Getting Started With ...
Haskell for Knowledge Representation

Jan van Eijck

jve@cwi.nl

December 19, 2005

Abstract

The purpose of this lecture is to give a lightning introduction to the functional programming language Haskell, and to make preparations for using Haskell for knowledge representation.

Learning Something New: Key ingredients

New Facts You will learn a few facts about how functional programs are written.

New Skills The main focus of this lecture.

- skills in (functional) computation, in learning to think functionally
- skills in representation, in getting from definitions to programs, in 'seeing' the program hidden in a definition.
- skills in working with 'the stuff of knowledge representation'.

Attitude The most important thing. But how do you acquire it? Once you have acquired the correct attitude you can learn to do anything.

Using the Hugs Haskell Interpreter

```
[jve@pc4 lai0506]$ hugs
___   __ __  __  ____   ___      ----------------------------------------
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: March 2005     ----------------------------------------

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs.Base>
```

http://haskell.org/hugs

Using the GHCI Haskell Interpreter

```
[jve@pc4 lai0506]$ ghci
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.4.1, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base-1.0 ... linking ... done.
Prelude>
```

http://www.haskell.org/ghc/

Haskell

These slides form a literate program. The text you are reading is the documentation. The actual code is the part typeset in frames. This is how the code begins:

```
module LAI9

where
import List
import Char
```

This declares a module and imports two other modules. The code of the module consists of the text in frames.

Loading the module

```
[jve@pc4 lai0506]$ ghci
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.4.1, f
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base-1.0 ... linking ... done.
Prelude> :l LAI9
Compiling LAI9             ( LAI9.lhs, interpreted )
Ok, modules loaded: LAI9.
*LAI9>
```

About Haskell

Haskell was named after the logician Haskell B. Curry. Curry, together with Alonzo Church, laid the foundations of functional computation in the era BC (Before the Computer), around 1940.

Haskell is a functional programming language, and a member of the Lisp family. Others family members are Scheme, ML, Occam, Clean. Haskell98 is intended as a standard for lazy functional programming.

With Haskell, the step from formal definition to program is particularly easy. This presupposes, of course, that you are at ease with formal definitions.

Our reason for combining training in reasoning with an introduction to functional programming is that your programming needs will provide motivation for improving your reasoning skills.

Implementation of a Prime Number Test

```
ld n = ldf 2 n

divides d n = rem n d == 0

ldf k n | divides k n = k
        | k^2 > n     = n
        | otherwise   = ldf (k+1) n

prime n | n < 1     = error "not a positive integer"
        | n == 1    = False
        | otherwise = ld n == n
```

Trying it out

```
somePrimes    = filter prime [1..1000]

primesUntil n = filter prime [1..n]

allPrimes     = filter prime [1..]
```

More on the `filter` function below.

```
LAI9> primesUntil 50
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
LAI9>
```

Type Declarations and Function Definitions

The truth values true and false are rendered in Haskell as `True` and `False`, respectively. The type of a truth value is called `Bool`.

All function definitions are typed: in a type declaration we indicate the type of the argument or arguments and the type of the value. A function `foo` that takes an integer as its first argument, and an integer as its second argument and yields a truth value has type

`Integer -> Integer -> Bool`.

Here is a type declaration for such a function, together with the actual definition:

```
divides :: Integer -> Integer -> Bool
divides m n = rem n m == 0
```

The type `Integer -> Integer -> Bool` should be read as `Integer -> (Integer -> Bool)`.

A type of the form `a -> b` classifies a procedure that takes an argument of type `a` to produce a result of type `b`.

Thus, `divides` takes an argument of type `Integer` and produces a result of type `Integer -> Bool`.

The result of applying `divides` to an integer is a function that takes an argument of type `Integer`, and produces a result of type `Bool`.

```
Main> :t divides 5
divides 5 :: Integer -> Bool
Main> :t divides 5 7
divides 5 7 :: Bool
Main>
```

Lambda Abstraction

Take the statement Diana loves Charles. By means of abstraction, we can get all kinds of properties and relations from this statement:

- 'loving Charles'
- 'being loved by Diana'
- 'loving'
- 'being loved by'

This works as follows. We replace the element that we abstract over by a variable, and we bind that variable by means of a lambda operator.

Lambda Abstraction – 2

Like this:

- '$\lambda x.\ x$ loves Charles' expresses 'loving Charles'.
- '$\lambda x.$ Diana loves x' expresses 'being loved by Diana'.
- '$\lambda x \lambda y.\ x$ loves y' expresses 'loving'.
- '$\lambda y \lambda x.\ x$ loves y' expresses 'being loved by'.

