CIS352 Course Wrapup

May 3, 2022
Projects

• Want to start by talking about project…

• Projects are the core part of the course

• Independent experience broadly related to the class

• Why do we do the projects?
  • Main goal: require you to learn debugging
  • The projects become conceptually nontrivial, and even the most experienced programmers will make mistakes
  • Understand how to make hypotheses about what may be buggy is crucial—I would like to do a better job of teaching this in subsequent (smaller) classes
Project Performance

- **Overall quite good project performance**
- About 85% of the class did P1
  - Almost everyone got ~100%
- About 85% of the class did P2
  - >75% of those got >85% (many >100)
- About 75% of the class did P3
  - 2/3 of those got >80%, rest got ~40-80
- About 70% of the class did P4
  - On average people did better than P3
Project Thoughts

- One was a warmup project with Racket / Autograder / …
  - Reiterating lessons of recursion, symbols, lists
- Second is PageRank:
  - Accumulating a hash—immutable maps are a key concept
- Three was a Scheme interpreter:
  - Functionally implement set! via “threading” store through recursive interpreter—this is an instance of the State monad (Haskell)
- Last, the Church encoder
  - Teaches concepts of compiler design: consume syntax as input, transform to new syntax to be executed as lambda calculus
Project Design Aspects

• Lots of the course was just **learning** Racket’s mix of features
  • As a design feature of the course, this has upsides and downsides
• Projects get **harder** and **more open-ended** as they progress
  • Different students report different projects hardest
• I think the right order is:
  • P3 (hardest, coding-wise, lots of places to make mistakes)
  • P4 (easier coding, conceptually harder, *trickiest to debug*)
  • P1 (learning Racket is hard, can be tedious, fast-paced)
  • P2 (surprisingly, many find this easy **once they understand folds**)
http://coursefeedback.syr.edu/
Some Course Concepts
Program with **Expressions** rather than **Statements**

- One significantly underrated aspect of functional programming
- Which of the following looks better?

```scheme
(define (foo x)
  (if x #t #f))

(define (foo x) x)
```

Why are we so tempted to write code that looks like the first?

(Potential) answer: common idiom from **statement-based** languages (Python/Java/…)—use sequence of `if/else/switch` to set a flag to return
Folds are specific kind of loop

- Folds are akin to a for loop that iterates over an ordered sequence and accumulates a value
- Trivial extensions: iterate over a set (call set->list), accumulate a hash / pair / set of values
Every **fold** corresponds to a **for** loop and **tail-recursive** function

```
(define (rec-reverse l)
  (define (h l acc)
    (match l
      ['() acc]
      [`(,hd . ,tl) (h tl (cons hd acc))]]
    (h l '())))
```
Every fold corresponds to a for loop and tail-recursive function

```
(define (fold-reverse l)
  (define (rec-reverse l)
    (define (h l acc)
      (match l
        ['() acc]
        [`,(,hd . ,tl) (h tl (cons hd acc))]))
    (h l '()))
  (define (foldl (lambda (x acc) (cons x acc)) '() l))
```
Every **fold** corresponds to a **for** loop and **tail-recursive** function

```
(define (for-reverse l)
  (define acc '())
  (for ([i l])
    (set! acc (cons i acc))
    x)

;; (for-reverse '(1 2 3))
```

```
(define (fold-reverse l)
  (foldl (lambda (x acc) (cons x acc))
         '(')
         l))
```

```
(define (rec-reverse l)
  (define (h l acc)
    (match l
      ['(()) acc]
      [`(~hs . ,tl) (h tl (cons hd acc))]
      (h l '())))
  (h l '()))
```

```
(define (define (rec-reverse l))
  (define (h l acc)
    (match l
      ['(()) acc]
      [`(~hs . ,tl) (h tl (cons hd acc))]
      (h l '())))
  (h l '()))
```
Every **fold** corresponds to a **for** loop and **tail-recursive** function

```
(define (for-reverse l)
  (define acc '())
  (for ([i l])
    (set! acc (cons i acc)))
)

;; (for-reverse '(1 2 3))

(define (fold-reverse l)
  (foldl (lambda (x acc) (cons x acc)
    '()) l))

(define (rec-reverse l)
  (define (h l acc)
    (match l
      ['() acc]
      ['(~(,hd . ,tl) (h tl (cons hd acc)))]))
  (h l '()))
```
Representing / Manipulating Syntax

