Chapter 2: Basic Elements of the Language Haskell

Learning targets of this chapter:

1. Haskell syntax
2. Understanding the most frequent error messages
3. Case analyses
4. Local definitions
5. Functions as arguments and results
6. Tuples, lists and list comprehensions
7. Polymorphism, overloading
8. Algebraic data types
9. Type classes and instances

Variables, Value Sets, Types

paradigm	variable stands for
imperative	updatable memory location
functional	carrying a unique value, or undefined ($\perp$)

Predefined types in Haskell:

type	value set
`()` "unit"	$\{\perp,$ `()` $\}$
`Bool`	$\{\perp,$ `False`, `True` $\}$
`Int`	$\{\perp\} \cup$ `[(minBound::Int)..(maxBound::Int)]`
`Integer`	$\{\perp\} \cup \mathbb{Z}$

Others: printable: `Char`/`String`, floating-point numbers: [`Float`|`Double`], rational numbers: `Rational`, complex numbers: `Complex` [`Float`|`Double`]

Haskell Expressions and Type Specifications

test with interpreter `ghci`

- Arithmetic expressions as usual: `2+3`, `7*(5-2)`, ...
- Type specifications postfixed with `::`*`Typname`*
  - `(2^12)^5` ⇝ `1152921504606846976` (default: `Integer`)
  - `((2^12)^5)::Int` ⇝ `0` (overflow not specified, here possibly modulo 2^{32})
- Arithmetic operators force type equality,
  constants and operators are overloaded (e.g., `2` ∈ `Int`, `2` ∈ `Float`, . . .)
  - `(2::Int)+3`

    `3` and the result are of type `Int`
  - `((2::Int)+3)::Integer`

    type error, no implicit type change (coercion) in Haskell!
  - `(fromIntegral ((2::Int)+3))::Integer`

    okay, explicit type change (conversion) via `fromIntegral`

Interpreter `ghci`

- Evaluation (return the value of 3+5)

```
Prelude> 3+5
8
```

- Definition (let x have value 10001; value of $x * (3x - 5 + x^2)$?)

```
Prelude> let x=10001
Prelude> x*(3*x-5+x*x)
1000600039999
```

- Type check (what is the type of expression `(+)` ?)

```
Prelude> :t (+)
forall a. (Num a) => a -> a -> a
```

Response: for all admissible types `a` must hold that `a` is a number type (`Num`) and the arguments and the result of `(+)` have the same type `a`

Scopes of Variables in ghci

```
Prelude> let x=5      let the value of x be 5
Prelude> let y=x+1    let the value of y be x+1, i.e., 6
Prelude> y==x+1       is y equal to x+1 ?
True                  response: True, type: Bool
Prelude> let x=0      new variable whose name is also x
Prelude> y==x+1       is y equal to x+1 ?
False                 response: False, type: Bool
Prelude> y            what is the value of y ?
6                     response: 6
```

- New definition (`let x =`) causes occlusion of old name
- Referential transparency holds (consider scope of `x`!)
- Static variable binding (`y` to `x+1`, not to `x+1`)

Syntax of Function Evaluation (Application)

```
Prelude> let f x = x^2-3*x+1  define function f
Prelude> f 2   apply f to number 2 by
-1                     juxtaposition (separated by space(s))
Prelude> f2    Fehler: f2 is considered a name (identifier)
Prelude> f(2)  allowed, but don’t use parentheses as token separators
-1
Prelude> f (f 2)      apply f to the result of f 2
5
Prelude> f f 2        type error, juxtaposition is left-associative,
                      and f cannot be the first argument of f
Prelude> (f . f) 2    function composition: f . f for f∘f
Prelude> f $ f 2      right-associative application operator $ has
5                     (like all operators) lower binding power than
                      the application via juxtaposition
```

Note: `$`/`.` (no spaces) have different meaning (`$f` splice, `A.` namespace)!

