Lecture 12: Functional programming

Haskell

January 10, 2018

Expressions, values, and types

Haskell is a **functional programming** language

⇒ computation = evaluation of **expressions** to yield values (the results of computations)

- The syntactic expressions for types are called type expressions.
- Every value has a **type**. A type is a set of values with common properties. Typical examples of type expressions and corresponding values are:
  - `Bool` (booleans): `True`, `False`
  - `Char` (characters): `'a'`, `'b'`, ...
  - `Integer` (integers): `0`, `-4767`, `2018`, ...
  - `[Integer]` (list of integers): `[]`, `[1,2,3]`, ...
  - `(Integer,Char)` (pairs of an integer and a character): `('a',196)`, `('+',-1024)`, ...
  - `Integer -> Integer` (functions from integers to integers): `\x -> x+1`

Values are “first-class”

This means that they can be

- passed as arguments to functions
- returned as results of function calls
- placed in data structures (e.g., list of functions)

Types are not first class: they are used to describe values. The association of a value with its type is called **typing**.

- Examples of typing declarations in Haskell:

```
    5    :: Integer
    'a'  :: Char
    inc  :: Integer -> Integer
[1,2,3]  :: [Integer]
('b',4)  :: (Char,Integer)
```

The intended reading of “`::`” is “has type.”

Functions are values

Most often, functions are defined by a series of *equations*.

Example (A simple numeric function)

```
inc :: Integer -> Integer
inc n = n+1
```

The definition of function `inc` consists of two declarations:

1. First line: declaration of the type signature of `inc`
2. Second line: an equation that defines the behavior of `inc`.

To indicate the evaluation of an expression e_1 to another expression or value e_2, we write $e_1 \Rightarrow e_2$. For example:

```
inc (inc 3)
```
$\Rightarrow$ 5

Haskell is statically typed

Haskell has a type system which, at compile time, detects if all expressions in the program ere well-typed.

Benefits of having a statically typed system:

- Many programming errors are detected at compile time
- It aids the used to reason about programs:
  - The user-supplied type signatures are a useful documentation about the behavior of programs; the type systems checks that they are correct
- It assists the compiler to generate more efficient code (e.g., no run-time type tags or tests are required)

Remarks:

- Not all errors are detected by the type system. For example, the expression `1/0` is well-typed but its evaluation yields an error at run-time.
- Haskell's type system allows us to avoid writing type signatures, because **it can infer** the correct types for us. However, writing type signatures is a very effective form of documentation.

Polymorphic types

A **polymorphic type** is a universally quantified type; it describes a family of types.

EXAMPLE 1: the type expression `[a]` is an abbreviation for the universally quantified type `(∀a)[a]` consisting of, for every type `a`, the type of lists of `a`:

- Lists of integers (e.g., `[1,2,3]`), lists of lists of characters (e.g., `[['a'],['b','c'],[]]`), etc., are members of this family of types.
- The expression `['a',1]` is not a member of this family of types: there is no single type that contains both `'a'` and `1`.

EXAMPLE 2: the type expression `a -> a` consists of, for every type `a`, the type of functions from `a` to `a`. For example, the type inferred for the function `id` defined by the equation

```
id x = x
```

is described by the type expression `a -> a`

Lists in Haskell

Lists are the most commonly used data structure in functional programming

- `[]` is the empty list.
  `x:xs` is the list with head `x` and tail `xs`.

  the operator "`:`" is right associative

- $[x_1, x_2, \ldots, x_n]$

  is a convenient abbreviation for the list

  $x_1 : x_2 : \ldots : x_n : []$

Definitions by pattern matching
Example: computing the length of a function

```
length :: [a] -> Integer
length []     = 0
length (x:xs) = 1+length xs
```

- `length` is a function defined by two equations.
- The left hand sides of the equations contain **patterns** such as `[]` and `x:xs`.
  `x` and `xs` are called **pattern variables**.
- When `length` is applied, these patterns are matched against the input argument of `length`:
  - `[]` only matches the empty list
  - `x:xs` matches any list with at least one element, binding `x` to the first element and `xs` to the rest of the list. Equations are tried top down, and for the first equation with successful match, the right side is evaluated and returned as result.

