Array Programming

CS242
Lecture 15
Review

• We’ve studied two function-based programming calculi
  • SKI combinators
  • Lambda Calculus

• In practice, lambda calculus has proven far more popular
  • The basis for functional languages
  • Used to model and understand most programming features
    • State, exceptions, continuations, ...

• But combinator programming is not just theoretical
Overview

• In practice, combinator programming is used most with collections
  • And particularly arrays

• Benefits
  • Conciseness: Bulk operations over the entire collection
    • Iteration/recursion is “baked in” to the operations
  • Performance: Leave the details of the implementation the underlying system
    • Might be very different for different hardware, e.g., CPUs or GPUs
An Example

• Two combinators
  • \( o \) function composition
  • \( \text{map} \) apply a function to every element of a list/array

• Semantics
  • \( \text{map } f \ [1, 2, 3] = [ f 1, f 2, f 3] \)
  • \( \text{map } (+ 1) \ [1, 2, 3] = [2, 3, 4] \)
Consider the program: 

\[(\text{map } f) \circ (\text{map } g)\]

In a conventional language:

```plaintext
array a[n], b[n], c[n]
for i = 1, a.len {
    b[i] = f(a[i])
}
for j = 1, a.len {
    c[j] = g(b[j])
}
```
Consider the program:

\[(\text{map } f) \circ (\text{map } g)\]

Much more concise!

Why: Conventional version uses general control structures. Combinator version uses a higher-order function (\text{map}) that captures exactly the specific iteration pattern needed.

In a conventional language

array a[n], b[n], c[n]
for i = 1, a.len {
    b[i] = f(a[i])
}
for j = 1, a.len {
    c[j] = g(b[j])
}
Comparison, Part I

Consider the program:

$$(\text{map } f) \circ (\text{map } g)$$

Easier to optimize!

An algebraic law:
$$(\text{map } f) \circ (\text{map } g) = \text{map } (f \circ g)$$

This transformation eliminates the intermediate list/array.

Much harder to recognize when written with explicit for-loops.

In a conventional language

```plaintext
array a[n], b[n], c[n]
for i = 1, a.len {
    b[i] = f(a[i])
}
for j = 1, a.len {
    c[j] = g(b[j])
}
```
A Digression

An algebraic law:

\((\text{map } f) \circ (\text{map } g) = \text{map } (f \circ g)\)

But what if we are programming in some monad?

E.g., with state or exceptions?
History (Review)

• First combinator-based programming language was APL
  • “A Programming Language”
  • Designed by Ken Iverson in the 1960’s

• Designed for expressing pipelines of operations on bulk data
  • Array programming
  • Basic data type is the multidimensional array

• The average of a vector of numbers: \[ \left\{ \frac{\sum \omega}{\#\omega} \right\} \]
APL’s Legacy

• Marketed by IBM starting in 1968
  • Eventually other companies also offered APL products

• Very influential
  • At least 50 subsequent array programming languages
  • Recent increased interest with the rising importance of array-based applications (e.g., deep learning) and GPUs

• Trivia: You can buy special APL keyboards today!
From APL to NumPy

• In practice, combinator programming is used most with collections
  • And particularly arrays

• Benefits
  • Conciseness: Bulk operations over the entire collection
    • Iteration/recursion is “baked in” to the operations
  • Performance: Leave the details of the implementation to the underlying system
    • Might be very different for different hardware, e.g., CPUs or GPUs

• The most popular of these interfaces today is NumPy
  • But note, python has imperative features
  • So programs tend to be a mix of styles, including using variables, state, etc.
A Brief NumPy Tutorial

A short overview of NumPy arrays

• Defining
• Shape
• Broadcasting
• Views
• Filters
Using NumPy

# This line will always appear in a NumPy program
import numpy as np
Defining an Array

import numpy as np

# initialize an array A of 10 elements with the integers 0..9
A = np.arange(0,10)
Example: Adding Arrays

```python
import numpy as np
A = np.arange(0,10)

# addition is pointwise if the dimensions match
np.add(A,A)
```
Reshaping

import numpy as np
A = np.arange(0,10)

# Reshaping is a general operation that changes array dimensions.
# Normally defines a view: creates an alias of the array -- does
# not make a copy.

# view the elements of A as a 2x5 array
A.reshape(2,5)

# view the elements of A as a 10x1 (column) array
A.reshape(10,1)

# Note that reshaping would be very difficult in a static type system!
Example: Outer Product

import numpy as np
A = np.arange(0,10)

# We can use a combination of reshape and broadcast to define a
# concise outer product.

np.multiply(A,A.reshape(10,1))
Broadcasting

- Broadcasting takes two arrays of possibly different dimensions and casts them to arrays of the same dimension.