Lambda Abstraction – 3

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

The intention is that variabele `x` stands proxy for a number of type `Int`. The result, the squared number, also has type `Int`. The function `sqr` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`. This is precisely what the type-indication `Int -> Int` expresses.

List processing in Haskell

`Integer` is the type of arbitrary precision integers, `Int` the type of fixed precision integers.

`[Integer]` is the type of lists of `Integers`, `[Int]` the type of lists of `Ints`.

Here is a function that gives the minimum of a list of integers:

```
mnmInt :: [Int] -> Int
mnmInt [] = error "empty list"
mnmInt [x] = x
mnmInt (x:xs) = min x (mnmInt xs)
```

This uses a predefined function `min` for the minimum of two integers. It also uses pattern matching for lists:

- The list pattern `[]` matches only the empty list,
- the list pattern `[x]` matches any singleton list,
- the list pattern `(x:xs)` matches any non-empty list.

Haskell Types

The basic Haskell types are:

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded. That's why we used this type in the implementation of the prime number test.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.
- `Char` is the type of characters.

Note that the name of a type always starts with a capital letter.

To denote arbitrary types, Haskell allows the use of type variables. For these, `a`, `b`, ..., are used.

New types can be formed in several ways:

- By list-formation: if `a` is a type, `[a]` is the type of lists over `a`. Examples: `[Int]` is the type of lists of integers; `[Char]` is the type of lists of characters, or strings.
- By pair- or tuple-formation: if `a` and `b` are types, then `(a,b)` is the type of pairs with an object of type `a` as their first component, and an object of type `b` as their second component. If `a`, `b` and `c` are types, then `(a,b,c)` is the type of triples with an object of type `a` as their first component, an object of type `b` as their second component, and an object of type `c` as their third component ...
- By function definition: `a -> b` is the type of a function that takes arguments of type `a` and returns values of type `b`.
- By defining your own datatype from scratch, with a `data` type declaration. More about this in due course.

Implementing Prime Factorisation in Haskell

Note the use of `div` for integer division. and the use of `where` to introduce an auxilary function.

```
factors :: Integer -> [Integer]
factors n | n < 1     = error "arg not positive"
          | n == 1    = []
          | otherwise = p : factors (div n p)
                        where p = ld n
```

```
LAI9> factors 84
[2,2,3,7]
LAI9> factors 557940830126698960967415390
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

Working with Lists: The `map` and `filter` Functions

If you use the Hugs command `:t` to find the types of the function `map`, you get the following:

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Sections

In general, if op is an infix operator, `(op x)` is the operation resulting from applying op to its righthand side argument, `(x op)` is the operation resulting from applying op to its lefthand side argument, and `(op)` is the prefix version of the operator. Thus `(2^)` is the operation that computes powers of 2, and `map (2^) [1..10]` will yield

```
[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

Similarly, `(>3)` denotes the property of being greater than 3, and `(3>)` the property of being smaller than 3.

map

If p is a property (an operation of type `a -> Bool`) and `l` is a list of type `[a]`, then `map p l` will produce a list of type `Bool` (a list of truth values), like this:

```
Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>
```

`map` is predefined in Haskell.

Home-made definition:

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

filter

Another useful function is `filter`, for filtering out the elements from a list that satisfy a given property. This is predefined, but here is a home-made version:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

Here is an example of its use:

```
LAI9> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

List comprehension

List comprehension is defining lists by the following method:

```
[ x | x <- xs, property x ]
```

This defines the sublist of `xs` of all items satisfying `property`. It is equivalent to:

```
filter property xs
```

```
somePrimes    = [ x | x <- [1..1000], prime x ]

primesUntil n = [ x | x <- [1..n], prime x ]

allPrimes     = [ x | x <- [1..], prime x ]
```

Equivalently:

```
somePrimes    = filter prime [1..1000]

primesUntil n = filter prime [1..n]

allPrimes     = filter prime [1..]
```

sort

```
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)
```

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x <= y    = x:y:ys
                | otherwise = y: insert x ys
```

nub

`nub` removes duplicates, as follows:

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

Contained in

$$A \subseteq B \;:\equiv\; \forall x \in A : x \in B.$$

```
containedIn :: Eq a => [a] -> [a] -> Bool
containedIn xs ys = all (\ x -> elem x ys) xs
```

elem, all

`elem` and `all` are predefined.

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys

all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

Note the use of `(.)` for function composition (predefined).

