**4: Parameterised Data Types**

Generic programming is a style of programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. This approach permits writing common functions that differ only in the set of types on which they operate when used, thus reducing duplication. Such software entities are known as *generics* in Kotlin. They are also known as *parametric polymorphism*; *templates*; and *parameterised types*.

**4.1 Parameterised types**

*Parametric polymorphism* makes Kotlin yet more expressive while still maintaining its static type safety. Using parametric polymorphism a function or class can be expressed generically so that it can handle values identically without depending on their type. Such functions and classes are called *generic functions* and *generic classes* respectively. Generic types are the subject of this section.

A *monomorphic* type system is one where functions can only be applied to one type. For example, the function **max** can only be called with **Int** actual parameters:

fun max(a: Int, b: Int): Int = if(a > b) a else b

Such a restricted type system has one immediate consequence: should we wish to define a **max** function for **Double**s, then we would have to introduce an identical definition except for the parameter and return types.

Parametric polymorphism seeks to overcome the inflexibility of monomorphism and still retain the advantages of static type checking. Parametric polymorphism greatly increases our code reuse.

Consider the simple class **Pair** introduced in Example 01a. The generic *type parameters* **A** and **B** are used in the declaration for the properties **first** and **second**. The scope of the identifiers **A** and **B** is the entire class declaration. In function **main** three instances of **Pair** are introduced each with a different combination of types. The function **swap** is also generic used to switch the order of the values in a **Pair**.

**Example 01a**: *Generic Pair class*

package example01a

import kotlin.test.\*

data class Pair<A, B>(val first: A, val second: B)

fun <A, B> swap(pr: Pair<A, B>): Pair<B, A> =

Pair(pr.second, pr.first)

fun main(args: Array<String>) {

val numbers: Pair<Int, Int> = Pair(12, 34)

val names: Pair<String, String> = Pair("Ken", "Barclay")

val mixed: Pair<String, Int> = Pair("Ken", 25)

assertEquals(Pair(34, 12), swap(numbers))

assertEquals(Pair("Barclay", "Ken"), swap(names))

assertEquals(Pair(25, "Ken"), swap(mixed))

}

The set of all possible types that can be substituted for a given type parameter may be restricted by *generic type constraints*. Probably the most common is an *upper bound* constraint. In Example 01b the type specified after the colon is the upper bound: only subtypes of **Comparable<A>** may be substituted for **A**. The commented assert would elicit the error message that **Pair** is not a subclass of **Comparable**.

**Example 01b**: *Type constraint*

package example01b

import kotlin.test.\*

class Pair<A, B>(val first: A, val second: B)

fun <A: Comparable<A>> maximum(a: A, b: A, c: A): A {

fun maximum(a: A, b: A): A =

if (a.compareTo(b) > 0) a else b

return maximum(a, maximum(b, c))

} // maximum

fun main(args: Array<String>) {

assertEquals(56, maximum(12, 34, 56))

assertEquals("Ken", maximum("Ken", "John", "Jessie"))

//assertEquals(Pair(5, 6), maximum(Pair(1, 2), Pair(3, 4), Pair(5, 6))) // compile error

}

**4.1.1 Parameterised sum types**

The product and sum types let us represent many different kinds of structured information. The sum types can be generalised through parameterisation. For example, a sophisticated dictionary of key/value pairs uses a number of **Node** types in its implementation. A simplification of the nodes is given by the sealed class declaration:

sealed class Node<K, V> {

class EmptyNode<K, V> : Node<K, V>()

class LeafNode<K, V>(val key: K, val value: V) : Node<K, V>()

}

The nodes are parameterised by the key type **K** and the value type **V**. Here the two examples are an **EmptyNode** with no data and a **LeafNode** with a key and value. Other node types would be added to this representation.

Example 02 presents an implementation of this illustration. Note the implementation of the **mapWithKey** function which transforms a Node<K, V> into a Node<K, W> given the transformation function **f**. Function **equals** compares two nodes for equality by extending the pattern matching across both node parameters.

