CSC 372 - Project 1

Introduction to Haskell

Anh Nguyen Phung
Trong Nguyen

1. Overview

- Haskell is a general-purpose, statically typed, purely functional programming language with type inference and lazy evaluation.
- Designed for teaching, research and industrial application, Haskell has pioneered a number of advanced programming language features such as type classes, which enable type-safe operator overloading.

Haskell

- Haskell's main implementation is the Glasgow Haskell Compiler (GHC).
- Haskell is named after logician Haskell Curry.
- Haskell's semantics are historically based on those of the Miranda programming language.
- Haskell is used in academia and industry. As of May 2021, Haskell was the 28th most popular programming language in terms of Google searches.
- Haskell belongs to the purely functional paradigm.

2. History of Origin

- Following the release of Miranda (a programming language) by Research Software Ltd. in 1985, interest in lazy functional languages grew.
- By 1987, more than a dozen non-strict, purely functional programming languages existed. Miranda, which is lazy and purely functional programming language, was the most widely used, but it was proprietary software.

- Therefore, at the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, there was a strong consensus that a committee be formed to define an open standard for such languages.
- The committee's purpose was to consolidate existing functional languages into a common one to serve as a basis for future research in functional-language design.

=> Haskell was invented as a result of the committee.

3. History of Development

Haskell 1.0 to 1.4

- Type classes, which enable type-safe operator overloading, were first proposed by Philip Wadler and Stephen Blott for Standard ML but were first implemented in Haskell between 1987 and version 1.0.
- The first version of Haskell ("Haskell 1.0") was defined in 1990.
- The committee's efforts resulted in a series of language definitions (1.0, 1.1, 1.2, 1.3, 1.4).

Haskell 98

- In late 1997, the series culminated in Haskell 98, intended to specify a stable, minimal, portable version of the language and an accompanying standard library for teaching, and as a base for future extensions.
- The committee expressly welcomed creating extensions and variants of Haskell 98 via adding and incorporating experimental features.

Haskell 2010

- Haskell 2010 is an incremental update to the language, mostly incorporating several well-used and uncontroversial features previously enabled via compiler-specific flags.
- Some important updates include hierarchical module names, foreign function interface (FFI), fixed syntax issues, and more relaxed rules of type inferences.

4. Main Features

Purely Functional

- Every function in Haskell is a function in the mathematical sense.
- Even side-effecting IO operations are but a description of what to do, produced by pure code.
- There are no statements or instructions, only expressions which cannot mutate variables (local or global) nor access state like time or random numbers.

Purely Functional

- The following function takes an integer and returns an integer. By the type it cannot do any side-effects whatsoever, it cannot mutate any of its arguments.

```haskell
square :: Int -> Int
square x = x * x
```

Statistically Typed

- Every expression in Haskell has a type which is determined at compile time.
- All the types composed together by function application have to match up. If they don't, the program will be rejected by the compiler.
- Types become not only a form of guarantee, but a language for expressing the construction of programs.

Statistically Typed

- All Haskell values have a type

```
char = 'a'      :: Char
int = 123       :: Int
fun = isDigit :: Char -> Bool
```

Type Inference

- We don't have to explicitly write out every type in a Haskell program. Types will be inferred by unifying every type bidirectionally.
- We can write out types if we choose, or ask the compiler to write them for us for handy documentation.

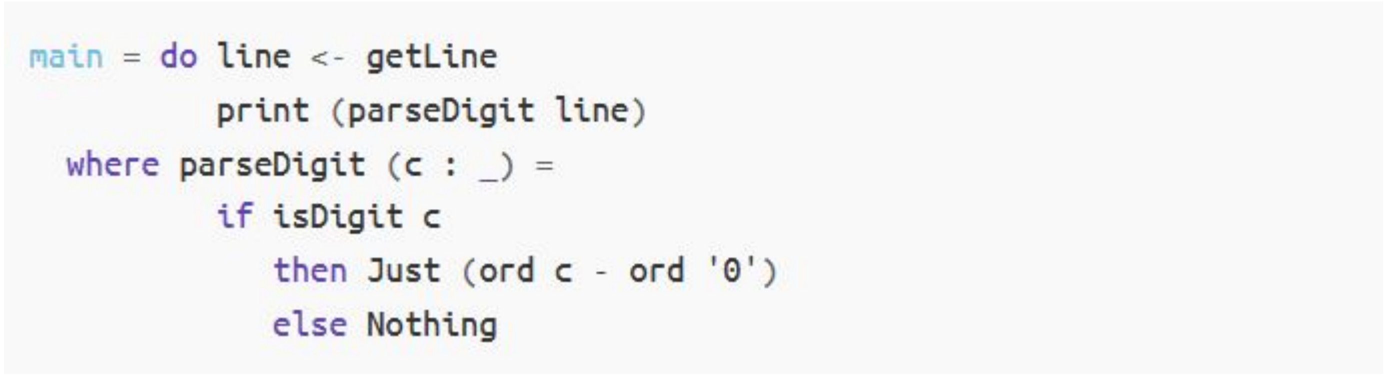
Type Inference

- The below example has a type signature for every binding:

```
main :: IO ()
main = do line :: String <- getLine
          print (parseDigit line)
  where parseDigit :: String -> Maybe Int
        parseDigit ((c :: Char) : _) =
          if isDigit c
            then Just (ord c - ord '0')
            else Nothing
```

Type Inference

- Instead of declaring every type, we can rewrite the example as follow:



```haskell
main = do line <- getLine
          print (parseDigit line)
  where parseDigit (c : _) =
          if isDigit c
             then Just (ord c - ord '0')
             else Nothing
```

Lazy Evaluation

- Functions in Haskell don't evaluate their arguments.
- Programs in Haskell can compose together very well, with the ability to write control constructs (such as if/else) just by writing normal functions.
- The purity of Haskell code makes it easy to fuse chains of functions together, allowing for performance benefits.

