# Sets, types and functions

*Computational Semantics with Functional Programming*

Jan van Eijck (CWI, Amsterdam) & Christina Unger (UiL-OTS, Utrecht)

LOT Summer School
Leiden, June 2009

## What the lecture today is about

- refreshing basics about sets, relations, functions and types
- getting a taste of how to work with them in Haskell
- based on that, specifying a model that we will use tomorrow to interpret natural language expressions

## Outline

## Sets

## Sets and their members

Sets are collections of definite, distinct objects.

- the set of words in Michael Ende's *The Neverending Story*
- the set of Jason and the argonauts
- the set of even natural numbers greater than 47
- the set consisting of
  - the set of letters in the Greek alphabet
  - the set of kanji
- the set of Guybrush Threepwood, the Danish national anthem, a deck of Skat cards, and all slides of today
- the empty set (written as $\emptyset$)

## Sets and their members

The elements of a set are called members.

- If $a$ is an element of a set $A$, we write $a \in A$.
- If $a$ is not an element of $A$, we write $a \notin A$.

Sets are fully determined by their members. To check whether two sets $A, B$ are the same we have to check whether they have the same members:

- Is every element of $A$ also an element of $B$?
- Is every element of $B$ also an element of $A$?

If every element of $A$ is also an element of $B$, then $A$ is a subset of $B$, written as $A \subseteq B$.

Now, $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

## Different notations

There are several ways to specify a set.

- Description

  the set of colors of the Dutch flag

- Enumeration

  {red,white,blue}
  which is the same as {blue,white,red,red}

- Set comprehension

  $E = \{2n \mid n \in \mathbb{N}\}$
  $O = \{n \in \mathbb{N} \mid n \notin E\}$

- By means of operations on other sets

  $O = \mathbb{N} - E$

## Lists in Haskell

In our implementations we use lists. (The order is taken to be an arbitrary one. If necessary, we use a function `nub` to remove all duplicates.)

- `[]` is the empty list
- `[x]` is the singleton list containing only `x`
- `(x:xs)` is a non-empty list with `x` its head and `xs` its tail
- `[ x | x <- U, p x ]` is the list of all `x` in `U` with property `p`

## Examples

Here are some examples of lists of type `[Int]`, given by enumeration...

- `[2,3,5]`, `2:(3:(5:[]))`
- `[1..47]`, `['a'..'z']`
- `[1..]` (Long lives laziness!)

... and list comprehension:

- `[ x | x <- [1..], even x]`
- `[ square x | x <- [1..]]`
- `[ (x+y) | x <- [1..10], y <- [11..20], even x && odd y]`
- `[ [x..y] | x <- [2,3], y <- [4,7]]`

What about `[even x | x <- [1..]]`?

## Strings as lists of characters

Strings are of type `String`, which is an abbreviation for `[Char]`. That is, `"Hello world"` is the same like:

```
['H','e','l','l','o',' ','w','o','r','l','d']
```

A simple example using strings:

```
stems,suffixes,adjectives :: [String]

stems    = ["use","faith"]
suffixes = ["ful","less"]

adjectives = [ x ++ y | x <- stems, y <- suffixes]
```

## Relations

## Relations

There are numerous ways in which objects are related to each other.

- people: *is friends with*
- numbers: *is divisible by*
- colors and people: *is the favorite color of*
- numbers and cities: *is the number of inhabitants of*
- ...

Formally, a relation between two sets $A$ and $B$ is a set of ordered pairs $(a, b)$ such that $a \in A$ and $b \in B$.

The set of all such ordered pairs is called the Cartesian product of $A$ and $B$, written as $A \times B$.

A relation between $A$ and $B$ is a subset of $A \times B$.

