Synthesis

Reading Reflection

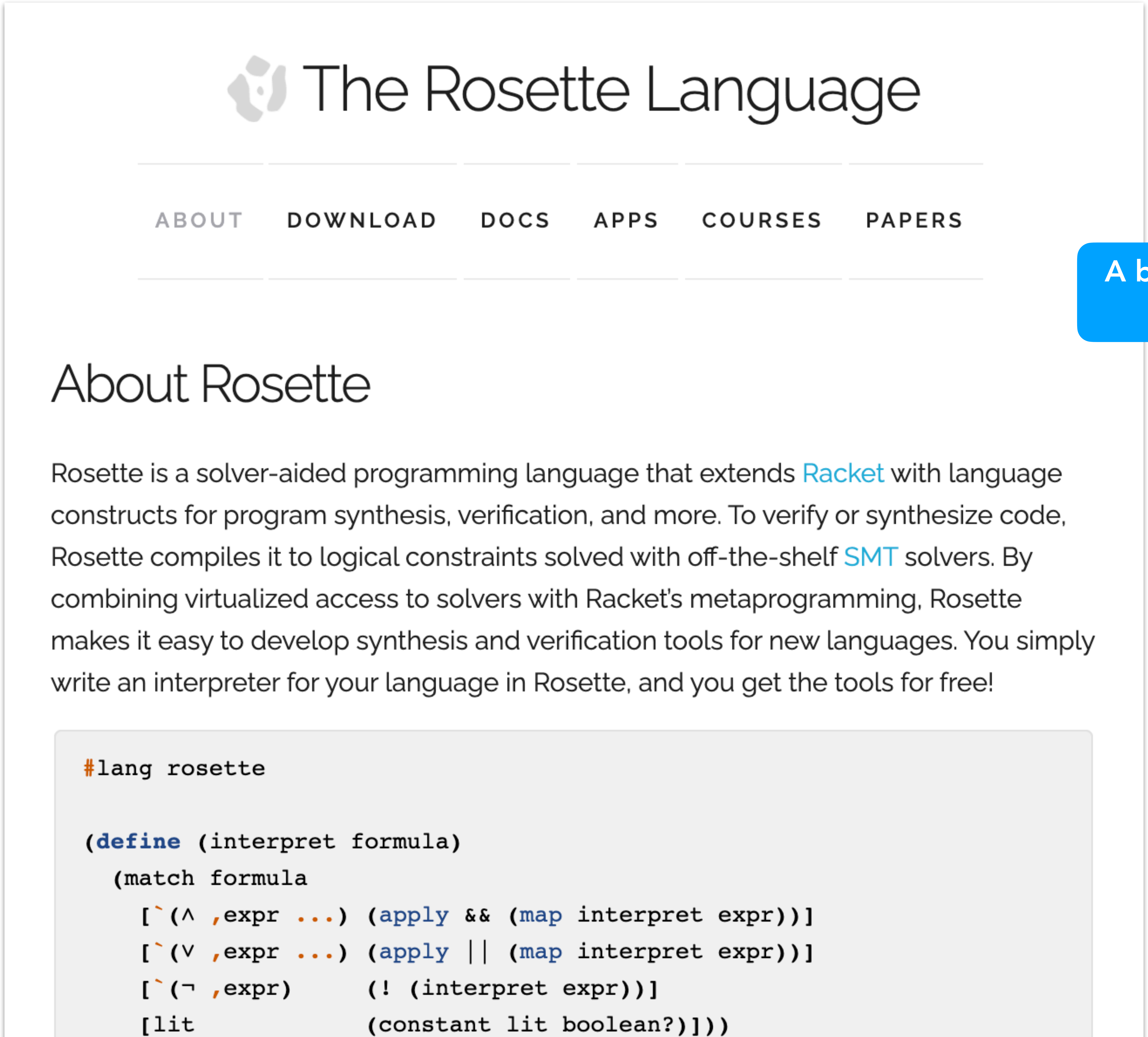
Discuss in groups

- So far, do you find enumerative search (tracking the subset of the program space explored so far) more natural? Or symbolic search (representing the space of all valid programs)?
- How are you feeling about viewing programs as manipulable objects rather than text?
    - Have you thought before about the fact that programs themselves are also data and can be treated as inputs to other programs?

