Lab 8

Advanced Functional Programming with Haskell and Standard ML

Goals

In this lab you will learn to:

1. Transition from Elm to using Haskell and SML
2. Understand the key differences between Elm, Haskell and SML
3. Use the unique features of Haskell and SML

Resources

Table 8.1: Haskell Resources

Resource	Link
Learn you a Haskell	`http://learnyouahaskell.com/chapters`
Haskell base library	`https://hackage.haskell.org/package/base-4.14.0.0`
Haskell `Data.List` module	`https://hackage.haskell.org/package/base-4.14.0.0/docs/Data-List.html`
Haskell `Data.Maybe` module	`https://hackage.haskell.org/package/base-4.14.0.0/docs/Data-Maybe.html`
Haskell `Data.String` module	`https://hackage.haskell.org/package/base-4.14.0.0/docs/Data-String.html`

Table 8.2: SML Resources

Resource	Link
Programming in Standard ML	`http://www.cs.cmu.edu/~rwh/isml/book.pdf`
Standard ML Basis library	`https://smlfamily.github.io/Basis/overview.html`
Standard ML `List` structure	`https://smlfamily.github.io/Basis/list.html`
Standard ML `Option` structure	`https://smlfamily.github.io/Basis/option.html`
Standard ML `String` structure	`https://smlfamily.github.io/Basis/string.html`

Preparations

Skim lectures #1 to #6 for quick recap of Haskell and SML syntax and features. Try to identify features of Haskell and SML that are not present in Elm.

8.1 Meet Standard ML

Standard ML is considered to be one of the first modern Functional Programming languages and influenced the design of the majority of modern Functional Programming languages available today, including Haskell, Elm, F#, Scala, OCaml and other modern languages like Swift and Rust.

One of the most unique aspects of Standard ML is that its semantics are *formally specified and verified*, which is a major advantage for writing theorem provers, compilers and formal verification programs.

A variety of implementations are available and are still maintained, including Standard ML of New Jersey, PolyML, MLTon and SML#. Newer implementations include WebML and SOSML.

From a pure Functional Programming point of view its main disadvantages are that it supports mutable references and exceptions.

8.2 Meet Haskell

Haskell is often considered a research experiment that escaped the labs: it was mainly designed for teaching and language design research, which is very clearly reflected in their motto:

> Avoid success at all costs!

Despite all this, it still gained enough popularity to become an industry-strength language, used by companies like Facebook and Microsoft.

Its main unique feature is *lazy evaluation* and it pioneered *type classes* (type-safe function overloading).

The main Haskell implementation today is the Glasgow Haskell Compiler (GHC), which also includes numerous (optional) *language extensions* to the Haskell 2010 standard.

As a note, Elm's design was inspired by Haskell, so Haskell's syntax should seem familiar.

8.3 Modules in Haskell

Question 8.3.1

Open the `Date.elm` file from the 3rd lab and `Date.hs`.
How many differences can you spot?

8.3.1 Exports in Haskell

Exports in Haskell work exactly as in Elm, with a minor syntax difference: instead of the `exposing` keyword we use `where` and place it *after* the list.

Table 8.3: Exports in Elm and Haskell

Export	Elm	Haskell
Export all symbols	`module Mod exposing (..)`	`module Mod where`
Export function	`module Mod exposing (func)`	`module Mod (func) where`
Export type without its variants	`module Mod exposing (Ty)`	`module Mod (Ty) where`
Export type with its variants	`module Mod exposing (Ty(..))`	`module Mod (Ty(..)) where`

8.3.2 Imports in Haskell

Table 8.4: Imports in Elm and Haskell

Import	Elm	Haskell
Import all symbols	`import Mod exposing (..)`	`import Mod`
Open import	`import Mod`	`import qualified Mod`
Qualified import	`import Mod as M`	`import qualified Mod as M`
Globally import a type without its variants	`import Mod exposing (Ty)`	`import Mod (Ty)`
Globally import a type with its variants	`import Mod exposing (Ty(..))`	`import Mod (Ty(..))`
Globally import a function	`import Mod exposing (func)`	`import Mod (func)`
Import all except a type or function	N/A	`import Mod hiding (Ty, func)`

8.3.3 Loading modules in GHCi (Haskell REPL)

Haskell REPL

```
Prelude> :l Date.hs
[1 of 1] Compiling Date             ( Date.hs, interpreted )
Ok, one module loaded.
*Date>
```

Note 8.3.1

When loading modules in the GHCi, visibility rules are **ignored** (i.e. all definitions are visible in the REPL)!

