Ordered Pairs, Products, Sets versus Lists, Lambda Abstraction, Database Query

Jan van Eijck

June 10, 2012

Abstract

Ordered pairs, products, from sets to lists, from lists to sets.
Next, we take a further look at lambda abstraction, explain the use of lambda abstraction for database query, and demonstrate how lambda abstracts can be used in Haskell.

Ordered Pairs

The two sets $\{a, b\}$ and $\{b, a\}$ are equal: it follows from the extensionality principle that order of presentation does not count.

The *ordered pair* of objects a and b is denoted by

$$(a, b) .$$

Here, a is the *first* and b the *second coordinate* of (a, b).

Ordered Pairs

The two sets $\{a, b\}$ and $\{b, a\}$ are equal: it follows from the extensionality principle that order of presentation does not count.

The *ordered pair* of objects a and b is denoted by

$$(a, b) \; .$$

Here, a is the *first* and b the *second coordinate* of (a, b).

Ordered pairs behave according to the following rule:

$$(a, b) = (x, y) \implies a = x \wedge b = y.$$

Ordered Pairs

The two sets $\{a, b\}$ and $\{b, a\}$ are equal: it follows from the extensionality principle that order of presentation does not count.

The *ordered pair* of objects a and b is denoted by

$$(a, b) \ .$$

Here, a is the *first* and b the *second coordinate* of (a, b).

Ordered pairs behave according to the following rule:

$$(a, b) = (x, y) \implies a = x \wedge b = y.$$

Note that $(a, b) = (b, a)$ only holds when $a = b$.

Cartesian Products

The (Cartesian) *product* of the sets A and B is the set of all pairs (a, b) where $a \in A$ and $b \in B$. In symbols:

$$A \times B = \{\, (a, b) \mid a \in A \wedge b \in B \,\}.$$

Cartesian Products

The (Cartesian) *product* of the sets A and B is the set of all pairs (a, b) where $a \in A$ and $b \in B$. In symbols:

$$A \times B = \{\, (a, b) \mid a \in A \land b \in B \,\}.$$

Instead of $A \times A$ one usually writes A^2.

Cartesian Products

The (Cartesian) *product* of the sets A and B is the set of all pairs (a, b) where $a \in A$ and $b \in B$. In symbols:

$$A \times B = \{ (a, b) \mid a \in A \land b \in B \}.$$

Instead of $A \times A$ one usually writes A^2.

Here is an implementation of the list product operation in Haskell

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

This gives:

```
Main> listproduct [1..4] ['A'..'C']
[(1,'A'),(1,'B'),(1,'C'),(2,'A'),(2,'B'),(2,'C'),
 (3,'A'),(3,'B'),(3,'C'),(4,'A'),(4,'B'),(4,'C')]

Main> listproduct [1..4] [True, False]
[(1,True),(1,False),(2,True),(2,False),(3,True),
 (3,False),(4,True),(4,False)]
```

Useful Product Laws

For arbitrary sets A, B, C, D the following hold:

1. $(A \times B) \cap (C \times D) = (A \times D) \cap (C \times B)$,
2. $(A \cup B) \times C = (A \times C) \cup (B \times C)$; $(A \cap B) \times C = (A \times C) \cap (B \times C)$,
3. $(A \cap B) \times (C \cap D) = (A \times C) \cap (B \times D)$,
4. $(A \cup B) \times (C \cup D) = (A \times C) \cup (A \times D) \cup (B \times C) \cup (B \times D)$,
5. $[(A - C) \times B] \cup [A \times (B - D)] \subseteq (A \times B) - (C \times D)$.

Example Proof

To be proved: $(A \cup B) \times C = (A \times C) \cup (B \times C)$:

$\subseteq$:

Suppose that $p \in (A \cup B) \times C$.

Then $a \in A \cup B$ and $c \in C$ exist such that $p = (a, c)$.

Thus (i) $a \in A$ or (ii) $a \in B$.

(i). In this case, $p \in A \times C$, and hence $p \in (A \times C) \cup (B \times C)$.

(ii). Now $p \in B \times C$, and hence again $p \in (A \times C) \cup (B \times C)$.

Thus $p \in (A \times C) \cup (B \times C)$.

Therefore, $(A \cup B) \times C \subseteq (A \times C) \cup (B \times C)$.

Example Proof, continued

$\supseteq$:

Conversely, assume that $p \in (A \times C) \cup (B \times C)$.

Thus (i) $p \in A \times C$ or (ii) $p \in B \times C$.

(i). In this case $a \in A$ and $c \in C$ exist such that $p = (a, c)$; *a fortiori*, $a \in A \cup B$ and hence $p \in (A \cup B) \times C$.