Polymorphic functions

- `length` is a **polymorphic function**: It can be applied to lists of elements of any type, e.g., `[Integer]`, `[Char]`, or `[[Bool]]`.

```
length [1,2,3]                                ⇒ 3
length['a','b','c']                           ⇒ 3
length [[True,False],[False],[False,False]] ⇒ 3
```

- Some other useful polymorphic functions:

```
head :: [a] -> a
head (x:xs) = x

tail : [a] -> [a]
tail (x:xs) = xs
```

Unlike `length`, these functions are not defined for all possible values of their argument:

A runtime error occurs when when they are applied to `[]`

Principal types
The Hindley-Milner type system

Given two polymorphic types T_1 and T_2, we say that T_1 is **more general** than T_2 if the set of values of type T_1 is larger than the set of values of type T_2. In general, T_1 is more general than T_2 if T_2 can be obtained from T_1 by a suitable substitution of type variables:

`[a]` is more general than `[Char]`. Type `[Char]` is obtained from `[a]` by the substitution $\{$`a` $\rightarrow$ `Char`$\}$

- `a -> b` is more general that `[a] -> a`. Type `[a] -> a` is obtained from `a -> b` by the substitution $\{$`a` $\rightarrow$ `[a]`, `b` $\rightarrow$ `a`$\}$

Haskell, ML, Miranda, and many other functional programming languages have a type system based on the **Hindley-Milner** type system, which has 2 important properties:

1. Every well–typed expression has a unique most general type, called its **principal type**
2. The principal type of a well-typed expression can be computed automatically.

EXAMPLES:

- The principal type of `head` is `[a] -> a`
- The principal type of `tail` is `[a] -> [a]`

User-defined types

Users can define their own types using `data` declarations. Every `data` declaration defines simultaneously two things:

- a **type constructor**, and
- one or more **data constructors** for the new type.

Examples (some are predefined):

- Enumerated types: they have finitely many nullary constructors

```
data Bool  = True | False
data Color = Red | Green | Blue | Indigo | Violet
```

- A polymorphic type

```
data Point a = Pt a a
```

- Some recursive types

```
data List a  = Null | Cons a (List a)
data BTree a = Leaf a | Node a (BTree a) (BTree a)
```

More about user-defined types

- The application of a type constructor yields a type; the application of a data constructor yields a value
- It is mandatory to start with uppercase letter the names of the **type constructors** (`Bool`, `Color`, `List`, `BTree`) and the names of the **data constructors** (`True`, `False`, `Red`, `Green`, `Blue`, `Indigo`, `Violet`, `Null`, `Cons`, `Leaf`, `Node`)
- The type system assigns corresponding types to data constructors, e.g.:
  `Cons :: a -> List a -> List a`
- Type constructors and data constructors are in separate namespaces. This implies that the same name can be used for both a type constructor and a data constructor. For example, we can define

  ```
  data Point a = Point a a
  ```

  Here, `Point` is both the name of a type constructor, and the name of a data constructor.
- For better readability, some predefined type constructors and data constructors have special syntax:
  - The function type constructor "`->`" and the data constructor "`:`" are like right-associative infix operators
  - The type constructors and data constructors for tuples have mixfix syntax

Recursive functions on recursive types
Example

Define the function `fringe` that takes as input a tree of type `BTree a` and returns the list of all elements of the tree, as seen by an inorder traversal of the tree. Remember the definition of polymorphic type `BTree`:

```
data BTree a = Leaf a | Node a (BTree a) (BTree a)
```

The definition of `fringe` is by recursion on the structure of `BTree`:

```
fringe :: BTree a -> [a]
fringe (Leaf x)              = [x]
fringe (Node x left right) = fringe left ++ [x] ++ fringe right
```

where "++" is the infix operator that concatenates two lists.

REMARK: By default, operators (like '++) haw lower binding occurrence than function calls. Therefore:

```
fringe left ++ [x] ++ fringe right
```

is parsed as

```
(fringe left) ++ [x] ++ (fringe right)
```

Type synonyms

A type synonym is a new name for an existing type. Type synonyms are defined using `type` declarations. For example:

```
type String  = [Char]
type Person  = (Name,Address)
type Name    = String
type Address = None | Addr String
```

`Person` is synonym with `(String,Address)`, therefore `Person->Name` is equivalent to `(String,Address)->String`

Remarks

- Type names improve readability of programs by being more mnemonic.
- We can even give names to polymorphic types, e.g.:

```
type AssocList a b = [(a,b)]
```

