Automating the Design Recipe

Hazel Levine   Sam Tobin-Hochstadt

Indiana University

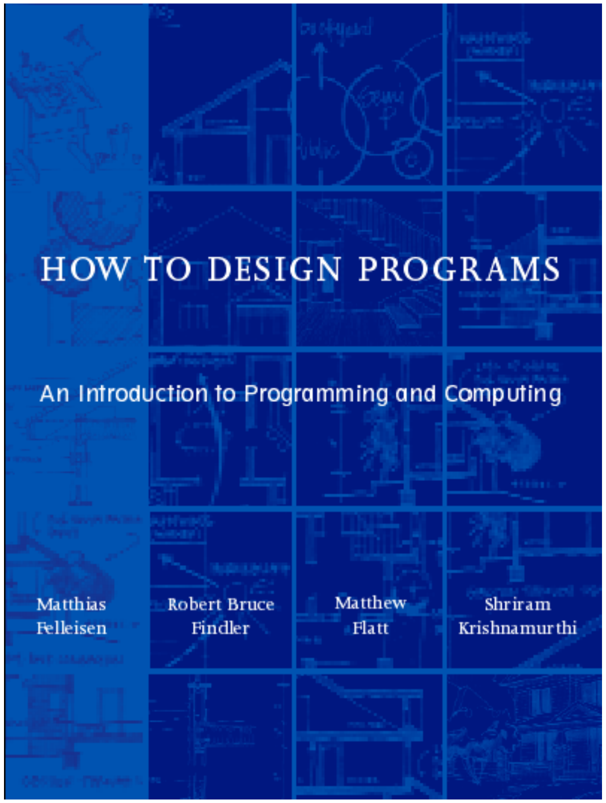
Any advice for future students? Think back to 3 months ago when you were just starting the course. What would have helped you then? Please share your wisdom here, and we will pass it on to the next generation.

Always follow the design recipe.

1. How to Design Programs

What is HtDP?

- A curriculum involving pedagogical subsets of the Racket language
- Used at Indiana for the C211 course
- Emphasizes reasoning based on the *design recipe*: a "formula" ensuring signature and test driven design



The template

We have a correspondence between the shape of a data definition, and the shape of structural decomposition of that data definition:

```
; A Date is a (make-date Number String Number)
(define-struct date (year month day))
```

```
(define (process-date d)
  (... (date-year d) ...
       (date-month d) ...
       (date-day d) ...))
```

The template

We have a correspondence between the shape of a data definition, and the shape of structural decomposition of that data definition:

```
; A TrafficLight is one of:
; - "red"
; - "yellow"
; - "green"
```

```
(define (process-trafficlight tl)
  (cond [(string=? tl "red") ...]
        [(string=? tl "yellow") ...]
        [(string=? tl "green") ...]))
```

The template

We have a correspondence between the shape of a data definition, and the shape of structural decomposition of that data definition:

```
; A TreeOfDates is one of:
; - (make-leaf)
; - (make-node TreeOfDates Date TreeOfDates)
(define-struct leaf ())
(define-struct node (left date right))
```

```
(define (process-treeofdates tod)
  (cond [(leaf? tod) ...]
        [(node? tod)
         (... (process-treeofdates (tod-left tod))
              (process-date (tod-date tod))
              (process-treeofdates (tod-right tod)))]))
```

The design recipe

1. Data definitions

```
; A ListOfNumbers is one of:
; - empty
; - (cons Number ListOfNumbers)
```

The design recipe

1. Data definitions
2. Signature, purpose

```
; A ListOfNumbers is one of:
; - empty
; - (cons Number ListOfNumbers)

; multiply-by : ListOfNumbers Number -> ListOfNumbers
; multiplies every element in the list by n
```

The design recipe

1. Data definitions
2. Signature, purpose
3. Unit tests/examples

```
; A ListOfNumbers is one of:
; - empty
; - (cons Number ListOfNumbers)

; multiply-by : ListOfNumbers Number → ListOfNumbers
; multiplies every element in the list by n
(check-expect (multiply-by empty 3) empty)
(check-expect (multiply-by (list 1 2 3) 2) (list 2 4 6))
(check-expect (multiply-by (list 7 2 1) 3) (list 21 6 3))
```