(ii). Now $b \in B$ and $c \in C$ exist such that $p = (b, c)$; *a fortiori* $b \in A \cup B$ and hence, again, $p \in (A \cup B) \times C$.

Thus $p \in (A \cup B) \times C$.

Therefore, $(A \times C) \cup (B \times C) \subseteq (A \cup B) \times C$.

The required result follows using Extensionality.

Ordered n-tuples

Ordered n-tuples over some base set A, for every $n \in \mathbb{N}$. Definition by recursion.

1. $A^0 := \{\emptyset\}$,
2. $A^{n+1} := A \times A^n$.

Note that ordered n-tuples are pairs.

Ordered n-tuples

Ordered n-tuples over some base set A, for every $n \in \mathbb{N}$. Definition by recursion.

1. $A^0 := \{\emptyset\}$,
2. $A^{n+1} := A \times A^n$.

Note that ordered n-tuples are pairs.

From Sets to Lists

Finally, let $A^* = \bigcup_{n \in \mathbb{N}} A^n$. Then A^* is the set of all finite lists over A. Note that the list $[a, b, c, d]$ gets represented as the pair $(a, (b, (c, (d, \emptyset))))$.

Taking Lists as Basic

Definition of lists in Haskell:

- `[]` is a list.
- If x is an object and l is a list, then $x : l$ is a list, provided that the types agree.

Taking Lists as Basic

Definition of lists in Haskell:

- [] is a list.
- If x is an object and l is a list, then $x : l$ is a list, provided that the types agree.

The type of a list is derived from the type of its objects: if $x :: a$, then lists of objects of that type have type $[a]$.

Taking Lists as Basic

Definition of lists in Haskell:

- [] is a list.
- If x is an object and l is a list, then $x : l$ is a list, provided that the types agree.

The type of a list is derived from the type of its objects: if $x :: a$, then lists of objects of that type have type $[a]$.

Under the typing restrictions of Haskell, it is impossible to put objects of different types together in one list.

Taking Lists as Basic

Definition of lists in Haskell:

- `[]` is a list.
- If x is an object and l is a list, then $x : l$ is a list, provided that the types agree.

The type of a list is derived from the type of its objects: if $x :: a$, then lists of objects of that type have type $[a]$.

Under the typing restrictions of Haskell, it is impossible to put objects of different types together in one list.

The operation `(:)` has type `a -> [a] -> [a]`.

Lists and List Equality

```
data [a] = [] | a : [a] deriving (Eq, Ord)
```

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

```
instance Eq a => Eq [a] where
    []     == []     =  True
    (x:xs) == (y:ys) =  x==y && xs==ys
    _      == _      =  False
```

List Ordering

```
instance Ord a => Ord [a] where
  compare []     (_:_)  = LT
  compare []     []     = EQ
  compare (_:_)  []     = GT
  compare (x:xs) (y:ys) =
             primCompAux x y (compare xs ys)
```

```
primCompAux      :: Ord a =>
                    a -> a -> Ordering -> Ordering
primCompAux x y o =
   case compare x y of EQ -> o; LT -> LT; GT -> GT
```

List indexing

Consider the list indexing function `(!!) :: [a] -> Int -> a` that does the following:

```
Prelude> ['a'..] !! 0
'a'
Prelude> ['a'..] !! 3
'd'
```

How would you implement this?

List indexing

Consider the list indexing function `(!!) :: [a] -> Int -> a` that does the following:

```
Prelude> ['a'..] !! 0
'a'
Prelude> ['a'..] !! 3
'd'
```

How would you implement this?

```
(!!)                   :: [a] -> Int -> a
(x:_)   !! 0           = x
(_:xs)  !! n | n>0     = xs !! (n-1)
(_:_)   !! _           = error "!!: negative index"
[]      !! _           = error "!!: index too large"
```

Fundamental List Operations

```
head                    :: [a] -> a
head (x:_)               = x
tail                    :: [a] -> [a]
tail (_:xs)              = xs
last                    :: [a] -> a
last [x]                 = x
last (_:xs)              = last xs
init                    :: [a] -> [a]
init [x]                 = []
init (x:xs)              = x : init xs
null                    :: [a] -> Bool
null []                  = True
null (_:_)               = False
```

Lambda Abstraction

A very convenient notation for function construction is by means of lambda abstraction. In this notation, $\lambda x.x+1$ encodes the specification $x \mapsto x+1$. The lambda operator is a variable binder, so $\lambda x.x+1$ and $\lambda y.y+1$ denote the same function.

