Sets, Types and Lists

Jan van Eijck

June 7, 2012

Abstract

Topics of today: Lazy list processing, operations on sets, set theoretic reasoning, set theory and paradoxes, the use of types to avoid paradoxes, how sets relate to types and to lists, operations on lists.

Lazy list processing... The Sieve of Eratosthenes

Start with the list of all natural numbers ≥ 2:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,$$
$$21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,$$
$$36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, \ldots$$

Lazy list processing... The Sieve of Eratosthenes

Start with the list of all natural numbers ≥ 2:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,$$
$$21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,$$
$$36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, \ldots$$

In the first round, mark 2 (the first number in the list) as prime, and mark all multiples of 2 for removal in the remainder of the list (marking for removal indicated by overlining):

$$\mathbb{2}, 3, \overline{4}, 5, \overline{6}, 7, \overline{8}, 9, \overline{10}, 11, \overline{12}, 13, \overline{14}, 15, \overline{16}, 17, \overline{18}, 19, \overline{20},$$
$$21, \overline{22}, 23, \overline{24}, 25, \overline{26}, 27, \overline{28}, 29, \overline{30}, 31, \overline{32}, 33, \overline{34}, 35,$$
$$\overline{36}, 37, \overline{38}, 39, \overline{40}, 41, \overline{42}, 43, \overline{44}, 45, \overline{46}, 47, \overline{48}, \ldots$$

In the second round, mark 3 as prime, and mark all multiples of 3 for removal in the remainder of the list:

$$\mathbb{2}, \mathbb{3}, \overline{4}, 5, \overline{6}, 7, \overline{8}, \overline{9}, \overline{10}, 11, \overline{12}, 13, \overline{14}, \overline{15}, \overline{16}, 17, \overline{18}, 19, \overline{20},$$
$$\overline{21}, \overline{22}, 23, \overline{24}, 25, \overline{26}, \overline{27}, \overline{28}, 29, \overline{30}, 31, \overline{32}, \overline{33}, \overline{34}, 35, \ldots$$
$$\overline{36}, 37, \overline{38}, \overline{39}, \overline{40}, 41, \overline{42}, 43, \overline{44}, \overline{45}, \overline{46}, 47, \overline{48}, \ldots$$

In the second round, mark 3 as prime, and mark all multiples of 3 for removal in the remainder of the list:

$$\mathbb{2}\,, \mathbb{3}\,, \overline{4}, 5, \overline{6}, 7, \overline{8}, \overline{9}, \overline{10}, 11, \overline{12}, 13, \overline{14}, \overline{15}, \overline{16}, 17, \overline{18}, 19, \overline{20},$$
$$\overline{21}, \overline{22}, 23, \overline{24}, 25, \overline{26}, \overline{27}, \overline{28}, 29, \overline{30}, 31, \overline{32}, \overline{33}, \overline{34}, 35, \ldots$$
$$\overline{36}, 37, \overline{38}, \overline{39}, \overline{40}, 41, \overline{42}, 43, \overline{44}, \overline{45}, \overline{46}, 47, \overline{48}, \ldots$$

In the third round, mark 5 as prime, and mark all multiples of 5 for removal in the remainder of the list:

$$\mathbb{2}\,, \mathbb{3}\,, \overline{4}, \mathbb{5}\,, \overline{6}, 7, \overline{8}, \overline{9}, \overline{10}, 11, \overline{12}, 13, \overline{14}, \overline{15}, \overline{16}, 17, \overline{18}, 19, \overline{20},$$
$$\overline{21}, \overline{22}, 23, \overline{24}, \overline{25}, \overline{26}, \overline{27}, \overline{28}, 29, \overline{30}, 31, \overline{32}, \overline{33}, \overline{34}, \overline{35},$$
$$\overline{36}, 37, \overline{38}, \overline{39}, \overline{40}, 41, \overline{42}, 43, \overline{44}, \overline{45}, \overline{46}, 47, \overline{48}, \ldots$$

And so on.

In the Haskell implementation we mark numbers in the sequence `[2..]` for removal by replacing them with 0. When generating the sieve, these zeros are skipped.

```
sieve :: [Integer] -> [Integer]
sieve (0 : xs) = sieve xs
sieve (n : xs) = n : sieve (mark (xs, n-1, n-1))
  where
  mark (x : xs, 0, m) = 0 : mark (xs, m,   m)
  mark (x : xs, n, m) = x : mark (xs, n-1, m)

primes :: [Integer]
primes = sieve [2..]
```

Extensionality and Subsets

A set is a collection into a whole of definite, distinct objects of our intuition or of our thought. The objects are called the elements (members) of the set.

