Practical Combinator Languages

CS242
Lecture 3
Combinator Calculus: Discussion

• Combinator calculus has the advantage of having no variables
  • Compositional!

• All computations are local rewrite rules
  • Compute by pattern matching on the shape and contents of a tree
  • All operations are local and there are few cases
  • No need to worry about variables, scope, renaming ...

• Many proofs of properties are easier in combinator systems
  • E.g., confluence
Discussion

• Combinator calculus has the disadvantage of having no variables

• Consider the $S$ combinator:  $S \ x \ y \ z \ \rightarrow \ (x \ z) \ (y \ z)$

• Note how $z$ is “passed” to both $x$ and $y$ before the final application

• In a combinator calculus, this is the only way to pass information
  • In a language with variables, we would simply stash $z$ in a variable and use it in $x$ and $y$ as needed
  • In a combinator-based language, $z$ must be explicitly passed down to all parts of the computation that need it
Discussion

• Thus, what can be done in one step with a variable requires many steps (in general) in a pure combinator system

• Why does this matter?
  • Combinator calculus is not a direct match to the way we build machines
    • Our machines have memory locations and can store things in them
    • Languages with variables take advantage of this fact
Discussion

• Another advantage of combinators is working at the function level
  • Avoid reasoning about individual data accesses

• A natural fit for parallel and distributed bulk operations on data
  • Map a function over all elements of a dataset
  • Reduce a dataset to a single value using an associative operator
  • Transpose a matrix
  • Convolve an image
  • ...

• Note that in parallel/distributed operations, variables can be a problem ...
History

•SKI calculus was developed by Schoenfinkel in the 1920’s
  • One of Hilbert’s students

•Rediscovered by Curry in the 1930’s

•The properties of SKI were known before any computers were built ...
History

• First combinator-based programming language was APL
  • Designed by Ken Iverson in the 1960’s

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

• Trivia: Special APL keyboards accommodated the many 1 character combinators
  • APL programs can be unreadable strings of Greek letters

• Highly influential
  • On functional programming (several languages), successors to APL, MatLab, MapReduce
An Example APL Program

\{ (+\omega) ÷ \neq \omega \}
Another Example

• Blend two images I2 and I2 using a weight w between 0 and 1

• Note implicit promotion of scalar operations to entire data structures

\[(I1 \times w) + (I2 \times (1 \times w))\]
FP

• John Backus’s Turing Award lecture brought new attention to combinator languages

• Backus developed FORTRAN, the first successful high-level language

• But late in his career he advocated for functional programming
  • Encouraging thinking at the function level
  • And getting away from a word-at-a-time model of computation

• Proposed FP, a combinator-based functional language
  • And emphasized an “algebra of programs”
FP’s Algebra

• FP’s basic data structure is the vector (or list)

• map f [x, y, z] = [f x, f y, f z]

• Combinators enable simple and powerful program transformations:

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

• Think about how you could describe this transformation in C!
Primitive Recursion

• Note that none of the examples use explicit looping
  • Apply a function to every element of an image
  • Combine two images pointwise
  • Reduce the values of a list to a scalar using a binary function

• These are all examples of primitive recursion
  • The extent of the “looping” is defined by the arguments themselves
  • Intuitively, the bound of the iteration is known when the iteration starts
  • Primitive recursion is easier to understand and analyze automatically
    • Termination is guaranteed
  • Primitive recursion can be used in any language, but is particularly encouraged by combinator-based programming
Primitive vs. General Recursion

• The difference between primitive and general recursion is the difference between a for loop and a while loop

```java
for i = 1, n
```

```java
while not_empty(queue)
    item = head(queue)
    if process(item) then
        item2 = new Item
        insert(item2, queue)
```
MapReduce

• A very simple and very popular combinator language

• Introduced in 2003 for ``big data’’ processing
  • Developed at Google

• The problem
  • How to process data center-scale data?
  • The data does not even fit in one machine
    • Distribution/parallelism is a must
  • But most programmers don’t know how to write parallel/distributed code
The Programming Model

• Conceptually, every program has the form
  \[(\text{map } (\text{reduce } g)) \circ \text{group\_by\_key} \circ (\text{map } f)\]

• The programmer writes f and g
  • Map, reduce, and group\_by\_key are built in

• In the map phase
  • The map function $f$ is applied to every data “chunk”
  • Output is a set of \(<key,value>\) pairs

• In the reduce phase
  • The reduce function $g$ is applied once to all values with the same key
Picture
Example: Longest String

Compute the longest string in a list of strings:

map function
\[ f(str) = < 0, \text{len}(str) > \]

reduce function
\[ g(v1,v2) = \max(v1,v2) \]
Example: Word Count

• Count the frequency of each word in a corpus

• map function
  \[ f(\text{word}) = \langle \text{word}, 1 \rangle \]

• reduce function
  \[ g(\text{count1}, \text{count1}) = \text{count1} + \text{count2} \]
Example: Distributed Grep

• Search for all lines containing a particular substring s
  • Input to the map function is a file

• map function
  \[ f(\text{file}) = <0, \text{grep} \ s \ \text{file}> \]

• reduce function
  \[ g(\text{lines1}, \text{lines2}) = \text{concat} (\text{lines1}, \text{lines2}) \]
Example: Reverse Edges in a Web Graph

• Input to map is a list of URLs and their contents
• map function
  \( f(\text{sourceurl}, \text{filecontents}) = <<\text{url1}, \text{sourceurl}>>, \ldots, <<\text{urln}, \text{sourceurl}>> >\)
  where \( \text{urli} \) is a URL in \( \text{filecontents} \)
• reduce function
  \( g(\text{urls1}, \text{urls2}) = \text{append}(\text{urls1}, \text{urls2}) \)

Note: Building an inverted index that maps words to the files they appear in is the same problem
Multiple Phases

• Most Mapreduce applications consist of multiple Mapreduce stages

• For example
  • Stage 1: Compute the frequency of every word in a set of files
  • Stage 2: Keep only words that occur frequently (above a threshold)
  • Stage 3: Build an inverted index of frequent words and the files they appear in
MapReduce Advantages

• Automatic parallelization and distribution

• I/O scheduling

• Fault tolerance
  • a big deal when running on thousands of nodes

• Monitoring and feedback
Implementation
Failures

• Several different things can fail ...

• A map worker fails
  • Reschedule the job, notify the reduce workers that the map worker has failed and where the new results can be found

• A reduce worker fails
  • Reschedule the reduce jobs of that worker

• The master fails
  • Abort with an error
Why MapReduce?

• Why use MapReduce for these problems?

• Why not write in Python, C++, Java?
  • Pick your favourite Turing language here ...
Observation

• Turing languages aren’t going away
  • Because Turing hardware isn’t going away
  • But increasingly the traditional CPU is just one part of a much more complex system

• At larger scales, machines are fundamentally parallel
  • Built out of many sequential (or even parallel) components
  • This is where functional languages have some advantages over the traditional approach
Summary

- Combinator calculi are among the simplest formal computation systems

- Also important in practice for large scale distributed/parallel programming
  - Where thinking in terms of bulk operations is beneficial

- Not used as a model for sequential computation
  - Where we often want to take advantage of temporary storage/variables

- 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