Reading Key Takeaways

- Inductive reasoning makes broad generalizations from specific observations
  - → inductive synthesis is about generalizing from ambiguous specifications
- The difference between finding *a* program that satisfies the spec and finding the program the user actually wants
- A nice review of ASTs and the importance of the DSL design for the program space size (although we touched on these in last class)
- A friendly introduction to symbolic search (representing the space of all valid programs instead of tracking the subset of programs explored so far)

Enumerative → Symbolic (Constraint-Based)



The Rosette Language

ABOUT DOWNLOAD DOCS APPS COURSES PAPERS

About Rosette

Rosette is a solver-aided programming language that extends Racket with language constructs for program synthesis, verification, and more. To verify or synthesize code, Rosette compiles it to logical constraints solved with off-the-shelf SMT solvers. By combining virtualized access to solvers with Racket's metaprogramming, Rosette makes it easy to develop synthesis and verification tools for new languages. You simply write an interpreter for your language in Rosette, and you get the tools for free!

```
#lang rosette

(define (interpret formula)
  (match formula
    [`(∧ ,expr ...) (apply && (map interpret expr))]
    [`(∨ ,expr ...) (apply || (map interpret expr))]
    [`(¬ ,expr)     (! (interpret expr))]
    [lit            (constant lit boolean?)]))
```

A brilliant language from Emina Torlak

If you want to get *really* into Rosette, I recommend…

- https://courses.cs.washington.edu/courses/cse507/19au/index.html

Here be dragons
Calandradua

Let’s tour Rosetteland!

- Can you run this program in DrRacket? Please try to help each other debug if you can’t!
    -

```
#lang rosette/safe

(require rosette/lib/synthax)
(current-bitwidth #f)
```

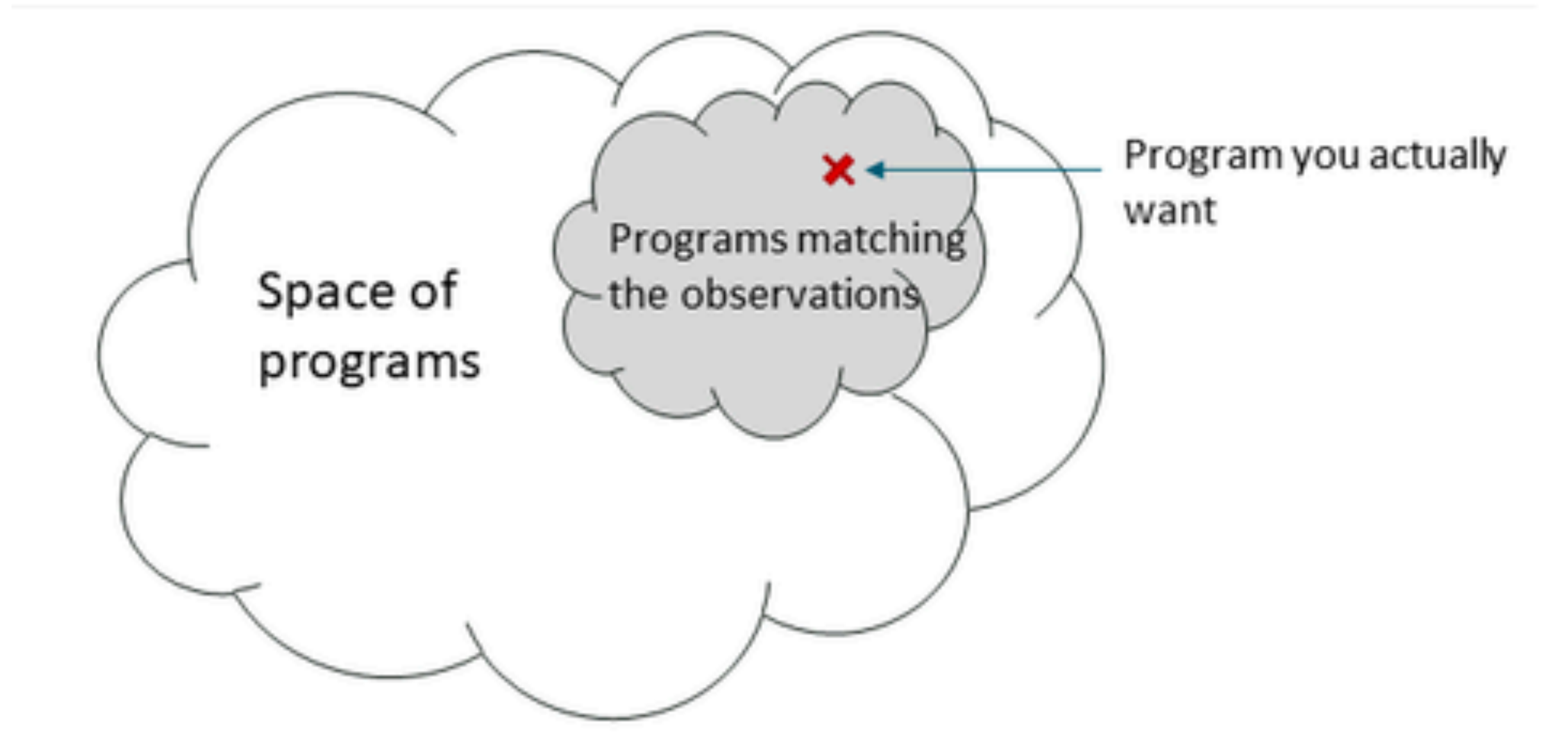
(1) individual Rosette intro activity
(2) group Rosette activity

