# Haskell for Knowledge Representation

Jan van Eijck

jve@cwi.nl

August 4, 2008

**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.

## Course Homepage

http://www.cwi.nl/~jve/courses/esslli08/

## 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@vuur:~/courses/esslli08$ 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: 20050308       _________________________________________

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@vuur:~/courses/esslli08$ ghci
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.6, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... 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 HFKR

where
import List
```

This declares a module and imports another module. The code of the module `HFKR` consists of the text in frames in the slides that follow.

# Loading the module

```
jve@vuur:~/courses/esslli08$ hugs HFKR
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      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: 20050308       _________________________________________

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

Type :? for help
HFKR>
```

## 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 an introduction to epistemic logic with an introduction to functional programming is to enable you to 'play' with the formal definitions on a computer. This will greatly speed up your learning process.

## Implementation of a Prime Number Test

A natural number $n$ is prime if $n > 1$ and $n$ has only $1$ and itself as proper divisors.

A proper divisor of a natural number $n$ is a number $m$ such that dividing $n$ by $m$ leaves no remainder.

The Haskell command `rem n m` gives the remainder of $n/m$.

Here is our first Haskell program:

```
prime n =
  n > 1 && all (\ x -> rem n x /= 0) [2..n-1]
```

Looking at the ingredients one by one, we see that this is an almost literal rendering of the definition of being a prime.

```
prime n =
  n > 1 && all (\ x -> rem n x /= 0) [2..n-1]
```

- `&&` denotes conjunction.
- `rem n x /= 0` expresses that the remainder of the process of dividing `n` by `x` is non-zero. In other words, `x` is not a proper divisor of `n`.
- `(\ x -> rem n x /= 0)` is the property of not being a proper divisor of `n`.
- `[2..n-1]` denotes the list of integers from 2 until (and including) `n-1`,
- `all` denotes a check that a property holds of all members of a list.

## Trying it out

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

primesUntil n = filter prime [1..n]

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

More on the `filter` function below.

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

## 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`.

```
HFKR> :t divides 5
divides 5 :: Integer -> Bool
HFKR> :t divides 5 7
divides 5 7 :: Bool
HFKR> divides 5 7
False
HFKR> divides 5 10
True
```

## Lambda Abstraction

Take the statement Hillary respects Barack. By means of abstraction, we can get all kinds of properties and relations from this statement:

- ‘respecting Barack’
- ‘being respected by Hillary’
- ‘respecting’
- ‘being respected 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:

- ‘$\lambda x.\ x$ respects Barack’ expresses ‘respecting Barack’.
- ‘$\lambda x.$ Hillary respects $x$’ expresses ‘being respected by Hillary’.
- ‘$\lambda x \lambda y.\ x$ respects $y$’ expresses ‘respecting’.
- ‘$\lambda y \lambda x.\ x$ respects $y$’ expresses ‘being respected by’.

## Lambda Abstraction – 3

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

We have already seen an example: `(\ x -> rem n x /= 0)`.

In Haskell, abstractions can be used as nameless functions.

But we can also give them names, as in the following example:

```
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 size integers, `Int` the type of fixed size 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.

## 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:

```
HFKR> :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:

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

`map` is predefined in Haskell. Home-made definition:

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

Note the use of `:` for placing an element at the head of a list.

## 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
```

Note the use of `| p x` and `| otherwise` for making a case distinction.

```
HFKR> 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
```

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

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

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

Equivalently:

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

primesUntil n = filter prime [1..n]

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

## sort

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

```haskell
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)]`.
- Characteristic functions, type `a -> a -> Bool`
- Characteristic functions of pairs, type `(a,a) -> Bool`.
- Range functions, type `a -> [a]`
- And so on.

Choice does not matter much, as these can easily converted into each other.

We will (mostly) use lists of pairs.

## 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

A relation $R$ is euclidian if $\forall xyz((Rxy \wedge Rxz) \rightarrow Ryz)$.

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.

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

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

## Test for Seriality

A relation $R$ is serial if $\forall x \exists y Rxy$ holds.

Here is a test:

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

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

## Testing for Transitivity

A relation $R$ is transitive if $\forall xyz((Rxy \wedge Ryz) \rightarrow Rxz)$.

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

## Testing for KD45

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

```
isKD45 :: Eq a => Rel a -> Bool
isKD45 r = transR r && serialR r && euclR r
```

## Testing for S5

An accessibility relation is S5 if it is an equivalence.

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
```

## A Datatype for Epistemic Models

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

## Tomorrow: . . .

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

Background reading: [2], [1], [8], [3], [7], [6], [4], [5].

## Homework Exercises

1. Implement a test for reflexivitity of a relation on a given domain. The type declaration is:

   `reflexiveR :: Eq a => [a] -> Rel a -> Bool`.

   The constraint `Eq a =>` expresses that a has to be a type for which equality is defined. The first argument gives the domain.

2. Implement a test for symmetry of relations. The type declaration is:

   `symmR :: Eq a => Rel a -> Bool`.

3. Implement a test for transitivity of relations. Here is the type declaration:

   `transR :: Eq a => Rel a -> Bool`

4. Implement a test for being an S5 relation, on a given domain. The type declaration is:

   `isS5 :: Eq a => [a] -> Rel a -> Bool`.

   The first argument gives the domain.

## References

[1] Hal Daume. Yet another Haskell tutorial. www.cs.utah.edu/~hal/docs/daume02yaht.pdf.

[2] K. Doets and J. van Eijck. The Haskell Road to Logic, Maths and Programming, volume 4 of Texts in Computing. College Publications, London, 2004.

[3] The Haskell Team. The Haskell homepage. http://www.haskell.org.

[4] P. Hudak, J. Fasel, and J. Peterson. A gentle introduction to Haskell. Technical report, Yale University, 1996. Online version: http://www.haskell.org/tutorial/.

[5] Mark P. Jones, Alastair Reid, et al. The Hugs98 user manual. http://cvs.haskell.org/Hugs/pages/hugsman/index.html.

[6] S. Peyton Jones, editor. Haskell 98 Language and Libraries; The Revised Report. Cambridge University Press, 2003.

[7] S. Peyton Jones, J. Hughes, et al. Report on the programming language Haskell 98. Available from the Haskell homepage: http://www.haskell.org, 1999.

[8] The GHC Team. The Glasgow Haskell compiler (GHC). http://www.haskell.org/ghc/.