Haskell Module File

Datei `Mean.hs`

```
module Mean where   -- headline with module name  Mean
-- type definitions
sum3, mean3 :: Double -> Double -> Double -> Double
{-             arg.1     arg.2     arg.3     result
function definitions (without let) -}
sum3   x  y  z  =  x+y+z
mean3  x  y  z  =  sum3 x y z  / 3
```

Use with `ghci`

```
Prelude> :l Mean           load definitions
Compiling Mean             ( Mean.hs, interpreted )
Ok, modules loaded: Mean.
Mean> mean3 3 9 1          call of mean3
4.333333333333333
```

Error Messages (1)

```
Prelude> let x = 9
Prelude> x -7
2
Prelude> id -7   error
<interactive>:1:
    No instance for (Num (a -> a))
      arising from use of ‘-’ at <interactive>:1
    In the definition of ‘it’: it = id - 7
```

Parser does not analyze types: `-` is taken as infix operator, but `id` is a function, not a number.

Better:

```
Prelude> id (-7)
-7
```

Error Messages (2)

```
Prelude> let { f::Integer->Integer; f x = x+1 }
Prelude> f 5
6
Prelude> f 4.0   error
<interactive>:1:
    No instance for (Fractional Integer)
      arising from the literal ‘4.0’ at <interactive>:1
    In the first argument of ‘f’, namely ‘4.0’
    In the definition of ‘it’: it = f 4.0
```

- `f` expects an argument of type `Integer`
- `4.0` is in type class `Fractional` (number types with restless division)
- `Integer` $\notin$ `Fractional` (no instance for `(Fractional Integer)`)

Error Messages (3)

```
Prelude> let { f :: Integer -> Integer; f x y = x+y } error
<interactive>:1:
    Couldn't match `Integer' against `t -> t1'
        Expected type: Integer
        Inferred type: t -> t1
    In the definition of `f': f x y = x + y
```

- According to the signature, `f x` is of type `Integer`
- According to the defining equation, `f x` is of type `t -> t1`, i.e., a function (expects `y` as additional argument)

Error Messages (4)

```
Prelude> let { f :: a->a; f x = x^2 }
<interactive>:1:
    Could not deduce (Num a) from the context ()
      arising from use of ‘^’ at <interactive>:1
    Probable fix:
        Add (Num a) to the type signature(s) for ‘f’
    In the definition of ‘f’: f x = x ^ 2
```

Type specification `f :: a->a` is to general:

`^2` requires restriction to number types (`a` ∈ `Num`).

Remedy: introduction of a new context `Num a =>`

```
Prelude> let { f :: Num a => a->a; f x = x^2 }
```

The Type Class of Numbers (`Num a`)

```
Prelude> let { square :: Num a => a->a; square x = x^2 }
Prelude> square (2::Int)
4
Prelude>  square 2.5
6.25
```

`Prelude> square ((Data.Ratio.%) 2 3)` rational number $\frac{2}{3}$

`4 % 9` $\frac{4}{9}$

`Prelude>  square ((Data.Complex.:+) 0 1)` complex number i (0 + 1i)

`(-1.0) :+ 0.0` -1

`Prelude> square '2'` error: '2' is a character, not a number

```
<interactive>:1:
    No instance for (Num Char)
      arising from use of `square' at <interactive>:1
    In the definition of `it': it = square '2'
```

Type Definition, not Search of Type Errors (1)

Contexts can be deduced automatically:

```
Prelude> let square x = x^2
Prelude> :t square
square :: forall a. (Num a) => a -> a
```

Why specify a type?

```
Prelude> let sumup n = n*(n+1)/2        without type specification
Prelude> sumup (10^20)
5.0e39                                  not exact
Prelude> sumup ((10^20)::Integer)       type error
<interactive>:1:
    No instance for (Fractional Integer)
      arising from use of ‘sumup’ at <interactive>:1
    In the definition of ‘it’: it = sumup ((10^20) :: Integer)
```

Why the demand for `Fractional`?

