

Proofs about functions

Function consuming A is related to proof about A

- Q: How to prove two lists are equal?
A: Prove they are both '()' or that they are both cons cells cons-ing equal car's to equal cdr's
- Q: How to prove two *functions* equal?
A: Prove that when applied to equal arguments they produce equal results.

What is tail position?

Tail position is defined inductively:

- The body of a function is in tail position
- When `(if e1 e2 e3)` is in tail position, so are `e2` and `e3`
- When `(let (...) e)` is in tail position, so is `e`, and similar for `letrec` and `let*`.
- When `(begin e1 ... en)` is in tail position, so is `en`.

Idea: The **last thing that happens**

Tail-call optimization

Before executing a call in tail position,
abandon your stack frame

Results in asymptotic space savings

Works for any call!

Example of tail position

```
(define reverse (xs)
  (if (null? xs)
      '()
      (append (reverse (cdr xs))
               (list1 (car xs))))))
```

Example of tail position

```
(define reverse (xs)
  (if (null? xs)
      '()
      (append (reverse (cdr xs))
              (list1 (car xs))))))
```

Reversal by accumulating parameters

Moves recursive call to tail position

Contract:

$$(\text{revapp } xs \ ys) = (\text{append } (\text{reverse } xs) \ ys)$$

Laws:

$$(\text{revapp } '() \ ys) == ys$$
$$(\text{revapp } (\text{cons } z \ zs) \ ys) == \\ (\text{revapp } zs \ (\text{cons } z \ ys))$$

Reversal by accumulating parameters

```
; laws: (revapp ' ()          ys) = ys
;        (revapp (cons z zs) ys) =
;           (revapp zs (cons z ys))
```

```
(define revapp (xs ys)
  ; return (append (reverse xs) ys)
  (if (null? xs)
      ys
      (revapp (cdr xs)
               (cons (car xs) ys))))
```

```
(define reverse (xs) (revapp xs ' ()))
```

Tail position in revapp

```
(define revapp (xs zs)
  (if (null? xs)
      zs
      (revapp (cdr xs) (cons (car xs) zs))))
```


Tail position in revapp

```
(define revapp (xs zs)
  (if (null? xs)
      zs
      (revapp (cdr xs) (cons (car xs) zs))))
```

Values **xs** and **zs** go in machine registers.

Code compiles to a loop.

Are tail calls familiar?

In your past, what did you call a construct that

- 1. Transfers control to a point in the code?**
- 2. Uses no stack space?**

Design Problem: Missing Value

Provide a **witness** to existence:

`(witness p? xs) == x, where (member x xs),
provided (exists? p? xs)`

Problem: What if there exists no such x ?

Solution: A New Interface

Success and failure continuations!

Contract written using properties (not algorithmic):

```
(witness-cps p? xs succ fail) = (succ x)  
  ; where x is in xs and (p? x)
```

```
(witness-cps p? xs succ fail) = (fail)  
  ; where (not (exists? p? xs))
```

From contract to laws

```
(witness-cps p? xs succ fail) = (succ x)
  ; where x is in xs and (p? x)
```

```
(witness-cps p? xs succ fail) = (fail)
  ; where (not (exists? p? xs))
```

Where do we have forms of data?

```
(witness-cps p? ' () succ fail) = ?
```

```
(witness-cps p? (cons z zs) succ fail) = ?
  ; when (p? z)
```

```
(witness-cps p? (cons z zs) succ fail) = ?
  ; when (not (p? z))
```

Coding witness with continuations

```
(define witness-cps (p? xs succ fail)
  (if (null? xs)
      (fail)
      (let ([z (car xs)])
        (if (p? z)
            (succ z)
            (witness-cps p? (cdr xs) succ fail))))))
```

“Continuation-Passing Style”

All tail positions are continuations or recursive calls

```
(define witness-cps (p? xs succ fail)
  (if (null? xs)
      (fail)
      (let ([z (car xs)])
        (if (p? z)
            (succ z)
            (witness-cps p? (cdr xs) succ fail))))))
```

Compiles to tight code

Example Use: Instructor Lookup

```
-> (val 2016f ' ((Fisher 105) (Hescott 170) (Chow 116)))  
-> (instructor-info 'Fisher 2016f)  
(Fisher teaches 105)  
-> (instructor-info 'Chow 2016f)  
(Chow teaches 116)  
-> (instructor-info 'Souvaine 2016f)  
(Souvaine is-not-on-the-list)
```


Instructor Lookup: The Code

```
; info has form: '(Fisher 105)
; classes has form: '(info_1 ... info_n)
(define instructor-info (instructor classes)
  (let (
    [s ; success continuation ]
    [f ; failure continuation ]
    (witness-cps pred
                 classes s f))
  )
)
```

Instructor Lookup: The Code

```
; info has form: '(Fisher 105)
; classes has form: '(info_1 ... info_n)
(define instructor-info (instructor classes)
  (let (
    [s          ; success continuation
      ]
    [f          ; failure continuation
      ]
    (witness-cps (o ((curry =) instructor) car)
                  classes s f))
  ]))
```

Instructor Lookup: The Code

```
; info has form: '(Fisher 105)
; classes has form: '(info_1 ... info_n)
(define instructor-info (instructor classes)
  (let (
    [s (lambda (info) ; success continuation
          (list3 instructor 'teaches (cadr info)))]
    [f           ; failure continuation
          ])
    (witness-cps (o ((curry =) instructor) car)
                  classes s f))
```

Instructor Lookup: The Code

```
; info has form: '(Fisher 105)
; classes has form: '(info_1 ... info_n)
(define instructor-info (instructor classes)
  (let (
    [s (lambda (info) ; success continuation
          (list3 instructor 'teaches (cadr info)))]
    [f (lambda ()      ; failure continuation
          (list2 instructor 'is-not-on-the-list))])
    (witness-cps (o ((curry =) instructor) car)
                 classes s f)))
```