**Example 02**: *Parameterised sum type*

package example02

import kotlin.test.\*

sealed class Node<K, V> {

class EmptyNode<K, V> : Node<K, V>()

class LeafNode<K, V>(val key: K, val value: V) : Node<K, V>()

}

fun <K, V> isEmpty(node: Node<K, V>): Boolean =

when (node) {

is Node.EmptyNode -> true

is Node.LeafNode -> false

}

fun <K, V> numberOfElements(node: Node<K, V>): Int =

when (node) {

is Node.EmptyNode -> 0

is Node.LeafNode -> 1

}

fun <K, V, W> mapWithKey(node: Node<K, V>, f: (K) -> (V) -> W): Node<K, W> =

when (node) {

is Node.EmptyNode -> Node.EmptyNode()

is Node.LeafNode -> Node.LeafNode(node.key, f(node.key)(node.value))

}

fun <K, V> equals(node1: Node<K, V>, node2: Node<K, V>): Boolean =

when (node1) {

is Node.EmptyNode -> when (node2) {

is Node.EmptyNode -> true

is Node.LeafNode -> false

}

is Node.LeafNode -> when (node2) {

is Node.EmptyNode -> false

is Node.LeafNode -> {

val key1: K = node1.key

val key2: K = node2.key

val value1: V = node1.value

val value2: V = node2.value

(key1 == key2) && (value1 == value2)

}

}

}

fun main(args: Array<String>) {

val empty: Node.EmptyNode<String, Int> = Node.EmptyNode()

val leaf: Node.LeafNode<String, Int> = Node.LeafNode("Ken", 25)

assertEquals(true, isEmpty(empty))

assertEquals(false, isEmpty(leaf))

assertEquals(0, numberOfElements(empty))

assertEquals(1, numberOfElements(leaf))

assertTrue(equals(Node.EmptyNode(), mapWithKey(empty){k: String -> {v: Int -> (k.length + v)}}))

assertTrue(equals(Node.LeafNode("Ken", 28), mapWithKey(leaf){k: String -> {v: Int -> (k.length + v)}}))

}

**4.2 The Option data type**

If you have programmed with Java it is very likely that you have experienced a **NullPointerException**. Usually this occurs because some function returns **null** when you were not expecting it. The **null** value is also used in Java to represent an absent value. For example, the function to look up a value for a given key in a map will return **null** if the key is absent.

Some languages, including Kotlin, treat **null** values in a special way to work safely with them. For instance Kotlin has the null safe operator for accessing properties, so that foo?.bar?.baz will not throw an exception if either **foo** or its **bar** property is **null**, instead directly returning **null**.

Another approach to this problem is to provide a type for representing optional values. The basic class declaration for **Option** is:

sealed class Option<A> {

object None : Option<Nothing>()

class Some<A>(val value: A) : Option<A>()

}

where the parameterised type is introduced in the brackets as shown.

Values that may be present or absent are supported by the **Option<A>** sealed class. **Option<A>** is a container for an optional value of type **A**. If the value of type **A** is present, then the **Option<A>** is an instance of **Some<A>** containing the value of type **A**. If the value is absent, then the **Option<A>** is the instance **None**.

The **Option** sealed class is outlined in Example 03a. The subtypes are the object **None** and the **Some** class which wraps the value. Annotating the type parameter **A** with the **out** keyword is discussed later.

**Example 03a**: *The Option class*

package example03a

import kotlin.test.\*

sealed class Option<out A> {

object None : Option<Nothing>()

class Some<out A>(val value: A) : Option<A>()

} // Option

data class Person(val firstName: String, val middleName: Option<String>, val lastName: String) {

constructor(firstName: String, middleName: String, lastName: String) : this(firstName, Option.Some(middleName), lastName)

constructor(firstName: String, lastName: String) : this(firstName, Option.None, lastName)

} // Person

fun fullName(person: Person): String {

return when(val middle = person.middleName) {

is Option.None -> "${person.firstName} ${person.lastName}"

is Option.Some -> "${person.firstName} ${middle.value} ${person.lastName}"

}

} // fullName

fun main(args: Array<String>) {

val kb: Person = Person("Ken", "Andrew", "Barclay")

val js: Person = Person("John", "Savage")

assertEquals("Ken Andrew Barclay", fullName(kb))

assertEquals("John Savage", fullName(js))

}

**4.2.1 Parameterised member functions**

Kotlin member functions can also be declared as generic, parameterised on one or more type parameters. This is similar to declaring a generic class type, but the type parameter's scope is limited to the function where it is declared. All Kotlin functions may be defined generically. This includes class member functions, package level functions as well as extension functions.

