Haskell: variables, data types, patterns, and recursion

Julian Grove

September 6, 2023

FACTS.lab, University of Rochester

Introduction

Last time

We looked at some basic data types, functions, fixity, and type inference.

This time

We'll look at some of the stuff Haskell was made for.

This time

We'll look at some of the stuff Haskell was made for.

- Anonymous functions (i.e., function literals)

This time

We'll look at some of the stuff Haskell was made for.

- Anonymous functions (i.e., function literals)
- Role-your-own data types (i.e., algebraic data types)

This time

We'll look at some of the stuff Haskell was made for.

- Anonymous functions (i.e., function literals)
- Role-your-own data types (i.e., algebraic data types)
- Recursion via pattern matching

Variables

`let` bindings

A `let` binding can be used to define a local variable anywhere you want.

where clauses

A `where` clause can be used to define a local variable inside of another definition.

Anonymous functions

Functions are first-class in Haskell, so they are treated like other data.

Anonymous functions

Functions are first-class in Haskell, so they are treated like other data.

This means that we can write function *literals*.

Anonymous functions

Functions are first-class in Haskell, so they are treated like other data.

This means that we can write function *literals*.

We do this by binding the variad to make `Fruit` an instance of the typeclass `Show`.

- Two ways to do this…

Sum types

`Fruit` is what is called a *sum type*.

Sum types

`Fruit` is what is called a *sum type*.

- It enumerates all values it can have in different branches, delimiting them with a `|`.

Sum types

`Fruit` is what is called a *sum type*.

- It enumerates all values it can have in different branches, delimiting them with a `|`.
- In each branch is what is called a *data constructor*.

Sum types

`Fruit` is what is called a *sum type*.

- It enumerates all values it can have in different branches, delimiting them with a `|`.
- In each branch is what is called a *data constructor*.
- The name of a data constructor in Haskell must begin with a capital letter.

N-ary constructors

The `Fruit` sum type is an odd special case, in that the data constructors don't carry an extra data besides their identity.

N-ary constructors

The `Fruit` sum type is an odd special case, in that the data constructors don't carry an extra data besides their identity.

Something more common might have data constructors carry additional data, e.g., one data constructor could carry a `Bool` and one could carry a `String`.

N-ary constructors

The `Fruit` sum type is an odd special case, in that the data constructors don't carry an extra data besides their identity.

Something more common might have data constructors carry additional data, e.g., one data constructor could carry a `Bool` and one could carry a `String`.

- This can allow us to write functions that take *either* a `Bool` *or* a `String` as its input, using *pattern matching*.

N-ary constructors

The `Fruit` sum type is an odd special case, in that the data constructors don't carry an extra data besides their identity.

Something more common might have data constructors carry additional data, e.g., one data constructor could carry a `Bool` and one could carry a `String`.

- This can allow us to write functions that take *either* a `Bool` *or* a `String` as its input, using *pattern matching*.

N-ary constructors

The `Fruit` sum type is an odd special case, in that the data constructors don't carry an extra data besides their identity.

Something more common might have data constructors carry additional data, e.g., one data constructor could carry a `Bool` and one could carry a `String`.

- This can allow us to write functions that take *either* a `Bool` *or* a `String` as its input, using *pattern matching*.

By the way, you might be wondering: if a data constructor can take an argument, does that mean it's a function?

N-ary constructors

The `Fruit` sum type is an odd special case, in that the data constructors don't carry an extra data besides their identity.

Something more common might have data constructors carry additional data, e.g., one data constructor could carry a `Bool` and one could carry a `String`.

- This can allow us to write functions that take *either* a `Bool` *or* a `String` as its input, using *pattern matching*.

By the way, you might be wondering: if a data constructor can take an argument, does that mean it's a function?

- The answer is "yes"!

Pattern matching: order matters

Pattern branches get checked in top-to-bottom order.

Pattern matching: order matters

Pattern branches get checked in top-to-bottom order.

- For example…

Pattern matching: order matters

Pattern branches get checked in top-to-bottom order.

- For example…
- Flipping the branches makes the definition effectively stop at the first branch, since `str` is a wildcard over all possible strings.

Case expressions

You can also use a *case expression* to do pattern matching.

Case expressions

You can also use a *case expression* to do pattern matching.

Case expressions do more than just pattern match—they also evaluate the expression between the `case` and the `of`…

As patterns

An *as pattern* (written with an `@` sign) allows you to bind an identifier to the an argument which has been deconstructed into a pattern…

Pattern guards

Pattern guards are useful when you want to further restrict the applicability of a branch of a definition to patterns that satisfy some boolean condition.

Pattern guards

Pattern guards are useful when you want to further restrict the applicability of a branch of a definition to patterns that satisfy some boolean condition.

