Paradigms of Programming

The Functional Paradigm

Example Languages: Lisp, Haskell, F#. Erlang, Scheme, OCaml, Scala

# Key Features

* Composed of Functions which take in parameters and return a value. They should have no ‘side-effects’.
* Leans towards statelessness.
* Uses recursion instead of loops.
* Functions are 1st class (can be used as arguments and returns, like closures).
* Makes use of currying.

# Description

Code which adopts the functional paradigm leans toward statelessness. The idea is that computation can be expressed as a series of function applications on immutable values. Each of these function applications must not have any side effects. As in the procedural paradigm, values are passed from one level of a program to another; however, lack of side effects makes it easy to deterministically compose larger functions from smaller ones. When state is not modified, we must turn our programs into the composition of “little transformers” which take values from one set of types and turn them to into values of another type.

It often takes quite a bit of practice to start to see design in a functional way. Many of us are quite used thinking about programs as large stateful stores populated by smaller stores (objects or data structures) with identity and state. In functional programming, identity and entities become more abstract. We see an entity as a series of values transformed over time. We cannot point at one thing in a program and say that it is an entity.

Consider the following example, a key-value list:

final ImmutableList<Person> peopleList = new ImmutableList<Person>() {{

add(new Person(1, "Fred"));

add(new Person(2, "Sally"));

}};

We can say that the variable peopleList represents the list, but in pure functional programming, it represents the value of the list at a moment in time. That value is immutable. We cannot say:

peopleList.add(new Person(3, "Joe"));

Rather, we must form a new list which consists of the previous list items plus a new element:

final ImmutableList<Person> newPeopleList = peopleList.newListAppending(new Person(3, "Joe"));

After this operation, the 'peopleList' reference is still pointing at a list containing Fred and Sally. However, newPeopleList points at a list containing Fred, Sally, and Joe.

Superficially, this appears extremely inefficient but in a good functional language implementation, pieces of these immutable data structures are shared. The runtime never has to make complete copies. One of the challenges in adopting the functional paradigm in a non-functional language is that these efficiencies are not available to you without special libraries. You may, in many cases, find yourself making a trade-off between immutability and efficiency. The important thing to remember, however, is that the key value which immutability enables is cognitive locality - the ability to look at a piece of code and have confidence that it does not have more than local effect. You can achieve that goal by other means as you may use mutability and still be within the functional paradigm.

Another structuring mechanism which is common in functional programming is 'partial function application' (also known as ‘currying’). It is a mechanism which allows you to create new functions which consist of other functions applied to a couple of their arguments. In Haskell, this can be shown with a simple example:

sum i j = i + j

The sum function does pretty much what you'd expect. You call it like this:

sum 3 4

and get 7 back.

However, we can create a function increment like this:

increment = sum 1

Since sum takes two arguments and we've bound one of them, our new function (increment) takes a single argument.

In Java, we can do partial function application in a much wordier way, through a chained call:

public int sum(int i, int j) { return i + j; }

public int increment(int value) { return sum(1, value); }

In functional programming we can often use partial function application to configure a set of functions for a particular local context.