Extensionality and Subsets

A set is a collection into a whole of definite, distinct objects of our intuition or of our thought. The objects are called the elements (members) of the set.

Sets that have the same elements are equal.

For all sets A and B, it holds that:

$$\forall x(x \in A \iff x \in B) \implies A = B.$$

This is called the principle of extensionality.

Extensionality and Subsets

A set is a collection into a whole of definite, distinct objects of our intuition or of our thought. The objects are called the elements (members) of the set.

Sets that have the same elements are equal.

For all sets A and B, it holds that:

$$\forall x(x \in A \iff x \in B) \implies A = B.$$

This is called the principle of extensionality.

The set A is a *subset* of the set B, and B a *superset* of A; notations: $A \subseteq B$, and $B \supseteq A$, if every member of A is also a member of B.

$$\forall x\ (x \in A \implies x \in B).$$

If $A \subseteq B$ and $A \neq B$, then A is a *proper* subset of B.

Proving that two sets are different

Note that $A = B$ iff $A \subseteq B$ and $B \subseteq A$. To show that $A \neq B$ we therefore either have to find an object c with $c \in A, c \notin B$ (in this case c is a witness of $A \not\subseteq B$), or an object c with $c \notin A, c \in B$ (in this case c is a witness of $B \not\subseteq A$).

Proving that two sets are different

Note that $A = B$ iff $A \subseteq B$ and $B \subseteq A$. To show that $A \neq B$ we therefore either have to find an object c with $c \in A, c \notin B$ (in this case c is a witness of $A \not\subseteq B$), or an object c with $c \notin A, c \in B$ (in this case c is a witness of $B \not\subseteq A$).

Proving that two sets are equal

To show $A = B$ we have to prove both $A \subseteq B$ and $B \subseteq A$.

Given: ...
To be proved: $A = B$.
Proof:
$\subseteq$: Let x be an arbitrary object in A.
To be proved: $x \in B$.
Proof:
...
Thus $x \in B$.

$\supseteq$: Let x be an arbitrary object in B.
To be proved: $x \in A$.
Proof:
...
Thus $x \in A$.

Thus $A = B$.

Set Enumeration

A set that has only few elements $a_1, \ldots, a_n$ can be denoted as

$$\{a_1, \ldots, a_n\}.$$

Extensionality ensures that this denotes exactly one set, for by extensionality the set is uniquely determined by the fact that it has $a_1, \ldots, a_n$ as its members.

Note that $x \in \{a_1, \ldots, a_n\}$ iff $x = a_1 \vee \cdots \vee x = a_n$.

List Enumeration in Haskell

An analogue to the enumeration notation is available in Haskell, where $[n..m]$ can be used for generating a list of items from `n` to `m`. This presupposes that `n` and `m` are of the same type, and that enumeration makes sense for that type.

List Enumeration in Haskell

An analogue to the enumeration notation is available in Haskell, where $[n..m]$ can be used for generating a list of items from `n` to `m`. This presupposes that `n` and `m` are of the same type, and that enumeration makes sense for that type.

Another possibility is enumeration from a given element: `['A'..]`.

This may create infinite lists: `[0..]`.

Set Comprehension

Given a universe of objects U, select the elements from U that satisfy the predicate E:

$$\{x \mid x \in U, E(x)\}$$

Set Comprehension

Given a universe of objects U, select the elements from U that satisfy the predicate E:

$$\{x \mid x \in U, E(x)\}$$

Example: $A = \{n \mid n \in \mathbb{N}, \text{even}(n)\}$.

List Comprehension in Haskell

As an analogue, we have list comprehension in Haskell.

Assume `list :: [a]` and `property :: a -> Bool`. Then a new list can be defined with:

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

List Comprehension in Haskell

As an analogue, we have list comprehension in Haskell.

Assume `list :: [a]` and `property :: a -> Bool`. Then a new list can be defined with:

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

Example:

```
evens1 = [ n | n <- [0..], even n ]
```

List Comprehension in Haskell

As an analogue, we have list comprehension in Haskell.

Assume `list :: [a]` and `property :: a -> Bool`. Then a new list can be defined with:

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

Example:

```
evens1 = [ n | n <- [0..], even n ]
```

This can also be done with `filter`:

```
evens2 = filter even [0..]
```

Notation

If f is an operation, then

$$\{\, f(x) \mid P(x) \,\}$$

denotes the set of things of the form $f(x)$ where the object x has the property P. For instance,

$$\{\, 2n \mid n \in \mathbb{N} \,\}$$

is another notation for the set of even natural numbers.