Example 03b includes the member function **map** which applies a transformer function to the content of an option type. The function signature includes the type parameter **B** to describe the value wrapped in the resulting **Option**.

**Example 03b**: *Generic member function*

package example03b

import kotlin.test.\*

sealed class Option<out A> {

val isEmpty: Boolean =

when (this) {

is None -> true

is Some -> false

}

fun <B> map(f: (A) -> B): Option<B> =

when (val op = this) {

is None -> None

is Some -> Some(f(op.value))

}

object None : Option<Nothing>()

class Some<out A>(val value: A) : Option<A>()

} // Option

// extension function

fun <A> Option<A>.getOrElse(defaultValue: A): A =

when (val op = this) {

is Option.None -> defaultValue

is Option.Some -> op.value

}

fun main(args: Array<String>) {

assertEquals(true, Option.None.map{str: String -> str.length}.isEmpty)

assertEquals(10, Option.Some(5).map{n: Int -> 2 \* n}.getOrElse(7))

}

A fully developed **Option** class is provided by the custom Dogs library. Its complete signature is:

sealed class Option<out A : Any>

which restricts the types of values contained by **Option** to be any non-null type. This same restriction is applied to all other types in the Dogs library. Not only does Dogs offer **Option** as an alternative to using **null**, it also forbids including **null** values in its data types. For brevity we will omit this constraint unless pertinent to the discussion.

**4.3 The List data type**

A data structure is a way of storing and organising data so that is can be used efficiently. Different kinds of data structures are suited to different applications. For example, compilers usually employ *hash tables* to manage information about program identifiers.

Ordinary data structures are *ephemeral* in the sense that making a change to the structure destroys the old version, leaving only the new one. This is the case for the **List** and **Map** types in the Kotlin standard library. These *imperative data structures* operate through in-place mutation of data ─ the data structure is *mutable*. In our everyday usage of Kotlin data structures we mutate lists, arrays and any other type to realise the algorithm that we implement.

An important property of *functional data structures* is that they always *persist* ─ updating a functional data structure does not destroy the existing version, but rather creates a new version that coexists with the old version. Persistence is achieved by copying the affected nodes of a data structure and making all changes in the copy rather than in the original. Because they are never modified directly, all nodes that are unaffected by an update can be shared between the old and new versions of the structure without worrying that a change in one version will inadvertently be visible to the other. Functional data structures are *immutable* and *persistent*.

**4.3.1 Motivation**

Functional languages exploit persistent data structures, many of them based on the seminal book by Chris Okasaki [http://www.cambridge.org/catalogue/catalogue.asp?isbn=0521663504]. Persistent data structures are being embraced by imperative programming languages. Persistent data structures should be part of every Kotlin programmer’s toolset, especially one interested in concurrency and parallelism.

Immutability is important for a number of reasons:

* immutable data makes the code more predictable
* immutable data is easier to work with
* immutable data encourages a transformational approach

With immutability there can be no side effects. If there are no side effects it is much easier to reason about the correctness of the code ─ something we have reiterated throughout the text. The code is then more predictable than its wholly imperative equivalent.

If data is immutable many common tasks become much easier: the code is easier to write and to maintain. Fewer unit tests are required and you only have to check that the function works in isolation. Concurrency is simpler since you are not worrying about update conflicts.

Immutable data promotes a different programming approach. More emphasis is given to transforming the data instead of in-place mutation of the data. A greater emphasis on transformation can lead to more elegant, more modular and more scalable solutions.

Every data structure has its own unique performance characteristics. A naïve implementation of a persistent data structure would deliver a very poor performance. For example, the Kotlin standard library extension function **drop** on **List**s delivers a new result **List** copying potentially large parts of the original.