Lambda Abstraction

A very convenient notation for function construction is by means of lambda abstraction. In this notation, $\lambda x.x+1$ encodes the specification $x \mapsto x+1$. The lambda operator is a variable binder, so $\lambda x.x+1$ and $\lambda y.y+1$ denote the same function.

In fact, every time we specify a function `foo` in Haskell by means of

```
foo x y z = t
```

we can also define `foo` by means of:

```
foo = \ x y z -> t
```

Lambda Abstraction

A very convenient notation for function construction is by means of lambda abstraction. In this notation, $\lambda x.x+1$ encodes the specification $x \mapsto x+1$. The lambda operator is a variable binder, so $\lambda x.x+1$ and $\lambda y.y+1$ denote the same function.

In fact, every time we specify a function `foo` in Haskell by means of

```
foo x y z = t
```

we can also define `foo` by means of:

```
foo = \ x y z -> t
```

If the types of x, y, z, t are known, this also specifies a domain and a range. For if `x :: a`, `y :: b`, `z :: c`, `t :: d`, then $\lambda xyz.t$ has type `a -> b -> c -> d`.

Lambda Abstraction (2)

Haskell allows construction of functions by means of lambda abstraction:

```
Prelude> (\x -> x + 1) 4
5
```

Lambda Abstraction (2)

Haskell allows construction of functions by means of lambda abstraction:

```
Prelude> (\x -> x + 1) 4
5

Prelude> (\s -> "hello, " ++ s) "dolly"
"hello, dolly"
```

Lambda Abstraction (2)

Haskell allows construction of functions by means of lambda abstraction:

```
Prelude> (\x -> x + 1) 4
5

Prelude> (\s -> "hello, " ++ s) "dolly"
"hello, dolly"

Prelude> :t (\s -> "hello, " ++ s)
\s -> "hello, " ++ s :: [Char] -> [Char]
```

Lambda Abstraction (2)

Haskell allows construction of functions by means of lambda abstraction:

```
Prelude> (\x -> x + 1) 4
5

Prelude> (\s -> "hello, " ++ s) "dolly"
"hello, dolly"

Prelude> :t (\s -> "hello, " ++ s)
\s -> "hello, " ++ s :: [Char] -> [Char]

Prelude> (\x y -> x^y) 2 4
16
```

Lambda Abstraction (3)

Such functions can be passed as arguments:

```
Prelude> map (\x -> x + 3) [1..5]
[4,5,6,7,8]
```

Lambda Abstraction (3)

Such functions can be passed as arguments:

```
Prelude> map (\x -> x + 3) [1..5]
[4,5,6,7,8]

Prelude> filter (\x -> x^2 < 20) [1..10]
[1,2,3,4]
```

Lambda Abstraction (3)

Such functions can be passed as arguments:

```
Prelude> map (\x -> x + 3) [1..5]
[4,5,6,7,8]

Prelude> filter (\x -> x^2 < 20) [1..10]
[1,2,3,4]

Prelude> ((\x -> x + 1) . (\y -> y + 2)) 5
8
```

List Comprehension and Database Query

```
module DB
where
type WordList = [String]
type DB       = [WordList]
db :: DB
db = [
 ["release", "Blade Runner", "1982"],
 ["release", "Alien", "1979"],
 ["release", "Titanic", "1997"],
 ["release", "Good Will Hunting", "1997"],
 ["release", "Pulp Fiction", "1994"],
 ["release", "Reservoir Dogs", "1992"],
 ["release", "Romeo and Juliet", "1996"],
```

```
["direct", "Brian De Palma", "The Untouchables"],
["direct", "James Cameron", "Titanic"],
["direct", "James Cameron", "Aliens"],
["direct", "Ridley Scott", "Alien"],
["direct", "Ridley Scott", "Blade Runner"],
["direct", "Ridley Scott", "Thelma and Louise"],
["direct", "Gus Van Sant", "Good Will Hunting"],
["direct", "Quentin Tarantino", "Pulp Fiction"],
{- ... -}
```

```
["play", "Leonardo DiCaprio",
         "Romeo and Juliet", "Romeo"],
["play", "Leonardo DiCaprio",
         "Titanic", "Jack Dawson"],
["play", "Robin Williams",
         "Good Will Hunting", "Sean McGuire"],
["play", "John Travolta",
         "Pulp Fiction", "Vincent Vega"],
["play", "Harvey Keitel",
         "Reservoir Dogs", "Mr White"],
{- ... -}
```

