Functional Data Structures

Functional Data Structures

Functional data structures are

- immutable,
- recursive (only way to have arbitrary size), and
- share data (else cost of copying would be prohibitive).

The simplest, most fundamental functional data structure is the singly-linked list.

Functional Lists from First Principles

A list is

- empty, or
- contains an element (head) and a pointer to a list (tail)

This is a *sum* type in the language of algebraic data types.

In code:

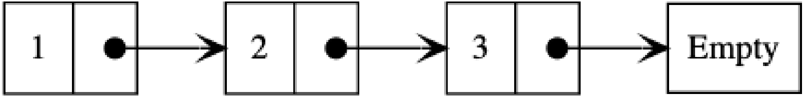
```
sealed trait FunList[+T]
case object Empty extends FunList[Nothing]
case class Cons[+T](head: T, tail: FunList[T]) extends FunList[T]
```

List `Cons`truction

Given the previous definition of a functional list, we can create a list like this:

```
val xs = Cons(1, Cons(2, Cons(3, Empty)))
```

Which creates a list that looks like this in memory:

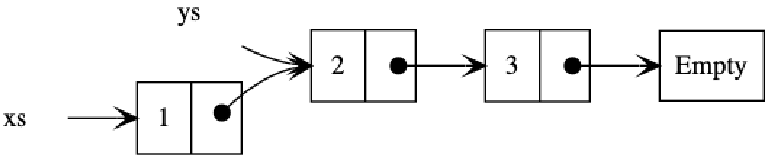


Data Sharing

When we reference a part of an existing data structure, data are shared between the two.

```
val xs = Cons(1, Cons(2, Cons(3, Empty)))
val ys = xs.tail
```

Creates:



Convenient List Construction

Of course we can make list construction more convenient:

```
object FunList {
  def apply[T](xs: T*): FunList[T] =
    if (xs.isEmpty) Empty
    else Cons(xs.head, apply(xs.tail: _*))
}
```

So instead of

```
val xs = Cons(1, Cons(2, Cons(3, Empty)))
```

we can

```
val xs = FunList(1, 2, 3)
```

Functional List Algorithms

We process sum types with pattern matching:

```
def sum(ints: FunList[Int]): Int = ints match {
  case Empty => 0
  case Cons(x,xs) => x + sum(xs)
}

def product(ds: FunList[Double]): Double = ds match {
  case Empty => 1.0
  case Cons(x, xs) => x * product(xs)
}
```

Notice that there is a case for each of the alternatives of the sum type. If we leave one out, the compiler complains because `FunList` is sealed.

Generalized List Algorithms

Look at these two list-processing functions again:

```
def sum(ints: FunList[Int]): Int = ints match {
  case Empty => 0
  case Cons(x,xs) => x + sum(xs)
}

def product(ds: FunList[Double]): Double = ds match {
  case Empty => 1.0
  case Cons(x, xs) => x * product(xs)
}
```

- Each function has a case to handle the "zero" of the list, and
- a recursive step that applies a function to successive elements of the list.

We can extract this pattern into a more general function.

Folding

Study this code:

```
def foldRight[A, B](xs: FunList[A], z: B)(f: (A, B) => B): B =
  xs match {
    case Empty => z
    case Cons(h, t) => f(h, foldRight(t, z)(f))
  }
```

We use parameters to represent

- the “zero” value, and
- the function to be applied to successive elements of the list. Notice how the return type of the function is the return type of the fold – it’s the type of the value we "reduce’ the list to.

Now we can implement `sum` and `product` in terms of fold.

```
def foldRightSum(xs: FunList[Int]) = foldRight(xs, 0)(_ + _)
```

FoldRight versus FoldLeft

Look at `foldRight` again:

```
def foldRight[A, B](xs: FunList[A], z: B)(f: (A, B) => B): B =
  xs match {
    case Empty => z
    case Cons(h, t) => f(h, foldRight(t, z)(f))
  }
```

Is `foldRight` tail recursive?

Exercise: write `foldLeft`

Is `foldLeft` tail recursive?

Standard Library `List`

Writing a functional list class is instructive but, of course, there is a standard library `List` class which you should use in your everyday programming.

Functional Trees

A tree is

- a leaf containing a data element, or
- a node with a left and right branch

In code:

```
sealed trait Tree[+T]
final case class Leaf[T](e: T) extends Tree[T]
final case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]
```

Tree Algorithms

```
def size[T](t: Tree[T]): Int =
  t match {
    case Leaf(_) => 1
    case Node(left, right) => size(left) + size(right)
  }

def treeToString[T](tree: Tree[T]): String =
  tree match {
    case Leaf(e) => e.toString
    case Node(left, right) =>
      treeToString(left) + "," + treeToString(right)
  }
```

Exercises:

- Write `reverseTree[T](tree: Tree[T]): Tree[T]`,which returns a `Tree` with same elements as `tree`, but in reverse order.

Closing Thoughts

Two options for modeling domain objects:

- Classes with polymorphic methods
- Agebraic data types (sum and product types) using pattern matching

Use ADTs when the set of classes is fixed.