Logic and Computation – CS 2800 Fall 2019

Lecture 2

Intro continued

Designing Programs

Introduction to ACL2s

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- Introduction and motivation continued
- Course web pages, logistics
- Designing programs, review
- Introduction to ACL2s

Recap: software science

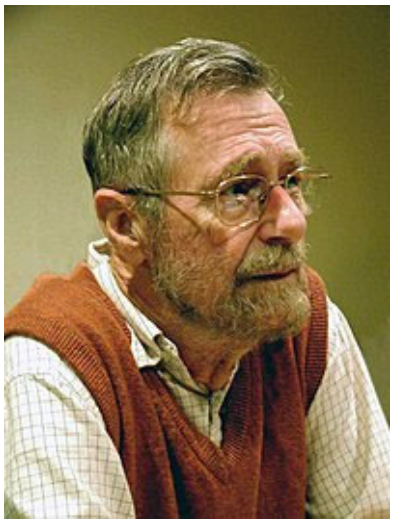
- What predictions can we make about the programs (software) that we write?

- Can we predict:
  - Whether our program will terminate?
  - That it will not crash? Never? In some cases? Under which conditions exactly?
  - That it will produce the correct result? What exactly is the correct result?
  - That it will never attempt to divide by zero?
  - That it will not access forbidden parts of the memory?
  - That it will not leak private information?
  - …

What about testing?

- Can’t we just write code and test it?

Edsger Dijkstra
[1930 – 2002]



“Program testing can be used to show the presence of bugs, but never to show their absence!” [Dijkstra, 1970]

- Testing is fundamental, and a whole discipline by itself. It is also the workhorse of industrial software engineering practice.
- In this class we will look at testing, but also focus on **proving**.

Still, do we need a software "science"?

- Do we need physics, chemistry, biology, ... ?
  - Humans have lived without science for millennia or more.
  - You don't need physics to predict that an apple will fall from a tree.
  - You don't need physics to learn to ride a bike.
  - You don't need physics/chemistry/biology/geology to build a hut, grow plants, cook your food.

  - But you need these sciences if you want to do harder things:
    - Build a bridge, a skyscraper, …
    - Build trains, automobiles, planes, rockets, …
    - Cure cholera, malaria, cancer, …

- The stronger the science the strongest (less trivial) the predictions it can make.
  - In this course we will learn techniques which can produce far stronger predictions than testing

Still, why do we need stronger predictions?

- After all, software always seems to come with "no warranty, no liability, …"

- This is only partly true:
  - Safety-critical software has stricter compliance requirements than your usual software
    - E.g., the standard *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*

- Even where true, it signals a so-far failure of computer science => it should motivate us to improve !

- New applications are forcing us to improve
  - **Cyber-physical systems**
  - Robots, autonomous cars, "smart" buildings, smart power-grid, …

Cyber-physical systems: future

Courtesy `https://vimeo.com/bsfilms`

Thanks to Christos Cassandras for recommending this video

Cyber-physical systems: present

Autonomous car driving through red light

Lives and costs

- Safety-critical systems: lives at risk
- Other systems: dollars at risk

What could go wrong?

- Ariane 5 rocket: 10 year development, $7B cost, explodes due to software bug. No casualties, cost $2B.
- Space shuttle: testing costs $1000 per LOC.
- Functional verification accounts for > 40% of total chip costs (after Intel $475M FDIV bug).
- Toyota found guilty by Oklahoma jury for death due to software bugs.
- NIST: cost of software bugs is $59.5B a year.
- Gulf War: Patriot misses incoming Iraqi Scud: 28 soldiers killed; 100s injured; GAO: software bug.
- USS Vincennes, using Aegis system, shot down an Iranian aircraft with pilgrims to Mecca: 290 dead.
- People given lethal doses of radiation from Therac-25 machines (1980s); Multidata software (2000s).
- Knight capital lost $500M in 1/2 an hour due to bug in trading software.
- 1982 Soviet gas pipeline: CIA sabotaged trans-Siberian gas pipeline software: largest non-nuclear explosion.
- Love virus: Virus from email attachment with subject “ILOVEYOU” infected millions of computers: $9B cost.
- ...

The science of software and systems

- Theses:
  - Almost all systems today rely heavily on software
  - Almost all system design is software design
    - When you design traditional software you program in C, Java, Python, …
    - When you design hardware you write Verilog code: Verilog programs are software
    - When you design embedded controllers you write Matlab/Simulink code: this too is software
    - When you design a robot?
    - When you design a self-driving car?
    - When you design a market trading algorithm?
    - When you design a drug?
    - …
- The fundamental principles to reason formally about programs and systems are the same!
  - Bonus: what you learn in this course will help you in non-traditional software domains

CS 2800 course overview

CS 2800: course overview

- Goals in a nutshell:
  - In Fundies 1 you learned how to program.
  - In 2800 you will learn how to reason about your programs (make predictions, prove stuff).