The database can be used to define the following lists of database objects, with list comprehension.

```
characters = nub [ x    | ["play",_,_,x]  <- db ]
movies     =     [ x    | ["release",x,_] <- db ]
actors     = nub [ x    | ["play",x,_,_]  <- db ]
directors  = nub [ x    | ["direct",x,_]  <- db ]
dates      = nub [ x    | ["release",_,x] <- db ]
universe   = nub (characters
                  ++ actors
                  ++ directors
                  ++ movies
                  ++ dates)
```

Next, define lists of tuples, again by list comprehension:

```
direct     = [ (x,y)   | ["direct",x,y]  <- db ]
act        = [ (x,y)   | ["play",x,y,_]  <- db ]
play       = [ (x,y,z) | ["play",x,y,z]  <- db ]
release    = [ (x,y)   | ["release",x,y] <- db ]
```

Finally, define one placed, two placed and three placed predicates by means of lambda abstraction.

```
charP       = \ x       -> elem x characters
actorP      = \ x       -> elem x actors
movieP      = \ x       -> elem x movies
directorP   = \ x       -> elem x directors
dateP       = \ x       -> elem x dates
actP        = \ (x,y)   -> elem (x,y) act
releaseP    = \ (x,y)   -> elem (x,y) release
directP     = \ (x,y)   -> elem (x,y) direct
playP       = \ (x,y,z) -> elem (x,y,z) play
```

Example Queries

'Give me the actors that also are directors.'

```
q1 = [ x | x <- actors, directorP x ]
```

Example Queries

‘Give me the actors that also are directors.’

```
q1 = [ x | x <- actors, directorP x ]
```

‘Give me all actors that also are directors, together with the films in which they were acting.’

```
q2 = [ (x,y) | (x,y) <- act, directorP x ]
```

'Give me all directors together with their films and their release dates.' The following is *wrong*.

```
q3 = [ (x,y,z) | (x,y) <- direct, (y,z) <- release ]
```

'Give me all directors together with their films and their release dates.' The following is *wrong*.

```
q3 = [ (x,y,z) | (x,y) <- direct, (y,z) <- release ]
```

The problem is that the two ys are unrelated. In fact, this query generates an infinite list. This can be remedied by using the equality predicate as a link:

```
q4 = [ (x,y,z) | (x,y) <- direct,
                 (u,z) <- release,
                 y == u ]
```

A Datatype for Sets

```
module SetEq (Set,emptySet,isEmpty,inSet,subSet,
              insertSet,deleteSet,powerSet,takeSet,
              list2set,(!!!))

where

import List

newtype Set a = Set [a]

instance Eq a => Eq (Set a) where
  set1 == set2 = subSet set1 set2
                 && subSet set2 set1
```

```haskell
subSet :: (Eq a) => Set a -> Set a -> Bool
subSet (Set [])      _   = True
subSet (Set (x:xs)) set = (inSet x set)
                            && subSet (Set xs) set

inSet  :: (Eq a) => a -> Set a -> Bool
inSet x (Set s) = elem x s
```

This gives:

```
Main> Set [2,3,3,1,1,1] == Set [1,2,3]
True
```

```
instance (Show a) => Show (Set a) where
    showsPrec _ (Set s) str = showSet s str

showSet []     str = showString "{}" str
showSet (x:xs) str =
  showChar '{' (shows x (sh xs str))
    where sh []     str = showChar '}' str
          sh (x:xs) str = showChar ','
                            (shows x (sh xs str))
```

This gives:

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

```
emptySet  :: Set a
emptySet = Set []

isEmpty  :: Set a -> Bool
isEmpty (Set []) = True
isEmpty  _       = False
```

```
insertSet :: (Eq a) => a -> Set a -> Set a
insertSet x (Set ys) | inSet x (Set ys) = Set ys
                     | otherwise        = Set (x:ys)

deleteSet :: Eq a => a -> Set a -> Set a
deleteSet x (Set xs) = Set (delete x xs)

list2set :: Eq a => [a] -> Set a
list2set [] = Set []
list2set (x:xs) = insertSet x (list2set xs)
```

```
powerSet :: Eq a => Set a -> Set (Set a)
powerSet (Set xs) = Set (map (\xs -> (Set xs))
                               (powerList xs))

takeSet :: Eq a => Int -> Set a -> Set a
takeSet n (Set xs) = Set (take n xs)

infixl 9 !!!
(!!!) :: Eq a => Set a -> Int -> a
(Set xs) !!! n = xs !! n
```

Five Levels From the Set Theoretic Universe

```
module Hierarchy where

import SetEq

data S = Void deriving (Eq,Show)
empty :: Set S
empty = Set []
v0    = empty
v1    = powerSet v0
v2    = powerSet v1
v3    = powerSet v2
v4    = powerSet v3
v5    = powerSet v4
```