The design recipe

1. Data definitions
2. Signature, purpose
3. Unit tests/examples
4. Template

```
; A ListOfNumbers is one of:
; - empty
; - (cons Number ListOfNumbers)

; multiply-by : ListOfNumbers Number → ListOfNumbers
; multiplies every element in the list by n
(check-expect (multiply-by empty 3) empty)
(check-expect (multiply-by (list 1 2 3) 2) (list 2 4 6))
(check-expect (multiply-by (list 7 2 1) 3) (list 21 6 3))

(define (multiply-by ls n)
  (cond [(empty? ls) ...]
        [(cons? ls) (... (first ls)
                         (multiply-by (rest ls) n) ...)]))
```

The design recipe

1. Data definitions
2. Signature, purpose
3. Unit tests/examples
4. Template
5. Function definition

```
; A ListOfNumbers is one of:
; - empty
; - (cons Number ListOfNumbers)

; multiply-by : ListOfNumbers Number → ListOfNumbers
; multiplies every element in the list by n
(check-expect (multiply-by empty 3) empty)
(check-expect (multiply-by (list 1 2 3) 2) (list 2 4 6))
(check-expect (multiply-by (list 7 2 1) 3) (list 21 6 3))

(define (multiply-by ls n)
  (cond [(empty? ls) empty]
        [(cons? ls) (cons (* (first ls) n)
                          (multiply-by (rest ls) n))]))
```

The design recipe

1. Data definitions
2. Signature, purpose
3. Unit tests/examples
4. Template
5. Function definition
6. Testing

```
; A ListOfNumbers is one of:
; - empty
; - (cons Number ListOfNumbers)

; multiply-by : ListOfNumbers Number -> ListOfNumbers
; multiplies every element in the list by n
(check-expect (multiply-by empty 3) empty)
(check-expect (multiply-by (list 1 2 3) 2) (list 2 4 6))
(check-expect (multiply-by (list 7 2 1) 3) (list 21 6 3))

(define (multiply-by ls n)
  (cond [(empty? ls) empty]
        [(cons? ls) (cons (* (first ls) n)
                          (multiply-by (rest ls) n))]))
```

```
All 3 tests passed!
```

The design recipe

1. Data definitions (input)
2. Signature, purpose (input)
3. Unit tests/examples (input)
4. Template (output)
5. Function definition (???)
6. Testing (output)

```
; A ListOfNumbers is one of:
; - empty
; - (cons Number ListOfNumbers)

; multiply-by : ListOfNumbers Number → ListOfNumbers
; multiplies every element in the list by n
(check-expect (multiply-by empty 3) empty)
(check-expect (multiply-by (list 1 2 3) 2) (list 2 4 6))
(check-expect (multiply-by (list 7 2 1) 3) (list 21 6 3))

(define (multiply-by ls n)
  (cond [(empty? ls) empty]
        [(cons? ls) (cons (* (first ls) n)
                          (multiply-by (rest ls) n))]))

All 3 tests passed!
```

Steps 4-6 as a closed loop

```
; A TreeOfNumbers is one of:
; - (make-leaf)
; - (make-node TreeOfNumbers Number TreeOfNumbers)
(define-struct leaf ())
(define-struct node (left value right))

; depth : TreeOfNumbers -> Number
; computes the maximum depth of the tree
(check-expect (depth (make-leaf)) 0)
(check-expect (depth (make-node (make-leaf) 1 (make-leaf))) 1)
(check-expect (depth (make-node (make-leaf)
                                4
                                (make-node (make-leaf) 1 (make-leaf))))
              2)

(define (depth tree)
  {... : Number})
```