Notation

If f is an operation, then

$$\{\, f(x) \mid P(x) \,\}$$

denotes the set of things of the form $f(x)$ where the object x has the property P. For instance,

$$\{\, 2n \mid n \in \mathbb{N} \,\}$$

is another notation for the set of even natural numbers.

Haskell counterpart for lists:

```
evens3 = [ 2*n | n <- [0..] ]
```

But note the difference:

```
naturals = [0..]

small_squares1 = [ n^2 | n <- [0..999] ]

small_squares2 = [ n^2 | n <- naturals , n < 1000 ]
```

The Russell Paradox

Define 'ordinary sets' with $R = \{ x \mid x \notin x \}$. Question: is R itself ordinary . . . ?

The Russell Paradox

Define 'ordinary sets' with $R = \{ x \mid x \notin x \}$. Question: is R itself ordinary . . . ?

If R is an ordinary set, then $R \in R$. Applying the definition of 'ordinary', this gives $R \in \{ x \mid x \notin x \}$. In other words, $R \notin R$, i.e., R is not ordinary. Contradiction.

The Russell Paradox

Define 'ordinary sets' with $R = \{ x \mid x \notin x \}$. Question: is R itself ordinary ...?

If R is an ordinary set, then $R \in R$. Applying the definition of 'ordinary', this gives $R \in \{ x \mid x \notin x \}$. In other words, $R \notin R$, i.e., R is not ordinary. Contradiction.

If R is not an ordinary set, then $R \notin R$. Applying the definition of 'ordinary', this gives $R \notin \{x | x \notin x\}$. In other words, $R \in \{x | x \in x\}$. But then $R \in R$, i.e., R is ordinary. Contradiction.

The Russell Paradox

Define 'ordinary sets' with $R = \{ x \mid x \notin x \}$. Question: is R itself ordinary . . . ?

If R is an ordinary set, then $R \in R$. Applying the definition of 'ordinary', this gives $R \in \{ x \mid x \notin x \}$. In other words, $R \notin R$, i.e., R is not ordinary. Contradiction.

If R is not an ordinary set, then $R \notin R$. Applying the definition of 'ordinary', this gives $R \notin \{x | x \notin x\}$. In other words, $R \in \{x | x \in x\}$. But then $R \in R$, i.e., R is ordinary. Contradiction.

Both the assumption $R \in R$ and the assumption $R \notin R$ lead to a contradiction. Conclusion: there is something wrong with the definition of 'ordinary set'.

The Halting Problem

Suppose there is a function `halt :: String -> String -> Bool` that checks whether a function (a program in some language, given by a string) is defined on a given input (also given by a string). Consider:

```
funny x | halts x x = undefined  -- Caution: this
        | otherwise = True       -- will not work
```

The Halting Problem

Suppose there is a function `halt :: String -> String -> Bool` that checks whether a function (a program in some language, given by a string) is defined on a given input (also given by a string). Consider:

```
funny x | halts x x = undefined  -- Caution: this
        | otherwise = True       -- will not work
```

`undefined` is predefined in the Haskell prelude, as follows:

```
undefined         :: a
undefined | False = undefined
```

The Halting Problem

Suppose there is a function `halt :: String -> String -> Bool` that checks whether a function (a program in some language, given by a string) is defined on a given input (also given by a string). Consider:

```
funny x | halts x x = undefined  -- Caution: this
        | otherwise = True       -- will not work
```

`undefined` is predefined in the Haskell prelude, as follows:

```
undefined         :: a
undefined | False = undefined
```

Now what about `funny funny`?

There can be no universal halts predicate ...

Suppose `funny funny` does not halt. Then by the definition of `funny`, we are in the first case. This is the case where the argument of `funny`, when applied to itself, halts. But the argument of `funny` is `funny`. Therefore, `funny funny` does halt, and contradiction.

There can be no universal halts predicate ...

Suppose `funny funny` does not halt. Then by the definition of `funny`, we are in the first case. This is the case where the argument of `funny`, when applied to itself, halts. But the argument of `funny` is `funny`. Therefore, `funny funny` does halt, and contradiction.

Suppose `funny funny` does halt. Then by the definition of `funny`, we are in the second case. This is the case where the argument of `funny`, when applied to itself, does not halt. But the argument of `funny` is `funny`. Therefore, `funny funny` does not halt, and contradiction.