## Example

- $A = \{a, b, c, d, e, f, g, h\}$
- $B = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- the set $P$ of positions on a chess board:

$$A \times B = \{(a, 1), (a, 2), \ldots, (b, 1), (b, 2), \ldots, (h, 1), \ldots, (h, 8)\}$$

- the set of colors: $C$=\{White,Black\}
- the set of chess figures: $F$=\{King,Queen,Knight,Rook,Bishop,Pawn\}
- the set of chess pieces: $C \times F$
  e.g. (White,King)
- the set of piece positions on a board: $(C \times F) \times P$
  e.g. ((White,King),($e$,1))
- the set of moves on a chess board: $(C \times F) \times (P \times P)$
  e.g. ((White,King),((e,1),(f,2)))

## Relations

We can generalize relations to sets of n-tupels.

- A unary relation (also called property) is the set that contains all elements having the property.
  - properties of people: *is a wizard*
  - properties of numbers: *is prime*
- A ternary relation is the set of triples that stand in the relation.
  - relating people: *a introduced b to c*
  - relating numbers: *x is the product of y and z*
  - relating people, books and numbers: *a read b more than x times*
- An *n*-ary relation is the set of *n*-tuples that stand in the relation.

## Implementing relations

We can implement:

- unary relations as lists of entities
- binary relations as lists of pairs of entities
- *n*-ary relations as lists of *n*-tuples of entities

Here is an implementation of a universe with 27 entities:

```
data Entity = A | B | C | D | E | F | G
            | H | I | J | K | L | M | N
            | O | P | Q | R | S | T | U
            | V | W | X | Y | Z | Unspec
            deriving (Eq,Show,Bounded,Enum)

entities = [minBound..maxBound]
```

## Implementing relations

Properties (unary relations) are lists of entities, i.e. of type `[Entity]`.

```
girl     = [A,D,G,S]           child   = girl ++ boy
boy      = [M,Y]
princess = [E]                 brave   = [D,Y]
dwarf    = [B,R]               laugh   = [A,G,E]
giant    = [T]                 cheer   = [M,D]
wizard   = [W,V]               shudder = [S]
```

```
is,does :: (Eq a) => a -> [a] -> Bool
is   = elem
does = elem
```

## Implementing relations

Binary relations are lists of pairs of entities, i.e. of type `[(Entity,Entity)]`.

```
help   = [(W,W),(V,V),(S,B),(D,M)]

admire = [(x,G) | x <- entities, x `is` child]

defeat = [(x,y) | x <- entities,
                  y <- entities,
                  x `is` dwarf && y `is` giant]
```

## Functions

## Functions

Functions are special relations: for any $(a, b)$ and $(a, c)$ in the relation it has to hold that $b = c$.

Thus a function from a set $A$ (domain) to a set $B$ (range) is a relation between $A$ and $B$ such that for each $a \in A$ there is one and only one associated $b \in B$.

In other words, a function is a mechanism that maps an input value to a uniquely determined output value.

## Functions: extensional view

| Kelvin | Celsius | Fahrenheit | |
|---|---|---|---|
| 0 | -273.15 | -459.67 | (absolute zero) |
| 273.15 | 0 | 32 | (freezing point of water) |
| 310.15 | 37 | 98.6 | (human body temperature) |
| 373.13 | 99.98 | 211.96 | (boiling point of water) |
| 505.9 | 232.8 | 451 | (paper auto-ignites) |
| 5778 | 5504.85 | 9940.73 | (surface temperature of the sun) |

Functions can be seen as sets of pairs of input and output values.

- function from Kelvin to Celsius: $\{(0, -273.15), \ldots\}$
- function from Celsius to Fahrenheit: $\{(-273.15, -459.67), \ldots\}$

## Functions: intensional view

Functions can also be seen as instructions for computation.

- function from Kelvin to Celsius: $x \mapsto x - 273.15$

  ```
  kelvin2celsius :: Float -> Float
  kelvin2celsius x = x - 273.15
  ```

- function from Celsius to Fahrenheit: $x \mapsto x \times \frac{9}{5} + 32$

  ```
  celsius2fahrenheit :: Float -> Float
  celsius2fahrenheit x = x * (9/5) + 32
  ```

  Example: $37 \times \frac{9}{5} + 32 \rightarrow 66.6 + 32 \rightarrow 98.6$

## Function composition

Function composition $f \cdot g$ is defined as $x \mapsto f(g(x))$.

