Introduction to Programming, Aug-Dec 2006

Lecture 2, Thursday 10 Aug 2006

Multiple definitions

Haskell does not limit us to a single definition for a function. We can give multiple definitions which are scanned from top to bottom. The first definition that matches is used to compute the value of the output. For instance, here is an alternative definition of `xor`.

```
xor :: Bool -> Bool -> Bool
xor True False = True
xor False True = True
xor b1 b2 = False
```

When does a function invocation match a definition? We have to check that it matches for each argument. If the definition has a variable for an argument, then any value supplied when invoking the function matches on that argument and the value supplied is uniformly substituted for the variable throughout the definition. On the other hand, if the definition has a constant value for an argument, the value supplied when invoking the function must match precisely.

For instance, in the revised definition of `xor`, if we invoke the function as `xor False True`, the first definition does not match, but the second one does. If we invoke the function as `xor True True`, the first two definitions both fail to match and we end up using the third one.

We can use multiple definitions to define a function inductively. For instance, here is a definition of the function *factorial*.

```
factorial :: Int -> Int -> Int
factorial 0 = 1
factorial n = n*(factorial (n-1))
```

If we write, for instance, `factorial 3`, then only the second definition matches, leaving us with the expression `3*(factorial 2)`, after uniformly substituting `3` for `n` and simplifying `(3-1)` to `2`. We use the second definition two more times to get `3*(2*(factorial 1))` and then `3*(2*(1*(factorial 0)))`. Now, the first definition matches, and we get `3*(2*(1*(1)))` which Haskell can evaluate using its built-in rules for `*` to return `6`.

Notice that there is no guarantee that an inductive definition in Haskell is correct, nor that it terminates on all inputs. Reflect, for instance, on what would happen if we invoked our function as `factorial (-1)`.

Observe the bracketing in the second defintion above. We write `n*(factorial (n-1))`. This says we should compute `(n-1)`, then feed this to `factorial` and multiply the result by `n`. If, instead, we write `n*(factorial n-1)`, Haskell would interpret this as `n*((factorial n)-1)`—in other words, feed `n` to factorial, subtract `1` from the result and then multiply by `n`. For arithmetic and relational expressions, the normal precedence rules of arithmetic apply, so an unbracketed expression such as `x <= 5 || y > 6` would be implicitly bracketed correctly as `(x <= 5) || (y > 6)`. However, function application binds more tightly than arithmetic operators, so `factorial n-1` is interpreted as `(factorial n)-1` rather than `factorial (n-1)`.

Function definitions with guards

Often, a function definition applies only if certain conditions are satisfied by the values of the inputs. Here is an example of how to define `factorial` to work with negative inputs. If the input is negative, we negate it and invoke `factorial` on the corresponding positive quantity.

```
factorial :: Int -> Int

factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

In this version of `factorial` , the second definition has two options depending on the value of `n`. If `n < 0`, the first definition applies. If `n > 0`, the second definition applies. These conditions are called *guards*, since they restrict entry to the definition that follows. Each guarded definition is signalled using `|`. Notice that lines beginning with `|` are indented. This tells Haskell that these lines are continuations of the current definition.

Observe that we can combine definitions of different types. In this example, the first definition, `factorial 0` is a simple expression while the second defintion is a conditional one.

The guards in a conditional definition are scanned from top to bottom. They may overlap, in which case the definition that is used is the one corresponding to the first guard that is satisfied. For instance, we could write:

```
factorial :: Int -> Int

factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
  | n > 0 = n * (factorial (n-1))
```

Now, `factorial 2` would match the guard `n > 1` while `factorial 1` would match the guard `n > 0`.

The guards in a conditional defintion may also not cover all cases. For instance, suppose we write:

```
factorial :: Int -> Int

factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
factorial 1 = 1
```

Now, the invocation `factorial 1` matches neither guard and falls through (fortunately) to the third definition. If we had not supplied the third definition, any invocation other than `factorial 0` would eventually have tried to evaluate `factorial 1`, for which no match would have been found, leading to the Haskell interpreter printing an error message like the following:

```
Program error: pattern match failure: factorial 1
```

Often, we do want to catch all leftover cases in the last guard. Rather than tediously specify the options that have been left out, we can use the word `otherwise`, as in the following definition of `xor`:

```
xor :: Bool -> Bool -> Bool

xor b1 b2
  | b1 && not(b2) = True
  | not(b1) && b2 = True
  | otherwise     = False
```

In this definition, note that since `b1` and `b2` are of type `Bool`, we can directly write `b1 && not(b2)` instead of the more explicit version `b1 == True && b2 == False`.

More on pattern matching

When we match a function invocation with a defintion involving variables, the variables are uniformly substituted by the values supplied. However, each input variable in the function definition would be distinct. Consider the following function, which checks if both its inputs are equal:

```
isequal :: Int -> Int -> Bool
isequal x y = (x == y)
```

It is tempting to try and rewrite this function as follows:

```
isequal :: Int -> Int -> Bool
isequal x x = True
isequal x y = False
```

The idea would be that the first definition implicitly checks whether both arguments are equal by forcing them to both match `x` and hence match each other. However, this is illegal in Haskell: each variable on the left hand side of a definition should be distinct.

Sometimes, an argument is not used on the right hand side of a definition. Consider the following definition that computes `x^n`.

```
power :: Float -> Int -> Float

power x 0 = 1.0
power x n | n > 0 = x * (power x (n-1)
```

Here, the value of `x^0` is `1.0` for all values of `x`. In such a situation, we can use a special variable `_` that matches any argument but cannot be used on the right hand side of a definition.

```
power :: Float -> Int -> Float

power _ 0 = 1.0
power x n | n > 0 = x * (power x (n-1)
```

Unlike normal variables, we can use more than one copy of `_` in a definition, since the corresponding value cannot be used on the righthand side in any case. As an example, here is a function that checks if at least two of its three `Bool` arguments are `True`.

```
twoofthree :: Bool -> Bool -> Bool -> Bool
twoofthree True True _ = True
twoofthree True _ True = True
twoofthree _ True True = True
twoofthree _ _ _  = False
```

How Haskell "computes"

Computation in Haskell is like simplifying expressions in algebra. Relatively early in school, we learn that $(a + b)^2$ is $a^2 + 2ab + b^2$. This means that wherever we see $(x + y)^2$ in an expression, we can replace it by $x^2 + 2xy + y^2$.

In the same way, Haskell computes by rewriting expressions using functions and operators. We say *rewriting* rather than *simpliyfing* because it is not clear, sometimes, that the rewritten expression is "simpler" than the original one!

To begin with, Haskell has rewriting rules for operations on built-in types. For instance, the fact that $6 + 2$ is 8 is embedded in a Haskell rewriting rule that says that `6+2` can be rewritten as `8`. In the same way, `True && False` can be rewritten to `False`, etc.

In addition to the builtin rules, the function definitions that we supply are also used for rewriting. For instance, given the following definition of `factorial`

```
factorial :: Int -> Int -> Int
factorial 0 = 1
factorial n = n*(factorial (n-1))
```

here is how "factorial 3" would be evaluated. In the following, we use $\leadsto$ to denote *rewrite to*:

```
factorial 3  ⇝  3 * (factorial (3-1))
             ⇝  3 * (factorial (2))
             ⇝  3 * (2 * factorial (2-1))
             ⇝  3 * (2 * factorial (1))
             ⇝  3 * (2 * (1 * factorial (1-1)))
             ⇝  3 * (2 * (1 * factorial (0)))
             ⇝  3 * (2 * (1 * 1))
             ⇝  3 * (2 * 1)
             ⇝  3 * 2
             ⇝  6
```

When rewriting expressions, brackets may be opened up to change the order of evaluation. Sometimes, more than one rewriting path may be available. For instance, we could have completed the computation above as follows.

