Generative Recursion and Tail Recursion

CS 350

Dr. Joseph Eremondi

Last updated: July 24, 2024

Broad Goals

Overview

- Objectives

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts
  - Generative recursion

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts
  - Generative recursion
  - Tail-calls

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts
  - Generative recursion
  - Tail-calls
  - Tail-call elimination

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts
  - Generative recursion
  - Tail-calls
  - Tail-call elimination
  - Folds

Generative Recursion

The Problem

- Some problems don’t obviously map to the style of recursion we’ve seen so far

The Problem

- Some problems don’t obviously map to the style of recursion we’ve seen so far
  - Or do, but not efficiently

The Problem

- Some problems don’t obviously map to the style of recursion we’ve seen so far
  - Or do, but not efficiently
- Especially problems that are about incrementally updating a value

The Problem

- Some problems don’t obviously map to the style of recursion we’ve seen so far
  - Or do, but not efficiently
- Especially problems that are about incrementally updating a value
  - No good way to talk about “the result so far”

The Problem

- Some problems don't obviously map to the style of recursion we've seen so far
  - Or do, but not efficiently
- Especially problems that are about incrementally updating a value
  - No good way to talk about "the result so far"
- e.g. What's the recursive version of:

The Problem

- Some problems don't obviously map to the style of recursion we've seen so far
  - Or do, but not efficiently
- Especially problems that are about incrementally updating a value
  - No good way to talk about "the result so far"
- e.g. What's the recursive version of:

The Problem

- Some problems don't obviously map to the style of recursion we've seen so far
  - Or do, but not efficiently
- Especially problems that are about incrementally updating a value
  - No good way to talk about "the result so far"
- e.g. What's the recursive version of:

```
int x = startVal;
for (int i = 0; i < n; i++)
{
  x = f(x);
}
```

Example: Reversing a List The Naive Way

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list
- Reversing a list with one element means

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list
- Reversing a list with one element means
  - Reversing the tail of the list

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list
- Reversing a list with one element means
  - Reversing the tail of the list
  - Putting the first element at the end of the new list

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list
- Reversing a list with one element means
  - Reversing the tail of the list
  - Putting the first element at the end of the new list
- $O(n^2)$: Each `append` has to walk through the whole list

Reverse: What we want (conceptually)

- Start with an empty list

Reverse: What we want (conceptually)

- Start with an empty list
- Pop an element off the first list, then add it to our result list so far

Reverse: What we want (conceptually)

- Start with an empty list
- Pop an element off the first list, then add it to our result list so far
- When reach the end, have a reversed list

An efficient reverse

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
  - Base case: finished, produce the list we've built so far

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
  - Base case: finished, produce the list we've built so far
  - Recursive case: keep building, but on the tail of the list

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
  - Base case: finished, produce the list we've built so far
  - Recursive case: keep building, but on the tail of the list
    - Call with head appended to `listSoFar`

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
  - Base case: finished, produce the list we've built so far
  - Recursive case: keep building, but on the tail of the list
    - Call with head appended to `listSoFar`
- Main function calls the helper with the start value

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
  - Base case: finished, produce the list we've built so far
  - Recursive case: keep building, but on the tail of the list
    - Call with head appended to `listSoFar`
- Main function calls the helper with the start value
  - Empty list `'()`

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Called the *accumulator*

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Called the *accumulator*
    - Often named `accum`

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Called the *accumulator*
    - Often named `accum`
  - Has same type as return type

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Called the *accumulator*
    - Often named `accum`
  - Has same type as return type
    - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Called the *accumulator*
    - Often named `accum`
  - Has same type as return type
    - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Called the *accumulator*
    - Often named `accum`
  - Has same type as return type
    - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive
  - Base case: return the accumulator

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Called the *accumulator*
    - Often named `accum`
  - Has same type as return type
    - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive
  - Base case: return the accumulator
  - Recursive case: call the helper recursively on the sub-value

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Called the *accumulator*
    - Often named `accum`
  - Has same type as return type
    - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive
  - Base case: return the accumulator
  - Recursive case: call the helper recursively on the sub-value
    - Pass the updated value as the new accumulator