Example:

Let $g$ be the function from Kelvin to Celsius and $f$ be the function from Celcius to Fahrenheit. Then $f \cdot g$ is a function from Kelvin to Fahrenheit, given by:

$$x \mapsto (x - 273.15) \times \frac{9}{5} + 32$$

In Haskell:

```
kelvin2fahrenheit x = celcius2fahrenheit (kelvin2celsius x)
```

Or shorter:

```
kelvin2fahrenheit = celsius2fahrenheit .  kelvin2celcius
```

## Functions in Haskell

Let us define an addition function `plus :: Int -> Int -> Int`.

- `plus x y = x + y`

Based on this, we can define a successor function.

- `succ y = plus 1 y`
- `succ = plus 1`
- `succ = (+ 1)`
- `succ = (1 +)`

So actually we could have defined `plus` like this:

- `plus = (+)`

## Characteristic functions

The characteristic function of a subset $A$ of some universe $U$ is a function that maps:

- all members of $A$ to the truth-value **True**
- all elements of $U$ that are not members of $A$ to **False**

Characteristic functions characterize membership of a set.

## Characteristic functions

Since we specified relations as sets, this means we can represent every relation as a characteristic function.

```
laugh' :: Entity -> Bool
laugh' x = x `elem` [A,G,E]

help'   :: (Entity,Entity) -> Bool
help' (x,y) = (x,y) `elem` [(W,W),(V,V),(S,B),(D,M)]

help''  :: Entity -> Entity -> Bool
help'' x y  = (x,y) `elem` [(W,W),(V,V),(S,B),(D,M)]

help''' :: Entity -> Entity -> Bool
help''' y x = (x,y) `elem` [(W,W),(V,V),(S,B),(D,M)]
```

## Lambda calculus

## Lambda calculus

The lambda calculus is a formal system for defining and investigating functions.

Functions are represented by means of function abstraction.
Consider $x^2 + y$.

- $\lambda x \mapsto x^2 + y$
- $\lambda y \mapsto x^2 + y$
- $\lambda x \mapsto (\lambda y \mapsto x^2 + y)$

Function application corresponds to substitution:

$$((\lambda x \mapsto (\lambda y \mapsto x^2 + y))\ 4)\ 7\ ((\lambda x \mapsto (\lambda y \mapsto x^2 + y))\ 4)\ 7$$
$$\rightarrow\ (\lambda y \mapsto 4^2 + y)\ 7 \rightarrow\ (\lambda y \mapsto 4^2 + y)\ 7$$
$$\rightarrow\ 4^2 + 7$$

## Lambda calculus (formal definition)

Expressions:

$$v ::= x \mid v'$$
$$E ::= v \mid (E\ E) \mid (\lambda v \mapsto E)$$

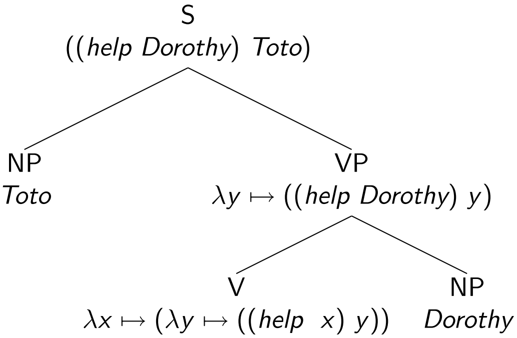
In Haskell we write `E E` and `\ x -> E`.

Reduction rule: $(\lambda x \mapsto E)\ A \rightarrow E[x := A]$

(When substituting expressions, we have to make sure that no variables get accidentally captured.)

## Lambda calculus in semantics

The lambda calculus allows to illustrate how the semantic derivation of a sentence proceeds in accordance with its syntactic structure.


S
$((\mathit{help}\ \mathit{Dorothy})\ \mathit{Toto})$
NP
$\mathit{Toto}$
VP
$\lambda y \mapsto ((\mathit{help}\ \mathit{Dorothy})\ y)$
V
$\lambda x \mapsto (\lambda y \mapsto ((\mathit{help}\ \ x)\ y))$
NP
$\mathit{Dorothy}$

## Two observations

First, reductions need not come to an end.