Steps 4-6 as a closed loop

```
; A TreeOfNumbers is one of:
; - (make-leaf)
; - (make-node TreeOfNumbers Number TreeOfNumbers)
(define-struct leaf ())
(define-struct node (left value right))

; depth : TreeOfNumbers -> Number
; computes the maximum depth of the tree
(check-expect (depth (make-leaf)) 0)
(check-expect (depth (make-node (make-leaf) 1 (make-leaf))) 1)
(check-expect (depth (make-node (make-leaf)
                                4
                                (make-node (make-leaf) 1 (make-leaf))))
              2)
```

```
(define (depth tree)
  {... : Number})
```

Steps 4-6 as a closed loop

```
; A TreeOfNumbers is one of:
; - (make-leaf)
; - (make-node TreeOfNumbers Number TreeOfNumbers)
(define-struct leaf ())
(define-struct node (left value right))

; depth : TreeOfNumbers -> Number
; computes the maximum depth of the tree
(check-expect (depth (make-leaf)) 0)
(check-expect (depth (make-node (make-leaf) 1 (make-leaf))) 1)
(check-expect (depth (make-node (make-leaf)
                                4
                                (make-node (make-leaf) 1 (make-leaf))))
              2)
```

```
(define (depth tree)
  0)
```

Two tests failed!

Steps 4-6 as a closed loop

```
; A TreeOfNumbers is one of:
; - (make-leaf)
; - (make-node TreeOfNumbers Number TreeOfNumbers)
(define-struct leaf ())
(define-struct node (left value right))

; depth : TreeOfNumbers -> Number
; computes the maximum depth of the tree
(check-expect (depth (make-leaf)) 0)
(check-expect (depth (make-node (make-leaf) 1 (make-leaf))) 1)
(check-expect (depth (make-node (make-leaf)
                                4
                                (make-node (make-leaf) 1 (make-leaf))))
              2)
```

```
(define (depth tree)
  (cond [(leaf? tree) {... : Number}]
        [(node? tree) {... : Number}]))
```

Steps 4-6 as a closed loop

```
; A TreeOfNumbers is one of:
; - (make-leaf)
; - (make-node TreeOfNumbers Number TreeOfNumbers)
(define-struct leaf ())
(define-struct node (left value right))

; depth : TreeOfNumbers -> Number
; computes the maximum depth of the tree
(check-expect (depth (make-leaf)) 0)
(check-expect (depth (make-node (make-leaf) 1 (make-leaf))) 1)
(check-expect (depth (make-node (make-leaf)
                                4
                                (make-node (make-leaf) 1 (make-leaf))))
              2)
```

```
(define (depth tree)
  (cond [(leaf? tree) {... : Number}]
        [(node? tree) (max {... : Number}
                           {... : Number})]))
```

Steps 4-6 as a closed loop

```
; A TreeOfNumbers is one of:
; - (make-leaf)
; - (make-node TreeOfNumbers Number TreeOfNumbers)
(define-struct leaf ())
(define-struct node (left value right))

; depth : TreeOfNumbers -> Number
; computes the maximum depth of the tree
(check-expect (depth (make-leaf)) 0)
(check-expect (depth (make-node (make-leaf) 1 (make-leaf))) 1)
(check-expect (depth (make-node (make-leaf)
                                4
                                (make-node (make-leaf) 1 (make-leaf))))
              2)
```

```
(define (depth tree)
  (cond [(leaf? tree) 0]
        [(node? tree) (max (depth (node-left tree))
                           (depth (node-right tree)))]))
```

```
All 3 tests passed!
```

2. Implementation

The Myth program synthesizer

Peter-Michael Osera and Steve Zdancewic.
Type-and-Example-Directed Program Synthesis.
(PLDI ’15)

- Describes a program synthesizer for a much simpler, ML-like language
- Extremely performant, but has some constraints that make it not what we want

How Myth gets really close

```
(* Type signature for natural numbers and lists *)
type nat =              type list =
  | O                     | Nil
  | S of nat              | Cons of nat * list
(* Goal type refined by input/output examples *)
let stutter : list -> list |>
  { []    => []
  | [0]   => [0;0]
  | [1;0] => [1;1;0;0]
  } = ?
```