• To define semantics / language features

• **Interpreters**—consume syntax and produce **values**

• **Compilers**—consume syntax and produce **programs**
  
  • Subsequently run via lower-level machine, preserve semantics
(define (scoped-λ-term? t ρ)
  (match t
    [(? symbol? x) (set-member? ρ x)]
    [`(~(,t0 ,t1)
      (and (scoped-λ-term? t0 ρ) (scoped-λ-term? t1 ρ))]
    [`(lambda (,(? symbol? xs) ...) ,e)
      (scoped-λ-term? e (set-union ρ (list->set xs)))]
  )
)

(scoped-λ-term? '(lambda (x) (x x)) (set))
(scoped-λ-term? '(((lambda (x) (lambda (y) (y x)))
    (lambda (z x y) (x y)))
  (set))
(scoped-λ-term? '(((lambda (x) (lambda (y) (z x)))
    (lambda (z x y) (x y)))
  (set))
Metacircular Interpreters (P3)

- Write an interpreter for a target language in a source language reusing features of source language
- Upside: expressive, succinct, straightforward to implement
- Downsides: (may be) slow if defining (meta) language is slow
Metacircular Interpreters (P3)

• Write an interpreter for a target language in a source language reusing features of source language

• Upside: expressive, succinct, straightforward to implement

• Downsides: (may be) slow if defining (meta) language is slow

• Most dynamic languages (Pearl, Ruby, Python, …) have relatively-fast interpreters that use high-performance native (C++/Rust/…) data structures but follow these same principles

• Compilation has mostly focused on lower-level memory-unsafe languages (C++) with the addition of compilation to bytecode (compile to IR; interpret IR w/ very-efficient interpreter)
A language with two extra ops: getstk and printstk.

Assume $\rho$ is Variable -> Value

Value ::= (closure $\rho$ e) (stack e ...)

e is source expressions

e ::= x (e e) (lambda (x) e) (getstk) (printstk e)

stk ::= list of expressions (stack e)
(define (eval-λ+stack e ρ stk)
  (match e
    [(? symbol? x) (hash-ref ρ x)]
    [`(lambda ,(x) ,e-body)
      `(closure ,e ,ρ)]
    [`(getstk) `(stack ,stk)]
    [`(printstk ,e+)
      (define stk-v (eval-λ+stack e+ ρ (cons e stk)))
      (displayln "Captured stack:")
      (for ([expr stk-v])
        (pretty-print expr))]
    [`(,e0 ,e1)
      (define v-e0 (eval-λ+stack e0 ρ stk))
      (match v-e0
        [`(closure (lambda ,(x) ,e-body) ,ρ+)
          (define v-a (eval-λ+stack e1 ρ stk))
          (eval-λ+stack e-body (hash-set ρ+ x v-a) (cons e stk))]
        [_ (error (format "can't apply ~a" v-e0))]])))
Debugging

We want you to form hypotheses for broken code

“When I have a piece of broken code, how can I interact with it to test a hypothesis about what it is doing?”

Why is this hard? A: debugging difficulty / frustration is often related to the amount of time between experiments

