The University of Hong Kong

COMP3259: Principles of Programming Languages

Tutorial 1

Tutor
Litao Zhou `ltzhou@cs.hku.hk`

Instructor
Bruno Oliveira `bruno@cs.hku.hk`

31 January 2023

Table of Contents

1 Introduction

The goal of this tutorial is to get the students familiar with Haskell programming. Students are encouraged to bring their own laptops to go through the installation process of Haskell and corresponding editors, especially if they haven't tried to install Haskell before or if they had problems with the installation. In any case the lab machines will have Haskell installed and students can also use these machines for the tutorial.

2 Install the Haskell Toolchain

If you have not installed Haskell yet, you can try to install it now. If you have problems we can try to help you.

The easiest way to install Haskell is via GHCup:

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

Follow the appropriate installation instructions depending on your operating system. With GHCup you get a comprehensive development environment for Haskell:

1. **GHC**: the *de facto* standard compiler for Haskell.
2. **Cabal**: a system for building and packaging Haskell libraries and programs.
3. **Stack**: an alternative to cabal-install. Stack uses the Cabal library but with a curated version of the Hackage repository called Stackage. Stack even downloads GHC for you and keeps it in an isolated location.
4. **HLS**: the official Haskell IDE support via the Language Server Protocol.

In our tutorials and assignments, we mainly use **Stack** to build a Haskell project.

3 Basic Steps in Haskell

In this tutorial, you are going to write your first (maybe not?) Haskell code.

3.1 Preamble

Download the project `tutorial1.zip` from Moodle and unzip it. Inside it is the standard structure of a Haskell project. We don't care about `tutorial1.cabal`, `stack.yaml`, and so on for the moment. What we are really interested in is all inside the `src` directory.

Before continuing, let's try to build this project first (yes, we can build it even we haven't written a single bit!). Now open the terminal, go to the project root and run `stack build`. If everything is OK as it should be, you will see the following message, though not exactly the same:

- tutorial1-0.1.0.0: unregistering ...
- tutorial1-0.1.0.0: configure
- Configuring tutorial1-0.1.0.0...
- tutorial1-0.1.0.0: build
- ...
- Registering tutorial1-0.1.0.0...

If you run `stack build` again, you will see nothing showing up. This is because the project hasn't changed from the last build. Stack is smart enough to detect it and not actually build it again.

`stack build` is one of the most important Stack commands that we use throughout the development of Haskell projects. We will see another important command shortly when we talk about testing.

Now we are ready to write some code! Go inside the `src` directory, and open the file `Tutorial1.hs` using your favourite editor. In the first line, you will see the module header as follows:

```
module Tutorial1 where
```

3.2 First Definitions in Haskell

In last week's lecture we have seen some basic Haskell definitions. For starters, let's take a look at a function that computes the absolute value of a number:

```
absolute :: Int -> Int
absolute x = if x < 0 then -x else x
```

Everything is an Expression in Haskell. Generally speaking, Haskell definitions have the following basic form:

- $name\ arg_1\ \cdots\ arg_n = expression$

Note that in the right side (the body of the definition) what you have is an expression. This is different from a conventional imperative language, where the body of a definition is usually a statement. In fact, there are no statements in Haskell, and in particular, the `if` construct in `absolute` is also an expression.

Question 1. Are both of the following Haskell programs valid?

```
nested_if x = if absolute x <= 10
              then x
              else error "Only numbers between [-10,10] allowed"

nested_if' x = if (if x < 0 then -x else x) <= 10
               then x
               else error "Only numbers between [-10,10] allowed"
```

Once you have thought about it, you can try these definitions on your Haskell file and see if they are accepted or not.

Question 2. Can you have nested `if` statements in Java, C or C++? For example, would this be valid in Java?

```
int f(int x) {
    if ((if (x < 0) -x; else x) > 10) return x; else return 0;
}
```

3.3 Parametric Polymorphism and Type Inference

We have seen that Haskell supports definitions with a type signature or without. When a definition does not have a signature, Haskell infers one and it is still able to check

whether some type errors exist or not. For example, for the definition:

```
newline s = s ++ "\n"
```

Haskell is able to infer the type: `String -> String`.