Type Definition, not Search of Type Errors (2)

Question: why the demand for `Fractional`?

```
Prelude> let sumup n = n*(n+1)/2
Prelude> :t sumup
forall a. (Fractional a) => a -> a
```

Answer: because of the definition of `sumup` can be recognized earlier!

```
Prelude> let { sumup :: Integer->Integer; sumup n = n*(n+1)/2 }
<interactive>:1:
    No instance for (Fractional Integer)
      arising from use of ‘/’ at <interactive>:1
    In the definition of ‘sumup’: sumup n = (n * (n + 1)) / 2
```

the cause is /

Solution: replace `/` by integer division `div` (type class `Integral`)

```
Prelude> let {sumInt :: Integral a => a->a; sumInt n = n*(n+1)‘div‘2}
Prelude> sumInt ((10^20)::Integer)
5000000000000000000050000000000000000000
```

Type Definition, not Search of Type Errors (3)

1. Settle on the type of a function, before you implement it.
2. Compare your choice with the type delivered by the Haskell interpreter.

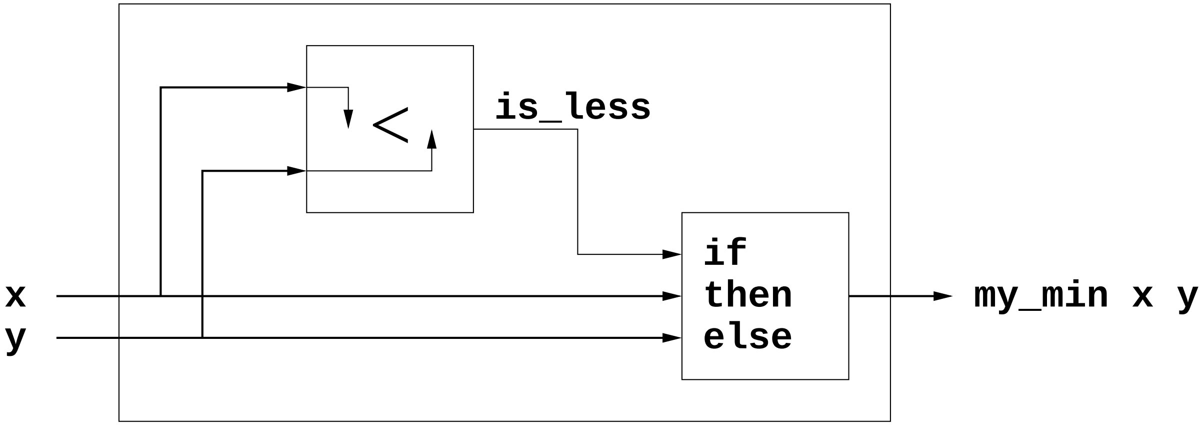
Rule: the type should be as general as possible (reusability), but the algorithm may require restrictions.

- Domain restrictions: `Integer` rather than type class `Integral`
- Accuracy: `Double` rather than type class `Fractional`

For complex functions, the Haskell interpreter may not be able to identify the most general type. If so, supply the specific type you need. Type error message, e.g.: `...is not polymorphic enough`. Not learning target of this chapter and dependent on the version of the interpreter, therefore no example here.

Stepwise Design

1. Structural layout (usually just imaginary)



2. Specification of type (not possible? ⇒ design error, back to 1.)

```
my_min :: Double -> Double -> Double
```

3. Specification of defining equation(s)

```
my_min x y = let is_less = x<y
             in  if is_less then x else y
```

Case Distinctions

1. `if-then-else`

```
implies x y = if x then y else True
```

2. Multiple equations

```
implies True  y = y
implies False _ = True
```

3. Guards

```
implies x y | x     = y
            | not x = True
```

4. `case`

```
implies x y = case x of
                True -> y
                False -> True
```

`if` *cond* `then` x `else` y

- Syntax: expression, maybe partial, `else`-branch must be present.
  Bsp.: `7 + (if doubleIt then (*2) else (*1)) x - y`

- Types:
  type(*cond*) = `Bool`, type(x) = type(y) = type(`if-then-else`)

- Semantics:

  1. if *cond* $= \perp$, result $\perp$ (non-termination)
  2. if *cond* = `True`, result x
  3. if *cond* = `False`, result y

  Only one branch is evaluated, the value of the other may be undefined (essential for termination of a recursion!).

