Student Online Teaching Advice Notice

1 Week 1

1.1 Class 1

1.1.1 Timetable 2020

Course Timetable (2020-21)

- Timetable:
  - Mon 2pm Online Lecture/Tutorial
  - Wed 4pm Online Lecture/Tutorial
  - Thu 2pm Online Lecture/Tutorial
  - Face2Face "pods" Weeks 4(?),7,10
- The timetable is possibly still in a state of flux, and the *authoritative* version of the timetable is in SITS (my.tcd.ie).
- Class Management: Blackboard
- Assessment

- Exam : 80%, 2hr
- Continuous Assessment : 20%

1.1.2 Haskell Version

Haskell for CSU34016 (2020)

- We shall use the GHC compiler
- Coursework will be based on the use of the *stack* tool `https://www.stackage.org` `https://docs.haskellstack.org/en/stable/README/`
- Install *stack* and let it install ghc, at least as far as this course is concerned (see Lab00, to come).

1.1.3 FP Marketing

What is a functional programming language?

- Basic notion of computation: the application of functions to arguments.
- Basic idea of program: writing function definitions
- Functional languages are declarative: more emphasis on *what* rather than *how*.

Defining Haskell values

- Function definitions are written as equations
- ```
  double x = x + x
  quadruple x = double (double x)
  ```
- compute the length of a list

  ```
  length [] = 0
  length (x:xs) = 1 + length xs
  ```

  recursion is the natural way to describe repeated computation
- Haskell can infer types itself (Type Inference)
  ```

Type Polymorphism

- What is the type of `length`?

```
> length [1,2,3]
3
> length ['a','b','c','d']
4
> length [[],[1,2],[3,2,1],[],[6,7,8]]
5
```

- `length` works for lists of elements of arbitrary type `length :: [a] -> Int` Here '`a`' denotes a type variable, so the above reads as "`length` takes a list of (arbitrary) type `a` and returns an `Int`".
- A similar notion to "generics" in O-O languages, but builtin without fuss.

Laziness

- What's wrong with the following (recursive) definition ?

  ```
  from n =  n : (from (n+1))
  ```

  Nothing ! It just generates an infinite list of ascending numbers, starting from `n`.
- `take n list` — return first `n` elements of `list`.
- What is `take 10 (from 1)` ?

  ```
  > take 10 (from 1)
  [1,2,3,4,5,6,7,8,9,10]
  ```

- Haskell is a *lazy* language, so values are evaluated only when needed.

Program Compactness

- Sorting the empty list gives the empty list:

  ```
  qsort [] = []
  qsort (x:xs)
    = qsort [y | y <- xs, y < x]
      ++ [x]
      ++ qsort [z | z <- xs, z >= x]
  ```

- We have used Haskell list comprehensions `[y | y <- xs, y < x ]` "build list of `y`s, where `y` is drawn from `xs`, such that `y < x`"
- Try that in Java !

Whistle . . . Stop!

- Haskell is powerful, and quite different to most mainstream languages
- It allows very powerful programs to be written in a concise manner
- These languages originally developed for theorem provers and rewrite systems
- Very popular now for:
  - software static checkers, e.g., Facebook's `infer` (fbinfer.com)
  - quantitative analysis in financial services
  - Domain-Specific Languages (DSLs)
  - Front-end language handling and transformation.

1.2 Class 2

1.2.1 Lambda Calculus

The λ-Calculus

- Invented by Alonzo Church in 1930s
- Intended as a form of logic
- Turned into a model of computation
- Not shown completely sound until early 70s !

λ-Calculus: Syntax

Infinite set $Vars$, of variables:

$$u, v, x, y, z, \ldots, x_1, x_2, \ldots \quad \in \quad Vars$$

Well-formed λ-calculus expressions $LExpr$ is the smallest set of strings matching the following syntax:

$$\begin{array}{rcl} M, N, \ldots \in LExpr & ::= & v \\ & | & (\lambda x \bullet M) \\ & | & (M\ N) \end{array}$$

Read: a λ-calculus expression is either (i) a variable (v); (ii) an *abstraction* of a variable from an expression ($\lambda x \bullet M$); or (iii) an *application* of one expression to another ($(M\ N)$).

COMMENTARY: Think of this a recipe to build valid λ-calculus expressions. Pick a variable a. This gives us a λ-calculus expression. We can use it to build an application so: $(a\ a)$. Now we can use those two to get three more applications:

$$(a\ (a\ a)) \quad ((a\ a)\ a) \quad ((a\ a)\ (a\ a))$$

We can continue like this indefinitely. We can also pick other variables (b, c, say) and mix it up. We can do all of this with abstraction too:

$$(\lambda a \bullet a) \quad (\lambda c \bullet (a\ a)) \quad ((\lambda c \bullet (a\ c))\ (b\ a))$$

λ-Calculus: Free/Bound Variables

- A variable *occurrence* is free in an expression if it is not mentioned in an *enclosing abstraction*.

  $$x \qquad (\lambda y \bullet (yz))$$

- A variable *occurrence* is bound in an expression if is mentioned in an *enclosing abstraction*.

  $$x \qquad (\lambda y \bullet (yz))$$

- A variable can be both free and bound in the same expression

  $$(x(\lambda x \bullet (xy)))$$

Think of bound variables as being like local variables in a program.

COMMENTARY: Consider the following C function:

```
int f(int x) {int y; return (g*x+y); }
```

Here x and y are bound (local) variables. while g is a free (global) variable.

λ-Calculus: α-Renaming

We can change a binding variable and its bound instances provided we are careful not to make other free variables become bound.

$$
\begin{array}{rcl}
(\lambda x \bullet (\lambda y \bullet (x\ y))) & \stackrel{\alpha}{\to} & (\lambda u \bullet \lambda v \bullet (u\ v))) \\
(\lambda x \bullet (x\ y)) & \stackrel{\alpha}{\not\to} & (\lambda y \bullet (y\ y)) \\
 & & \text{formerly free } y \text{ has been ``captured'' !}
\end{array}
$$

This process is called α-Renaming or α-Substitution, and leaves the meaning of a term unchanged.

It's the same as changing the name of a local variable in a program (fine, but you need to take care if there is a global variable of the same name hanging around)

λ-Calculus: Substitution

We define the notion of substituting an expression N for all free occurrences of x, in another expression M, written:

$$M[N/x]$$

$$(x\ (\lambda y \bullet (z\ y)))\ [\ (\lambda u \bullet u)\ /\ z\ ] \quad \stackrel{\rho}{\to} \quad (x\ (\lambda y \bullet ((\lambda u \bullet u)\ y)))$$

$$
\begin{array}{rcl}
(x\ (\lambda y \bullet (z\ y)))\ [\ (\lambda u \bullet u)\ /\ y\ ] & \stackrel{\rho}{\to} & (x\ (\lambda y \bullet (z\ y))) \\
 & & y \text{ was not free anywhere}
\end{array}
$$

λ-Calculus: Careful Substitution!

When doing (general) substitution $M[N/x]$, we need to avoid variable "capture" of free variables in N, by bindings in M:

$$(x\ (\lambda y \bullet (z\ y)))[(y\ x)/z] \quad \stackrel{\rho}{\not\to} \quad (x\ (\lambda y \bullet ((y\ x)\ y)))$$

If N has free variables which are going to be inside an abstraction on those variables in M, then we need to α-Rename the abstractions to something else first, and then substitute:

$$\begin{array}{ll} & (x\ (\lambda y \bullet (z\ y)))[(y\ x)/z] \\ \overset{\alpha}{\to} & (x\ (\lambda w \bullet (z\ w)))[(y\ x)/z] \\ \overset{\rho}{\to} & (x\ (\lambda w \bullet ((y\ x)\ w))) \end{array}$$

The Golden Rule: A substitution should never make a free occurrence of a variable become bound, or vice-versa.

λ-Calculus: β-Reduction

We can now define the most important "move" in the λ-calculus, known as β-Reduction:

$$(\lambda x \bullet M)\ N \quad \overset{\beta}{\to} \quad M[N/x]$$

We define an expression of the form $(\lambda x \bullet M)\ N$ as a "$(\beta-)$redex" (*red*ucible *ex*pression).

λ-Calculus: Normal Form

An expression is in "Normal-Form" if it contains no redexes.

The object of the exercise is to reduce an expression to its normal-form (*if it exists*).

$$\begin{array}{ll} & ((\underline{(\lambda x \bullet (\lambda y \bullet (y\ x)))\ u)}\ v) \\ \overset{\beta}{\to} & \underline{((\lambda y \bullet (y\ u))\ v)} \\ \overset{\beta}{\to} & (v\ u) \end{array}$$

Not all expressions have a normal form — e.g.: $((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))$

What about:

$$(\ (\ (\lambda x \bullet (\lambda y \bullet y))\ \ ((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))\ )\ \ w\ )\quad ?$$

$$(\ (\ (\lambda x \bullet (\lambda y \bullet y)) \ \ ((\lambda x \bullet (x \ x)) \ (\lambda x \bullet (x \ x))) \ ) \ \ w \ )$$

- Do innermost redex first

$$\begin{array}{ll} & (\ (\ (\lambda x \bullet (\lambda y \bullet y)) \ \ \underline{((\lambda x \bullet (x \ x)) \ (\lambda x \bullet (x \ x)))} \ ) \ \ w \ ) \\ \stackrel{\beta}{\to} & (\ (\ (\lambda x \bullet (\lambda y \bullet y)) \ \ ((\lambda x \bullet (x \ x)) \ (\lambda x \bullet (x \ x))) \ ) \ \ w \ ) \end{array}$$

  We can keep doing this forever!

- Do outermost (leftmost) redex first

$$\begin{array}{ll} & (\ \underline{(\ (\lambda x \bullet (\lambda y \bullet y)) \ \ ((\lambda x \bullet (x \ x)) \ (\lambda x \bullet (x \ x))) \ )} \ \ w \ ) \\ \stackrel{\beta}{\to} & (\ (\lambda y \bullet y) \ \ w \ ) \\ \stackrel{\beta}{\to} & w \end{array}$$

λ-Calculus and Computability

- So What ? Why do we look at this weird calculus anyway?
- We can use it to encode booleans, numbers, and functions over same.
- In fact, we can encode any computable function this way!
- λ-Calculus is Turing-complete
  - or is it that Turing machines are Church-complete?
- It is one of a number of *equivalent* models of computation that emerged in the 1930s

And this has what to do with functions, exactly?

Consider a "conventional" function definition and application of that function to an argment:

$$f(x) = 2x + 1 \qquad f(42)$$

$$
\begin{aligned}
& f(42) \\
= \quad & \text{substitute } 42 \text{ for } x \text{ in definition r.h.s.} \\
& (2x+1)[42/x] \\
= \quad & \text{perform substituion} \\
& 2 \times 42 + 1
\end{aligned}
$$

This is basically β-reduction!

What the λ-calculus captures is function definition and application ($f = \lambda x \bullet 2x+1$)

Lambda abstraction in Haskell

The Haskell notation is designed to reflect how it looks in lambda-calculus

Since these values are themselves functions, we just apply them to values to compute something

