Functional Programming in Haskell

λ

Narendra Joshi

February 5, 2015

Outline

- Functional view of the world
- Haskell and its brief history
- Perks of being a Haskeller
- Logistics of the workshop

The Enterprise of Education

Education should prepare young people for jobs that do not yet exist, using technologies that have not been invented, to solve problems of which we are not yet aware.

Imperative World by Example

```
/* Adding numbers from 1 to 5, inclusive */

int acc = 0;
int i = 1;

while (i <= 5) {
     acc = acc + 1;
     i = i + 1;
}
```

Let's think about it for a while.

- What is the model of computation in our mind?
- What are the elements that make up that model?
- Is it all relevant to our problem of adding up a sequence of numbers?

Functional World by Example

```haskell
-- Sum up the *sequence* 1 to 5, inclusive
sum [1..5]

-- Definition of the sum function
sum [] = 0
sum listOfNumbers = head listOfNumbers + sum (tail listOfNumbers)
```

Answers to previous questions for you:

- Computation by calculation. Not commands and their execution.
- Hides details of execution. Lets us have more time to think about the problem.

What gives?

```c
/* Adding numbers from 1 to 5, inclusive */

int acc = 0;
int i = 1;

while (i <= 5) {
     acc = acc + 1;
     i = i + 1;
}
```

- Book-keeping of events in time.
- Details of the machine spill up to our mental model.

```haskell
-- Sum up the *sequence* 1 to 5, inclusive
sum [1..5]

-- Definition of the sum function
sum [] = 0
sum (x:xs) = x + sum xs
```

- Nice clean functional abstraction.
- Order of events that happen is based on data dependencies.

Yet Another Example

Imperative Code

```
/* Find first 2 primes in the range [1000, 1000000] */

int count = 0;
int primes[2] = {0};

for (int i = 1000; i <= 1000000 && count < 2; ++i) {
    if (is_prime(i))
        primes[count++] = i;
}
```

Functional Code

```
-- Filter primes in the range [1000, 1000000]
primesInRange = filter isPrime [1000..1000000]

-- Take 2 out of them
twoPrimesInRange = take 2 primesInRange
```

Key Points to Notice

Imperative Code

```
/* Find first 2 primes in the range [1000, 1000000] */

int count = 0;
int primes[2] = {0};

for (int i = 1000; i <= 1000000 && count < 2; ++i) {
     if (is_prime(i))
          primes[count++] = i;
}
```

- Filtering primes and collecting them interwined. Looks more efficient.
- We lost modularity when the above two operations mixed.

Functional Code

```
-- Filter primes in the range [1000, 1000000]
primesInRange = filter isPrime [1000..1000000]

-- Take 2 out of them
twoPrimesInRange = take 2 primesInRange
```

- First filter primes from the sequence. Then pick two out of them. Perfect!
- Is it less efficient? Are you checking primality for every number in the range?

That's being lazy with style!

What's Haskell?

Haskell is

- **Purely Funcional**
  You have definitions not assignments. No mutation.
- **Lazy**
  If something doesn't need to be computed, it will never be.
- **Higher Order**
  Functions are first-class people. Like values, they can be input to other functions.
- **General Purpose**
  It's not specific to any domain, e.g. SQL or html.

A little bit of History

Inspired by the paper: Being lazy with class, a history of Haskell.