You use a | after the relevant pattern and then state the condition…

Recursive definitions

Lists

We talked a little about lists last time.

Lists

We talked a little about lists last time.

Lists are deeply baked into Haskell, so we can't look at the source code.

Lists

We talked a little about lists last time.

Lists are deeply baked into Haskell, so we can't look at the source code.

But we can roll our own…

Haskell lists

For convenience, Haskell lets you type, e.g., `['a', 's', 'd', 'f']` for a list literal.

Haskell lists

For convenience, Haskell lets you type, e.g., `['a', 's', 'd', 'f']` for a list literal.

When you see this, you should have in mind the following:

```
('a' : ('s' : ('d' : ('f' : []))))
```

Haskell lists

For convenience, Haskell lets you type, e.g., `['a', 's', 'd', 'f']` for a list literal.

When you see this, you should have in mind the following:

```
('a' : ('s' : ('d' : ('f' : []))))
```

Everything is one of two cases; either:

Haskell lists

For convenience, Haskell lets you type, e.g., `['a', 's', 'd', 'f']` for a list literal.

When you see this, you should have in mind the following:

```
('a' : ('s' : ('d' : ('f' : []))))
```

Everything is one of two cases; either:

- any empty list

Haskell lists

For convenience, Haskell lets you type, e.g., `['a', 's', 'd', 'f']` for a list literal.

When you see this, you should have in mind the following:

```
('a' : ('s' : ('d' : ('f' : []))))
```

Everything is one of two cases; either:

- any empty list
- something cons-ed onto a list

Appending stuff

Let's define our first recursive function: append.

`[a]` to `List a`

How could we write a recursive function that maps values of type `List a` to values of type `[a]`?

map

Haskell has a built-in function `map` for mapping functions of type `a -> b` to functions from lists of `a`'s to lists of `b`'s.

```
map :: (a -> b) -> [a] -> [b]
```

How does `map` work?...

`map`

Haskell has a built-in function `map` for mapping functions of type `a -> b` to functions from lists of `a`'s to lists of `b`'s.

```
map :: (a -> b) -> [a] -> [b]
```

How does `map` work?...

- We need a branch in the definition that applies to the empty list.

`map`

Haskell has a built-in function `map` for mapping functions of type `a -> b` to functions from lists of `a`'s to lists of `b`'s.

```
map :: (a -> b) -> [a] -> [b]
```

How does `map` work?...

- We need a branch in the definition that applies to the empty list.
- We need a branch in the definition that applies to non-empty lists.

`filter`

Filter takes a predicate, i.e., a function of from `a`'s to `Bool`'s, along with a list of `a`'s, in order to give back a list of the `a`'s that satisfy the predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

How does `filter` work?...

`filter`

Filter takes a predicate, i.e., a function of from `a`'s to `Bool`'s, along with a list of `a`'s, in order to give back a list of the `a`'s that satisfy the predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

How does `filter` work?...

- We need a branch in the definition that applies to the empty list.

`filter`

Filter takes a predicate, i.e., a function of from `a`'s to `Bool`'s, along with a list of `a`'s, in order to give back a list of the `a`'s that satisfy the predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

How does `filter` work?...

- We need a branch in the definition that applies to the empty list.
- We need a branch in the definition that applies to non-empty lists.

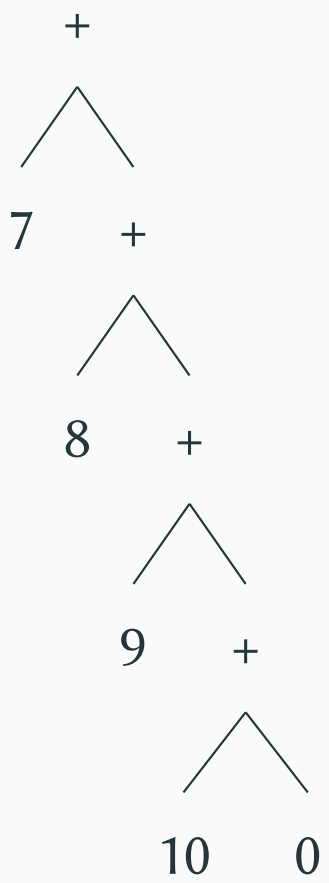
`foldr` and `foldl`

Haskell has functions `foldr` and `foldl` that each take a two-place operation, a starting value, and some list, in order to iteratively apply the function to the elements of the list, one-by-one.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (b -> a -> b) -> b -> [a] -> b
```

`foldr`

`foldr`, in a way, conceptualizes a list as right-branching.



`foldl`

`foldl` conceptualizes it as left-branching.