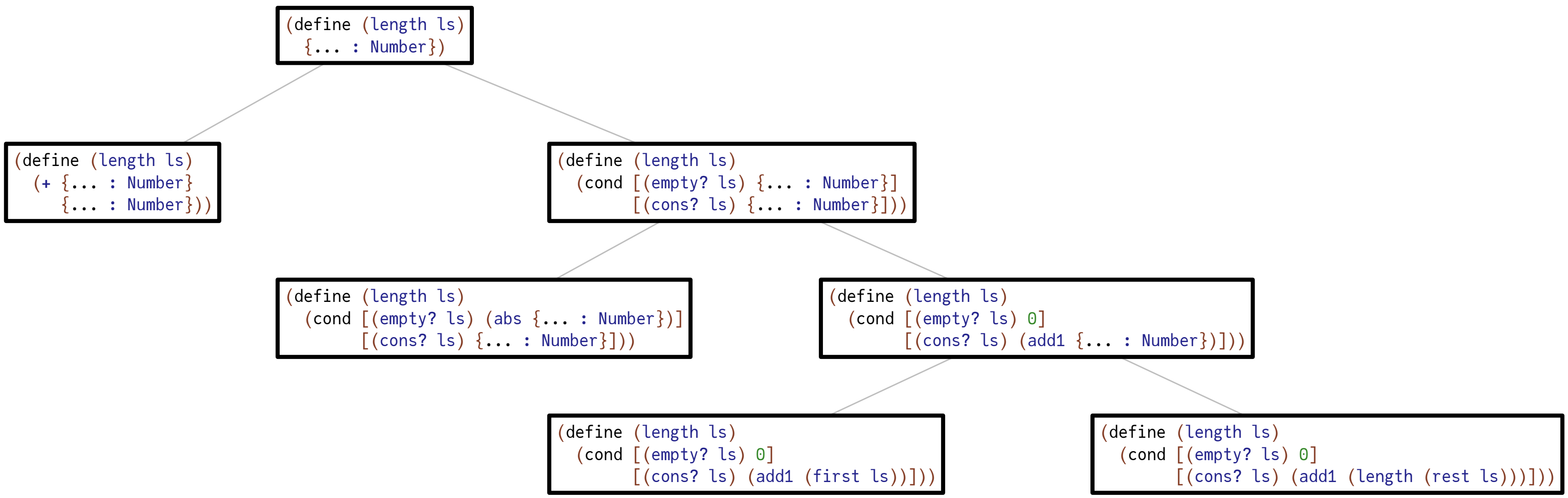
```
(* Output: synthesized implementation of stutter *)
let stutter : list -> list =
let rec f1 (l1:list) : list =
  match l1 with
   | Nil -> l1
   | Cons(n1, l2) -> Cons(n1, Cons(n1, f1 l2))
in f1
```

Figure 1. An example program synthesis problem (above) and the resulting synthesized implementation (below).

What's different in Beginning Student?

- Non-inductive data
- Constants
- Signatures that don’t behave like types
- A large collection of primitives

The core algorithm



Our refinements

- `introduce-lambda`: for any `X -> Y` signature
- `guess-var`: given a matching var in the environment, plug the hole with it
- `guess-const`: guess a known constant or one from a unit test
- `guess-app`: given an `X -> Y`, and an `X` hole, make a `Y` hole
- `guess-template`: given an inductive `X` in the environment, try its template and extend the environment with any recursion

3. Future directions

What's this good for?

```
(warn*
 [(!defined land-rocket) "the function named \"land-rocket\" is not defined"]
 [(!test (image? (land-rocket  50))) "the function named \"land-rocket\" is not defined as a function that takes a time and returns an image"]
 [(!test (image? (land-rocket 150))) "the function named \"land-rocket\" is not defined as a function that takes a time and returns an image"]
 [(!test (image? (land-rocket 201))) "the function named \"land-rocket\" is not defined as a function that takes a time and returns an image"]
 [(!test (image? (land-rocket 202))) "the function named \"land-rocket\" is not defined as a function that takes a time and returns an image"]
 [(!test (not (image=? (land-rocket 50) (land-rocket 150)))) (err-msg-incorrect-def 'land-rocket)]
 [(!test (image=? (land-rocket 201) (land-rocket 202))) (err-msg-incorrect-def 'land-rocket)]
```

