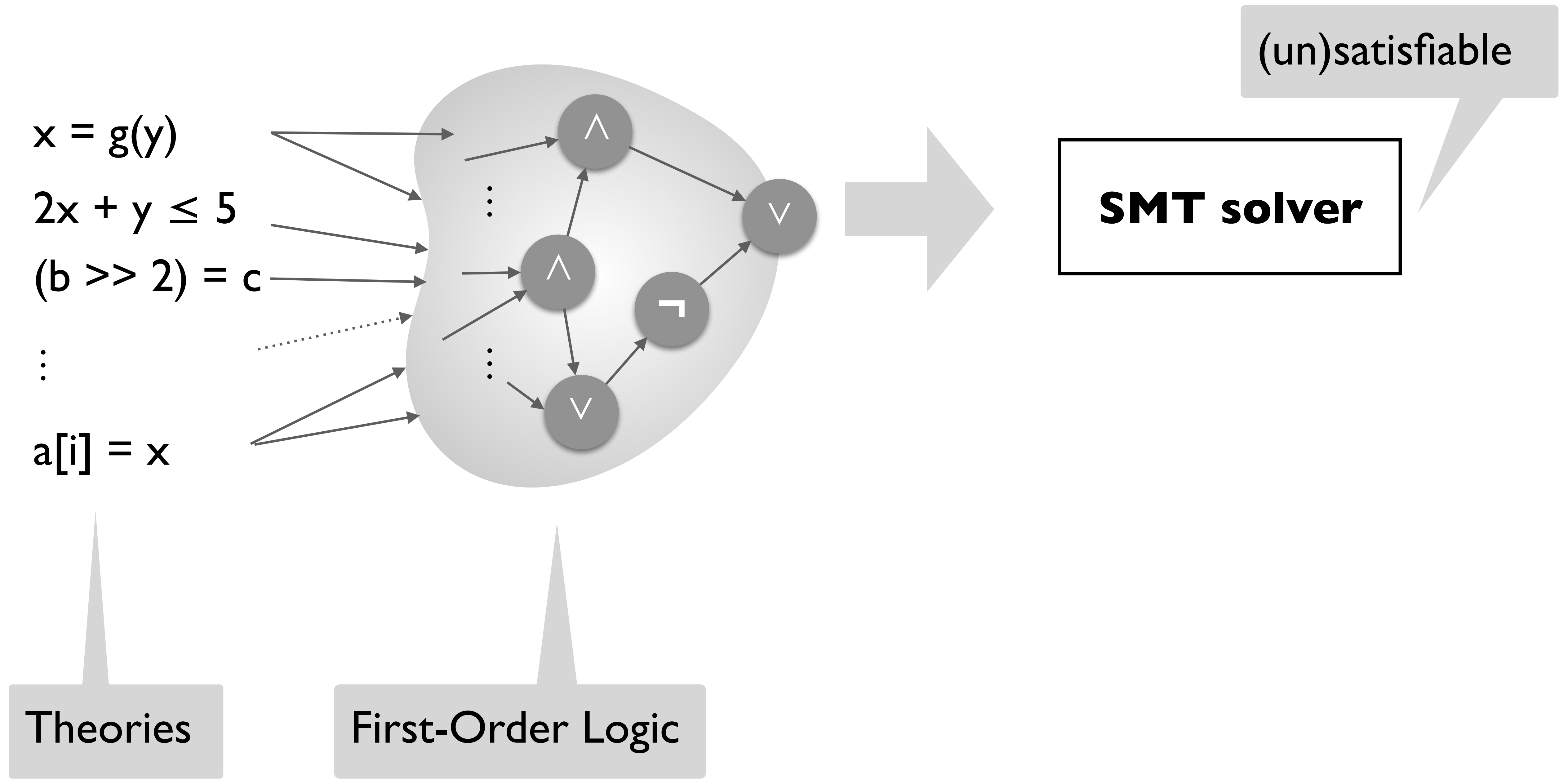
Satisfiability Modulo Theories (SMT)



Satisfiability Modulo Theories (SMT)



Satisfiability Modulo Theories (SMT)



Syntax of First-Order Logic (FOL)

Logical symbols

- Connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- Parentheses: ()
- ✗ Quantifiers: $\forall, \exists$

Non-logical symbols

- Constants: x, y, z
- N-ary functions: f, g
- N-ary predicates: p, q
- ✗ Variables: u, v, w

We will only consider the **quantifier-free** fragment of FOL.

In particular, we will consider quantifier-free **ground** formulas.

Semantics of FOL: example

Universe

- A non-empty set of values
- Finite or (un)countably infinite

Interpretation

- Maps a constant symbol c to an element of U: $I[c] \in U$
- Maps an n-ary function symbol f to a function $f_I : U^n \to U$
- Maps an n-ary predicate symbol p to an n-ary relation $p_I \subseteq U^n$

U = {☀, ☁}

I[x] = ☀

I[y] = ☁

I[f] = {☀ ↦ ☁, ☁ ↦ ☀}

I[p] = {⟨☀,☀⟩, ⟨☀,☁⟩}

$\langle U, I \rangle \vDash p(f(y), f(f(x)))$?

You decide!
Take 1 min, write in Zoom chat.

Satisfiability and validity of FOL

F is **satisfiable** iff M ⊨ F for some structure M = ⟨U, I⟩.

F is **valid** iff M ⊨ F for all structures M = ⟨U, I⟩.

Duality of satisfiability and validity:

F is valid iff ¬*F* is unsatisfiable.