Major studies have been done designing efficient persistent data structures. In many cases they closely match the performance of their mutable cousins.

**4.4 Immutable Lists**

The simple list is ubiquitous throughout programming and is probably the most-used data structure. The list is a collection of references to other objects. The list can grow dynamically. The list is defined generically so that it can handle values identically without depending on their type. Since the type of objects maintained by this collection is arbitrary, the elements of a list can be another list or map or etc. This way we can create useful data structures of arbitrary complexity.

The immutable list type can be defined recursively as an *algebraic data type*:

datatype List[A] = Nil

| Cons of A \* List[A]

Every list is constructed with just the two *value constructors* **Nil** and **Cons**. When a type has more than one value constructor they are referred to as *alternatives* or *sum types*. **Nil** is a synonym for the empty list while **Cons** makes a list by putting an element in front of an existing list. Every list is either Nil, if empty, or has the form Cons(x, xs), where x is the *head* of the list and xs is the *tail* of the list. The tail is another list.

Figure 4.1 is a sample list using an empty list (represented by the circle) and **Cons** cells (represented by the sub-divided rectangles). Each **Cons** cell stores a single value for its head and a reference to the remainder for its tail.

**Figure 4.1**: *Functional list*



This sample list would be constructed by creating an empty list with Nil() then cons-ing new elements on to the head with Cons(2, Nil()). The completed list is assembled with:

Cons(5, Cons(1, Cons(6, Cons(2, Nil()))))

Sum types are readily implemented in Kotlin with a class hierarchy (see Figure 4.2). The concrete object **Nil** and concrete class **Cons** implement the sealed class **List** which is covariant in it generic parameter. Most of the functionality for the type is defined by member functions in the immutable class **List**. The outline code is:

sealed class List<out A> {

object Nil : List<Nothing>() { … }

class Cons<A>(val hd: A, val tl: List<A>) : List<A>() { … }

// other behaviours

}

The immutable **List** class has the same functionality as that provided by the Kotlin extension functions on the Java **List**. Many are named the same. For example the immutable **List** class includes the member functions **contains**, **count**, **drop**, **dropWhile**, **filter**, **get**, **indexOf**, **isEmpty**, **last**, **map**, **take**, **takeWhile** and **zip** among others. Most bear the same signature as their Kotlin counterparts.

**Figure 4.2**: *Class hierarchy*

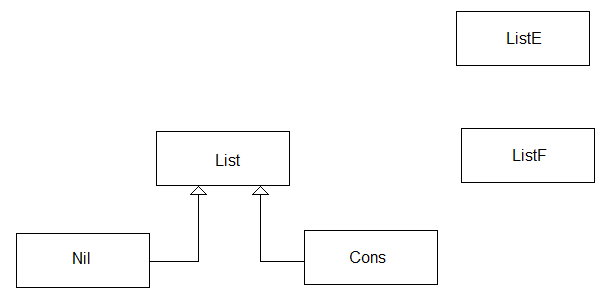


Figure 4.2 also includes the Kotlin file **ListE**. This file declares contravariant extension functions on the **List** class which would otherwise conflict with the covariant type parameter in class **List**. The file **ListF** defines a number of functions that operate on our functional list. These functions are packaged in the object declaration **ListF**. They are list-related functions that would not be a member of the **List** hierarchy. One such function is **replicate** that produces a list with repeated copies of a given value. Its declaration is:

fun <A> replicate(n: Int, a: A): List<A>

The sealed class hierarchy defines our immutable **List**. Kotlin’s pattern matching scheme over a sealed class hierarchy delivers regular and repeatable solutions. Since the data type is recursive, then recursive functions support the implementations. Two member function examples are:

fun dropWhile(predicate: (A) -> Boolean): List<A> {

tailrec

fun recDropWhile(predicate: (A) -> Boolean, ps: List<A>): List<A> {

return when(ps) {

is Nil -> Nil

is Cons -> if(!predicate(ps.head())) ps else recDropWhile(predicate, ps.tail())