- Rules for broadcast in an array operation $A \text{ op } B$
  - If one array has fewer dimensions, add dimensions of size 1 until both have the same number of dimensions.
  - For each dimension $i$
    - If $A$ and $B$ have the same size in dimension $i$, do nothing.
    - If one of $A$ and $B$ has size 1 in dimension $i$, replicate data in the dimension to the same size as the other array.
    - If $A$ and $B$ have different sizes in dimension $i$ and neither is 1, throw an error.

- Example
  - $A \times 5$
  - The 5 (a 0-D array) is promoted to a 1-D array of 5’s of the same length as $A$. 

Alex Aiken  CS 242  Lecture 15
Slicing

import numpy as np
A = np.arange(0,10)

# slicing defines views (aliases) of subsets of an array
A[3:]    # slice of 4th element to the end of the array
A[1:-1]  # slice of all but the first and last elements of the array
A.reshape(2,5)[0:2,1:3] # same slice written a different way
Example: Moving Average

```python
import numpy as np
A = np.arange(0, 10)

# cumulative sum is one of many NumPy built-in array functions
B = np.cumsum(A)

# moving average of A with a window of size 3
(B[3:] - B[:-3]) / 3.0
```
import numpy as np
A = np.arange(0,10)

# Using an array in a predicate returns an array of Boolean results
# Here broadcasting promotes 5 to a 1D array of 5’s
A > 5
A <= 5
(2 * A) == (A ** 2)
Filters

```python
import numpy as np
A = np.arange(0,10)

# Boolean arrays can be used as array indices to filter arrays
A[A > 5]                              # elements of A that are > 5
A[A <= 5]                             # elements of A that are <= 5
A[(2 * A) == (A ** 2)]               # elements x of A where 2*x == x ** 2
```
A Bigger Example: The Game of Life

• The Game of Life is played on 2D grid in time steps

• Grid cells are either live or dead

• A cell is live or dead at time $t+1$ based on its neighbors at time $t$
  • Cells at the world’s edge are always dead

• Defined by George Conway in 1969
  • An early example of cellular automata
Rules

• A live cell with < 2 neighbors dies
  • From loneliness

• A live cell with > 3 neighbors dies
  • From overcrowding

• A live cell with 2 or 3 neighbors survives

• A dead cell with 3 neighbors becomes live
The Game of Life

```python
import numpy as np
Z = np.zeros((300, 600))
Z[1:-1, 1:-1] = np.random.randint(0, 2, np.shape(Z[1:-1, 1:-1]))  # 0 is dead, 1 is live

while True:
    N = (Z[0:-2, 0:-2] + Z[0:-2, 1:-1] + Z[0:-2, 2:] +
         Z[1:-1, 0:-2] + Z[1:-1, 2:] +
    birth = (N == 3) & (Z[1:-1, 1:-1] == 0)
    survive = ((N == 2) | (N == 3)) & (Z[1:-1, 1:-1] == 1)
    Z[:, :] = 0
    Z[1:-1, 1:-1][birth | survive] = 1
```
\[
N = (Z[0:-2, 0:-2] + Z[0:-2, 1:-1] + Z[0:-2, 2:] + \\
    Z[1:-1, 0:-2] + Z[1:-1, 2:] + \\
\]

*Summing these 8 subarrays computes the number of live neighbors for each cell in the interior of the space.*
# N is a 2D array of the number of neighbors of each cell
# birth is a 2D Boolean array; a cell is true if it is has 3 neighbors and is dead
birth = (N == 3) & (Z[1:-1, 1:-1] == 0)

# survive is a 2D Boolean array; a cell is true if it is has 2 or 3 neighbors and is live
survive = ((N == 2) | (N == 3)) & (Z[1:-1, 1:-1] == 1)

# create a new generation
# the interior cells of Z are live if they are born or survive the previous time step
Z[:, :] = 0
Z[1:-1, 1:-1][birth | survive] = 1
The Game of Life

import numpy as np
Z = np.zeros((300, 600))
Z[1:-1, 1:-1] = np.random.randint(0, 2, np.shape(Z[1:-1, 1:-1]))  # 0 is dead, 1 is live

while True:
    N = (Z[0:-2, 0:-2] + Z[0:-2, 1:-1] + Z[0:-2, 2:] +
         Z[1:-1, 0:-2] + Z[1:-1, 2:] +
    birth = (N == 3) & (Z[1:-1, 1:-1] == 0)
    survive = ((N == 2) | (N == 3)) & (Z[1:-1, 1:-1] == 1)
    Z[:, :] = 0
    Z[1:-1, 1:-1][birth | survive] = 1
Summary

• Combinator calculi are important in practice for array/collection programming
  • Where thinking in terms of bulk operations with built-in iteration is useful
  • Often useful in parallel implementations
    • Because the combinators can be high-level enough that the programmer doesn’t need to be aware of parallelism at all

• Combinators are also important in program transformations
  • Much easier to design combinator-based transformation systems
  • Some compilers (Haskell’s GHC) even translate into an intermediate combinator-based form for some optimizations