Functional Programming with Haskell

Kenneth Lai

Brandeis University

September 14, 2022

Announcements

- Ken away next week
  - Classes
    - Ken teaching on Zoom
    - You are still encouraged to come to class in person!
  - Student hours
    - Fri 9/16 2:15-3:15pm, Ken, remote only
    - Tue 9/20 5-6pm, Ken, remote only
    - Wed 9/21 11am-noon, Bingyang, hybrid
    - Thu 9/22 4-5pm, Ken, remote only
    - Fri 9/23 2:15-3:15pm, Bingyang, hybrid
- HW1 due date moved to 9/28

Today's Plan

- Functional Programming with Haskell

Today's Plan

- Functional Programming with Haskell
- (we'll see how far we get...)

Today's Plan

- Functional Programming with Haskell
- (we'll see how far we get...)
- Types in Natural Language

Types, in Words

$$\tau ::= e \mid t \mid \tau \to \tau$$

Types, in Words

$$\tau ::= e \mid t \mid \tau \rightarrow \tau$$

- A type can be:
  - A basic type (e, t, etc.)
  - A functional type $\tau_1 \rightarrow \tau_2$, where τ_1 and τ_2 are types
    - This is the type of a function whose input is of type τ_1 and output is of type τ_2

Computational Semantics
Day 1: Getting Started with Haskell + Inference Engine for NL

Jan van Eijck[1] & Christina Unger[2]

[1]CWI, Amsterdam, and UiL-OTS, Utrecht, The Netherlands
[2]CITEC, Bielefeld University, Germany

ESSLLI 2011, Ljubljana

Haskell is functional

A program consists entirely of functions.

- The main program itself is a function with the program's input as argument and the program's output as result.
- Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives.

Running a Haskell program consists in evaluating expressions (basically functions applied to arguments).

A shift in thinking

Imperative thinking:

- Variables are pointers to storage locations whose value can be updated all the time.
- You give a sequence of commands telling the computer what to do step by step.

Examples:

- initialize a variable `examplelist` of type integer list, then add 1, then add 2, then add 3
- in order to compute the factorial of n, initialize an integer variable `f` as 1, then for all `i` from 1 to n, set `f` to `f`$\times$`i`

A shift in thinking

Functional thinking:

- Variables are identifiers for an immutable, persistent value.
- You tell the computer what things are.

Examples:

- `examplelist` is a list of integers containing the elements 1, 2, and 3
- the factorial of n is the product of all integers from 1 to n

A shift in thinking

Functional thinking:

- Variables are identifiers for an immutable, persistent value.
- You tell the computer what things are.

Examples:

- `examplelist` is a list of integers containing the elements 1, 2, and 3
- the factorial of n is the product of all integers from 1 to n

```
factorial :: Int -> Int
factorial n = product [1..n]
```

A shift in thinking

Stop thinking in variable assignments, sequences and loops.

Start thinking in functions, immutable values and recursion.

Characteristics of Haskell

1. "Functions are first-class citizens"
   - "Functions may be passed as arguments to other functions and also can be returned as the result of some function"

Characteristics of Haskell

1. "Functions are first-class citizens"
    - "Functions may be passed as arguments to other functions and also can be returned as the result of some function"
2. Recursion

Characteristics of Haskell

1. "Functions are first-class citizens"
   - "Functions may be passed as arguments to other functions and also can be returned as the result of some function"
2. Recursion
3. Lazy evaluation
   - "Arguments of functions are only evaluated when needed, if at all"

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.
- Haskell offers you a new perspective on programming, it is powerful, and it is fun.

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.
- Haskell offers you a new perspective on programming, it is powerful, and it is fun.
- The type system behind Haskell is close related to the type system behind Montague grammar.

Why use Haskell?

- Haskell allows for abstract, high order programming. (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.
- Haskell offers you a new perspective on programming, it is powerful, and it is fun.
- The type system behind Haskell is close related to the type system behind Montague grammar.
- Your Haskell understanding will influence the way you understand natural language semantics.