- Focus on the fundamentals:
  - Programming language **syntax**: how to write a program correctly
  - Programming language **semantics**: what the program does/means
  - **Specification**: making clear, unambiguous statements about our programs
  - Tests: testing those statements automatically (**property-based testing**)
  - **Proofs**: proving those statements
  - **Logic**: the language and methods to reason about our programs
    - Language to write specifications
    - Language and methods to do the proofs

- Material accessible to 1st year undergrads.
  - Hands-on experience using **ACL2s**.
  - Logic presented from a computational perspective.

CS 2800: course overview

- Rough list of novel concepts:
  - contracts
  - invariants
  - properties
  - testing
  - propositional logic
  - equational reasoning
  - recursion and induction
  - termination analysis
  - various proof techniques
  - software verification

There are many other topics in software science, that we will not have time to cover thoroughly or even touch upon in this course!

- Model checking
- Theorem proving (the internals)
- Testing
- Abstract interpretation
- Static analysis
- Controller/program synthesis
- Programming language theory
- Type theory
- …
- See more advanced CS courses, e.g.:
  - 4820: Computer-Aided Reasoning
  - 4830: System Specification, Verification, and Synthesis
  - 6535: Engineering Reliable Software
  - …

Logistics

Course web pages

- Review course web pages
- Notes:
  - Exams: book your dates
  - Quizzes: individual, graded, one or more per lecture
    - Install Poll Everywhere
  - Homeworks: graded, weekly, done in groups of 2-3
  - Piazza: enroll
  - ACL2s: install
  - Office hours:
    - Knock on my door during office hours

Class rules

- Lecture notes:
  - Try to read them **before class**.
  - Train to learn on your own.
  - Come prepared with questions ⇒ more effective use of lecture time.
- No recordings allowed
- No electronics in class except when used for quizzes (Poll Everywhere)
- **Please turn off your phones, tablets, laptops, ... now!**

Designing programs: review

Quiz (not graded)

- On a piece of paper, define the following function

```
;; rl: List x Nat -> List
;; Given a list, l, and a natural number, n,
;; rl rotates the list to the left n times

;; Primary consideration: correctness
;; No need to worry about efficiency

;; some tests:
(check= (rl (list 1 2 3) 1) (list 2 3 1))
(check= (rl (list 1 2 3) 2) (list 3 1 2))
(check= (rl (list 1 2 3) 3) (list 1 2 3))
```

Designing programs

```
;; len: List -> Nat
;; Given a list, return its length
```

1. Identify data definitions
2. Write a description

```
(definec len (l :tl) :nat
  (if (endp l)
    . . .
    (. . . (len (rest l)) . . .)))
```

4. Contracts
5. Data-driven definition template

```
(check= (len (list 1 2 3)) 3)
(check= (len ()) 0)
```

3. Test cases

Designing programs

```
;; len: List -> Nat
;; Given a list, return its length
```

1. Identify data definitions
2. Write a description

```
(definec len (l :tl) :nat
  (if (endp l)
    0
    (+ 1 (len (rest l)))))
```

4. Contracts
5. Data-driven definition template
6. Complete data-driven definition

```
(check= (len (list 1 2 3)) 3)
(check= (len ()) 0)
```

3. Test cases

ACL2s vs Racket

```
(definec len (l :tl) :nat
  (if (endp l)
    0
    (+ 1 (len (rest l)))))




(check= (len (list 1 2)) 2)
(check= (len (list)) 0)
```

```
(define (len l)
  (if (empty? l)
    0
    (+ 1 (len (rest l)))))




(check-expect (len (list 1 2)) 2)
(check-expect (len (list)) 0)
```

ACL2s vs Racket

What if we had these instead?

```
(definec len (l :tl) :nat
  (if (endp l)
    -1
    (+ 1 (len (rest l)))))



(check= (len (list 1 2)) 2)
(check= (len (list)) 0)
```

```
(define (len l)
  (if (empty? l)
    -1
    (+ 1 (len (rest l)))))



(check-expect (len (list 1 2)) 2)
(check-expect (len (list)) 0)
```

ACL2s will not accept the above definition, but Racket will.
Contracts allow ACL2s to check function signatures.

ACL2s

- Based on ACL2
  - Widely used in the industry, e.g., AMD, Rockwell, IBM, Intel, GE, Centaur, …
  - ACM Software System Award 2005

- ACL2s is:
  - A programming language with contracts
  - A logic to express properties of programs, theorems, etc
  - A theorem prover that
    - Automates reasoning about programs (proving properties, generating test cases, …)
    - Book-keeping to ensure we don't make mistakes

- Note: it is **not** the goal of this class to make you ACL2s experts
  - Focus on foundations
  - You should be able to adapt what you learn here to other tools

Next time

- Designing programs continued
- Tour of ACL2s continued
  - **Install ACL2s and experiment with it!**
  - **Try to finish your installation before Friday's lab**
  - **At the very least you should download everything before the lab (many hundreds of MBs of downloads, you won't have time to do it during lab)**

  - rl: define and test the function using ACL2s or Racket
- Skim the lecture notes