Local Definitions

Two options: `let` and `where`, preferably

- `let`, to highlight the steps of a computation,
- `where`, to specify an expression in more detail.

Sometimes only a single option:

- in an expression only `let`

```
f x = x + 3 * (let z = x^3-5 in  z*x+z) - 2
```

- with multiple guards only `where`

```
intpow :: Double -> Integer -> Double
intpow x n | n<0  = 1 / pow x (-n)
           | n>=0 = pow x n
  where pow x n = x^n
```

Ex. Roots of aX^2+bX+c

```
root :: Double -> Double -> Double -> Bool -> Double
root a b c chooselower
 = let p = b/a
       q = c/a
       disc  = (p/2)^2-q
       rootd = sqrt disc
   in if disc < 0
        then error "root: negative discriminant"
        else -p/2+(if chooselower then -rootd else rootd)
```

```
root 1 1 (-2) False ⇝ 1.0
root 1 1 (-2) True ⇝ -2.0
root 1 0 5 False
 ⇝ *** Exception: root: negative discriminant
```

`rootd = sqrt disc` just a definition; evaluated only at point of use.

The Layout Style of Haskell

without layout	with layout

`f x = let { y=x+1; z=y*y } in z`	`f x = let y=x+1`
	`          z=y*y`
	`      in z`

- Objective: make programs more legible
- Lines of a block must begin in the same column
- Applies to `let`, `where`, `case`, etc.
- Applicable for every block separately

Special Issues of let-Expressions

1. The order of definitions is immaterial.
2. The definition in the innermost block applies.
3. Computations happen only at the point of use.

```
f x = let y   = 4            local number value
          g w = w+y          local function
          z   = let a = x+y  (to 1: y of following line)
                    y = g x  (g uses outer x)
                    z = y-x  (to 2: y of previous line)
                    b = ⊥    (to 3: ignored)
                in z+x+(if g x > x then a else b)
          u = y+z            (to 2: y=4 of first line)
      in u*u
```

Functions as Arguments

Function `restricted_equal` is supposed to check whether its arguments, functions `f,g::Int->Int`, agree on the set $\{0, 1, 2, 3\}$.

```
restricted_equal :: (Int->Int) -> (Int->Int) -> Bool
restricted_equal f g =     (f 0 == g 0) && (f 1 == g 1)
                        && (f 2 == g 2) && (f 3 == g 3)
```

Remark: two polynomials of degree n are equal if they agree at $n+1$ points.

Equality check of two polynomials of degree 3:

```
Test> let p1 x = x^3-1
Test> let p2 x = (x-1)*(x^2+x+1)
Test> restricted_equal p1 p2
True
```

A Function as Result

Function `twice` is supposed to take a function `f::Int->Int` and apply it twice successively.

```
twice :: (Int->Int) -> (Int->Int)
twice f = f . f
```

Test:

```
Test> (+1) 6
7
Test> let f = twice (+1)     f corresponds to (+2)
Test> f 6
8
Test> let g = twice f        g corresponds to (+4)
Test> g 6
10
```

Ex.: Numeric Differentiation Operator

Type: function as argument and as result.

```
diff :: (Double->Double) -> ( Double->Double )
diff f = ...    ? the right side should be a function
```

Solution: specify how the function acts on its argument
(extensional definition, not necessary for `twice`)

```
diff f x = let h = 1.0e-10
           in (f (x+h) - f x) / h
```

Aditional argument okay: second pair of parentheses in the type unnecessary.

```
Test> let f = sin
Test> let f' = diff f
Test> f' 0                      f' ist cos
1.0
```

!!! `(diff . diff)` is functionally possible, but numerically *not* the 2. derivative !!!

Tuples and Lists

structure	# components	component type	type example
tuple	fixed	variable	`(Int,Bool)`
list	variable	fixed	`[Double]`

Examples:

- `(2,True) :: (Integer, Bool)`
- `[1.2, -3.456e23, 67, 7.31, 3.2E-5] :: [Double]`
- `[("inc",(+1)), ("double",(*2))] :: [(String, Int -> Int)]`

Tuples

Let the type of expression x_i be α_i;
then the following expression-type correspondences hold:

expression	type
(x_0,x_1)	(α_0,α_1)
(x_0,x_1,x_2)	$(\alpha_0,\alpha_1,\alpha_2)$
(x_0,x_1,x_2,x_3)	$(\alpha_0,\alpha_1,\alpha_2,\alpha_3)$
...	...

