Introduction to Haskell

Julian Grove

Computational semantics, August 30, 2023

University of Rochester

Haskell basics

Static typing

Haskell is statically typed. This means:

Static typing

Haskell is statically typed. This means:

- Every expression in the language is assigned a type.

Static typing

Haskell is statically typed. This means:

- Every expression in the language is assigned a type.
- If an expression is not well-typed (i.e., cannot be be assigned a type), the compiler will throw an error.

Type annotations

Expressions can be annotated with their types:

Type annotations

Expressions can be annotated with their types:

- Put the type signature on one line, and the definition of the value after it.

Type annotations

Expressions can be annotated with their types:

- Put the type signature on one line, and the definition of the value after it.
- The identifier for the value goes to the left of the '::', and the type of the value bound to the identifier goes to the right of the '::'.

Type annotations

Expressions can be annotated with their types:

- Put the type signature on one line, and the definition of the value after it.
- The identifier for the value goes to the left of the '::', and the type of the value bound to the identifier goes to the right of the '::'.
- We can think of `String` as a simple type.

Type annotations

Expressions can be annotated with their types:

- Put the type signature on one line, and the definition of the value after it.
- The identifier for the value goes to the left of the '::', and the type of the value bound to the identifier goes to the right of the '::'.
- We can think of `String` as a simple type.

Type annotations

Expressions can be annotated with their types:

- Put the type signature on one line, and the definition of the value after it.
- The identifier for the value goes to the left of the '::', and the type of the value bound to the identifier goes to the right of the '::'.
- We can think of `String` as a simple type. But we can also have more complicated types, e.g., those for functions.

Type annotations

Expressions can be annotated with their types:

- Put the type signature on one line, and the definition of the value after it.
- The identifier for the value goes to the left of the '::', and the type of the value bound to the identifier goes to the right of the '::'.
- We can think of `String` as a simple type. But we can also have more complicated types, e.g., those for functions.
    - In '$\alpha \to \beta$', α is the input type and β is the output type.

Type annotations

Expressions can be annotated with their types:

- Put the type signature on one line, and the definition of the value after it.
- The identifier for the value goes to the left of the '::', and the type of the value bound to the identifier goes to the right of the '::'.
- We can think of `String` as a simple type. But we can also have more complicated types, e.g., those for functions.
  - In '$\alpha \to \beta$', α is the input type and β is the output type.
- For a function with more than one input, the thing to the right of the final '$\to$' is the output, and everything to the left of that type is an input type.

Function application

Function application

- To call a function in languages like, e.g., Python, you use this syntax:

$$\texttt{addThenDouble}(2, 3)$$

- In Haskell, a function with multiple arguments is applied to them in order…
- So, if we want to define another function in terms of `addThenDouble`, that's the syntax we'd use for its definition…

Fixity

Fixity

- You can make an infix function a prefix function by wrapping it in parentheses…

Fixity

- You can make an infix function a prefix function by wrapping it in parentheses…
- And you can make any prefix function an infix function by wrapping it in backticks…

Lambdas

Lambdas

- The function definitions we have been writing so far are actually a convenient shorthand.

Lambdas

- The function definitions we have been writing so far are actually a convenient shorthand.
- Another way to write them…

Lambdas

- The function definitions we have been writing so far are actually a convenient shorthand.
- Another way to write them…
- Haskell provides this syntax for anonymous functions so that we may exploit the fact that functions are treated as *first-class citizens* of the language.

Lambdas

- The function definitions we have been writing so far are actually a convenient shorthand.
- Another way to write them…
- Haskell provides this syntax for anonymous functions so that we may exploit the fact that functions are treated as *first-class citizens* of the language.

Lambdas

- The function definitions we have been writing so far are actually a convenient shorthand.
- Another way to write them…
- Haskell provides this syntax for anonymous functions so that we may exploit the fact that functions are treated as *first-class citizens* of the language. That means you can…

Lambdas

- The function definitions we have been writing so far are actually a convenient shorthand.
- Another way to write them…
- Haskell provides this syntax for anonymous functions so that we may exploit the fact that functions are treated as *first-class citizens* of the language. That means you can…
    - write a function that takes another function as its input

Lambdas

- The function definitions we have been writing so far are actually a convenient shorthand.
- Another way to write them…
- Haskell provides this syntax for anonymous functions so that we may exploit the fact that functions are treated as *first-class citizens* of the language. That means you can…
    - write a function that takes another function as its input
    - write a function that returns another function as its output

Type inference

As we said, the usual syntax for type annotations is:

```
nameOfThing :: typeOfThing
```

Type inference

As we said, the usual syntax for type annotations is:

```
nameOfThing :: typeOfThing
```

- However, Haskell doesn't usually require that you give type annotations.

Type inference

As we said, the usual syntax for type annotations is:

```
nameOfThing :: typeOfThing
```

- However, Haskell doesn't usually require that you give type annotations.
- Why? The compiler can usually infer what type everything is supposed to be from your code.

Type inference

As we said, the usual syntax for type annotations is:

```
nameOfThing :: typeOfThing
```

- However, Haskell doesn't usually require that you give type annotations.
- Why? The compiler can usually infer what type everything is supposed to be from your code.
- But the usual practice is to always include explicit type annotations in your code.

Basic types

Booleans

Booleans are truth values, and there are two of them: `True` and `False`.

Integers

The `Integer` type allows you to express any integer.

Integers

The `Integer` type allows you to express any integer.

- Note that if you ask for the type of something that looks like an integer, weird stuff happens…

Integers

The `Integer` type allows you to express any integer.

- Note that if you ask for the type of something that looks like an integer, weird stuff happens…
- You can fix this with type annotations.

Chars

A `Char` is a character of text. In Haskell, this can be any Unicode character.

Strings

In Haskell, strings are lists of characters.

Strings

In Haskell, strings are lists of characters.

- This means that any function you're used to using on lists, you can use on strings…

Tuples

A tuple stores a number of different values, bundled up into a single value.

Tuples

A tuple stores a number of different values, bundled up into a single value.

- Tuples are very different from lists (which we'll get to later).