}

} // recDropWhile

return recDropWhile(predicate, this)

}

fun find(predicate: (A) -> Boolean): Option<A> {

tailrec

fun recFind(predicate: (A) -> Boolean, ps: List<A>): Option<A> {

return when(ps) {

is Nil -> Option.None

is Cons -> if (predicate(ps.head())) Option.Some(ps.head()) else recFind(predicate, ps.tail())

}

} // recFind

return recFind(predicate, this)

}

The **List** type is found in many applications. Here are some application examples. In the first illustration the **lines** property of class **Order** is a **List** of **OrderLine**s.

class OrderLine …

class Order(val id: OrderId, val lines: List<OrderLine>)

class Book …

class Library(val name: Name, val address: Address, val books: List<Book>)

class Staff …

class Department(val name: Name, val staff: List<Staff>)

Example 04a is a simple illustration. It constructs a list comprising three names then confirms various properties of that list. The list construction is achieved with the complementary **ListF** namespace functions **nil** and **cons**.

**Example 04a**: *Simple list*

**package** example04a  
  
**import** com.adt.kotlin.data.immutable.list.List  
**import** com.adt.kotlin.data.immutable.list.ListF.nil  
**import** com.adt.kotlin.data.immutable.list.ListF.cons  
  
**import** kotlin.test.\*  
  
**fun** main(args: Array<String>) {  
 **val** names: List<String> = cons(**"Ken"**, cons(**"John"**, cons(**"Jessie"**, nil())))  
  
 *assertEquals*(3, names.size())  
 *assertEquals*(**"Ken"**, names.head())  
 *assertEquals*(**"John"**, names.tail().head())  
}

Example 04b determines the names of the given capital cities that have a population that exceed one million. The two-stage solution is first to filter out those cities with populations over one million, then obtaining their names. The **val** bindings for **largeCities** and **nameLargeCities** do precisely that. The **largeCities** binding references a new **List**. We demonstrate in the next example how to avoid this.

**Example 04b**: *Transformational*

**package** example04b  
  
**import** com.adt.kotlin.data.immutable.list.List  
**import** com.adt.kotlin.data.immutable.list.ListF  
  
**import** kotlin.test.\*  
  
**class** City(**val name**: String, **val population**: Int)  
  
**val** *capitals*: List<City> = ListF.of(  
 City(**"Amsterdam"**, 730000),  
 City(**"Berlin"**, 3400000),  
 City(**"Brussels"**, 450000),  
 City(**"Lisbon"**, 670000),  
 City(**"London"**, 7000000),  
 City(**"Madrid"**, 3000000),  
 City(**"Paris"**, 2200000),  
 City(**"Stockholm"**, 720000)  
)  
  
**fun** citiesOver1M(cities: List<City>): List<String> {  
 **val** largeCities: List<City> = cities.filter**{**city **->** (city.**population** > 1000000)**}  
 val** nameLargeCities: List<String> = largeCities.map**{**city **->** city.**name}  
 return** nameLargeCities  
}  
  
**fun** main(args: Array<String>) {  
 *assertEquals*(ListF.of(**"Berlin"**, **"London"**, **"Madrid"**, **"Paris"**), *citiesOver1M*(*capitals*))  
}

Example 04c repeats the previous example and determines the names of the given capital cities that have a population that exceed one million. The two top level functions **filterC** and **mapC** are curried versions of the member functions **filter** and **map**. Both return a function result that transforms a **List** into a **List**.

In the function **citiesOver1M** the **val** binding **largeCitiesC** is a concrete version of **filterC** that transforms a List<City> into a List<City>. The val binding **cityNamesC** is a concrete version of **mapC** transforming a List<City> into a List<String>. The **namesOfCitiesOver1MC** binding then composes these two by pipelining the filtering through the mapping. The pipelining is achieved by forward composing (**forwardCompose**) the functions **largeCitiesC** and **cityNamesC** so that output of the former is input into the latter.

This is sometime known as the *point-free programming* style in which the function **namesOfCitiesOver1MC** does not include any reference to its (**List**) argument, but is defined using *combinators* and function composition.