Why use Haskell?

- Haskell allows for abstract, high order programming.
  (Ideally, more thinking and less writing and debugging.)
- Haskell is based on the lambda calculus, therefore the step from formal semantics theory to implementation is very small.
- Haskell offers you a new perspective on programming, it is powerful, and it is fun.
- The type system behind Haskell is close related to the type system behind Montague grammar.
- Your Haskell understanding will influence the way you understand natural language semantics.

Haskell is rich enough to be useful. But above all, Haskell is a language in which people play. In the end, we want to infect your brain, not your hard drive.

(Simon Peyton-Jones)

Getting started

Get GHCup:

- https://www.haskell.org/ghcup/

This includes the Glasgow Haskell Compiler (GHC) together with standard libraries and the interactive environment GHCi.

Using the Book Code

- Formerly available at the book website:
  http://www.computational-semantics.eu
- Currently available on LATTE

Using the Book Code

```
module FPH

where

import Data.List
import Data.Char
```

Haskell as a Calculator

Start the interpreter:

Haskell as a Calculator

Start the interpreter:

```
lucht:cmpsem jve$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

Haskell as a Calculator

Start the interpreter:

```
lucht:cmpsem jve$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

GHCi can be used to interactively evaluate expressions.

```
Prelude> 2 + 3
Prelude> 2 + 3 * 4
Prelude> 2^10
Prelude> (42 - 10) / 2
```

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

```
double :: Int -> Int
double n = 2 * n
```

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

   ```
   double :: Int -> Int
   double n = 2 * n
   ```

2. Inside GHCi, you can load the program with `:l first.hs`
   (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

```
double :: Int -> Int
double n = 2 * n
```

2. Inside GHCi, you can load the program with `:l first.hs` (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.
3. Now you can evaluate expressions like `double 5`, `double (2+3)`, and `double (double 5)`.

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

```
double :: Int -> Int
double n = 2 * n
```

2. Inside GHCi, you can load the program with `:l first.hs` (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.
3. Now you can evaluate expressions like `double 5`, `double (2+3)`, and `double (double 5)`.
4. With `:t` you can ask GHCi about the type of an expression.

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

```
double :: Int -> Int
double n = 2 * n
```

2. Inside GHCi, you can load the program with `:l first.hs` (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.
3. Now you can evaluate expressions like `double 5`, `double (2+3)`, and `double (double 5)`.
4. With `:t` you can ask GHCi about the type of an expression.
5. Leave the interactive environment with `:q`.

First Experiments

- In a module:

```
double :: Int -> Int
double n = 2 * n
```

First Experiments

- In a module:

```
double :: Int -> Int
double n = 2 * n
```

- In the interpreter:

```
Prelude> let double n = 2 * n
Prelude> double 5
10
```

First Experiments

- In a module:

```
double :: Int -> Int
double n = 2 * n
```

- In the interpreter:

```
Prelude> let double n = 2 * n
Prelude> double 5
10
```

- Combining the two statements:

```
Prelude> let double n = 2 * n in double 5
10
```

First Experiments

- Type declarations work in the interpreter too:

```
Prelude> let double :: Int -> Int; double n = 2 * n
Prelude> double 5
10
```

First Experiments

- "In Haskell it is not strictly necessary to always give explicit type declarations.
    - For instance, the definition of square would also work without the type declaration, since the system can infer the type from the definition.
- However, it is good programming practice to give explicit type declarations even when this is not strictly necessary.
    - These type declarations are an aid to understanding, and they greatly improve the digestibility of functional programs for human readers.
    - Moreover, by writing down the intended type of a function you constrain what you can implement, for you rule out all definitions that take arguments or yield values that do not agree with the type declaration.
    - If you try to write a definition with such a type conflict, the interpreter will immediately reject it."

