Haskell Crash Course Part I

From the Lambda Calculus to Haskell

+ builtin Int, Char...

+ types

+ run-time

+ compiler

+ ...

Programming in Haskell

Computation by Calculation

Substituting equals by equals

Computation via Substituting Equals by Equals

```
    (1 + 3) * (4 + 5)
                         -- subst 1 + 3 = 4
==>       4 * (4 + 5)
                         -- subst 4 + 5 = 9
==>       4 * 9
                         -- subst 4 * 9 = 36
==>       36
```

f x y = x+y

Computation via Substituting Equals by Equals

Equality-Substitution enables **Abstraction** via **Pattern Recognition**

Abstraction via Pattern Recognition

Repeated Expressions

$x * (y + z)$

```
31 * (42 + 56)
70 * (12 + 95)
90 * (68 + 12)
```

Recognize Pattern as λ-function

```
pat = \x y z -> x  * ( y + z )
```

Equivalent Haskell Definition

```
pat   x y z =  x  * ( y + z )
```

Function Call is Pattern Instance

```
pat 31 42 56 =*> 31 * (42 + 56) =*> 31 * 98  =*> 3038
pat 70 12 95 =*> 70 * (12 + 95) =*> 70 * 107 =*> 7490
pat 90 68 12 =*> 90 * (68 + 12) =*> 90 * 80  =*> 7200
```

Key Idea: Computation is *substitute* equals by equals.

$$foo\ x\ y = e \qquad foo\ e_1\ e_2 \Rightarrow e\left[\begin{matrix}x=e_1\\ y=e_2\end{matrix}\right]$$

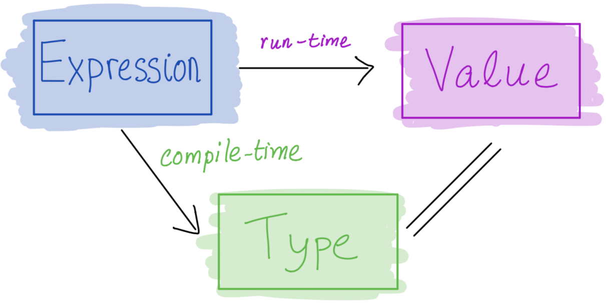
Programming in Haskell

Substitute Equals by Equals

Thats it! (*Do not* think of registers, stacks, frames etc.)

$$\boxed{130}$$

Elements of Haskell



- Core program element is an **expression**
- Every *valid* expression has a **type** (determined at compile-time)
- Every *valid* expression reduces to a *value* (computed at run-time)

Ill-typed* expressions are rejected at *compile-time* before execution

- *like* in Java
- *not like* λ-calculus or Python …

GHC

Haskel

compiler

"glorious"
"glasgow"

ghci

The Haskell Eco-System

- **Batch compiler:** `ghc` Compile and run large programs
- **Interactive Shell** `ghci` Shell to interactively run small programs online (https://repl.it/languages/haskell)
- **Build Tool** `stack` Build tool to manage libraries etc.

Interactive Shell: `ghci`

```
$ stack ghci
```

```
:load file.hs
:type expression
:info variable
```

A Haskell Source File

A sequence of **top-level definitions** `x1` , `x2` , …

- Each has *type* `type_1` , `type_2` , …
- Each defined by *expression* `expr_1` , `expr_2` , …

```
x_1 :: type_1
x_1 = expr_1

x_2 :: type_2
x_2 = expr_2

.
.
.
```

Basic Types

```
ex1 :: Int
ex1 = 31 * (42 + 56)   -- this is a comment

ex2 :: Double
ex2 = 3 * (4.2 + 5.6)  -- arithmetic operators "overloaded"

ex3 :: Char
ex3 = 'a' , 'b'        -- 'a', 'b', 'c', etc. built-in `Char` values

ex4 :: Bool
ex4 = True             -- True, False are builtin Bool values

ex5 :: Bool
ex5 = False
```

QUIZ: Basic Operations

```
ex6 :: Int
ex6 = 4 + 5

ex7 :: Int
ex7 = 4 * 5

ex8 :: Bool
ex8 = 5 > 4

quiz :: ~~???~~ Int
                      Int           Int
quiz = if ex8 then ex6 else ex7
```

if cond then e_1
else e_2

$$\boxed{\underbrace{cond}_{Bool}\ ?\ \underbrace{e_1}_{T} : \underbrace{e_2}_{T}} :: T$$

What is the *type* of `quiz` ?

A. `Int`

B. `Bool`

C. Error!

QUIZ: Basic Operations

```
ex6 :: Int
ex6 = 4 + 5

ex7 :: Int
ex7 = 4 * 5

ex8 :: Bool
ex8 = 5 > 4
```

TRUE

```
quiz :: ???
quiz = if ex8 then ex6 else ex7
```

TRUE

9

What is the *value* of `quiz`?

A. 9

B. `20`

C. Other!

Function Types

In Haskell, a **function is a value** that has a type

```
A -> B
```

A function that

- takes *input* of type `A`
- returns *output* of type `B`

For example

```
isPos :: Int -> Bool
isPos = \n -> (x > 0)
```

Define **function-expressions** using `\` like in λ-calculus!

But Haskell also allows us to put the parameter on the *left*

```
isPos :: Int -> Bool
isPos n = (x > 0)
```

(Meaning is **identical** to above definition with `\n -> ...`)

Multiple Argument Functions

A function that

- takes three *inputs* `A1`, `A2` and `A3`
- returns one *output* `B` has the type

```
A1 -> A2 -> A3 -> B
```

For example

```
pat :: Int -> Int -> Int -> Int
pat = \x y z -> x * (y + z)
```

which we can write with the params on the *left* as

```
pat :: Int -> Int -> Int -> Int
pat x y z = x * (y + z)
```

QUIZ

What is the type of `quiz` ?

```
quiz :: ???
quiz x y = (x + y) > 0
```

A. `Int -> Int`

B. `Int -> Bool`

C. `Int -> Int -> Int`

D. `Int -> Int -> Bool`

E. `(Int, Int) -> Bool`

Function Calls

A function call is *exactly* like in the λ-calculus

```
e1 e2
```

where `e1` is a function and `e2` is the argument. For example

```
>>> isPos 12
True

>>> isPos (0 - 5)
False
```

Multiple Argument Calls

With multiple arguments, just pass them in one by one, e.g.

```
(((e e1) e2) e3)
```

For example

```
>>> pat 31 42 56
3038
```