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
f . g = \ x -> f (g x)
```

Representing Relations

Various options:

- Lists of pairs, type `[(a,a)]`.
- Sets of pairs, type `Set (a,a)`.
- Characteristic functions, type `a -> a -> Bool`
- Range functions, type `a -> [a]` or `a -> Set a`.

Relations as Lists of Pairs

```
type Rel a = [(a,a)]
```

Example relations:

```
r1 = [(1,2),(2,1)]

r2 = [(1,2),(2,1),(2,1)]
```

These relations have the same pairs, so they are in fact equal.

Test for equality of relations

```
sameR :: Ord a => Rel a -> Rel a -> Bool
sameR r s = sort (nub r) == sort (nub s)
```

Operations on relations: converse

Relational converse $R^{\smile}$ is given by:

$$R^{\smile} = \{(y, x) \mid (x, y) \in R\}$$

Implementation

```
cnv :: Rel a -> Rel a
cnv r = [ (y,x) | (x,y) <- r ]
```

Operations on relations: composition

The relational composition of two relations R and S on a set A:

$$R \circ S = \{(x, z) \mid \exists y \in A(xRy \wedge ySz)\}$$

For the implementation, it is useful to declare a new infix operator for relational composition.

```
infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s =
  nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

Note that `(@@)` is the prefix version of `@@`.

Testing for Euclideanness

Proposition:

R is euclidean iff $R^{\smile} \circ R \subseteq R$.

Proof:

$\Rightarrow$. Suppose R is euclidean. Assume $(x, y) \in R^{\smile} \circ R$. Then for some z, $(x, z) \in R^{\smile}$ and $(z, y) \in R$. Then $(z, x) \in R$ and $(z, y) \in R$, so by euclideanness of R, $(x, y) \in R$. This proves $R^{\smile} \circ R \subseteq R$.

$\Leftarrow$. Suppose $R^{\smile} \circ R \subseteq R$. We must show that R is euclidean. Assume $(x, y) \in R, (x, z) \in R$. We must show that $(y, z) \in R$. This follows immediately from $(y, x) \in R^{\smile}, (x, z) \in R$ and $R^{\smile} \circ R \subseteq R$.

Use this proposition for a test of Euclideanness:

```
euclR :: Eq a => Rel a -> Bool
euclR r = (cnv r @@ r) `containedIn` r
```

Note the use of backquotes to make `containedIn` an infix operator.

This gives:

```
LAI9> euclR [(1,2),(1,3)]
False
LAI9> euclR [(1,2),(1,3),(2,3)]
False
LAI9> euclR [(1,2),(1,3),(2,3),(3,2)]
False

LAI9> euclR [(1,2),(1,3),(2,3),(3,2),(2,2),(3,3)]
True
```

Test for Seriality

```
serialR :: Eq a => Rel a -> Bool
serialR r =
  all (not.null)
    (map (\ (x,y) -> [ v | (u,v) <- r, y == u]) r)
```

```
LAI9> serialR [(1,2)]
False
LAI9> serialR [(1,2),(2,3)]
False
LAI9> serialR [(1,2),(2,3),(3,2)]
True
```

Testing for KD45

Implementation of test for transitivity `transR` is left for you as a computer lab exercise.

Once we have tests for seriality, transitivity and euclideanness we can implement the test for their combination as follows:

```
isTSE :: Eq a => Rel a -> Bool
isTSE r = transR r && serialR r && euclR r
```

Implementing a test for being an equivalence relation is left for you as a computer lab exercise.

Representing Epistemic Models: Agents

```
data Agent = A | B | C | D | E deriving (Eq,Ord,Enum)

a,alice, b,bob, c,carol, d,dave, e,ernie  :: Agent
a = A; alice = A
b = B; bob   = B
c = C; carol = C
d = D; dave  = D
e = E; ernie = E

instance Show Agent where
  show A = "a"; show B = "b"; show C = "c";
  show D = "d" ; show E = "e"
```

Representing Epistemic Models: Basic Propositions

```
data Prop = P Int | Q Int | R Int deriving (Eq,Ord)

instance Show Prop where
  show (P 0) = "p"; show (P i) = "p" ++ show i
  show (Q 0) = "q"; show (Q i) = "q" ++ show i
  show (R 0) = "r"; show (R i) = "r" ++ show i
```

Datatype for Epistemic Models

```
data EpistM state = Mo
              [state]
              [Agent]
              [(state,[Prop])]
              [(Agent,state,state)]
              [state]  deriving (Eq,Show)
```

Next time: ...

- Representing formulas
- Implementing evaluation of formulas in epistemic models
- Public Announcement Logic
- Representing public announcements.
- ...

Background reading: [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**].