```
> (\x -> 2 * x + 1) 42
85
```

Essentially we can view Haskell as being the (typed) lambda-calculus with *LOTS* of syntactic sugar.

1.3 Class 3

1.3.1 Getting GHC

Getting GHC

- Can't wait for the 1st exercise in order to get going?
- Strongly recommended: install `stack` (see `https://docs.haskellstack.org/en/stable/README/`)
- Follow the Quickstart guide for Unix/OS X the default behaviour is usually fine for Windows read the Windows stuff carefully

1.3.2 Basics

Haskell Language Structure

- Haskell is built on top of a simple functional language (Haskell "Core")
- Haskell Core is itself built on top of an extended form of the λ-calculus, that has value, types, primitive operations, and pattern-matching added on.
- A lot of syntactic sugar is added
- A large collection of standard types and functions are predefined and automatically loaded (the Haskell "Prelude")
- There are a vast number of libraries that are also available
- See `www.haskell.org`

Patterns in Mathematics

In mathematics we often characterise something by laws it obeys, and these laws often look like patterns or templates:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)!, \qquad n > 0 \end{aligned}$$

$$\begin{aligned} len(\langle\rangle) &= 0 \\ len(\ell_1 \frown \ell_2) &= len(\ell_1) + len(\ell_2) \end{aligned}$$

Here $\langle\rangle$ denotes an empty list, and $\frown$ joins two lists together.

Pattern matching is inspired by this (but with some pragmatic differences).

Factorial! as Patterns

Math:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \end{aligned}$$

Haskell (without patterns):

```
factorial_nop n  =  if n==0 then 1 else n * factorial_nop (n-1)
```

Haskell (with patterns):

```
factorial 0  =  1
factorial n  =  n * factorial (n-1)
```

Formal argument `0` is shorthand saying check the argument to see if it is zero. If so, do my righthand side.

Formal argument `n` says, take the argument, and refer to it in my righthand side as `n`.

Lists in Haskell

Lists of things are a very common datatype in functional languages similar to Haskell.

There is a standard approach to constructing lists:

- An empty list using: `[]`
- Given a term `x` and a list `xs` we can construct a list consisting of `x` followed by `xs` as follows: `x:xs`
- So the list 1,2,3 can be built as `1:2:3:[]` Brackets show how it is built up: `1:(2:(3:[]))`
- Syntactic Sugar:
  - We can write `[1,2,3]` as a shorthand for the above list
  - Lists can contain characters: `['H','e','l','l','o']` For character lists we have more shorthand: `"Hello"`
- Here `[]` and `:` are list *constructors* For historical reasons, `:` is pronounced "cons".

Length with Patterns

Math:

$$
\begin{aligned}
len(\langle\rangle) &= 0 \\
len(\ell_1 \frown \ell_2) &= len(\ell_1) + len(\ell_2) \\
len(\langle \_ \rangle \frown \ell) &= 1 + len(\ell) \\
len(\langle \_ \rangle) &= 1
\end{aligned}
$$

Haskell:

```
mylength []      =  0
mylength (x:xs)  =  1 + mylength xs
```

The key idea in pattern-matching is that the syntax used to build values, can also be used to look at a value, determine how it was built, and extract out the individual sub-parts if required.

Compact "Truth Tables"

Patterns can be used to give an elegant expression to certain functions, for instance we can define a function over two boolean arguments like this:

```
myand True True = True
myand _    _    = False
```

Key point: patterns are matched in order, and the first one to succeed is used.