(**Note:** In Haskell strings are represented as lists of characters, i.e. `[Char]`. The operator `++` is a built-in function in Haskell that concatenates two lists.) For certain definitions, it appears as if there was not enough information to infer a type. For example, consider the definition:

```
id x = x
```

This is the definition of the identity function: the function that given some argument returns that argument unmodified. This is a perfectly valid definition, but what type should it have?

The answer is:

```
id :: a -> a
```

The function `id` is a **(parametrically) polymorphic** function. **Polymorphism** means that the definition works for multiple types; and this type of polymorphism is called **parametric** because it results from parametrizing over a type. In the type signature, `a` is a type parameter. In other words, it is a variable that can be replaced by some valid type (e.g. `Int`, `Char`, or `String`). Indeed, the `id` function can be applied to any type of values. Try the following in GHCi (run `stack ghci` in the terminal), and see what is the resulting type:

```
Tutorial1> id 3
Tutorial1> id 'c'
Tutorial1> id id
-- you may try instead ":t id id"
```

Question 3. Have you seen this form of polymorphism in other languages? Perhaps under a different name?

3.4 Pattern Matching

One feature that many functional languages support is pattern matching. Pattern matching plays a role similar to conditional expressions, allowing us to create definitions de-

pending on whether the input matches a certain pattern. For example, the function `hd` (to be read as "head") given a list returns the first element of the list:

```
hd :: [a] -> a
hd [] = error "cannot take the head of an empty list!"
hd (x:xs) = x
```

In this definition, the pattern `[]` denotes an empty list, whereas the pattern `(x:xs)` denotes a list where the first element (or the head) is `x` and the remainder of the list is `xs`. Note that instead of a single clause in the definition, there are now two clauses for two cases.

Detour: Test Test Test!

The comments in the source file are a bit, shall I say, verbose? For example, in the following code snippet, the comments tell the users how `tl` should work for some sample inputs. The best thing is, they also serve as unit tests for the function so that the compiler will help run those tests for us, and should some function fail the tests, it would let us know!

```
-- | The tail of a list
--
-- Examples:
--
-- >>> tl [1,2,3]
-- [2,3]
--
-- >>> tl [1]
-- []
--

t1 :: [a] -> [a]
t1 = error "TODO: question 4"
```

Now go to the terminal, run `stack test`. You should see the following

```
...
### Failure in .../Tutorial1.hs:22: expression `tl [1,2,3]'
expected: [2,3]
but got: *** Exception: TODO: question 4
...
Examples: 14 Tried: 10 Errors: 0 Failures: 10
```

It reads that we have 14 examples, the compiler tried 10 of them, and they *all* failed to pass the tests! (No surprise, we haven't started yet.) So here we go, let's make it right!

Question 4. Define a `tl` function that given a list, drops the first element and returns the remainder of the list. That is, the function should behave as follows for the sample inputs:

```
Tutorial1> tl [1,2,3]
[2,3]
Tutorial1> tl ['a','b']
['b']
```

Now if you run `stack test` again, chances are that you should see some improvements.

More Pattern Matching. Pattern matching can be used with different types. For example, here are two definitions with pattern matching on tuples and integers:

```
first :: (a,b) -> a
first (x,y) = x

isZero :: Int -> Bool
isZero 0 = True
isZero n = False
```

3.5 Recursion

In functional languages, mutable state is generally avoided and in the case of Haskell (which is purely functional), it is actually forbidden. So, how can we write many of the programs we are used to? In particular, how can we write programs that in a language like C would normally be written with some mutable state and some type of loop? For example:

```
int sum_array(int a[], int num_elements) {
    int sum = 0;
    for (int i = 0; i < num_elements; i++) {
        sum = sum + a[i];
    }
    return sum;
}
```

The answer is to use **recursive** functions. For example here is how to write a function that sums a list of integers:

```haskell
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

Question 5. The factorial sequence is (starting from 1):

1, 2, 6, 24, 120, ...

Implement a recursive function `factorial :: Int -> Int` that given a number returns the corresponding number in the factorial sequence.

Question 6. The Fibonacci sequence (index starting from 0) is:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Implement a recursive function `fibonacci :: Int -> Int` that given a number returns the corresponding number in the sequence.

Question 7. Write a function:

```haskell
mapList :: (a -> b) -> [a] -> [b]
```

that applies the function of type `a -> b` to every element of a list. For example:

```
Tutorial1> mapList absolute [4,-5,9,-7]
[4,5,9,7]
```

Question 8. Write a function that given a list of characters returns a list with the corresponding ASCII number of the character. Note that in Haskell, the function `ord`:

```haskell
ord :: Char -> Int
```

gives you the ASCII number of a character. To use it we need to add the following just after the module declaration (which is already been added for you):

```haskell
import Data.Char (ord)
```

Question 9. Write a function `filterList` that given a predicate and a list returns another list with only the elements that satisfy the predicate.

```haskell
filterList :: (a -> Bool) -> [a] -> [a]
```

For example, the following filters all the even numbers in a list (`even` is a built-in Haskell function):

```
Tutorial1> filterList even [1,2,3,4,5]
[2,4]
```

Question 10. In the `Prelude` library, there is a function `zip`:

```
zip :: [a] -> [b] -> [(a,b)]
```

that given two lists pairs together the elements in the same positions. For example:

```
Tutorial1 > zip [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
```

Write a definition `zipList` that implements the `zip` function.

Question 11. Define a function `zipSum`:

```
zipSum :: [Int] -> [Int] -> [Int]
```

that sums the elements of two lists at the same positions.

(Suggestion: You can define this function recursively, but a simpler solution can be found by combining some of the previous functions.)

3.6 User-Defined Datatypes

You may be wondering how we we can make our own types out of existing ones in Haskell. The short answer is: use datatype declaration. In fact, many built-in types in Haskell like Boolean and lists are actually defined using datatypes. For example, the built-in type `Bool` is defined in the standard library as follows:

```
data Bool = True | False
```

The keyword `data` introduces a new datatype. The part before `=` defines a new type, which in our case is `Bool`. The parts after `=` are data constructors. They tell us what values this type can have. The `|` symbol can be read as "or". So in the above example, `Bool` type can only have two values, i.e. either `True` or `False`. Note that both the type name and the data constructors must begin with capital letters.

A data constructor can take some parameters and produce a new value. In the `Bool` example, this is not the case since `True` and `False` take no parameters. Let us define a list datatype to illustrate the use of type parameters:

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

Here `a` is a type parameter, which can be used in the data constructors. Depending on what kind of values we want our own version of list type to hold, we can have different types of lists. Once the type parameter `a` is fixed to some concrete type, we can only have elements of that type in the list. For example, if we pass `Int` as the type parameter to `MyList`, we end up having a type `MyList Int`, and the two data constructors `Nil` and `Cons` have the following types correspondingly:

```
Nil :: MyList Int
Cons :: Int -> MyList Int
```

Note that the above types tell us that we can only make a list of integers, not a list of characters or some other type. So

```
Cons 1 (Cons 2 (Cons 3 Nil))
```

creates a list of integers, whose elements are 1, 2 and 3.

A `Maybe` datatype is defined as

```
data Maybe a = Nothing | Just a
```

Quite look like our `MyList` example. You can think of this datatype as representing computations which might "go wrong" by not returning a value, or "succeeded" in producing some value.

The `Maybe` datatype provides a way to make a safe version of a partial function. A partial function is simply a function that is undefined for some argument. For example, the built-in function `hd` is partial, because if we pass an empty list `[]` to it, it raises an exception, and makes our program crash. A safe version of `hd` won't do that:

```
safeHead :: [a] -> Maybe a
```

As can be seen from the type, it returns a value of `Maybe a`. So if we pass an empty list to `safeHead`, it return `Nothing`, otherwise, it wraps the head element of the list inside `Just`. Some examples below:

```
Tutorial1> safeHead []
Nothing
```

```
Tutorial1> safeHead [1,2,3]
Just 1
```

Question 12. Define the function `safeHead`.

Question 13. Define a `catMaybes` function, which takes a list of `Maybe`s and returns a list of all the `Just` values. For example:

```
Tutorial1> catMaybes [Just 1, Nothing, Just 2]
[1,2]
```