What did you learn from the Rosette activity?

A few learning goals

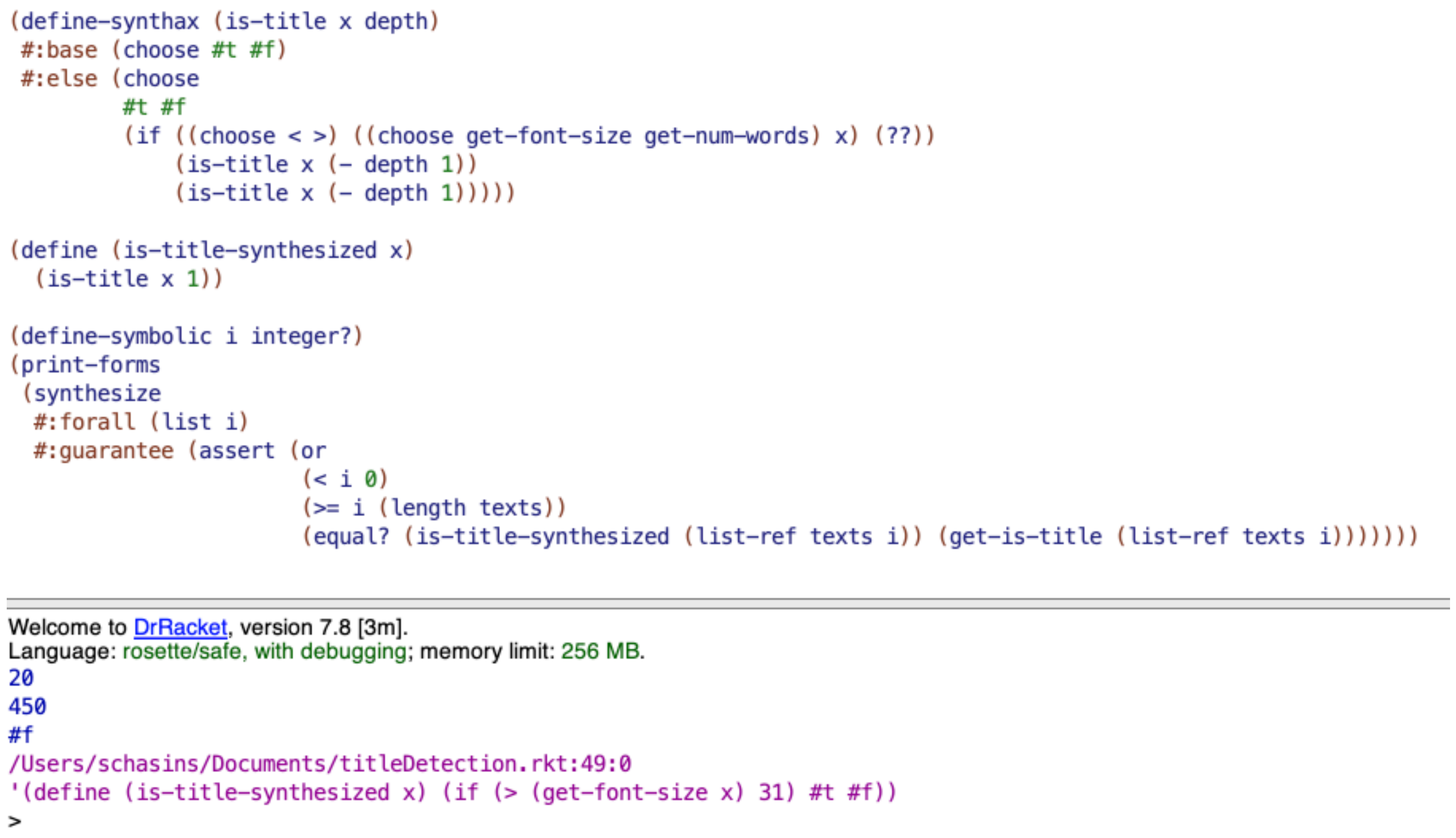
You might have learned…

- 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.