- $$\begin{aligned} & (\lambda x \mapsto x\ x)\ (\lambda x \mapsto x\ x) \\ \rightarrow\ & (\lambda x \mapsto x\ x)\ (\lambda x \mapsto x\ x) \\ \rightarrow\ & \ldots \end{aligned}$$
- and if you want things to get wild, try:
  $(\lambda x \mapsto x\ x\ x)\ (\lambda x \mapsto x\ x\ x)$

Second, we can build a lot of expressions that do not make sense.

- $4\ (\lambda x \mapsto x + 7)$
- $\mathit{Toto}\ (\lambda x \mapsto (\lambda y \mapsto ((\mathit{help}\ x)\ y)))$
- $(\lambda y \mapsto y^2)\ (\lambda x \mapsto x + 7)$

## Types

Types are sets of expressions, classifying them according to their combinatorial behaviour.

$$\tau ::= b \mid (\tau \rightarrow \tau)$$

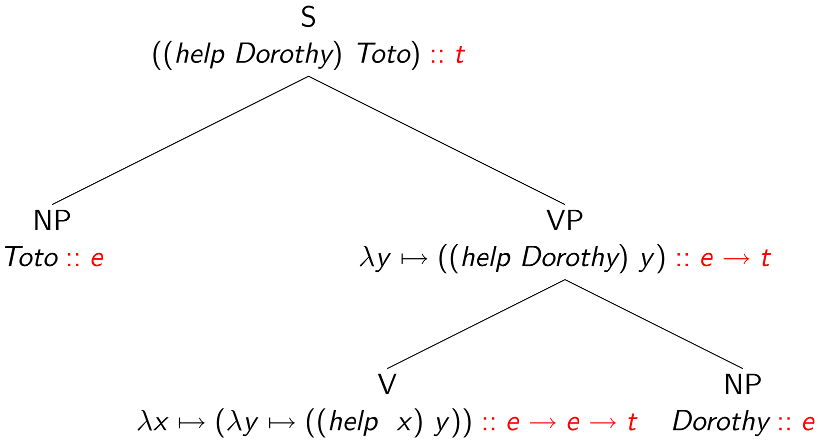
Basic types $b$:

- Haskell: `Int`, `String`, `Bool`, `[a]`, `(a,b)`, `Entity`, . . .
- Semantics: $e$ (`Entity`) and $t$ (`Bool`)

Examples:

```
help   ::  [(Entity,Entity)]
help'  ::   (Entity,Entity) -> Bool
help'' :: Entity -> Entity -> Bool
```

## Example


S
$((\mathit{help}\ \mathit{Dorothy})\ \mathit{Toto}) :: t$
NP
$\mathit{Toto} :: e$
VP
$\lambda y \mapsto ((\mathit{help}\ \mathit{Dorothy})\ y) :: e \rightarrow t$
V
$\lambda x \mapsto (\lambda y \mapsto ((\mathit{help}\ \ x)\ y)) :: e \rightarrow e \rightarrow t$
NP
$\mathit{Dorothy} :: e$

## Typed lambda calculus

Each lambda expression is assigned a type, specified as follows:

- Variables: For each type $\tau$ we have variables for that type, e.g. $x :: \tau$, $x' :: \tau$, and so on.
- Abstraction: If $x :: \delta$ and $E :: \tau$, then $(\lambda x \mapsto E) :: \delta \to \tau$.
- Application: If $E_1 :: \delta \to \tau$ and $E_2 :: \delta$, then $(E_1\ E_2) :: \tau$.

## Lambda calculus as a model of computation

The lambda calculus can also be thought of as a basic programming language, with functions corresponding to programs or procedures.

Functional programming languages like Haskell are indeed based on the lambda calculus. (You can even view them as an executable lambda calculus augmented with constants and datatypes.)

Now we have the formal tools at hand that we need in order to assign meanings to natural language expressions.

