ECE251: Programming Languages & Translators Fall 2010

Lecture 27 — November 23, 2010

Patrick Lam *version 1*

A one-line recap of higher-order functions: pass functions to functions, or get functions back from functions. Higher-order functions can state what we want to do to a data structure.

Note that we can write many higher-order function idioms using anonymous classes; e.g. think about how to write `List.map` in Java.

Here is another resource for information about functional programming and MapReduce:

`http://www.fincher.org/tips/General/SoftwareEngineering/FunctionalProgramming.shtml`

MapReduce

As promised, I'll briefly discuss MapReduce, which Google popularized. Some of this information is from the Hadoop MapReduce tutorial:

`http://hadoop.apache.org/mapreduce/docs/current/mapred_tutorial.html`

If you want to know more about programming for performance, take ECE459. The key idea behind MapReduce is to enable parallelization on huge datasets by distributing the data over a huge collection of commodity PCs. The input is a (large) set of (key, value) pairs, and the output is also a set of (key, value) pairs, with both keys and values possibly of different types.

The two key boxes are the `map` box and the `reduce` box. There is also an optional `combine` box between `map` and `reduce`, which transforms the `map` output in-place.

Map. The massively parallel step is taking the input (k, v) pairs and producing any number of new (k_2, v_2) pairs from each input. Each pair is mapped independently, so it's possible to run the billion input pairs on a million computers quite quickly.

The tutorial describes a word counting application; the input to the map is a list of lines. This mapper ignores the key and uses the value as the data. Its output maps words (as keys) to their counts (as values) in each line.

Reduce. The reduce phase is less parallelizable. First, MapReduce groups together all identical keys k from `map`, creating a list $\ell_k(v)$ of the values associated with each k. It then applies `reduce` to each of these $(k_2, \ell_k(v))$ pairs, obtaining values of a third type v_3 from the `reduce` calls.

The MapReduce framework then concatenates the v_3 values it obtains from all of the `reduce` calls.

In the tutorial's example, the `reduce` phase adds the different word counts for each word from the different lines. v_3 is a pair $\langle \text{word}, \text{count} \rangle$.

How does this compare to the map and reduce we saw last week?

Logic Programming

Chapter 11 of the textbook contains more information on logic programming, although it is still quite superficial. Consult a good logic programming or Prolog book for more information.

Typical Exam Question: I give you a logic program, tell me what the result will be. Expect 0.5 questions on the final exam on this topic.

Main Idea. Here are three steps involved in using a logic programming language:

- Programmer states a collection of *axioms*.
- User states a *goal*.
- Language implementation tells user about *how to* to reach that goal, in terms of values for variables; or that the goal is unreachable. ("No.")

General Logic Programming Concepts. We'll talk about axioms, resolution, terms, and unification.

Axioms are usually written as *Horn clauses*:

$$H \leftarrow B_1, B_2, \ldots, B_n,$$

where H is the head and the B_i are the body. When all of the B_i are true, then the axiom states that H is true as well. We read this as "H, if $B_1, B_2, \ldots,$ andB_n".

Resolution is used to derive new statements from axioms. For instance, we can derive:

$$\frac{\begin{array}{l} C \leftarrow A, B \\ D \leftarrow C \end{array}}{D \leftarrow A, B}$$

What is another example? What does this remind you of?

Terms can be constants (e.g. "Vancouver is rainy.") or predicates on atoms (`rainy(Vancouver)`) or variables (`rainy(X)`).

Unification gives values to free variables:

$$\frac{\begin{array}{l} \texttt{flowery}(X) \leftarrow \texttt{rainy}(X) \\ \texttt{rainy}(\texttt{Victoria}) \end{array}}{\texttt{flowery}(\texttt{Victoria})}$$