**Example 04c**: *Pipelining*

package example04c

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.fp.FunctionF.forwardCompose

import kotlin.test.\*

class City(val name: String, val population: Int)

val capitals: List<City> = ListF.of(

City("Amsterdam", 730000),

City("Berlin", 3400000),

City("Brussels", 450000),

City("Lisbon", 670000),

City("London", 7000000),

City("Madrid", 3000000),

City("Paris", 2200000),

City("Stockholm", 720000)

)

fun <A : Any>filterC(predicate: (A) -> Boolean): (List<A>) -> List<A> = {xs -> xs.filter(predicate)}

fun <A : Any, B : Any> mapC(f: (A) -> B): (List<A>) -> List<B> = {xs -> xs.map(f)}

fun citiesOver1M(cities: List<City>): List<String> {

val largeCitiesC: (List<City>) -> List<City> = filterC{city -> (city.population > 1000000)}

val cityNamesC: (List<City>) -> List<String> = mapC{city -> city.name}

val namesOfCitiesOver1MC: (List<City>) -> List<String> = forwardCompose(largeCitiesC, cityNamesC)

return namesOfCitiesOver1MC(cities)

}

fun main(args: Array<String>) {

assertEquals(ListF.of("Berlin", "London", "Madrid", "Paris"), citiesOver1M(capitals))

}

**4.4.1 Immutable lists and persistence**

Our list data structures are immutable. Member functions of the class **List** and the functions defined in **ListF** working with our list data structure do not modify the state of the structure. They can only return a new structure if they represent some change operation. In this setting the lists we create are immutable. A further consequence of this approach is that the structures are *shared*.

**Figure 4.3**: *Sharing*

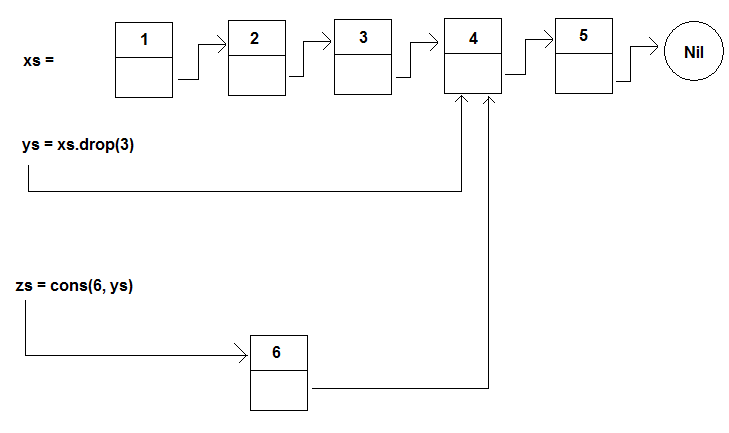


Figure 4.3 presents three lists named **xs**, **ys** and **zs**. The list **ys** is obtained by dropping the first three elements from the **xs** list sharing its final two elements. By comparison, the **drop** function from the Kotlin standard library implements this by making a copy of the sub-list. The figure also shows how the list **zs** also shares part of the original list **xs**.

In Example 05a we use KwikCheck to confirm that concatenating two arbitrary lists produces a list whose length is the sum of the lengths of the individual lists. This code fragment might represent a small part of a property based test for the properties of **List**s. In this test we might also want to ensure that **list1** is a prefix of the **concatenatedList** and that **list2** is a suffix of the **concatenatedList**.

**Example 05a**: *Concatenating lists*

package example05a

import com.adt.kotlin.dogs.data.immutable.list.\*

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.kwikcheck.generator.Gen

import com.adt.kotlin.kwikcheck.generator.GenF

import com.adt.kotlin.kwikcheck.property.Property

import com.adt.kotlin.kwikcheck.property.PropertyF.forAll

import com.adt.kotlin.kwikcheck.property.PropertyF.prop

import kotlin.test.\*

import org.junit.Test

class Example05a {

@Test fun concatenateationOperation() {

val genList1: Gen<List<Int>> = GenF.genList(GenF.genInt)

val genList2: Gen<List<Int>> = GenF.genList(GenF.genInt)

val property: Property = forAll(genList1, genList2){list1, list2 ->

val concatenatedList = list1.concatenate(list2)

prop(concatenatedList.size() == list1.size() + list2.size())

}

val checkResult = property.check()

assertTrue(checkResult.isPassed())

}

}

In Example 05b we use KwikCheck to confirm three properties when prepending an arbitrary value on to an arbitrary list, namely, the length of the new list exceeds the length of the original list by one, the head of the new list is the prepended value, and that the tail of the new list is the same as the original.