Note 8.3.2

When loading a module GHCi will also load all of its dependencies (imported modules) automatically.

8.4 Local declarations in Haskell with `where`

Besides `let ... in`, Haskell also has the `where` keyword for local declarations that are written *after* an expression:

Listing 8.4.1: Local declarations using `where` — Haskell code

```haskell
double a = d where d = a * a

heron a b c = sqrt (s * (s - a) * (s - b) * (s - c)) where
  s = (a + b + c) / 2

quadruple a = q where
  d = a * a
  q = d * d
```

8.5 Patterns in Haskell

Lets consider the (inefficient) Fibonacci function, first written using only `if` expressions:

Listing 8.5.1 of `Patterns.hs (fib)` — Haskell code

```haskell
fib n =
  if n == 0 then
    0
  else if n == 1 then
    1
  else
    fib (n-1) + fib (n-2)
```

We can also rewrite it using `case` expressions:

Listing 8.5.2 of `Patterns.hs (fibCase)` — Haskell code

```haskell
fibCase n =
  case n of
    0 -> 0
    1 -> 1
    n ->  fibCase (n-1) + fibCase (n-2)
```

8.5.1 Patterns in function definitions

Concept 8.5.1: Refutable and irrefutable patterns

Refutable - patterns that can fail to match (variants of a union type, literals).
Irrefutable - patterns that always match (record fields, data from a type with a single constructor, wildcard patterns, bindings).

Both of the previous methods were also available in Elm, but in Haskell we can also write function definitions with refutable patterns:

Listing 8.5.3 of `Patterns.hs (fibPat)` Haskell code

```haskell
fibPat 0 = 0
fibPat 1 = 1
fibPat n = fibPat (n-1) + fibPat (n-2)
```

You can think of this as writing the `case` expression directly in the function definition. This also works for multiple parameters:

Listing 8.5.4 of `Patterns.hs (safeDiv)` Haskell code

```haskell
safeDiv _ 0 = Nothing
safeDiv num d = Just (div num d)
```

8.5.2 Partially-defined functions

By allowing refutable patterns in function definitions and not checking for exhaustiveness, Haskell allows partially defined functions.

Listing 8.5.5 of `Patterns.hs (partialHead)` Haskell code

```haskell
partialHead (x:_) = x
```

Haskell REPL

```
> :l Patterns.hs
> partialHead [1, 2 ,3]
1
> partialHead []
*** Exception: <interactive>:1:1-21: Non-exhaustive patterns in function partialHead
```

Note 8.5.1: Rules for pattern matching and destructuring in Haskell

Variable names must be unique in pattern bindings!
The order of patterns is important!

- → The patterns are checked from top to bottom, until one pattern matches and that branch is chosen.
- → By default, Haskell won't warn you for redundant patterns!

By default, Haskell does not check for exhaustiveness! If no pattern matches you will get a runtime exception.

8.5.3 Pattern guards

In Haskell we also have *pattern guards*: expressions that can further refine pattern matching results.

Consider the `takeWhile` function, written with the pattern integrated into the function definition:

Listing 8.5.6 of `Patterns.hs (takeWhile)` Haskell code

```haskell
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs) =
  if p x then
    x : takeWhile p xs
  else
    []
```

We can use pattern guards to also skip the `if` expression, making the code even shorter:

Listing 8.5.7 of `Patterns.hs (takeWhileGuard)` Haskell code

```haskell
takeWhileGuard :: (a -> Bool) -> [a] -> [a]
takeWhileGuard _ [] = []
takeWhileGuard p (x:xs)
  | p x = x : takeWhile p xs
  | otherwise = []
```

Pattern guards may also be used with the patterns of `case` expressions:

Listing 8.5.8 of `Patterns.hs (checkDiv)` Haskell code

```haskell
checkDiv num d =
  case safeDiv num d of
    Nothing -> "Div by 0!"
    Just res
      | res == 0 -> "Smaller"
      | otherwise -> "Equal or Greater"
```

Note 8.5.2: Rules for pattern guards

There are 2 main rules for pattern guards:

1. We use `otherwise` as the wildcard ("catch-all") pattern.
   → By default, guards aren't checked for exhaustiveness either, so if no guard matches you will get a runtime exception.
2. The guard refinements *are part of the patterns*!
   → In the case of function definitions, we don't place the `=` after the line with the function definition, but after each guard.
   → In the case of `case` expressions, we don't place the `->` after the pattern, but after each guard.

