Recursion patterns, polymorphism, and the Prelude

CIS 194 Week 3
28 January 2013

While completing HW 2, you probably spent a lot of time writing explicitly recursive functions. At this point, you might think that's what Haskell programmers spend most of their time doing. In fact, experienced Haskell programmers *hardly ever* write recursive functions!

How is this possible? The key is to notice that although recursive functions can theoretically do pretty much anything, in practice there are certain common patterns that come up over and over again. By abstracting out these patterns into library functions, programmers can leave the low-level details of actually doing recursion to these functions, and think about problems at a higher level—that's the goal of *wholemeal programming*.

Recursion patterns

Recall our simple definition of lists of `Int` values:

```
data IntList = Empty | Cons Int IntList
  deriving Show
```

What sorts of things might we want to do with an `IntList`? Here are a few common possibilities:

- Perform some operation on every element of the list
- Keep only some elements of the list, and throw others away, based on a test
- "Summarize" the elements of the list somehow (find their sum, product, maximum…).
- You can probably think of others!

Map

Let's think about the first one ("perform some operation on every element of the list"). For example, we could add one to every element in a list:

Or we could ensure that every element in a list is nonnegative by taking the absolute value:

```
absAll :: IntList -> IntList
absAll Empty       = Empty
absAll (Cons x xs) = Cons (abs x) (absAll xs)
```

Or we could square every element:

```
squareAll :: IntList -> IntList
squareAll Empty       = Empty
squareAll (Cons x xs) = Cons (x*x) (squareAll xs)
```

At this point, big flashing red lights and warning bells should be going off in your head. These three functions look way too similar. There ought to be some way to abstract out the commonality so we don't have to repeat ourselves!

There is indeed a way—can you figure it out? Which parts are the same in all three examples and which parts change?

The thing that changes, of course, is the operation we want to perform on each element of the list. We can specify this operation as a *function* of type `Int -> Int`. Here is where we begin to see how incredibly useful it is to be able to pass functions as inputs to other functions!

We can now use `mapIntList` to implement `addOneToAll`, `absAll`, and `squareAll`:

```
exampleList = Cons (-1) (Cons 2 (Cons (-6) Empty))

addOne x = x + 1
square x = x * x

mapIntList addOne exampleList
mapIntList abs    exampleList
mapIntList square exampleList
```

Filter

Another common pattern is when we want to keep only some elements of a list, and throw others away, based on a test. For example, we might want to keep only the positive numbers:

Or only the even ones:

```
keepOnlyEven :: IntList -> IntList
keepOnlyEven Empty = Empty
keepOnlyEven (Cons x xs)
  | even x    = Cons x (keepOnlyEven xs)
  | otherwise = keepOnlyEven xs
```

How can we generalize this pattern? What stays the same, and what do we need to abstract out?

Fold

The final pattern we mentioned was to "summarize" the elements of the list; this is also variously known as a "fold" or "reduce" operation. We'll come back to this next week. In the meantime, you might want to think about how to abstract out this pattern!

Polymorphism

We've now written some nice, general functions for mapping and filtering over lists of `Int`s. But we're not done generalizing! What if we wanted to filter lists of `Integer`s? or `Bool`s? Or lists of lists of trees of stacks of `String`s? We'd have to make a new data type and a new function for each of these cases. Even worse, the *code would be exactly the same*; the only thing that would be different is the *type signatures*. Can't Haskell help us out here?

Of course it can! Haskell supports *polymorphism* for both data types and functions. The word "polymorphic" comes from Greek ( ) and means "having many forms": something which is polymorphic works for multiple types.

Polymorphic data types

First, let's see how to declare a polymorphic data type.

```
data List t = E | C t (List t)
```

(We can't reuse `Empty` and `Cons` since we already used those for the constructors of `IntList`, so we'll use `E` and `C` instead.) Whereas before we had `data IntList = ...`, we now have `data List t = ...` The `t` is a *type variable* which can stand for any type. (Type variables must start with a lowercase letter, whereas types must start with uppercase.) `data List t = ...` means that the `List` type is *parameterized* by a type, in much the same way that a function can be parameterized by some input.