List comprehensions

Syntax to to create lists inspired from mathematical notation. For example:

- `[f x | x<-xs]`
  creates the list of all `f x` where `x` is drawn from list `xs`. The subexpression `x<-xs` is a **generator**; we can create lists with many generators, e.g., we can compute the cartesian product of two lists `xs` and `ys`:

  ```
  [(x,y) | x<-xs, y<-ys]
  ```

  The elems. are selected as if the generators were nested from left to right, e.g.:

  `[(x,y)|x<-[1,2],y<-[3,4]]` ⇒ `[(1,3),(1,4),(2,3),(2,4)]`

- Besides generators, guards are also permitted. A **guard** is a boolean expression that places constraints on the elements generated. For example:

  `[(x,y)|x<-[1,2,3],y<-[1,2,3],x<y]` ⇒ `[(1,2),(1,3),(2,3)]`

  Another example: a concise definition of quicksort:

  ```
  quicksort []     =  0
  quicksort (x:xs) =  quicksort [y|y<-xs,y<x]
                   ++ [x]
                   ++ quicksort [y|y<-xs,y>=x]
  ```

Arithmetic sequences. Strings

Haskell has special syntax for arithmetic sequences. For example:

```
[1..10] ⇒ [1,2,3,4,5,6,7,8,9,10]
[1,3..9] ⇒ [1,3,4,5,6,7,8,9]
[1,3..] ⇒ [1,3,4,5,6,7,...]   -- a potentially infinite list
```

`String` is a type synonym for `[Char]`, therefore we can manipulate strings like lists. For example:

```
"Hello " ++ "World!" ⇒ "Hello World!"
```

Higher-order functions
Currying

The following definitions of `f` are equivalent:

`f x y = (x,y)` or `f = \x y->(x,y)` or `f = \x->\y->(x,y)`

Thus, `f x y` is an abbreviation for `(f x) y` and we can pass the input arguments to `f` one-by-one. The type system calculates the principal type of `f`, which is `a->b->(a,b)`
In particular, the value of `f "a" True` is that of `(f "a") True`:

`f "a"` $\Rightarrow$ `\y->("a",y)`

Note that `f "a"` evaluates to a function whose principal type is `b->([Char],b)`. Therefore, `(f "a") True` is a well-typed expression with principal type `([Char],Bool)`, and its value is

`(f "a") True` $\Rightarrow$ `(\y->("a",y)) True` $\Rightarrow$ `("a" True)`

Infix operators

An infix operator `op` is like a binary function, but we write `x op y` instead of `op x y`
Infix operators must consist entirely of non-alphanumeric "symbols", like `+-` or `+.+`.

- They are defined by equations, just like ordinary functions.

Example

```
(+-) :: Integer->Integer->Integer
x +- y = 2*x-3*y

  5 +- 3 ⇒ 1
```

There are several useful predefined infix operators. For example:

- Left-associative:
  - the arithmetic operators `(+)`, `(-)`, `(*)`, `(/)`
- Right associative:
  - list concatenation `(++)`
  - function composition `(.)`

Infix operators
Sections

Sections are partial applications of an infix operator. For example, the infix operator + has three sections (the parentheses are mandatory):

```
(x+)    ≡ \y ->x+y
(+y)    ≡ \x -> x+y
(+)     ≡ \x y -> x+y
```

Thus `(x+) y ≡ x (+y) ≡ (+) x y ≡ x+y`

Remarks

- The syntax `(op)` coerces an infix operator `op` into a binary function.
- Also, the syntax `` `fct` `` coerces a binary function `fct` into an infix operator:

  `fct x y ≡ x `fct` y`

Laziness

In Haskell, a definition `x = expr` is lazy: it defines `x` as `expr`.

- Only when some parts of `x` are needed, will `expr` be evaluated, and only until the needed parts become available.

In contrast, `x = expr` in a strict language is an assignment interpreted as follows: "compute the value of `expr` and store it in `x`."

Examples

- `bot = bot` is a valid definition. Any attempt to evaluate `bot` yields a nonterminating computation. Abstractly, it is assumed that
  - all nonterminating computations, and
  - evaluations of expressions that yield runtime errrors (e.g., `1/0`)

  compute an **error value** $\perp$ which can have any type `a`.
- Consider the function defined by
  `const1 x = 1`
  Then `const1 expr` $\Rightarrow$ `1` even if `expr` $\Rightarrow \perp$
  `expr` is not evaluated because it's value is not needed.