```
factorial 3              ⇝ 3 * (factorial (3-1))
                          ⇝ 3 * (factorial (2))
                        ⇝ 3 * (2 * factorial (2-1))
              ⇝ (3 * 2) * (factorial (2-1)) <== New expression
                         ⇝ 6 * (factorial (2-1)))
                          ⇝ 6 * (factorial (1))
                        ⇝ 6 * (1 * factorial (1-1))
                         ⇝ 6 * (1 * factorial (0))
                              ⇝ 6 * (1 * 1)
                                 ⇝ 6 * 1
                                   ⇝ 6
```

In Haskell, the "result" of a computation is an expression that cannot be further simplified. In general, it is guaranteed that any path we follow leads to the same "result", if a "result" is found. It could be that one choice of simplification could yield a result while another may not. For instance, using our definition of `power`

```
power :: Float -> Int -> Float
power _ 0 = 1.0
power x n | n > 0 = x * (power x (n-1)
```

we could consider the expression `power (8.0/0.0) 0`.

Using the first rule, this reduces as

```
power (8.0/0.0) 0 ⇝ 1.0
```

However, if we first try to simplify `(8.0/0.0)`, we get an expression without a value so, in a sense, we have

```
power (8.0/0.0) 0 ⇝ Error
```

Alternatively, we could even try to evaluate an expression such as

```
power (1.0 * factorial (-1)) 0
```

where the first rule for `power` yields the result `1.0` while repeatedly trying to simplify the argument `(1.0 * factorial (-1))` will go into an unending sequence of simplifications yielding no result.

Haskell uses a form of simplification that is called *lazy*—it does not simplify the argument to a function until the value of the argument is actually needed in the evaluation of the function. In particular, Haskell would evaluate both the expressions above to `1.0`. We will examine the consequences of having such a lazy evaluation strategy at a later stage in the course.

Lists

Suppose we want a function that finds the maximum of all values from a collection. We cannot use an individual variable to represent each value in the collection because when we write our function definition we have to fix the number of variables we use, which limits our function to work only with collections that have exactly that many variables.

Instead, we need a way to collectively associate a group of values with a variable. In Haskell, the most basic way of collecting a group of values is to form a list. A list is a sequence of values of a fixed type and is written within square brackets separated by commas. Thus, `[1,2,3,1]` is a list of `Int`, while `[True,False,True]` is a list of `Bool`. The underlying type of a list must be uniform: we cannot write lists such as `[1,2,True]` or `[3.0,'a']`. A list of underlying type `T` has type `[T]`. Thus, `[1,2,3,1]` is of type `[Int]`, `[True,False,True]` is of type `[Bool]`, ...

Lists can be nested: we can have lists of lists. For instance, `[[1,2],[3],[4,4]]` is a list each of whose members is a list of `Int`, so the type of this list is `[[Int]]`.

The empty list is uniformly denoted `[]` for all list types.

Internal representation of lists

Internally, Haskell builds lists incrementally, one element at a time, starting with the empty list. This incremental building can be done from left to right (each new element is tagged on at the end of the current list) or from right to left (each new element is tagged on at the beginning of the current list). For historical reasons, Haskell chooses the latter, so all lists are built up right to left, starting with the empty list.

The basic listbuilding operator, denoted `:`, takes an element and a list and returns a new list. For instance `1:[2,3,4]` returns `[1,2,3,4]`. As mentioned earlier, all lists in Haskell are built up right to left, starting with the empty list. So, internally the list `[1,2,3,4]` is actually `1:(2:(3:(4:[])))`. We always bracket the binary operator `:` from right to left, so we can unambiguously leave out the brackets and write `[1,2,3,4]` as `1:2:3:4:[]`. It is important to note that all the human readable forms of a list `[x1,x2,x3,...,xn]` are internally represented canonically as `x1:x2:x3:...:xn:[]`. Thus, there is no difference between the lists `[1,2,3]`, `1:[2,3]`, `1:2:[3]` and `1:2:3:[]`.

Defining functions on lists

Most functions on lists are defined by induction on the structure of the list. The base case specifies a value for the empty list. The inductive case specifies a way to combine the leftmost element with an inductive evaluation of the function on the rest of the list. The functions `head` and `tail` return the first element and the rest of the list for all nonempty lists. These functions are undefined for the empty list. We can use `head` and `tail` in our inductive definitions.

Here is a function that computes the length of a list of `Int`.

```
length :: [Int] -> Int

length [] = 0
length l  = 1 + (length (tail l))
```

Notice that if the second definition matches, we know that `l` is nonempty, so `tail l` retuns a valid value.

In general, the inductive step in a list based computation will use both the head and the tail of the list to build up the final value. Here is a function that computes the sum of the elements of a list of `Int`.

```
sum :: [Int] -> Int

sum [] = 0
sum l  = (head l) + (sum (tail l))
```