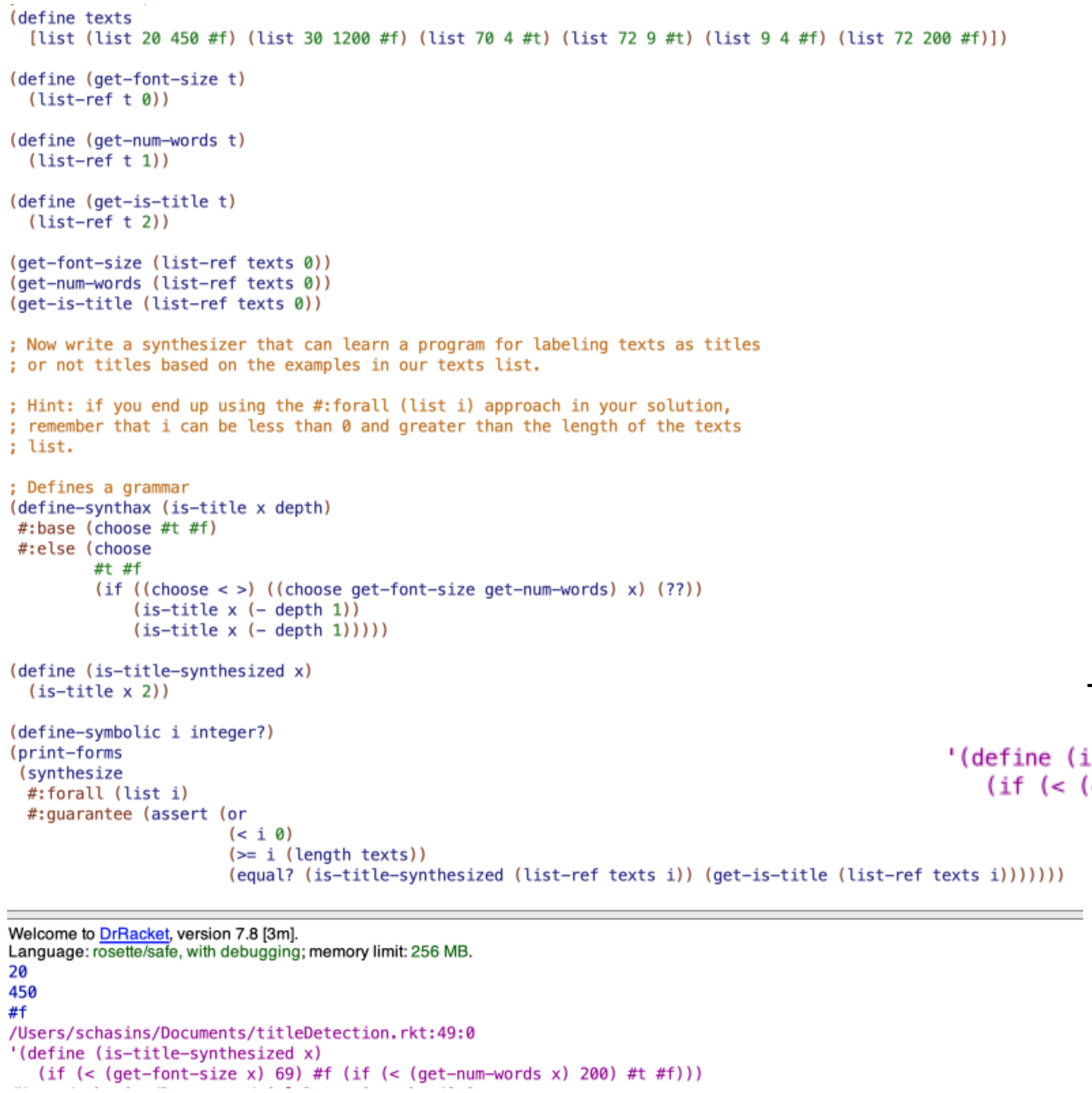
https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture2.htm
Armando Solar-Lezama