Question 8.5.1

Is `otherwise` a language keyword? If not what is it?

8.6 List comprehensions and ranges in Haskell

8.6.1 Ranges

Haskell also has ranges which, as the name suggests, can be used to represent a range of values.

For example, we can enumerate all integers between 1 and 10:

Haskell REPL

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

We can also enumerate every second number between 0 and 15:

Haskell REPL

```
> [0,2..15]
[0,2,4,6,8,10,12,14]
```

This happens by specifying the first element, the second element according to the a rule you would like and final element.

Note 8.6.1

You can only specify rules that take into consideration the first 2 elements. So if you tried to obtain the powers of 2 with the following code it wouldn't compile: `[1, 2, 4..100]`

8.6.2 List comprehensions

The other powerful feature for working with lists in Haskell are *list comprehensions*: concise syntax for filtering and transforming lists.

For example consider the `filter` and `map` functions written using list comprehensions:

Listing 8.6.1 of `Lists.hs (filterComp, mapComp)` Haskell code

```haskell
filterComp p l = [x | x <- l, p x]
mapComp f l = [f x | x <- l]
```

We can identity 3 important components of a list comprehension:

$$[\ map\_fn \mid generator_1, \ldots, generator_n, filter\_exp_1, \ldots, filter\_exp_n\ ]$$

1. The expression that is used to create the elements of the list: map_fn
2. Generators: the expression that is used to draw items that will be used:

   $$generator_1, \ldots, generator_n$$

   where $generator_i$ has the form $binding_i \leftarrow generator_i$
3. Guards: expressions used to filter the generated items:

   $$filter\_exp_1, \ldots, filter\_exp_n$$

   where each $filter\_exp_i$ expression returns value of type `Bool`

List comprehensions are not limited to only one list! We can implement the cartesian product of 2 lists using list comprehensions in a very concise way:

Listing 8.6.2 of `Lists.hs (cartesianComp)` Haskell code

```haskell
cartesianComp lx ly = [(x, y) | x <- lx, y <- ly]
```

Haskell REPL

```
> :l Lists.hs
> cartesianComp [1..3] ['a'..'c']
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c'),(3,'a'),(3,'b'),(3,'c')]
```

8.7 Syntax sugar in Haskell

8.7.1 Infix Operators

In Haskell we can also define infix operators. Let's define an operator `??` for the `Maybe.withDefault` function in Elm which is `fromMaybe` in Haskell:

Listing 8.7.1 of Sugar.hs (??) Haskell code

```haskell
infixl 3 ??

(Just x) ?? _ = x
_ ?? def = def
```

First we declare the operator is infix and left associative with the `infixl` keyword and its precedence (3). Then we can write the normal function definition.

8.7.2 Infix call syntax

In Haskell we can call function that take 2 arguments with a special *infix call syntax*, by putting the name of the function between backticks (the ` symbol, to the left of `1` on your keyboard):

Haskell REPL

```
> div 10 2
5
> 10 `div` 2
5
```

8.7.3 Sections

In Haskell infix operators may also be partially applied, which are called sections:

Haskell REPL

```
> map (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
> filter (<4) [1..10]
[1, 2, 3]
```

8.8 Arbitrary precision integers

To define the non-tail recursive factorial function in GHCi, we need to write it between `:{` and `:}` since it is a multiline function.

```
Haskell REPL

Prelude> :{
Prelude| fact 0 = 1
Prelude| fact n = n * fact (n - 1)
Prelude| :}
```