**Example 05b**: *The cons operation*

package example05b

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF.cons

import com.adt.kotlin.kwikcheck.generator.Gen

import com.adt.kotlin.kwikcheck.generator.GenF

import com.adt.kotlin.kwikcheck.property.Property

import com.adt.kotlin.kwikcheck.property.PropertyF.forAll

import com.adt.kotlin.kwikcheck.property.PropertyF.prop

import kotlin.test.\*

import org.junit.Test

class Example05b {

@Test fun consOperation() {

val genList: Gen<List<Int>> = GenF.genList(GenF.genInt)

val property: Property = forAll(genList, GenF.genInt){list, number ->

val consList = cons(number, list)

val lengthProp: Property = prop(consList.size() == 1 + list.size())

val headProp: Property = prop(consList.head() == number)

val tailProp: Property = prop(consList.tail() == list)

lengthProp and headProp and tailProp

}

val checkResult = property.check()

assertTrue(checkResult.isPassed())

}

}

**4.4.2 Processing immutable lists**

Example 06a demonstrates how easy it is to define one function in terms of those provided. Given two indices **from** and **to**, the **segment** is the list containing the elements between **from** and **to** inclusive. The functions **take** and **drop** are sufficient for this implementation.

**Example 06a**: *Simple function definitions*

package example06a

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF

import kotlin.test.\*

fun <A : Any> segment(from: Int, to: Int, xs: List<A>): List<A> = xs.drop(from).take(to - from + 1)

fun main(args: Array<String>) {

assertEquals(ListF.empty(), segment(3, 5, ListF.empty<Int>()))

assertEquals(ListF.empty(), segment(3, 5, ListF.of(1, 2, 3)))

assertEquals(ListF.of(1, 2, 3), segment(3, 5, ListF.of(1, 1, 1, 1, 2, 3, 3, 1, 1, 4, 5, 5, 5, 5)))

}

Every list is either empty represented by Nil, or is non-empty represented by Cons( ... ). In fact, every list is constructed by repeatedly applying **Cons** on to an empty list **Nil**. If we want to define a function over a list we distinguish between empty and non-empty cases. This leads to a *constructor pattern* over lists which will pattern match against either **Nil** or **Cons**.

Example 06b declares the functions **delete** and **pairwise**. Function **delete** removes the element at index position **n** from the **list**. It operates by using the **List** member function **splitAt** to split the original list into a pair of lists with the first list of length **n** and the second list the remainder. Function **pairwise** returns a list of pairs of the adjacent elements from the **list** parameter.

**Example 06b**: *Functions delete and pairwise*

package example06b

import com.adt.kotlin.dogs.data.immutable.list.\*

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF

import kotlin.test.\*

fun <A : Any> delete(n: Int, list: List<A>): List<A> {

return if (n < 0 || n >= list.size())

list

else {

val (prefix: List<A>, suffix: List<A>) = list.splitAt(n)

prefix.append(suffix.tail())

}

} // delete

fun <A : Any> pairwise(list: List<A>): List<Pair<A, A>> =

list.zip(list.tail())

fun main(args: Array<String>) {

assertEquals(ListF.of(0, 1, 3, 4, 5), delete(2, ListF.of(0, 1, 2, 3, 4, 5)))

assertEquals(ListF.of(1, 2, 3, 4, 5), delete(0, ListF.of(0, 1, 2, 3, 4, 5)))

assertEquals(ListF.of(0, 1, 2, 3, 4, 5), delete(-1, ListF.of(0, 1, 2, 3, 4, 5)))

assertEquals(ListF.of(0, 1, 2, 3, 4, 5), delete(6, ListF.of(0, 1, 2, 3, 4, 5)))

assertEquals(ListF.of(Pair(1, 2), Pair(2, 3), Pair(3, 4)), pairwise(ListF.of(1, 2, 3, 4)))

}

Example 06c declares the extension function **isDisjoint** to determine if two lists in ascending order are disjoint, i.e. their intersection is empty.