May have to modify code multiple times, hence multiple interactions
(define (bad-eval e ρ)
  (match e
    [(? number? n) n]
    [(? symbol? x) (hash-ref ρ x)]
    [(lambda (,x) ,e-body)
      ~(closure ,e ,ρ)]
    [`,(e0 ,e1)
      (match (bad-eval e0 ρ)
        [`,(closure (lambda (,x) ,e-body) ,ρ+)
          (define v-a (bad-eval e1 ρ))
          (bad-eval e-body (hash-set ρ+ x v-a))]]]
    [`(+ ,e0 ,e1)
      (+ (bad-eval e0 ρ) (bad-eval e1 ρ))]
    [`,(- ,e0)
      (- (bad-eval e0 ρ))]))
(define (bad-eval e ρ)
  (match e
    [(? number? n) n]
    [(? symbol? x) (hash-ref ρ x)]
    [`(lambda (,x) ,e-body)
      `(closure ,e ,ρ)]
    [`(,e0 ,e1)
      (match (bad-eval e0 ρ)
        [`(closure (lambda (,x) ,e-body) ,ρ+)
          (define v-a (bad-eval e1 ρ))
          (bad-eval e-body (hash-set ρ+ x v-a)))]
        [`(+) (bad-eval e0 ρ) (bad-eval e1 ρ)]
        [`(--) (bad-eval e0 ρ)])]]
(bad-eval '(((lambda (x) (+ x 2)) (+ 1 2)) (hash))
  ;; 5
(define (bad-eval e ρ)
  (match e
    [([? number? n) n]
    [([? symbol? x) (hash-ref ρ x)]
    [`(lambda ,(x) ,e-body)
      `(closure ,e ,ρ)]
    [`(,e0 ,e1)
      (match (bad-eval e0 ρ)
        [`(closure (lambda ,(x) ,e-body) ,ρ+)
          (define v-a (bad-eval e1 ρ))
          (bad-eval e-body (hash-set ρ+ x v-a)))]
    [`(+ ,e0 ,e1)
      (+ (bad-eval e0 ρ) (bad-eval e1 ρ))]
    [`(− ,e0)
      (− (bad-eval e0 ρ)))]
  )
)
(bad-eval '(((lambda (x) (+ x 2)) (+ 1 2)) (hash))
 ;; 5
Looks good; but crucially broken.
(define (bad-eval e ρ)
  (match e
    [(? number? n) n]
    [(? symbol? x) (hash-ref ρ x)]
    [`(lambda ,x ,e-body)
      `(closure ,e ,ρ)]
    [`,(,e0 ,e1)
      (match (bad-eval e0 ρ)
        [`(closure (lambda ,x ,e-body) ,ρ+)
          (define v-a (bad-eval e1 ρ))
          (bad-eval e-body (hash-set ρ+ x v-a)))]
        [`(+) ,e0 ,e1)
          (+ (bad-eval e0 ρ) (bad-eval e1 ρ))]
        [`(--) ,e0)
          (- (bad-eval e0 ρ)))]
  )
)

(bad-eval '((lambda (x) (+ (- x) 2)) (+ 1 2)) (hash))
;; hash-ref: no value found for key!
Now we look at the term and think: when does this case happen?

Based on the fact `hash-ref` is in the `symbol` case, it must be this subexpression

```
(bad-eval '((lambda (x) (+ (- x) 2)) (+ 1 2)) (hash))
```
But why would this cause problems?

Now we ask: what is the right thing that should happen?

We think: "it should be executing the - branch."

To test this hypothesis we edit the code...

```
(bad-eval '(((lambda (x) (+ (- x) 2)) (+ 1 2)) (hash))
```

(define (bad-eval e ρ)
  (match e
    [((? number? n) n]
    [((? symbol? x) (hash-ref ρ x))
     `(lambda ,(x), (e-body))
     `(closure ,e ,ρ))]
    `,(e0 ,e1)
    (match (bad-eval e0 ρ)
      [`(closure (lambda ,(x), (e-body)) ,ρ+)
       (define v-a (bad-eval e1 ρ))
       (bad-eval e-body (hash-set ρ+ x v-a)))]
    `(+ ,e0 ,e1)
    (+ (bad-eval e0 ρ) (bad-eval e1 ρ))]
    `(- ,e0)
    (displayln "(evaluating (- ...))")
    (- (bad-eval e0 ρ))))
Now we run the instrumented code with the same testcase

But we never see our new code

But how could that happen?

```
(define (bad-eval e ρ)
  (match e
    [([? number? n) n]
    [([? symbol? x) (hash-ref ρ x)]
      `(lambda (,x) ,e-body)
      `(closure ,e ,ρ)])
    [(`(,e0 ,e1)
        (match (bad-eval e0 ρ)
          [(`(closure (lambda (,x) ,e-body) ,ρ+)
              (define v-a (bad-eval e1 ρ)))
            (bad-eval e-body (hash-set ρ+ x v-a)))]
          [(`(+ ,e0 ,e1)
              (+ (bad-eval e0 ρ) (bad-eval e1 ρ))]
          [(`(- ,e0)
              (displayln "(evaluating (- ...))")
              (- (bad-eval e0 ρ)))]))))
```

```
(bad-eval '((lambda (x) (+ (- x) 2)) (+ 1 2)) (hash))
```
(define (bad-eval e ρ)
  (match e
   [(? number? n) n]
   [(? symbol? x) (hash-ref ρ x)]
   [`(lambda ,(x) ,e-body)
     `(closure ,e ,ρ)]
   [`(,e0 ,e1)
     (match (bad-eval e0 ρ)
       [`(closure (lambda ,(x) ,e-body) ,ρ+)
         (define v-a (bad-eval e1 ρ))
         (bad-eval e-body (hash-set ρ+ x v-a)))]
       [`(+) ,e0 ,e1]
         (+ (bad-eval e0 ρ) (bad-eval e1 ρ))]
       [`(- ,e0)
         (displayln ``(evaluating (- …))`)
         (- (bad-eval e0 ρ)))]
     )]
)