Because Haskell has arbitrary precision integers, the only limiting factor for our computations is time and memory:

```
Haskell REPL

Prelude> fact 1000
40238726007709377354370243392300398571937486421071463254379991042993851239862902
05920442084869694048004799886101971960586316668729948085589013238296699445909974
24504087073759918823627727188732519779505950995276120874975462497043601418278094
64649629105639388743788648733711918104582578364784997701247663288983595573543251
31853239584630755574091142624174743493475534286465766116677973966688202912073791
43853719588249808126867838374559731746136085379534524221586593201928090878297308
43139284440328123155861103697680135730421616874760967587134831202547858932076716
91324484262361314125087802080002616831510273418279777047846358681701643650241536
91398281264810213092761244896359928705114964975419909342221566832572080821333186
11681155361583654698404670897560290095053761647584772842188967964624494516076535
34081989013854424879849599533191017233555566021394503997362807501378376153071277
61926849034352625200015888535147331611702103968175921510907788019393178114194545
25722386554146106289218796022383897147608850627686296714667469756291123408243920
81601537808898939645182632436716167621791689097799119037540312746222899880051954
44414282012187361745992642956581746628302955570299024324153181617210465832036786
90611726015878352075151628422554026517048330422614397428693306169089796848259012
54583271682264580665267699586526822728070757813918581788896522081643483448259932
66043367660176999612831860788386150279465955131156552036093988180612138558600301
43569452722420634463179746059468257310379008402443243846565724501440282188525247
09351906209290231364932734975655139587205596542287497740114133469627154228458623
77387538230483865688976461927383814900140767310446640259899490222221765904339901
88601856652648506179970235619389701786004081188972991831102117122984590164192106
88843871218556461249607987229085192968193723886426148396573822911231250241866493
53143970137428531926649875337218940694281434118520158014123344828015051399694290
15348307764456909907315243327828826986460278986432113908350621709500259738986355
42771967428222487575867657523442202075736305694988250879689281627538488633969099
59826280956121450994871701244516461260379029309120889086942028510640182154399457
15680594187274899809425474217358240106367740459574178516082923013535808184009699
63725242305608559037006242712434169090041536901059339838357779394109700277534720
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000
```

Exercise 8.8.1 *

Test in GHCi: `fact 10000` and `fact 100000`. What takes longer, calculating the number or printing all of its digits?

8.9 Function and value declarations in SML

To declare a constant in SML we can use the `val` keyword:

Listing 8.9.1: Declarations in SML SML code

```
val x: int = 2;
```

To declare a curried function we can use the `fun` keyword:

Listing 8.9.2: Declarations in SML SML code

```
fun factAcc (acc: int) (n: int): int =
  if n = 0 then
    acc
  else
    factAcc (n*acc) (n-1);
```

As you can see, in SML type annotations are embedded in the declarations of the function. But SML also has type inference so you might skip them alltogether:

Listing 8.9.3: Declarations in SML SML code

```
fun factAcc acc n =
  if n = 0 then
    acc
  else
    factAcc (n*acc) (n-1);
```

Or even annotate your function partially:

Listing 8.9.4: Declarations in SML SML code

```
fun factAcc acc n: int =
  if n = 0 then
    acc
  else
    factAcc (n*acc) (n-1);
```

8.9.1 Lambda expressions

To declare a lambda expression we use the `fn` keyword:

SML REPL

```
- List.map (fn x => x + 1) [1, 2, 3];
val it = [2, 3, 4]: int List.list;
```

8.10 Types in SML

In SML type variables are prefixed with the ' character and are in front of the generic types:

Listing 8.10.1: Type variables in SML SML code

```
fun len (l: 'a list): int =
  if l = [] then
    0
  else
    1 + len (tl l)
```

8.11 Patterns in SML

8.11.1 `case` expressions

In SML patterns in `case` expressions are separated by vertical bars `|`:

Listing 8.11.1: `case` expression in SML SML code

```
fun take (n: int) (l: 'a list): 'a list =
  if n <= 0 then
    []
  else
    case l of
      [] => []
    | x::xs => x :: take (n - 1) xs;
```

8.11.2 Patterns in function definitions

Similar to Haskell, we can also use patterns in function definitions:

Listing 8.11.2: Patterns in function definitions SML code

```
fun take _ [] = []
  | take n (x::xs) =
      if n <= 0 then
        []
      else
        x :: take (n - 1) xs;

fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n-1) + fib (n-2);
```

Note 8.11.1

In SML vertical bars are used to separate patterns, not as pattern guards!

8.11.3 Patterns in lambda expressions

Listing 8.11.3: `case` expression in SML SML code

```
fun emptyOrLen2Lists ls =
  List.filter (fn [] => true | [_, _] => true | _ => false) ls;
```

8.11.4 Partial functions in SML

Standard ML also allows the definition of partial functions, but it will issue a warning for non-exhaustive patterns.