Infinite data structures

Functions and data constructors are lazy: they do not evaluate their arguments, unless their values are needed.

⇒ Data constructors can be used to define and work with conceptually infinite data structures:

```
ones         = num : ones
numsFrom n = n : numsFrom (n+1)
squares     = map (^2) (numsFrom 0)
fib          = 1 : 1 : [a+b | (a,b)<-zip fib (tail fib)]
```

where

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

zip :: [a] -> [b] -> [(a,b)]
zip [] []           = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Patterns

Patterns are expressions built only with data constructors and variables, such that no variable occurs twice in a pattern.

- They are used to describe the shape of data we want to process
- The matching of a pattern against an expression may succeed or fail:
  - when matching succeeds, the pattern variables are bound to the parts of the input which they match.
- Patterns that never fail to match are called irrefutable. There are three kinds of irrefutable patterns:
  - variables: they match and get bound to any input expression. The variables in a pattern are also called **formal parameters**.
  - the wild-card pattern _ : it matches any expression, but yields no binding
  - as-patterns (see next slide)

  The other kinds of patterns are refutable.

More about patterns

Wild-cards simplify the equational specification of some functions. For example:

```
head (x:_)  = x
tail (_:xs) = xs
```

An **as-pattern** is a mechanism used to name a pattern for use in the right-hand side of an equation, in order to avoid reconstructing a part of the matched expression. For example:

```
f (x:xs) = x:x:xs
```

Note that `x:xs` appears in both in the left- and right-hand side. This means that we waste time and memory space reconstructing the input argument of `f` in the right-hand side. We can avoid by using the as-pattern `x@(x:xs)` as follows:

```
f s@(x:xs) = x:s
```

Pattern matching
Semantics

Pattern matching is triggered by an attempt to apply a function defined by one or more equations. It can **succeed**, **fail**, or **diverge**.

- A successful match binds the formal parameters in the pattern.
- Divergence occurs when the value needed by the pattern contains $\perp$

Pattern matching is performed **top-down** and **left-to-right**:

- Failure of a pattern anywhere in one equation results in failure of the whole equation, and the next equation is then tried.
- If all equations fail, the value of the function application is $\perp$, and results in a run-time error.

Some illustrated examples

Consider the functions `take` and `take1` defined by the same equations but in a different order:

```
take 0 _       = []
take _ []      = []
take n (x:xs) = x : take (n-1) xs

take1 _ []      = []
take1 0 _       = []
take1 n (x:xs) = x : take1 (n-1) xs
```

Note that

```
take 0 bot  ⇒ []
take1 0 bot ⇒ ⊥

take bot 0  ⇒ ⊥
take1 bot 0 ⇒ []
```

Guards

The top-level patterns of equational definitions may also have a boolean guard that constrains the success of applying an equation. For example, the following is an abstract version of the function that returns a number's sign:

```
sign x | x > 0 =  1
       | x = 0 =  0
       | x < 0 = -1
```

This example shows a situation where a sequence of guards was provided for the same pattern:

- Like equations, they are tried top-down, and the first guard that evaluates to `True` results in a successful match.

Case expressions

Functions defined by equations with different patterns provide a way to decide what to compute based on the structural properties of a value.

Sometimes, we want to decide what to compute without defining a new function.

Haskell's case expressions solve this problem: Instead of defining

```
f p11 ... p1k = e1
...
f pn1 ... pnk = en
```

and calling `f` u_1 `...` u_k to compute a result, we can compute the same result by evaluating

```
case (u1 ... uk) of (p11 ... p1k) -> e1
                    ...
                    (pn1 ... pnk) -> en
```

For example, the function `take :: Integer->[a]->[a]` can also be defined by

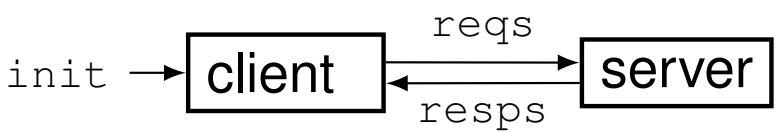
```
take m ys = case (m,ys) of (0,_)     -> []
                           (_,[])    -> []
                           (n,x:xs) -> x : take (n-1) xs
```

Lazy patterns

A lazy pattern is `~pat` where `pat` is a normal pattern.