There can be no universal halts predicate ...

Suppose `funny funny` does not halt. Then by the definition of `funny`, we are in the first case. This is the case where the argument of `funny`, when applied to itself, halts. But the argument of `funny` is `funny`. Therefore, `funny funny` does halt, and contradiction.

Suppose `funny funny` does halt. Then by the definition of `funny`, we are in the second case. This is the case where the argument of `funny`, when applied to itself, does not halt. But the argument of `funny` is `funny`. Therefore, `funny funny` does not halt, and contradiction.

Thus, there is something wrong with the definition of `funny`. The only peculiarity of the definition is the use of the `halts` predicate. This shows that such a `halts` predicate cannot be implemented.

Test for Equality of Functions

Such a test would solve the halting problem:

```
halts f x =  f /= g
  where g y | y == x    = undefined
            | otherwise = f y
```

Conclusion: functions cannot be in the class `Eq`.

Types

Types can be viewed as a regulation of the language to rule out paradoxes.

Types

Types can be viewed as a regulation of the language to rule out paradoxes.

A list counterpart to the Russell set $R = \{x \mid x \notin x\}$ cannot be defined in Haskell, for the function `notElem` has type `Eq a => a -> [a] -> B`

Types

Types can be viewed as a regulation of the language to rule out paradoxes.

A list counterpart to the Russell set $R = \{x \mid x \notin x\}$ cannot be defined in Haskell, for the function `notElem` has type `Eq a => a -> [a] -> B`

Type Classes

Type classes are sets of types for which certain common functions are defined. `Eq` is the class for which `==` and `/=` are defined. `Show` is the class for which `show` is defined. `Ord` is the class for which `compare` is defined. `Bounded` is the class for which `minBound` and `maxBound` are defined. `Num` is the class for which numerical operations like `(+)`, `(-)`, `(*)` are defined, and so on.

Empty Set, Singletons

A set A is empty if it has no elements. By extensionality, there is just one empty set, so we may give it a name: $\emptyset$.

A set A that has just one member d is called a singleton. The singleton whose only element is d is $\{d\}$. Do not confuse d with $\{d\}$.

Empty Set, Singletons

A set A is empty if it has no elements. By extensionality, there is just one empty set, so we may give it a name: $\emptyset$.

A set A that has just one member d is called a singleton. The singleton whose only element is d is $\{d\}$. Do not confuse d with $\{d\}$.

Empty List, Unit Lists

Empty list: `[]`. Unit list: `[d]`.

If `d :: a` then `[d] :: [a]`.

Operations on Sets

Intersection: $A \cap B = \{\, x \mid x \in A \wedge x \in B \,\}$

Union: $A \cup B = \{\, x \mid x \in A \vee x \in B \,\}$.

Difference: $A - B = \{\, x \mid x \in A \wedge x \notin B \,\}$.

Operations on Sets

Intersection: $A \cap B = \{ x \mid x \in A \wedge x \in B \}$

Union: $A \cup B = \{ x \mid x \in A \vee x \in B \}$.

Difference: $A - B = \{ x \mid x \in A \wedge x \notin B \}$.

Properties

$A \cap \emptyset = \emptyset$, $A \cup \emptyset = A$

$A \cap A = A$, $A \cup A = A$ (idempotence)

$A \cap B = B \cap A$, $A \cup B = B \cup A$ (commutativity)

$A \cap (B \cap C) = (A \cap B) \cap C$, $A \cup (B \cup C) = (A \cup B) \cup C$ (associativity)

$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

Powerset

The *powerset* of the set X is the set $\mathcal{P}(X) = \{ A \mid A \subseteq X \}$.

We have: $\emptyset \in \mathcal{P}(X)$ and $X \in \mathcal{P}(X)$.

Powerset

The *powerset* of the set X is the set $\mathcal{P}(X) = \{\, A \mid A \subseteq X \,\}$.

We have: $\emptyset \in \mathcal{P}(X)$ and $X \in \mathcal{P}(X)$.

The power set of $\{1, 2, 3\}$:

$$\left\{ \begin{array}{ccc} & \{1,2,3\} & \\ \{1,2\} & \{2,3\} & \{1,3\} \\ \{1\} & \{2\} & \{3\} \\ & \emptyset & \end{array} \right\}$$

Generalized Union and Intersection

Suppose that a set A_i has been given for every element i of a set I.

1. The *union* of the sets A_i is the set $\{x \mid \exists i \in I(x \in A_i)\}$.
   Notation: $\bigcup_{i \in I} A_i$.

Generalized Union and Intersection

Suppose that a set A_i has been given for every element i of a set I.