Differently parenthesized tuples have different types,
although the corresponding value sets are isomorphic:

$$\boxed{((\alpha,\beta),\gamma) \overset{\text{type}}{\neq} (\alpha,\beta,\gamma)}$$

Component Selection via Pattern Matching

Principle: (“constructor style”)

structure (here a tuple) on the left side of a defining equation

```
fst (x,y) = x
snd (x,y) = y
```

Application: (1) match actual with formal parameter

(2) assign each variable the corresponding value

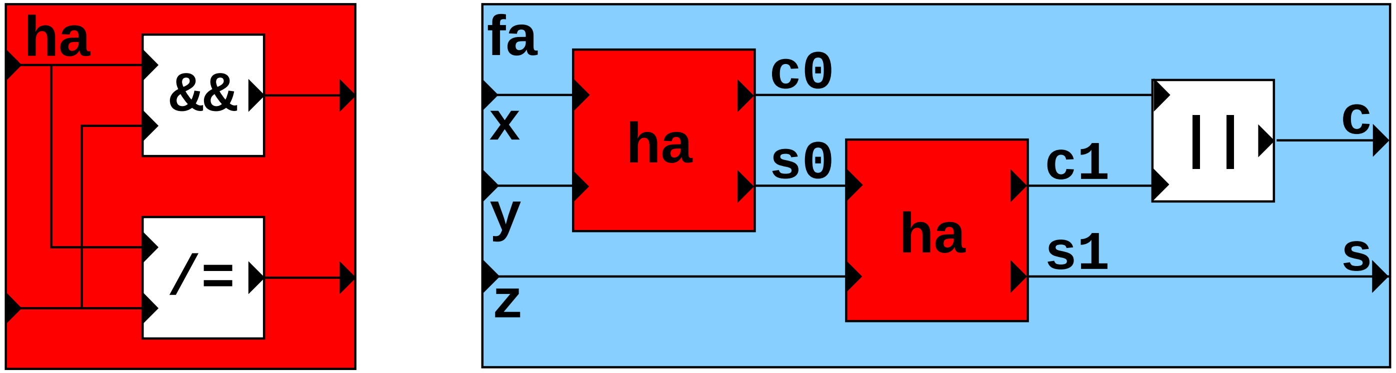
Beispiele:

```
proj_lastname_pay (_,(_,lastname),_,pay)
                   = (lastname,pay)

implies (True,False) = False     match with constants
implies _            = True      in the order specified
```

Tuple as the Result of Functions

Ex.: logic circuit



```
ha :: (Bool,Bool) -> (Bool,Bool)        half adder
ha (x,y) = (x && y, x /= y)

fa :: (Bool,Bool,Bool) -> (Bool,Bool)  full adder
fa (x,y,z) = let (c0,s0) = ha (x,y)
                 (c1,s1) = ha (s0,z)
             in (c0 || c1, s1)
```

Lists

- Homogeneous data structure (elements have a common type)
- Generated via two *data constructors*

name	meaning	symbol	type
nil	empty list	`[]`	$\forall\alpha.[\alpha]$
cons	add at list head	`(:)`	$\forall\alpha.\alpha \to [\alpha] \to [\alpha]$

- **cons** is usually a right-associative infix operator
- Syntactic sugar: `[1,2,3,4]` for `1:2:3:4:[]`
- List as argument with pattern matching