Given a type `t`, a `(List t)` consists of either the constructor `E`, or the constructor `C` along with a value of type `t` and another `(List t)`. Here are some examples:

```
lst1 :: List Int
lst1 = C 3 (C 5 (C 2 E))

lst2 :: List Char
lst2 = C 'x' (C 'y' (C 'z' E))

lst3 :: List Bool
lst3 = C True (C False E)
```

Polymorphic functions

Now, let's generalize `filterIntList` to work over our new polymorphic `List`s. We can just take code of `filterIntList` and replace `Empty` by `E` and `Cons` by `C`:

```
filterList _ E = E
filterList p (C x xs)
  | p x       = C x (filterList p xs)
  | otherwise = filterList p xs
```

Now, what is the type of `filterList`? Let's see what type `ghci` infers for it:

```
*Main> :t filterList
filterList :: (t -> Bool) -> List t -> List t
```

We can read this as: "for any type `t`, `filterList` takes a function from `t` to `Bool`, and a list of `t`'s, and returns a list of `t`'s."

What about generalizing `mapIntList`? What type should we give to a function `mapList` that applies a function to every element in a `List t`?

Our first idea might be to give it the type

```
mapList :: (t -> t) -> List t -> List t
```

This works, but it means that when applying `mapList`, we always get a list with the same type of elements as the list we started with. This is overly restrictive: we'd like to be able to do things like `mapList show` in order to convert, say, a list of `Int`s into a list of `String`s. Here, then, is the most general possible type for `mapList`, along with an implementation:

```
mapList :: (a -> b) -> List a -> List b
mapList _ E        = E
mapList f (C x xs) = C (f x) (mapList f xs)
```

One important thing to remember about polymorphic functions is that **the caller gets to pick the types**. When you write a polymorphic function, it must work for every possible input type. This—together with the fact that Haskell has no way to directly make make decisions based on what type something is—has some interesting implications which we'll explore later.

The Prelude

The `Prelude` is a module with a bunch of standard definitions that gets implicitly imported into every Haskell program. It's worth spending some time skimming through its documentation to familiarize oneself with the tools that are available.

Of course, polymorphic lists are defined in the `Prelude`, along with many useful polymorphic functions for working with them. For example, `filter` and `map` are the counterparts to our `filterList` and `mapList`. In fact, the `Data.List` module contains many more list functions still.

Another useful polymorphic type to know is `Maybe`, defined as

```
data Maybe a = Nothing | Just a
```

A value of type `Maybe a` either contains a value of type `a` (wrapped in the `Just` constructor), or it is `Nothing` (representing some sort of failure or error). The `Data.Maybe` module has functions for working with `Maybe` values.

Total and partial functions

Consider this polymorphic type:

```
[a] -> a
```

What functions could have such a type? The type says that given a list of things of type `a`, the function must produce some value of type `a`. For example, the Prelude function `head` has this type.

…But what happens if `head` is given an empty list as input? Let's look at the source code for `head`…

It crashes! There's nothing else it possibly could do, since it must work for *all* types. There's no way to make up an element of an arbitrary type out of thin air.

`head` is what is known as a *partial function*: there are certain inputs for which `head` will crash. Functions which have certain inputs that will make them recurse infinitely are also called partial. Functions which are well-defined on all possible inputs are known as *total functions*.

It is good Haskell practice to avoid partial functions as much as possible. Actually, avoiding partial functions is good practice in *any* programming language—but in most of them it's ridiculously annoying. Haskell tends to make it quite easy and sensible.

`head` is a mistake! It should not be in the `Prelude`. Other partial `Prelude` functions you should almost never use include `tail`, `init`, `last`, and `(!!)`. From this point on, using one of these functions on a homework assignment will lose style points!

What to do instead?

Replacing partial functions

Often partial functions like `head`, `tail`, and so on can be replaced by pattern-matching. Consider the following two definitions:

```
doStuff1 :: [Int] -> Int
doStuff1 []  = 0
doStuff1 [_] = 0
doStuff1 xs  = head xs + (head (tail xs))
```

```
doStuff2 :: [Int] -> Int
doStuff2 []         = 0
doStuff2 [_]        = 0
doStuff2 (x1:x2:_) = x1 + x2
```

These functions compute exactly the same result, and they are both total. But only the second one is *obviously* total, and it is much easier to read anyway.