Lambda Abstraction in Haskell

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

Lambda Abstraction in Haskell

In Haskell, \ x expresses lambda abstraction over variable x.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

- The intention is that variabele x stands proxy for a number of type Int.

Lambda Abstraction in Haskell

In Haskell, \ x expresses lambda abstraction over variable x.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

- The intention is that variabele x stands proxy for a number of type Int.
- The result, the squared number, also has type Int.

Lambda Abstraction in Haskell

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.
- The function `sqr` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`.

Lambda Abstraction in Haskell

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.
- The function `sqr` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`.
- This is precisely what the type-indication `Int -> Int` expresses.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.
- `Char` is the type of characters.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.
- `Char` is the type of characters.

Note that the name of a type always starts with a capital letter.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.
- `Char` is the type of characters.

Note that the name of a type always starts with a capital letter.

To denote arbitrary types, Haskell allows the use of *type variables*. For these, `a`, `b`, . . . , are used.

Haskell Derived Types

- By list-formation: if a is a type, [a] is the type of lists over a. Examples: [Int] is the type of lists of integers; [Char] is the type of lists of characters, or strings.

Haskell Derived Types

- By list-formation: if `a` is a type, `[a]` is the type of lists over `a`. Examples: `[Int]` is the type of lists of integers; `[Char]` is the type of lists of characters, or strings.
- By pair- or tuple-formation: if `a` and `b` are types, then `(a,b)` is the type of pairs with an object of type `a` as their first component, and an object of type `b` as their second component. If `a`, `b` and `c` are types, then `(a,b,c)` is the type of triples with an object of type `a` as their first component, an object of type `b` as their second component, and an object of type `c` as their third component . . .

Haskell Derived Types

- By list-formation: if a is a type, [a] is the type of lists over a. Examples: [Int] is the type of lists of integers; [Char] is the type of lists of characters, or strings.
- By pair- or tuple-formation: if a and b are types, then (a,b) is the type of pairs with an object of type a as their first component, and an object of type b as their second component. If a, b and c are types, then (a,b,c) is the type of triples with an object of type a as their first component, an object of type b as their second component, and an object of type c as their third component . . .
- By function definition: a -> b is the type of a function that takes arguments of type a and returns values of type b.

Haskell Derived Types

- By list-formation: if a is a type, [a] is the type of lists over a. Examples: [Int] is the type of lists of integers; [Char] is the type of lists of characters, or strings.
- By pair- or tuple-formation: if a and b are types, then (a,b) is the type of pairs with an object of type a as their first component, and an object of type b as their second component. If a, b and c are types, then (a,b,c) is the type of triples with an object of type a as their first component, an object of type b as their second component, and an object of type c as their third component ...
- By function definition: a -> b is the type of a function that takes arguments of type a and returns values of type b.
- By defining your own datatype from scratch, with a data type declaration. More about this in due course.

Question for Discussion

- Formal semantics is about building models of the (or a) world, and evaluating the truth of sentences in those models

Question for Discussion

- Formal semantics is about building models of the (or a) world, and evaluating the truth of sentences in those models
- What kinds of things do we want to represent in our models?
  - What are the corresponding kinds of linguistic expressions?

Question for Discussion

- Formal semantics is about building models of the (or a) world, and evaluating the truth of sentences in those models
- What kinds of things do we want to represent in our models?
  - What are the corresponding kinds of linguistic expressions?
- Try to assign each "kind of thing" a type
  - Which types should be basic types, and which types can be derived (functional) types?

Question for Discussion

- Formal semantics is about building models of the (or a) world, and evaluating the truth of sentences in those models
- What kinds of things do we want to represent in our models?
  - What are the corresponding kinds of linguistic expressions?
- Try to assign each "kind of thing" a type
  - Which types should be basic types, and which types can be derived (functional) types?
- If you have taken formal semantics before, you may remember "the answer"
  - That being said, try to keep an open mind. For example, what happens if you choose a different set of basic types?