- Making autograding scripts less manual and tedious to make
- Determining inconsistent sets of unit tests
- Checking that a function follows a certain "shape"

Making it faster

- Using Dijkstra's algorithm instead of breadth-first search
- Removing unit tests from consideration inside conditionals
- Using refinement trees from Myth

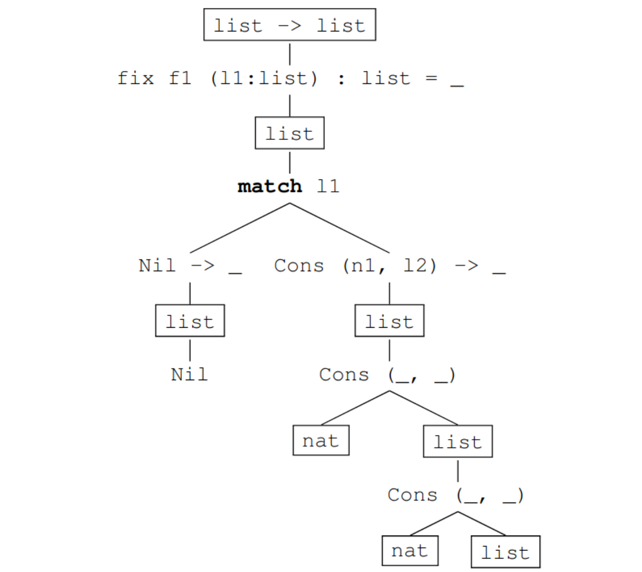


Figure 8. Example refinement tree for `stutter`.

Making it do more

- We can’t easily apply this algorithm for arithmetic/string functions
- Calling another synthesizer, like Rosette, would work
- This can be implemented as another refinement

```
; days->year : Number -> Year
; takes days since 1 Jan 0 and returns the year
; given: 364                                        expect: 0
; given: 365                                        expect: 1
; given: 736305                                     expect: 2017
; given: (year-month-day->days 1999 "December" 31) expect: 1999
```

Making it user-friendly

- We need some way to call this
- We provide a basic API that parses comments from student files
- We have a DrRacket Quickscript that runs the synthesizer

```racket
; A TreeOfNumbers is one of:
; - (make-leaf Number)
; - (make-node TreeOfNumbers TreeOfNumbers)
(define-struct leaf [n])
(define-struct node [left right])

; prod-tree : TreeOfNumbers -> Number
; multiplies all elements in a TreeOfNumbers
(define (prod-tree ton)
  ...)

(check-expect (prod-tree (make-leaf 3)) 3)
(check-expect (prod-tree (make-node (make-leaf 3) (make-leaf 9)))
              27)
(check-expect (prod-tree (make-node (make-node (make-leaf 3) (make-leaf 9))
                                    (make-leaf 3)))
              81)
```

```racket
; A TreeOfNumbers is one of:
; - (make-leaf Number)
; - (make-node TreeOfNumbers TreeOfNumbers)
(define-struct leaf [n])
(define-struct node [left right])

; prod-tree : TreeOfNumbers -> Number
; multiplies all elements in a TreeOfNumbers
(define (prod-tree g62301)
  (cond
   ((leaf? g62301) (leaf-n g62301))
   ((node? g62301)
    (* (prod-tree (node-right g62301)) (prod-tree (node-left g62301))))))

(check-expect (prod-tree (make-leaf 3)) 3)
(check-expect (prod-tree (make-node (make-leaf 3) (make-leaf 9)))
              27)
(check-expect (prod-tree (make-node (make-node (make-leaf 3) (make-leaf 9))
                                    (make-leaf 3)))
              81)
```

Thank you!