**Example 06c**: *Function isDisjoint*

package example06c

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.List.Nil

import com.adt.kotlin.dogs.data.immutable.list.List.Cons

import com.adt.kotlin.dogs.data.immutable.list.ListF

import kotlin.test.\*

fun <A : Comparable<A>> List<A>.isDisjoint(list: List<A>): Boolean {

fun recIsDisjoint(list1: List<A>, list2: List<A>): Boolean {

return when (list1) {

is Nil -> true

is Cons -> when (list2) {

is Nil -> true

is Cons -> {

val head1: A = list1.head()

val head2: A = list2.head()

if (head1 < head2)

recIsDisjoint(list1.tail(), list2)

else if (head1 > head2)

recIsDisjoint(list1, list2.tail())

else

false

}

}

}

} // recIsDisjoint

return recIsDisjoint(this, list)

} // isDisjoint

fun main(args: Array<String>) {

assertEquals(false, ListF.of(1, 2, 3).isDisjoint(ListF.of(2, 3)))

assertEquals(true, ListF.of(1, 2, 3).isDisjoint(ListF.of(4, 5)))

}

Example 06d introduces the functions **headOption** and **reverseMap**. Function **headOption** delivers the first element in the list. If the list is empty the function returns **None** otherwise it returns the head of the list wrapped in a **Some**. The implementation exploits the pattern matching of sealed classes.

The extension function **reverseMap** builds a new list by applying the function to all elements of the receiver, collecting the result in reverse order. The local function **recReverseMap** pattern matches against the empty and non-empty list. As the elements are transformed by the function **f** they are prepended on to the accumulating parameter **acc**.

**Example 06d**: *Pattern matched functions*

**package** example03d  
  
**import** com.adt.kotlin.data.immutable.list.List  
**import** com.adt.kotlin.data.immutable.list.List.Nil  
**import** com.adt.kotlin.data.immutable.list.List.Cons  
**import** com.adt.kotlin.data.immutable.list.ListF  
**import** com.adt.kotlin.data.immutable.list.ListF.cons  
  
**import** com.adt.kotlin.data.immutable.option.Option  
**import** com.adt.kotlin.data.immutable.option.Option.None  
**import** com.adt.kotlin.data.immutable.option.Option.Some  
  
**import** kotlin.test.\*  
  
**fun** <A> List<A>.headOption(): Option<A> {  
 **return when** (**this**) {  
 **is** Nil -> None  
 **is** Cons -> Some(**this**.head())  
 }  
} *// headOption***fun** <A, B> List<A>.reverseMap(f: (A) -> B): List<B> {  
 **tailrec  
 fun** recReverseMap(f: (A) -> B, xs: List<A>, acc: List<B>): List<B> {  
 **return when**(xs) {  
 **is** Nil -> acc  
 **is** Cons -> recReverseMap(f, xs.tail(), cons(f(xs.head()), acc))  
 }  
 } *// recReverseMap* **return** recReverseMap(f, **this**, ListF.empty())  
} *// reverseMap***fun** main(args: Array<String>) {  
 *assertEquals*(None, ListF.empty<Int>().*headOption*())  
 *assertEquals*(Some(**"Ken"**), ListF.of(**"Ken"**, **"John"**, **"Jessie"**, **"Irene"**).*headOption*())  
 *assertEquals*(ListF.of(16, 9, 4, 1), ListF.of(1, 2, 3, 4).*reverseMap***{**n **->** n \* n**}**)  
 *assertEquals*(ListF.empty(), ListF.empty<String>().*reverseMap***{**str **->** str.*toUpperCase*()**}**)  
}