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)])
```

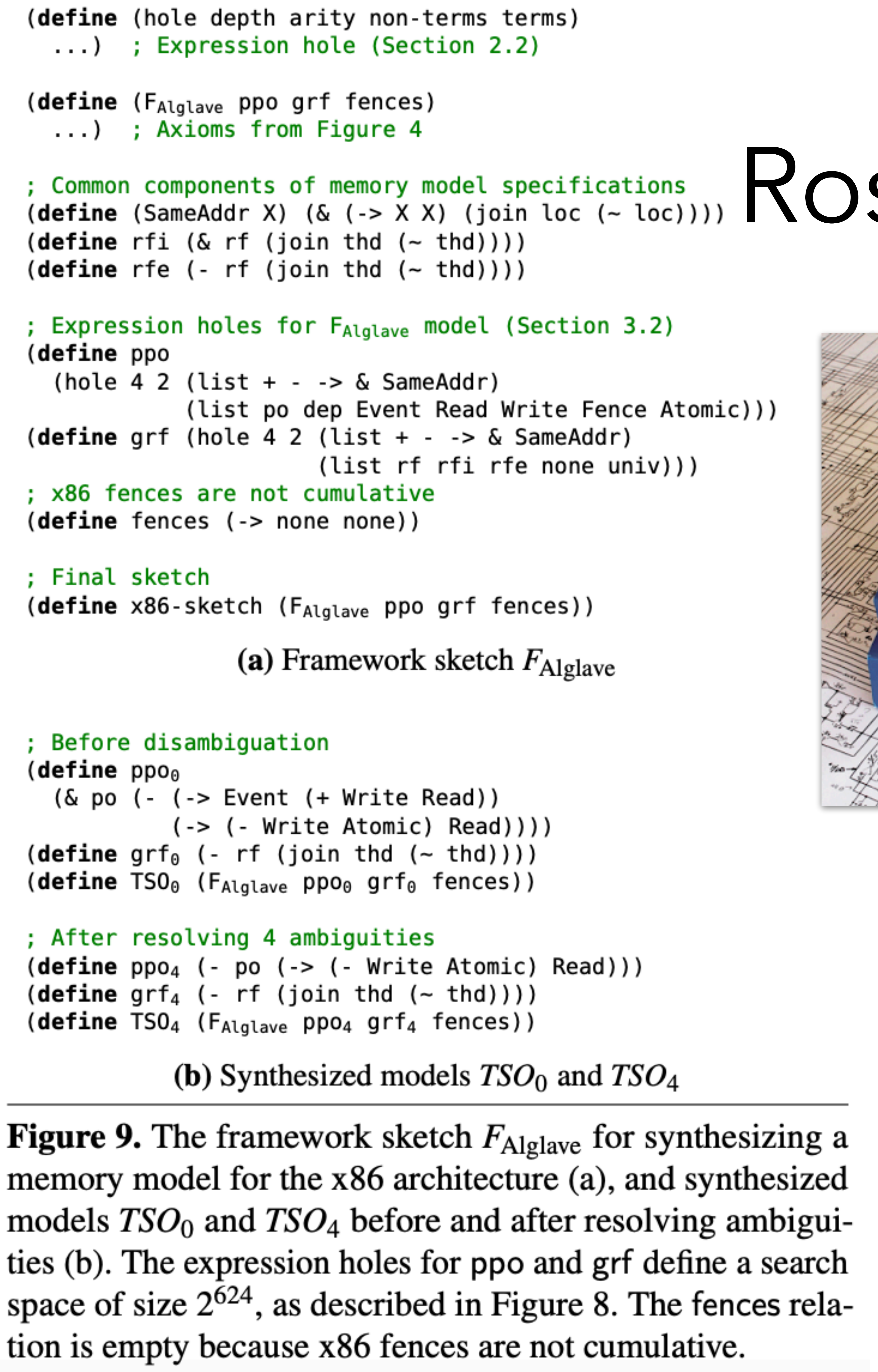
mavbe a footnote

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