Common theories

Equality (and uninterpreted functions)

- $x = g(y)$

Fixed-width bitvectors

- $(b >> 1) = c$

Linear arithmetic (over R and Z)

- $2x + y \leq 5$

Arrays

- $a[i] = x$

Theory of equality with uninterpreted functions

Signature: {=, x, y, z, …, f, g, …, p, q, …}

- The binary predicate = is *interpreted*.
- All constant, function, and predicate symbols are *uninterpreted.*

Axioms

- $\forall x.\ x = x$
- $\forall x, y.\ x = y \rightarrow y = x$
- $\forall x, y, z.\ x = y \land y = z \rightarrow x = z$
- $\forall x_1, \ldots, x_n, y_1, \ldots, y_n.\ (x_1 = y_1 \land \ldots \land x_n = y_n) \rightarrow (f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$
- $\forall x_1, \ldots, x_n, y_1, \ldots, y_n.\ (x_1 = y_1 \land \ldots \land x_n = y_n) \rightarrow (p(x_1, \ldots, x_n) \leftrightarrow p(y_1, \ldots, y_n))$

Deciding $T_=$

- Conjunctions of literals modulo $T_=$ is decidable in polynomial time.

$T_=$ example: checking program equivalence

```
int abs(int y) {
  return y<0 ? -y : y;
}

int sq(int y) {
  return y*y;
}

int sqabs(int y) {
  return abs(y)*abs(y);
}
```

$T_=$ example: checking program equivalence

```
int abs(int y) {
  return y<0 ? -y : y;
}

int sq(int y) {
  return y*y;
}

int sqabs(int y) {
  return abs(y)*abs(y);
}
```

Are **sq** and **sqabs** equivalent on all 128-bit integers?

$T_=$ example: checking program equivalence

```
int abs(int y) {
  return y<0 ? -y : y;
}

int sq(int y) {
  return y*y;
}

int sqabs(int y) {
  return abs(y)*abs(y);
}
```

Are **sq** and **sqabs** equivalent on all 128-bit integers?

Yes, but the solver takes a while to return an answer because reasoning about multiplication is expensive.

$T_=$ example: checking program equivalence

```
int abs(int y) {
  return y<0 ? -y : y;
}

int sq(int y) {
  return y*y;
}

int sqabs(int y) {
  return abs(y)*abs(y);
}
```

Are **sq** and **sqabs** equivalent on all 128-bit integers?

Yes, but the solver takes a while to return an answer because reasoning about multiplication is expensive.

What happens if we replace the multiplication with an uninterpreted function?

Theory of fixed-width bitvectors

Signature

- Fixed-width words modeling machine ints, longs, …
- Arithmetic operations: bvadd, bvsub, bvmul, …
- Bitwise operations: bvand, bvor, bvnot, …
- Comparison predicates: bvlt, bvgt, …
- Equality: =
- Expanded with all constant symbols: x, y, z, …

Deciding T_{BV}

- NP-complete.

Theories of linear integer and real arithmetic

Signature

- Integers (or reals)
- Arithmetic operations: multiplication by an integer (or real) number, +, -.
- Predicates: =, $\leq$.
- Expanded with all constant symbols: x, y, z, …

Deciding T_{LIA} and T_{LRA}

- NP-complete for linear integer arithmetic (LIA).
- Polynomial time for linear real arithmetic (LRA).
- Polynomial time for difference logic (conjunctions of the form $x - y \leq c$, where c is an integer or real number).

LIA example: compiler optimization

```
for (i=1; i<=10; i++) {
  a[j+i] = a[j];
}
```

A LIA formula that is unsatisfiable iff this transformation is valid:

```
int v = a[j];
for (i=1; i<=10; i++) {
  a[j+i] = v;
}
```

LIA example: compiler optimization

```
for (i=1; i<=10; i++) {
  a[j+i] = a[j];
}
```

```
int v = a[j];
for (i=1; i<=10; i++) {
  a[j+i] = v;
}
```

A LIA formula that is unsatisfiable iff this transformation is valid:

$(i \geq 1) \land (i \leq 10) \land$

$(j + i = j)$

Theory of arrays

Signature

- Array operations: read, write
- Equality: =
- Expanded with all constant symbols: x, y, z, …

Axioms

- $\forall a, i, v.\ \mathrm{read}(\mathrm{write}(a, i, v), i) = v$
- $\forall a, i, j, v.\ \neg(i = j) \rightarrow (\mathrm{read}(\mathrm{write}(a, i, v), j) = \mathrm{read}(a, j))$
- $\forall a, b.\ (\forall i.\ \mathrm{read}(a, i) = \mathrm{read}(b, i)) \rightarrow a = b$

Deciding T_A

- Satisfiability problem: NP-complete.
- Used in many software verification tools to model memory.

Basically…

- SAT lets us say simple things
- SMT lets us say…other simple things. But more complicated than SAT!
  - And it's enough that we can get to some interesting tasks
  - On Thursday we'll start playing around with some interesting tasks!

Install before Tuesday's class: Z3 SMT solver

We'll use the Python Z3 bindings. First make sure you have Python installed. Then install the Z3 bindings. (https://pypi.org/project/z3-solver/)

```
pip install z3-solver
```

OR

```
pip install z3-solver --user
```

Then make sure you can run this program, which I'll also upload in Slack.

```
from z3 import *

x = Int('x')
y = Int('y')
solve(x > 1, y > 1, x * y + 3 == 7)
```

To think about for next reading

- Like last reading, no need to memorize details—mostly want you to know these techniques exist and why we care about them.
- Our silly Python synthesizer from last week wasn't very scalable, but there are ways to make enumerative search scale!
- Hierarchical search is the fancy way of saying you can split the problem into multiple subproblems which you can solve separately—this is a key idea for many important synthesis tasks, and you can apply it yourself in many domains. This can improve scalability dramatically. Read the example extra carefully.