Writing partial functions

What if you find yourself *writing* a partial functions? There are two approaches to take. The first is to change the output type of the function to indicate the possible failure. Recall the definition of `Maybe`:

```
data Maybe a = Nothing | Just a
```

Now, suppose we were writing `head`. We could rewrite it safely like this:

```
safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

Indeed, there is exactly such a function defined in the `safe` package.

Why is this a good idea?

1. `safeHead` will never crash.

2. The type of `safeHead` makes it obvious that it may fail for some inputs.

3. The type system ensures that users of `safeHead` must appropriately check the return value of `safeHead` to see whether they got a value or `Nothing`.

In some sense, `safeHead` is still "partial"; but we have reflected the partiality in the type system, so it is now safe. The goal is to have the types tell us as much as possible about the behavior of functions.

OK, but what if we know that we will only use `head` in situations where we are *guaranteed* to have a non-empty list? In such a situation, it is really annoying to get back a `Maybe a`, since we have to expend effort dealing with a case which we "know" cannot actually happen.

The answer is that if some condition is really *guaranteed*, then the types ought to reflect the guarantee! Then the compiler can enforce your guarantees for you. For example:

```
data NonEmptyList a = NEL a [a]

nelToList :: NonEmptyList a -> [a]
nelToList (NEL x xs) = x:xs

listToNel :: [a] -> Maybe (NonEmptyList a)
listToNel []     = Nothing
listToNel (x:xs) = Just $ NEL x xs

headNEL :: NonEmptyList a -> a
headNEL (NEL a _) = a

tailNEL :: NonEmptyList a -> [a]
tailNEL (NEL _ as) = as
```

You might think doing such things is only for chumps who are not coding supergeniuses like you. Of course, *you* would never make a mistake like passing an empty list to a function which expects only non-empty ones. Right? Well, there's definitely a chump involved, but it's not who you think.

Recursion patterns, polymorphism, and the Prelude

CIS 194 Week 3
28 January 2013

While completing HW 2, you probably spent a lot of time writing explicitly recursive functions. At this point, you might think that's what Haskell programmers spend most of their time doing. In fact, experienced Haskell programmers *hardly ever* write recursive functions!

How is this possible? The key is to notice that although recursive functions can theoretically do pretty much anything, in practice there are certain common patterns that come up over and over again. By abstracting out these patterns into library functions, programmers can leave the low-level details of actually doing recursion to these functions, and think about problems at a higher level—that's the goal of *wholemeal programming*.

Recursion patterns

Recall our simple definition of lists of `Int` values:

```
data IntList = Empty | Cons Int IntList
  deriving Show
```

What sorts of things might we want to do with an `IntList`? Here are a few common possibilities:

- Perform some operation on every element of the list
- Keep only some elements of the list, and throw others away, based on a test
- "Summarize" the elements of the list somehow (find their sum, product, maximum…).
- You can probably think of others!

Map

Let's think about the first one ("perform some operation on every element of the list"). For example, we could add one to every element in a list:

Or we could ensure that every element in a list is nonnegative by taking the absolute value:

```
absAll :: IntList -> IntList
absAll Empty       = Empty
absAll (Cons x xs) = Cons (abs x) (absAll xs)
```

Or we could square every element:

```
squareAll :: IntList -> IntList
squareAll Empty       = Empty
squareAll (Cons x xs) = Cons (x*x) (squareAll xs)
```

At this point, big flashing red lights and warning bells should be going off in your head. These three functions look way too similar. There ought to be some way to abstract out the commonality so we don't have to repeat ourselves!

There is indeed a way—can you figure it out? Which parts are the same in all three examples and which parts change?

The thing that changes, of course, is the operation we want to perform on each element of the list. We can specify this operation as a *function* of type `Int -> Int`. Here is where we begin to see how incredibly useful it is to be able to pass functions as inputs to other functions!

We can now use `mapIntList` to implement `addOneToAll`, `absAll`, and `squareAll`:

```
exampleList = Cons (-1) (Cons 2 (Cons (-6) Empty))

addOne x = x + 1
square x = x * x

mapIntList addOne exampleList
mapIntList abs    exampleList
mapIntList square exampleList
```

Filter

Another common pattern is when we want to keep only some elements of a list, and throw others away, based on a test. For example, we might want to keep only the positive numbers:

Or only the even ones:

```
keepOnlyEven :: IntList -> IntList
keepOnlyEven Empty = Empty
keepOnlyEven (Cons x xs)
  | even x    = Cons x (keepOnlyEven xs)
  | otherwise = keepOnlyEven xs
```

How can we generalize this pattern? What stays the same, and what do we need to abstract out?

Fold

The final pattern we mentioned was to "summarize" the elements of the list; this is also variously known as a "fold" or "reduce" operation. We'll come back to this next week. In the meantime, you might want to think about how to abstract out this pattern!

Polymorphism

We've now written some nice, general functions for mapping and filtering over lists of `Int`s. But we're not done generalizing! What if we wanted to filter lists of `Integer`s? or `Bool`s? Or lists of lists of trees of stacks of `String`s? We'd have to make a new data type and a new function for each of these cases. Even worse, the *code would be exactly the same*; the only thing that would be different is the *type signatures*. Can't Haskell help us out here?

Of course it can! Haskell supports *polymorphism* for both data types and functions. The word "polymorphic" comes from Greek ( ) and means "having many forms": something which is polymorphic works for multiple types.

Polymorphic data types

First, let's see how to declare a polymorphic data type.

```
data List t = E | C t (List t)
```

(We can't reuse `Empty` and `Cons` since we already used those for the constructors of `IntList`, so we'll use `E` and `C` instead.) Whereas before we had `data IntList = ...`, we now have `data List t = ...` The `t` is a *type variable* which can stand for any type. (Type variables must start with a lowercase letter, whereas types must start with uppercase.) `data List t = ...` means that the `List` type is *parameterized* by a type, in much the same way that a function can be parameterized by some input.

Given a type `t`, a `(List t)` consists of either the constructor `E`, or the constructor `C` along with a value of type `t` and another `(List t)`. Here are some examples:

```
lst1 :: List Int
lst1 = C 3 (C 5 (C 2 E))

lst2 :: List Char
lst2 = C 'x' (C 'y' (C 'z' E))

lst3 :: List Bool
lst3 = C True (C False E)
```

Polymorphic functions

Now, let's generalize `filterIntList` to work over our new polymorphic `List`s. We can just take code of `filterIntList` and replace `Empty` by `E` and `Cons` by `C`:

```
filterList _ E = E
filterList p (C x xs)
  | p x       = C x (filterList p xs)
  | otherwise = filterList p xs
```

Now, what is the type of `filterList`? Let's see what type `ghci` infers for it:

```
*Main> :t filterList
filterList :: (t -> Bool) -> List t -> List t
```

We can read this as: "for any type `t`, `filterList` takes a function from `t` to `Bool`, and a list of `t`'s, and returns a list of `t`'s."

What about generalizing `mapIntList`? What type should we give to a function `mapList` that applies a function to every element in a `List t`?

Our first idea might be to give it the type

```
mapList :: (t -> t) -> List t -> List t
```

This works, but it means that when applying `mapList`, we always get a list with the same type of elements as the list we started with. This is overly restrictive: we'd like to be able to do things like `mapList show` in order to convert, say, a list of `Int`s into a list of `String`s. Here, then, is the most general possible type for `mapList`, along with an implementation:

```
mapList :: (a -> b) -> List a -> List b
mapList _ E        = E
mapList f (C x xs) = C (f x) (mapList f xs)
```

One important thing to remember about polymorphic functions is that **the caller gets to pick the types**. When you write a polymorphic function, it must work for every possible input type. This—together with the fact that Haskell has no way to directly make make decisions based on what type something is—has some interesting implications which we'll explore later.

The Prelude

The `Prelude` is a module with a bunch of standard definitions that gets implicitly imported into every Haskell program. It's worth spending some time skimming through its documentation to familiarize oneself with the tools that are available.

Of course, polymorphic lists are defined in the `Prelude`, along with many useful polymorphic functions for working with them. For example, `filter` and `map` are the counterparts to our `filterList` and `mapList`. In fact, the `Data.List` module contains many more list functions still.

Another useful polymorphic type to know is `Maybe`, defined as

