What is tail position?

Tail position is defined inductively:

- The body of a function is in tail position.
- When \((\text{if } e_1 e_2 e_3)\) is in tail position, so are \(e_2\) and \(e_3\).
- When \((\text{let } (...) e)\) is in tail position, so is \(e\), and similarly for \(\text{letrec}\) and \(\text{let*}\).
- When \((\text{begin } e_1 ... e_n)\) is in tail position, so is \(e_n\).

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!
(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:

\[(\text{revapp } \text{x}s \text{ } \text{y}s) = (\text{append } (\text{reverse } \text{x}s) \text{ } \text{y}s)\]

Laws:

\[(\text{revapp } \text{}() \text{ } \text{y}s) == \text{y}s\]
\[(\text{revapp } (\text{cons } \text{z} \text{ } \text{z}s) \text{ } \text{y}s) ==\]
\[\quad (\text{revapp } \text{z}s (\text{cons } \text{z} \text{ } \text{y}s))\]
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 \texttt{revapp}

\begin{verbatim}
(define revapp (xs zs)
  (if (null? xs)
      zs
      (revapp (cdr xs) (cons (car xs) zs)))))
\end{verbatim}
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:

\[(\text{witness } p? \; xs) \equiv x, \text{ where } (\text{member } x \; xs),\]

\[\text{provided } (\text{exists? } p? \; xs)\]

Problem: What if there exists no such \(x\)?
Solution: A New Interface

Success and failure continuations!

Contract written using properties (not algorithmic):

\[
\text{(witness-cps \ p? \ xs \ succ \ fail)} = (\text{succ} \ x) \\
\quad ; \text{where } x \text{ is in } xs \text{ and } (p? \ x)
\]

\[
\text{(witness-cps \ p? \ xs \ succ \ fail)} = (\text{fail}) \\
\quad ; \text{where } (\text{not} \ (\text{exists?} \ p? \ xs))
\]
From contract to laws

\[(\text{witness-cps } p\? \; \text{x}s \; \text{succ} \; \text{fail}) = (\text{succ} \; x)\]
\[\quad ; \text{where } x \text{ is in } \text{x}s \text{ and } (p\? \; x)\]
\[(\text{witness-cps } p\? \; \text{x}s \; \text{succ} \; \text{fail}) = (\text{fail})\]
\[\quad ; \text{where } (\text{not} \; (\text{exists?} \; p\? \; \text{x}s))\]

Where do we have forms of data?

\[(\text{witness-cps } p\? \; '() \; \text{succ} \; \text{fail}) = ?\]
\[(\text{witness-cps } p\? \; (\text{cons} \; z \; \text{zs}) \; \text{succ} \; \text{fail}) = ?\]
\[\quad ; \text{when } (p\? \; z)\]
\[(\text{witness-cps } p\? \; (\text{cons} \; z \; \text{zs}) \; \text{succ} \; \text{fail}) = ?\]
\[\quad ; \text{when } (\text{not} \; (p\? \; z))\]
Coding \texttt{witness} with continuations

\begin{verbatim}
(define witness-cps (p? xs succ fail)
  (if (null? xs)
      (fail)
      (let ([z (car xs)])
        (if (p? z)
            (if (p? z)
                (succ z)
                (witness-cps p? (cdr xs) succ fail))))))
\end{verbatim}
“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)
            (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
        (lambda (info)
          (list3 instructor 'teaches (cadr info)))
      ]
      [f  ; failure continuation
        (lambda ()
          (list2 instructor 'is-not-on-the-list))
      ]
    )
(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)))