…



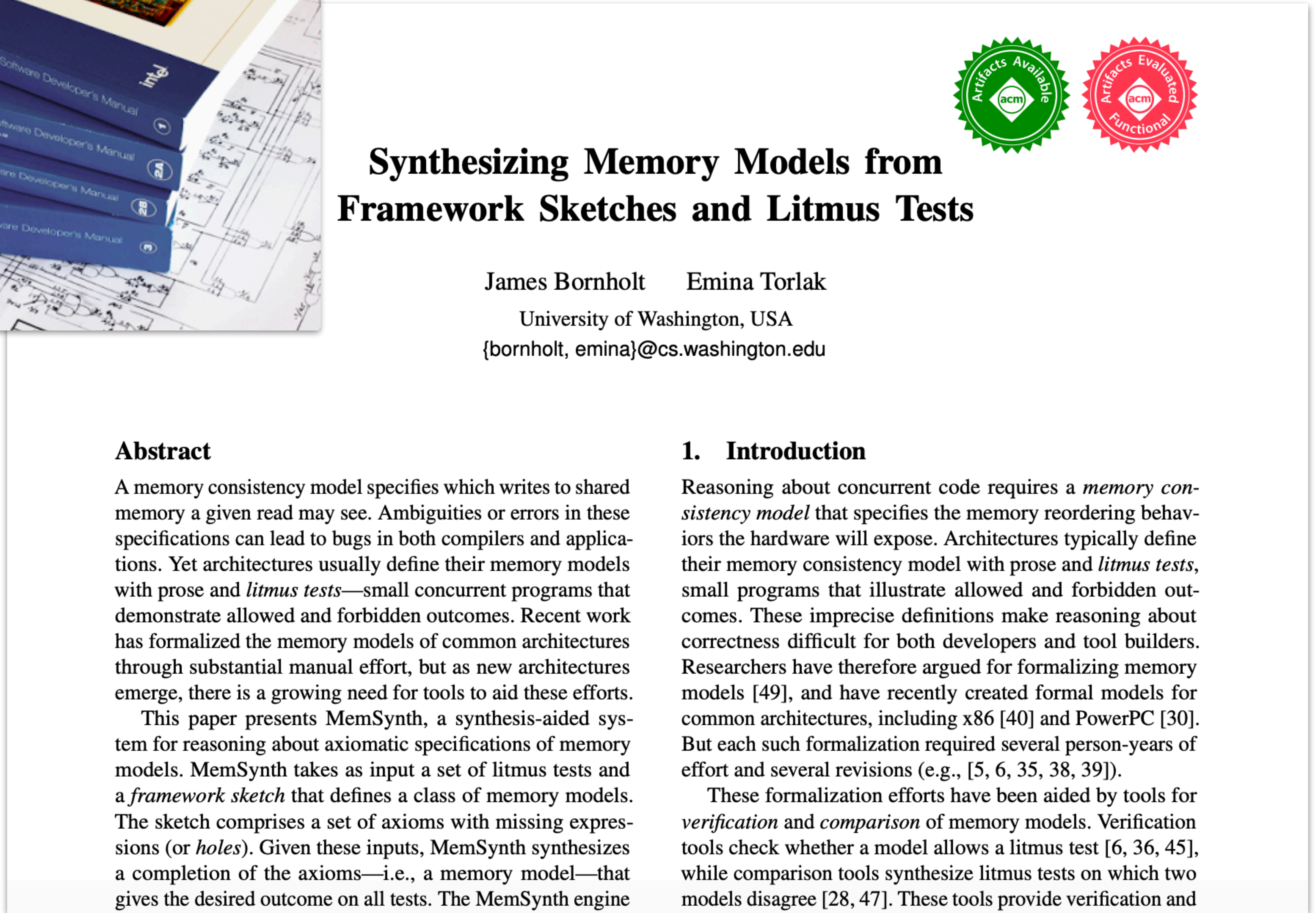
```
(define (hole depth arity non-terms terms)
  ...)  ; Expression hole (Section 2.2)