```haskell
data Maybe a = Nothing | Just a
```

A value of type `Maybe a` either contains a value of type `a` (wrapped in the `Just` constructor), or it is `Nothing` (representing some sort of failure or error). The `Data.Maybe` module has functions for working with `Maybe` values.

Total and partial functions

Consider this polymorphic type:

```haskell
[a] -> a
```

What functions could have such a type? The type says that given a list of things of type `a`, the function must produce some value of type `a`. For example, the Prelude function `head` has this type.

…But what happens if `head` is given an empty list as input? Let's look at the source code for `head`…

It crashes! There's nothing else it possibly could do, since it must work for *all* types. There's no way to make up an element of an arbitrary type out of thin air.

`head` is what is known as a *partial function*: there are certain inputs for which `head` will crash. Functions which have certain inputs that will make them recurse infinitely are also called partial. Functions which are well-defined on all possible inputs are known as *total functions*.

It is good Haskell practice to avoid partial functions as much as possible. Actually, avoiding partial functions is good practice in *any* programming language—but in most of them it's ridiculously annoying. Haskell tends to make it quite easy and sensible.

`head` is a mistake! It should not be in the `Prelude`. Other partial `Prelude` functions you should almost never use include `tail`, `init`, `last`, and `(!!)`. From this point on, using one of these functions on a homework assignment will lose style points!

What to do instead?

Replacing partial functions

Often partial functions like `head`, `tail`, and so on can be replaced by pattern-matching. Consider the following two definitions:

```haskell
doStuff1 :: [Int] -> Int
doStuff1 []  = 0
doStuff1 [_] = 0
doStuff1 xs  = head xs + (head (tail xs))
```

```
doStuff2 :: [Int] -> Int
doStuff2 []        = 0
doStuff2 [_]       = 0
doStuff2 (x1:x2:_) = x1 + x2
```

These functions compute exactly the same result, and they are both total. But only the second one is *obviously* total, and it is much easier to read anyway.

Writing partial functions

What if you find yourself *writing* a partial functions? There are two approaches to take. The first is to change the output type of the function to indicate the possible failure. Recall the definition of `Maybe`:

```
data Maybe a = Nothing | Just a
```

Now, suppose we were writing `head`. We could rewrite it safely like this:

```
safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

Indeed, there is exactly such a function defined in the `safe` package.

Why is this a good idea?

1. `safeHead` will never crash.
2. The type of `safeHead` makes it obvious that it may fail for some inputs.
3. The type system ensures that users of `safeHead` must appropriately check the return value of `safeHead` to see whether they got a value or `Nothing`.

In some sense, `safeHead` is still "partial"; but we have reflected the partiality in the type system, so it is now safe. The goal is to have the types tell us as much as possible about the behavior of functions.

OK, but what if we know that we will only use `head` in situations where we are *guaranteed* to have a non-empty list? In such a situation, it is really annoying to get back a `Maybe a`, since we have to expend effort dealing with a case which we "know" cannot actually happen.

The answer is that if some condition is really *guaranteed*, then the types ought to reflect the guarantee! Then the compiler can enforce your guarantees for you. For example:

```haskell
data NonEmptyList a = NEL a [a]

nelToList :: NonEmptyList a -> [a]
nelToList (NEL x xs) = x:xs

listToNel :: [a] -> Maybe (NonEmptyList a)
listToNel []     = Nothing
listToNel (x:xs) = Just $ NEL x xs

headNEL :: NonEmptyList a -> a
headNEL (NEL a _) = a

tailNEL :: NonEmptyList a -> [a]
tailNEL (NEL _ as) = as
```

You might think doing such things is only for chumps who are not coding super-geniuses like you. Of course, *you* would never make a mistake like passing an empty list to a function which expects only non-empty ones. Right? Well, there's definitely a chump involved, but it's not who you think.