(bad-eval `((lambda (x) (+ (- x) 2)) (+ 1 2)) (hash))

Answer: our match statement is broken! Function application eagerly matches (- x)

Thus, - is looked up via the symbol case.. and crashes
(define (bad-eval e ρ)
  (match e
    [(? number? n) n]
    [(? symbol? x) (hash-ref ρ x)]
    [`(lambda (,x) ,e-body)
      `(closure ,e ,ρ)]
    [`(+ ,e0 ,e1)
      (+ (bad-eval e0 ρ) (bad-eval e1 ρ))]
    [`(− ,e0)
      (− (bad-eval e0 ρ))]
    [`(,e0 ,e1)
      (match (bad-eval e0 ρ)
        [`(closure (lambda (,x) ,e-body) ,ρ+)
          (define v-a (bad-eval e1 ρ))
          (bad-eval e-body (hash-set ρ+ x v-a))]]))]

(bad-eval '(((lambda (x) (+ (- x) 2)) (+ 1 2)) (hash))
  ;; -1 🎉)
Compilers (P4)

• Traditionally, the C++-style compiler-engineering workforce was small

• As language technology evolves (Rust, WebAssembly, ...), the language design landscape has become more granular

• Developers harness application-specific algorithmic and hardware features

• Examples include GPGPU (General-Purpose GPU)
LLVM

- Compiler backend for C-like languages
- If you run a Mac, this is your native build toolchain
- Supersedes GCC in design methodology, robustness, & ease of extension
- Common compiler target that abstracts around register allocation, etc…
The future of Chips

- All languages ultimately execute in native instruction set of some chip
- From 90s-2020: x86 (Pentium/Core iX/… chips), x86-64 (AMD64)
  - AMD chips currently offer leading core-density via manufacture at TSMC
  - TSMC able to print chips at densest scale due to its use of ASML’s Extreme UltraViolet (EUV) photolithography
M1 Ultra (Apple)

- Apple has designed world-class chips since their experience w/ iPhone
- Built on ARM, RISC assembly, much simpler than X86-64 (TSMC)
- Instruction decoding much cheaper
- Modern system-on-chips (M1 Ultra) integrate CPU+GPU to achieve awesome speeds
- Application-specific instructions + toolchain integration (supports emulation)

http://hrtapps.com/blogs/20220427/
Languages Into the Future

• Fast, high-level abstractions
  • Highly-dynamic langs (Perl) intrinsically slow, good in-between spots (Rust)
  • Application-specific acceleration via GPUs/ISA/…
• Safety generally prevails once runtime overhead effectively mitigated
  • Garbage-collected langs: once GC fast enough
  • “Fancy types for memory” languages (Rust)—once community built / good compiler error msgs for type / borrow issues, etc…
• “Desktop OS” idea will become less dominant
  • Every app compiles its OS in, runs on a hypervisor situated on cloud/local server
  • Common components (libraries, runtime, GC) shared
Exams and Participation

- Quizzes can be stressful, but designed to be checkpoints to motivate you to study topics on a specific timeline
- Many students did corrections, almost all got 10/10
- Overall, most students averaging B to B- on exams
- Final will have 10 questions (like Q4)—up to 8 answers
  - Monday, May 9, LSC 105 (normal room), 5:15 to 7:15 PM
- Roughly half of students will get bump to + for participation, other half will see no change, very few will (possibly) get a -
Final Logistics

• Last call for projects is May 8, 2022 @ 11:59PM

• Consult grade calculator, may trade up to 15 points between categories

• In practice, I may average (i.e., let you take as many points as useful) the two categories

• I will be flexible on grading in practice, but when bumping students up I will prefer those with higher project grades vs. exam grades

• I may overlook late projects if they are otherwise correct

• I expect many As, many Bs, some Cs, and (possibly) a few <C-

• Great job in the course!