- Matching `v` against `~pat` always succeeds. It is like matching `v` against `pat`, but with the following difference: the variables in `pat` are bound to parts of `v` that would result if `v` successfully matches `pat`, and $\perp$ otherwise.

Illustrated example: A simulation of a client-server interaction through conceptually infinite lists of data (a.k.a. streams):



client is a process that behaves as follows:

> Initially, it sends to the server the element `init` received from somewhere else. Afterwards, whenever it receives a response `resp` from the server, it processes it by computing `next resp`, and sends `next resp` to the server.

server is a process that behaves as follows:

> Whenever it receives a request `req` from the client, it processes it by computing `process req`, and sends `process req` back to the client.

Lazy patterns
Example (continued): simulation of a client-server interaction

To be concrete, assume `init`, `next`, and `process` are defined by

```
init         = 0
next   resp = resp
process req = req+1
```

⇒ the server processes a stream of requests as follows:

```
server req : reqs = (process req) : server reqs
```

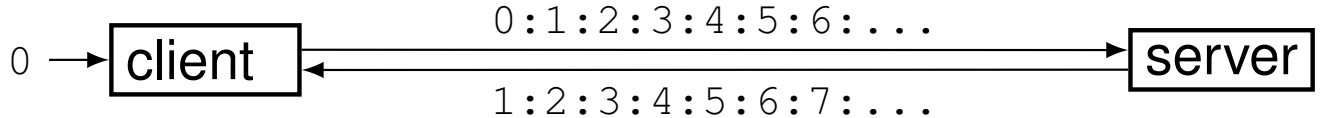
and the client handles an initial value and a stream of responses as follows:

```
client init  (resp : resps) = init : client (next resp) resps
```

If we define the streams `requests` and `responses` by mutual recursion:

```
requests  = client init responses
responses = server requests
```

where `init` is `0`, then we expect to have `requests` ⇒ `0:1:2:3:4:5:6:...`
`responses` ⇒ `1:2:3:4:5:6:7:...`



Unfortunately, there is a serious problem with the definition of `client` (see next slide)

Lazy patterns
Example (continued): simulation of a client-server interaction

```
requests  = client 0 responses
responses = server requests
```

where

```
client init (resp : resps) = init : client (next resp) resps
```

With this definition, the function call

```
client 0 responses
```

can not instantiate `requests` because the sub-pattern `resp:resps` fails to match `responses` (which is uninstantiated)

We can fix this problem by making the subpattern `resp:resps` lazy. Th revised (and working) definition of `client` is

```
client init ~(resp : resps) = init : client (next resp) resps
```

Pattern bindings are lazy

Consider the following definition of the stream of Fibonacci numbers:

```
fib@(1:tfib) = 1 : 1 : [a+b | (a,b)<-zip fib tfib]
```

Such an equation is a pattern binding because it is a top-level equation in which the entire left-hand side is a pattern.

Pattern bindings have an implicit `~` in front of them. This is the reason why this definition of `fib` works as expected.

`let` expressions

General syntax:

```
let local_declarations
in expr
```

where *local_declarations* are visible only in *expr*, and can be: type signatures, function bindings, and pattern bindings. The local declarations can be mutually recursive.

Example

```
let y   = a*b          -- pattern binding
    f x = (x+y)/y      -- function binding
in f c + f d
```

`where` clauses

A `where` clause introduces bindings visible only in the guards and right side of a guarded equation.

Example

```
f x y | y>z = ...
      | y==z = ...
      | y<z  = ...
  where z =  x*x
```

Ad-hoc polymorphism

Ad-hoc polymorphism, or overloading, allows to use the same literals (e.g., `1`, `2`, etc.), operators (e.g., `+` or `==`) to represent different things. For example:

- The literals `1`, `2` can represent both fixed and arbitrary precision integers
- `+` can be used for many purposes: to add integers, to add floating-point numbers, etc.
- `==` can be used to compare many kinds of elements: numbers, strings, characters, etc.

In Haskell, add-hoc polymorphism is defined with type classes.

References

- P. Hudak, J. Peterson, J. Fasel: A Gentle Introduction to Haskell.

  `https://www.haskell.org/tutorial/`