SML REPL

```
- fun partialHead (x::xs) = x;
stdIn:1.6-1.29 Warning: match nonexhaustive
          x :: xs => ...

val partialHead = fn : 'a list -> 'a
```

8.12 Local declarations in SML

8.12.1 `let ... in ... end` expressions

`let ... in ... end` expressions work exactly as they do in Elm and Haskell, with the syntactical difference that we also have the **`end`** keyword:

Listing 8.12.1: `let ... in` expressions SML code

```
fun fibTail n =
  let
    fun helper f1 f2 0 = f1
      | helper f1 f2 n = helper f2 (f1+f2) (n-1);
  in
    helper 0 1 n
  end;
```

8.12.2 `local ... in ... end` expressions

`local ... in ... end` expressions allow to define local, helper functions before they're used:

Listing 8.12.2: `local ... in` expressions SML code

```
local
  fun helper f1 f2 0 = f1
    | helper f1 f2 n = helper f2 (f1 + f2) (n-1);
in
  fun fibTailLocal n = helper 0 1 n;
end;
```

Note 8.12.1

In the declaration part (between `let` and `in` and `local` and `in`) all values and functions must be declared by using the `val` or `fun` keywords as appropriate!
Don't forget the `end` keyword!

8.13 Review questions

Question 8.13.1 *

Enumerate:

- 2 differences between Elm and Haskell
- 2 features of Haskell that are not present in SML
- 2 features of SML that are not present in Haskell

8.14 Practice problems

8.14.1 Haskell

Exercise 8.14.1 *

Rewrite the `sudan` function from lab 1 using `where` in Haskell. The definition is repeated here for your convenience:

$$S(n,x,y) = \begin{cases} x+y & \text{if n = 0} \\ x & \text{if n > 0 and y=0} \\ S(n-1, S(n,x,y-1), y+S(n,x,y-1)) & \text{otherwise} \end{cases}$$

Hint:
Try to find a function that is called twice with the same parameters.

Exercise 8.14.2 *

Define an infix operator called `!&` in Haskell, which implements the logical function `nand` (`not and`):

$$x\ nand\ y = \begin{cases} false & \text{if x=y=true} \\ true & \text{otherwise} \end{cases}$$

Exercise 8.14.3 *

Implement the `safeHead :: [a] -> Maybe a` and `safeTail :: [a] -> Maybe [a]` functions in Haskell which return `Nothing` if the list is empty and the result wrapped in `Just` if it isn't empty.

```
                                                     Haskell REPL
> safeHead []
Nothing
> safeHead [1, 2]
Just 1
> safeTail []
Nothing
> safeTail [1, 2]
Just [2]
```

Exercise 8.14.4 *

Write a function called `average :: [Int] -> Float`, which calculates the average of a list of integers.

```
Haskell REPL

> average [1..10]
5.5
```

Hint:

1. Check the signature of the `(/)` operator using the `:t (/)` command in GHCi
2. Use the `fromIntegral` function to convert `Int`s to `Float`s

Exercise 8.14.5 *

Write a function called `countVowels` which counts the number of vowels in a string:

```
Haskell REPL

> countVowels "lalala"
3
> countVowels "psst"
0
```

Exercise 8.14.6 **

Write a function called `addBigs` which adds two big numbers given as lists of digits. E.g.:

```
Haskell REPL

> addBigs [1,3,9] [2,2,2]
[3,6,1]
```

Exercise 8.14.7 ***

To format the result of `fact 1000` in LaTeX, I had to break the number up in groups of 80 digits.
Write a function `breakToLines lineLen str` with the signature `breakToLines :: Int -> String -> [String]` that takes a string as input an returns a list of strings which have length at most `lineLen` characters.

```
Haskell REPL

Prelude> breakToLines 5 (show (fact 20))
["24329","02008","17664","0000"]
```

Use the `show` function to turn the result of `fact 1000` into a string and write a function `formatLines :: [String] -> String` that takes a list of string, places a newline character between each string and concatenates the list into one string.

```
Haskell REPL

Prelude> formatLines ["First", "Second"]
"First\nSecond"
```

8.14.2 SML

Exercise 8.14.8 *

Write a function called `countChar` which counts how many times a given string contains a character. E.g.:

SML REPL

```
- countChar #"a" "lalala";
val it = 3 : int;
- countChar #"a" "hey";
val it = 0 : int;
```

Exercise 8.14.9 **

Write a function called `addBigs` which adds two big numbers given as lists of digits. E.g.:

SML REPL

```
- addBigs [1,3,9] [2,2,2];
[3,6,1]
```

Exercise 8.14.10 ***

Write a function called `toposort` which topologically sorts a dag given as parameter. E.g.:

SML REPL

```
- val dag = [("a","b"),("b","d"),("d","f"), ("a","c"),("c","e"),("e","f")];
- toposort dag;
val it = ["a","c","e","b","d","f"] : string list;
```