Lazy Evaluation

- Lazy evaluation allows us to define control structure easily

```haskell
when p m = if p then m else return ()
main = do args <- getArgs
          when (null args)
               (putStrLn "No args specified!")
```

Lazy Evaluation

- For a repeated expression pattern, we can give it a name to reuse later

```
and c t = if c then t else False
```

Lazy Evaluation

- For a repeated expression pattern, we can give it a name to reuse later

```haskell
and c t = if c then t else False
```

- It is common to get code re-use by using predefined functions in Haskell

```haskell
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p
```

Concurrent

- Haskell lends itself well to concurrent programming due to its explicit handling of effects.
- Its flagship compiler, GHC, comes with a high-performance parallel garbage collector and light-weight concurrency library containing a number of useful concurrency primitives and abstractions.

Haskell

Packages

- Open source contribution to Haskell is very active with a wide range of packages available on the public package servers.
- There are 6,954 packages freely available in Haskell.

5. Things learned from Haskell

Basics Operations & Boolean Logic

- Basic operations and boolean logic in Haskell follow the same standard of rules as other programming languages.
- More details in the code: https://github.com/anhnguyenphung/csc372-project1/blob/master/code/basics.hs

Types & Algebraic Data Types

- We can specify the types in Haskell when we declared the variables.
- Haskell allows programmers to create their own enumeration types.
- More details in the code:

  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/variables.hs

  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/algebraic_types.hs

  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/pairs_and_lists.hs

Functions & Recursion

- In Haskell, functions can be declared by specifying the arguments and return value type. In fact, we often use recursion patterns to define functions.
- More details in the code:

  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/function.hs

  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/recursion_Prelude.hs

Lazy Evaluation & High Order Functions

- Haskell follows lazy evaluation, which is an evaluation strategy which delays the evaluation of an expression until its value is needed.
- We can utilize anonymous function and function composition to write high order functions.
- More details in the code:

  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/lazy_evaluation.hs

  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/high_order_functions.hs

Functor & Polymorphism

- Functor in Haskell is a kind of functional representation of different Types which can be mapped over. It is a high level concept of implementing polymorphism.
- Haskell supports polymorphism for both data type and functions.
- More details in the code:

  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/functor.hs

  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/polymorphism.hs

I/O

- I/O is quite special in Haskell. Haskell is a lazy and pure functional language, so in Haskell, functions may not have any external effects and may not depend on external input.
- However, we can still handle the work with I/O through libraries in Haskell.
- More details in the code:
  https://github.com/anhnguyenphung/csc372-project1/blob/master/code/io.hs

6. Haskell and SML

Similarities between Haskell and standard ML

- Both have first-class functions (functions and values)

  Example of add function:

Haskell

```
add                         :: Integer -> Integer -> Integer
add x y                     =  x + y
```

standard ML

```
fun add(x, y) = x + y
```

Similarities between Haskell and standard ML

- Both of the languages are statically typed
- Both have polymorphic types
- Both have algebraic datatypes (& pattern matching)

```
data List a = Nil | Cons a (List a)
```

- Both have automatic type inference
- Both have “layered patterns” (“as” in ML and “@” in Haskell)
- Both support a fairly conventional set of basic types integers, reals, booleans, strings
- Both have list as a basic and heavily used type

Differences between Haskell and standard ML

- Haskell is pure, without side effects or exceptions, while standard ML has mutable data structures (refs and arrays) and exceptions (control effects).
- Haskell has lazy evaluation, while ML has strict evaluation

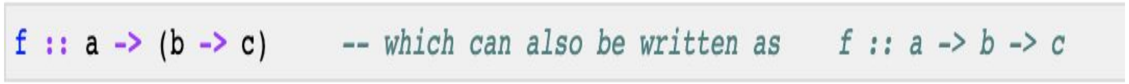
  Example of lazy evaluation in Haskell:

```
(1+2)*(1+2)
=> 3*(1+2)
=> 3*3
=> 9
```

- Haskell has more special notational support for list -- list comprehension
- Haskell has boolean guards associated with pattern matching

Differences between Haskell and standard ML

- Haskell functions are usually curried, while SML functions use tuple of record arguments for multi-arguments functions.



```
f :: a -> (b -> c)     -- which can also be written as    f :: a -> b -> c
```

- Haskell has type classes to support abstraction over types with associated interfaces, meanwhile, ML use modules and functors to support types with associated interfaces and abstraction.
- Haskell use monads to simulate imperative and effectful programming while ML support it directly.

7. Summary & Practical Applications

Why Haskell is so useful?

- Concise, high-level, practical and also extremely fast
- Concurrency is easy compared to other languages
- Haskell has many high-quality library
- Good tooling and package management
- Haskell provides a lot of extra safety and flexibility
- Haskell is a pleasant language to work with

Some other aspects of Haskell

- Folds and monoids
- Applicative functors
- Monads
- Monads Transformers
- Free Monads
- Coroutines

Would we recommend Haskell for other programmers?

Absolutely yes!

The programmers can use Haskell as the functional languages for many other purposes such gaming, financial math, desktop environment, and other applications.

There are many interesting things you can learn from Haskell and you will enjoy them a lot.

References

https://www.haskell.org/

https://en.wikipedia.org/wiki/Haskell_(programming_language)

https://www.classes.cs.uchicago.edu/archive/2012/spring/22300-1/lectures/lesson7.txt

https://www.cis.upenn.edu/~cis194/spring13/lectures.html

Thank you for watching the presentation!