(define (F_Alglave 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 F_Alglave 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 (F_Alglave ppo grf fences))
```

(a) Framework sketch F_{Alglave}

```
; Before disambiguation
(define ppo_0
  (& po (- (-> Event (+ Write Read))
           (-> (- Write Atomic) Read))))
(define grf_0 (- rf (join thd (~ thd))))
(define TSO_0 (F_Alglave ppo_0 grf_0 fences))

; After resolving 4 ambiguities
(define ppo_4 (- po (-> (- Write Atomic) Read)))
(define grf_4 (- rf (join thd (~ thd))))
(define TSO_4 (F_Alglave ppo_4 grf_4 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.

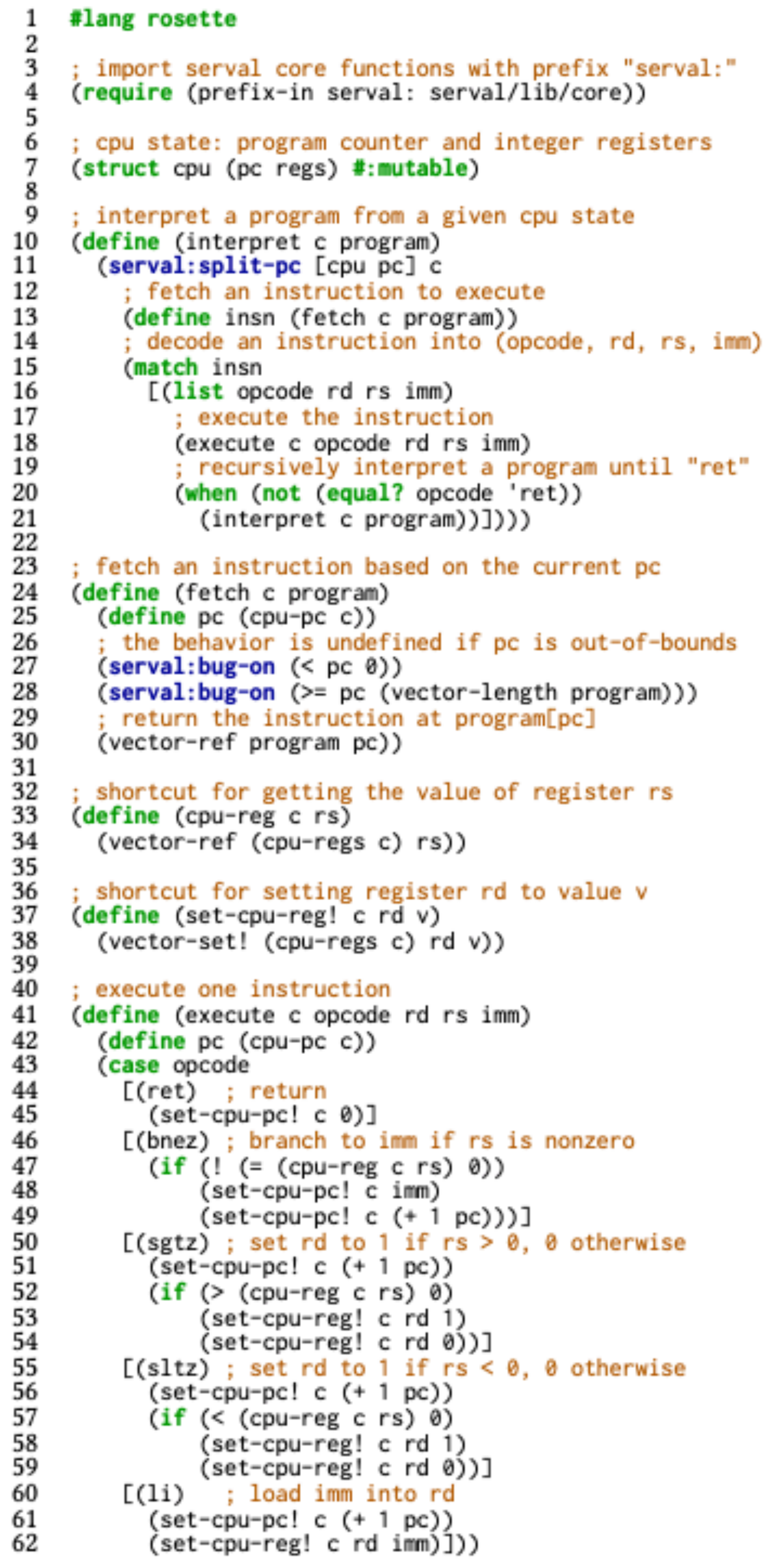
This paper presents MemSynth, a synthesis-aided system for reasoning about axiomatic specifications of memory models. MemSynth takes as input a set of litmus tests and a *framework sketch* that defines a class of memory models. The sketch comprises a set of axioms with missing expressions (or *holes*). 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]).