## Specifying a model

## Specifying a model

A model, with respect to which we interpret expressions, is a pair

$$M = (U, I)$$

where

- $U$ is a non-empty set of objects (the universe)
- $I$ is an interpretation function that maps
  - every proper name to an object in $U$
  - every $n$-ary predicate to an $n$-ary relation over $U$ (or, in our case, to its characteristic function)

## Specifying a model: the universe

As universe we take the one we introduced earlier:

```
data Entity = A | B | C | D | E | F | G
            | H | I | J | K | L | M | N
            | O | P | Q | R | S | T | U
            | V | W | X | Y | Z | Unspec
            deriving (Eq,Show,Bounded,Enum)

entities = [minBound..maxBound]
```

## Specifying a model: interpretation functions

Proper names are interpreted as entities.

```
alice,dorothy,goldilocks,littleMook,atreyu :: Entity

alice      = A
dorothy    = D
goldilocks = G
littleMook = M
atreyu     = Y
```

## Specifying a model: interpretation functions

One-place predicates are interpreted as characteristic functions of unary relations.

```
list2OnePlacePred :: [Entity] -> (Entity -> Bool)
list2OnePlacePred xs = \ x -> x ‘elem‘ xs

girl,boy,wizard,child,laugh,cheer,shudder :: Entity -> Bool

girl    = list2OnePlacePred [A,D,G,S]
boy     = list2OnePlacePred [M,Y]
wizard  = list2OnePlacePred [W,V]
child   = \ x -> (girl x || boy x)

laugh   = list2OnePlacePred [A,G,E]
cheer   = list2OnePlacePred [M,D]
shudder = list2OnePlacePred [S]
```

## Specifying a model: interpretation functions

Two-place predicates are interpreted as characteristic functions of binary relations.

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)

help,admire,defeat :: Entity -> Entity -> Bool

help   = curry (‘elem‘ [(W,W),(V,V),(S,B),(D,M)])
admire = curry (‘elem‘ [(x,G) | x <- entities, person x])
defeat = curry (‘elem‘ [(x,y) | x <- entities,
                                y <- entities,
                                dwarf x && giant y])
```

## A gimmick

## Argument reduction

The entity Unspec can be used to leave arguments implicit. We can thus define underspecified relations.

```
eat :: Entity -> Entity -> Bool
eat = curry (‘elem‘ [(A,I),(D,Unspec)])
```

We can also use it to define argument reducing functions, such as passivization.

```
passivize :: (Entity -> Entity -> Bool) -> (Entity -> Bool)
passivize r = \ x -> r Unspec x
```

But:
```
Relations> (passivize eat) I
False
```

## Fixing it

```
close :: [(Entity,Entity)] -> [(Entity,Entity)]
close r = r ++ [ (Unspec,y) | x <- entities, y <- entities,
                              (x,y) `elem` r ]
            ++ [ (x,Unspec) | x <- entities, y <- entities,
                              (x,y) `elem` r ]
```

```
Relations> close [(A,I),(D,Unspec)]
[(A,I),(D,Unspec),(Unspec,I),(Unspec,Unspec),(A,Unspec),
 (D,Unspec)]
```

```
eat' :: Entity -> Entity -> Bool
eat' = curry (`elem` (close [(A,I),(D,Unspec)]))
```

```
Relations> (passivize eat') I
True
```

## Argument reduction

Reflexive pronouns like *himself* and *herself* can also be seen as argument reducing functions. Applied to a two-place predicate, they unify the two argument positions, thus yielding a one-place predicate.

```
self :: (a -> a -> b) -> (a -> b)
self r = \ x -> r x x
```

Two nice consequences:

- Reflexive pronouns can only refer to co-arguments but not to antecedents that are ‘further away’.

  * *Alice$_i$ said that Dorothy likes herself$_i$.*

- Reflexives cannot occur in subject position.

  * *Herself likes Dorothy.*

## More

- This afternoon in Yoad Winter's course: more on types and model structure
- Tomorrow in this course: typed meanings for natural language