1. The *union* of the sets A_i is the set $\{x \mid \exists i \in I (x \in A_i)\}$.
   Notation: $\bigcup_{i \in I} A_i$.
2. The *intersection* of the sets A_i is the set $\{x \mid \forall i \in I (x \in A_i)\}$.
   Notation: $\bigcap_{i \in I} A_i$

Generalized Union and Intersection

Suppose that a set A_i has been given for every element i of a set I.

1. The *union* of the sets A_i is the set $\{x \mid \exists i \in I (x \in A_i)\}$.
   Notation: $\bigcup_{i \in I} A_i$.
2. The *intersection* of the sets A_i is the set $\{x \mid \forall i \in I (x \in A_i)\}$.
   Notation: $\bigcap_{i \in I} A_i$

Example: for $p \in \mathbb{N}$, let $A_p = \{mp \mid m \in \mathbb{N}, m \geq 1\}$. Then A_p is the set of all natural numbers that have p as a factor.

Generalized Union and Intersection

Suppose that a set A_i has been given for every element i of a set I.

1. The *union* of the sets A_i is the set $\{x \mid \exists i \in I (x \in A_i)\}$.
   Notation: $\bigcup_{i \in I} A_i$.
2. The *intersection* of the sets A_i is the set $\{x \mid \forall i \in I (x \in A_i)\}$.
   Notation: $\bigcap_{i \in I} A_i$

Example: for $p \in \mathbb{N}$, let $A_p = \{mp \mid m \in \mathbb{N}, m \geq 1\}$. Then A_p is the set of all natural numbers that have p as a factor.

What is $\bigcup_{i \in \{2,3,5,7\}} A_i$?

Generalized Union and Intersection

Suppose that a set A_i has been given for every element i of a set I.

1. The *union* of the sets A_i is the set $\{x \mid \exists i \in I (x \in A_i)\}$.
   Notation: $\bigcup_{i \in I} A_i$.
2. The *intersection* of the sets A_i is the set $\{x \mid \forall i \in I (x \in A_i)\}$.
   Notation: $\bigcap_{i \in I} A_i$

Example: for $p \in \mathbb{N}$, let $A_p = \{mp \mid m \in \mathbb{N}, m \geq 1\}$. Then A_p is the set of all natural numbers that have p as a factor.

What is $\bigcup_{i \in \{2,3,5,7\}} A_i$?

What is $\bigcap_{i \in \{2,3,5,7\}} A_i$?

The 'take' function

Consider the function `take :: Int -> [a] -> [a]` that does the following:

```
Prelude> take 10 [0..]
[0,1,2,3,4,5,6,7,8,9]
```

How would you implement this?

The 'take' function

Consider the function `take :: Int -> [a] -> [a]` that does the following:

```
Prelude> take 10 [0..]
[0,1,2,3,4,5,6,7,8,9]
```

How would you implement this?

```
take                  :: Int -> [a] -> [a]
take n _   | n <= 0   = []
take _ []             = []
take n (x:xs)         = x : take (n-1) xs
```

Representing Sets with Lists

Instead of taking sets as basic, and defining lists in terms of sets, we may also proceed the other way around, by representing sets as a special kind of lists.

Representing Sets with Lists

Instead of taking sets as basic, and defining lists in terms of sets, we may also proceed the other way around, by representing sets as a special kind of lists.

Removing duplicates with `nub`:

```
nub :: (Eq a) => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (remove x xs)
  where
  remove y []                  = []
  remove y (z:zs) | y == z     = remove y zs
                  | otherwise = z : remove y zs
```

Deleting Elements, Finding Elements

```
delete :: Eq a => a -> [a] -> [a]
delete x [] = []
delete x (y:ys) | x == y    = ys
                | otherwise = y: delete x ys
```

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

List Union and Intersection

```
union :: Eq a => [a] -> [a] -> [a]
union [] ys     = ys
union (x:xs) ys = x : union xs (delete x ys)

intersect :: Eq a => [a] -> [a] -> [a]
intersect []     s                = []
intersect (x:xs) s | elem x s     = x : intersect xs s
                   | otherwise    =     intersect xs s
```

Sublists: the Power List Operation

```
powerList  :: [a] -> [[a]]
powerList  [] = [[]]
powerList  (x:xs) = (powerList xs)
                    ++ (map (x:) (powerList xs))
```

```
Main> powerList [1,2,3]
[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
```