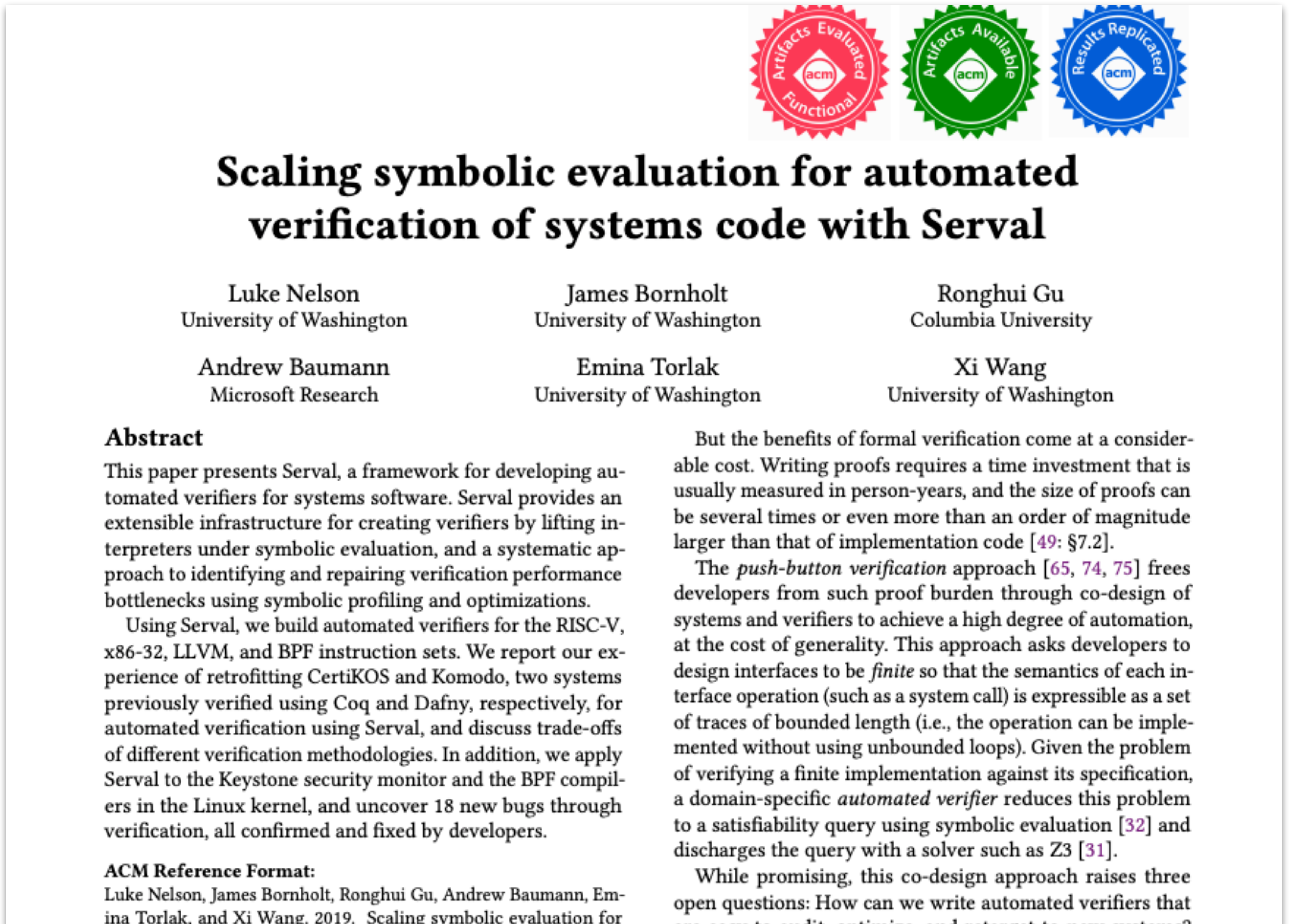
These formalization efforts have been aided by tools for *verification* and *comparison* of memory models. Verification tools check whether a model allows a litmus test [6, 36, 45], while comparison tools synthesize litmus tests on which two models disagree [28, 47]. These tools provide verification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

quick prep for next session

To think about for next reading

- You do *not* need to memorize or deeply understand details of these approaches!
  - I want you to recognize the key terms and know where to turn for a high-level overview of key techniques. Also, this chapter offers excellent pointers to examples of synthesis work, which you might find useful if you start tackling a synthesis project.
- Think about
  - How these different approaches would or wouldn't apply to the synthesis ideas you brainstormed last session
  - How these different approaches shape the user interaction

Install before next 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)
```