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

Popular Functions on Lists

name	type	example
null	[a]->Bool	null [] ⇝True
head	[a]->a	head [1,2,3,4] ⇝1
tail	[a]->[a]	tail [1,2,3,4] ⇝[2,3,4]
map	(a->b)->[a]->[b]	map (*2) [1,2,3] ⇝[2,4,6]
length	[a]->Int	length [1,3,5,7,9] ⇝5
(++)	[a]->[a]->[a]	[1,2,3]++[4,5] ⇝[1,2,3,4,5]
(!!)	[a]->Int->a	[8,5,7,3] !! 1 ⇝5
take	Int->[a]->[a]	take 3 [4,5,6,7,8] ⇝[4,5,6]
drop	Int->[a]->[a]	drop 3 [4,5,6,7,8] ⇝[7,8]
concat	[[a]]->[a]	concat [[1,5],[],[3,7,2]] ⇝[1,5,3,7,2]

Arithmetic Sequences

expression	generated list	stride
`[1..5]`	`[1,2,3,4,5]`	default: `1`
`[1,3..12]`	`[1,3,5,7,9,11]`	`2` (`|3-1|`)
`[4..2]`	`[]`	`-1`, but negative default prohibited
`[9,6..0]`	`[9,6,3,0]`	`-3`

- Arithmetic expressions allowed: `let x=7 in [x..2*x]`
- Also for other enumeration types `[False .. True]` space before `..`
- Even infinite lists definable and usable, but only a finite prefix can be output:

expression	generated list
`take 9 [1..]`	`[1,2,3,4,5,6,7,8,9]`
`take 7 [4,2..]`	`[4,2,0,-2,-4,-6,-8]`

Characters and Strings

- Unicode characters, type `Char`
  - `import Data.Char` required for many functions
  - constants, e.g., `'a'`, `'\n'` , `'\112'`
  - conversion to/from `Int`: `digitToInt`, `intToDigit`, `ord`, `chr`
  - form of character: `isAscii`, `isUpper`, `isLower`, `toUpper`, `toLower`
- Strings, character lists

  `type String = [Char]`   `type synonym`

  - syntactic sugar: `"Hallo"` for `['H','a','l','l','o']`
  - sequence: `['a','c'..'o']` ⇝ `"acegikmo"`
  - `show` function: `(Show a) => a->String`   e.g., `Float` ∈ `Show`

    ex.: `let x=27.5 in (Preis: "++ show x ++ "Euro")`

List Comprehensions (1)

generation of lists, akin to set comprehension

Ex.: pythagorean_triples $= \{ \, (x,y,z) \in \mathbb{N}^3 \mid x^2 + y^2 = z^2 \, \}$

Naive implementation with upper limit `n` for the numbers:

```
ptriple :: Integer -> [(Integer,Integer,Integer)]
ptriple n =
 [ (x,y,z)             element in the result list
 | x <- [1..n],        pick x successively
   y <- [1..n],        for each x pick y successively
   z <- [1..n],        for each x and y pick z successively
   x^2+y^2==z^2        accept choices of x, y, z, if equation holds
 ]
```

```
ptriple 15 ⇝ [(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(12,5,13)]
```

List Comprehensions (2)

correspondence to nested `for`-loops

- C:

```
count=0;
for (i=a;i<=b;i++)
   if (p(i)) { limit = f(i);
               for (j=0;j<=limit;j++)
                  result[count++] = exp(i,j); }
```

- Haskell:

```
result = [ exp (i,j) | i<-[a..b],
                       p i,
                       let limit = f i,
                       j <- [0..limit] ]
```

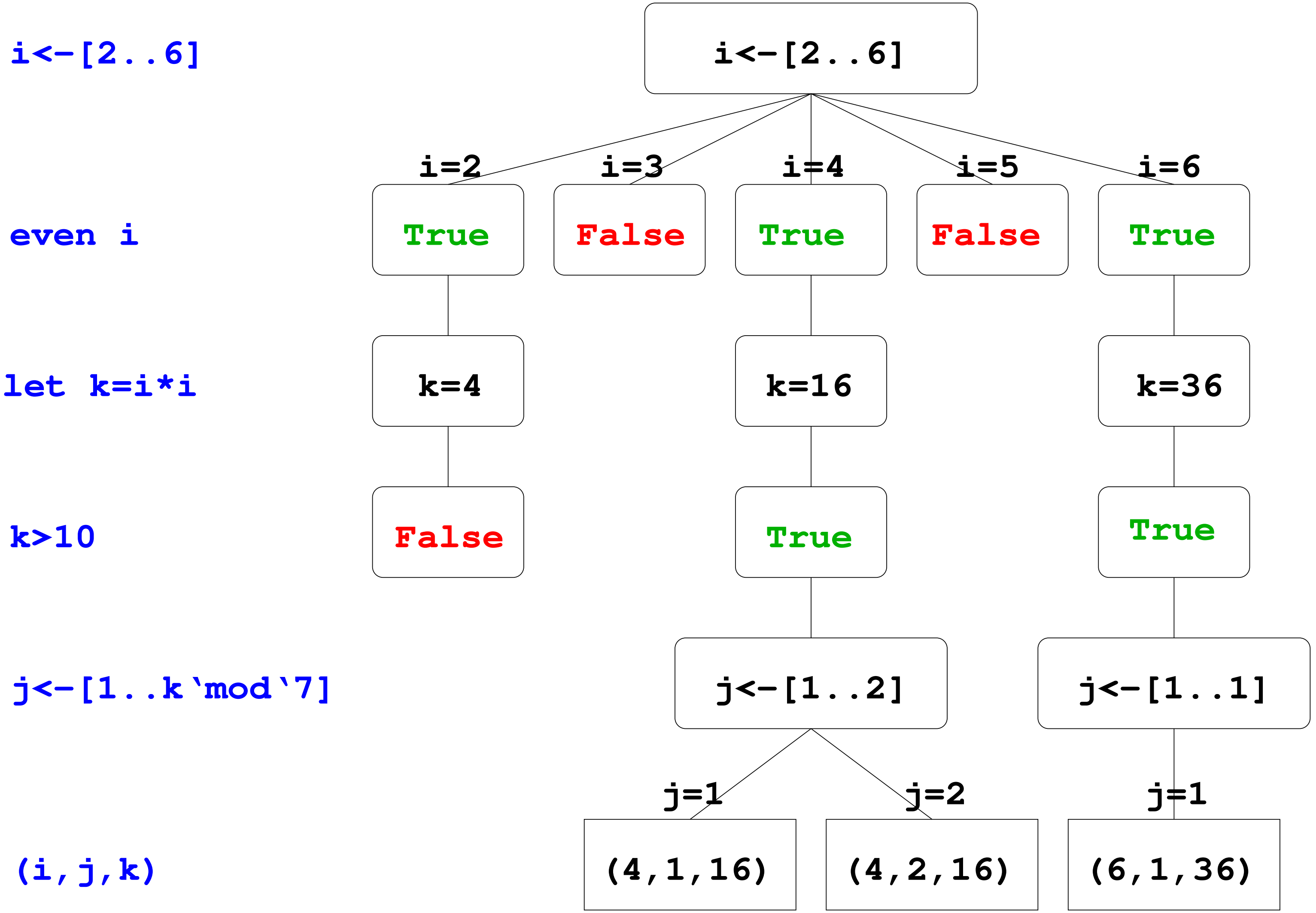
List Comprehensions (3)

- Syntax: `[` *exp* `|` q_0 `,` ... `,` q_n `]`,
  with *exp* element in the result list and q_i qualifier

- Semantics: generation of a decision tree of variable bindings with levels
  q_0,q_1..q_n,*exp*;
  result: list of values of *exp* at the leaves

- Types of qualifiers:
  - generator: *pattern* `<-` *list*
    for each list element: if pattern match succeeds, new subtree with bindings of the variables in the pattern
    Ex. `(True,x) <- [(True,4),(False,7),(True,2)]`
    generates two subtrees with the bindings `x=4` and `x=2`.
  - Guard: boolean expression (⇝`False`: truncation of subtree)
  - Local definition: `let` *pattern* `=` *expression* (no `in`)

Beispiel: `[(i,j,k) | i<-[2..6], even i, let k=i*i, k>10, j<-[1..k`mod`7]]`



Ergebnis: `[(4,1,16),(4,2,16),(6,1,36)]`

Semantics of List Comprehensions

Inductively defined: let `Q` be a sequence of qualifiers, `e` an expression, `b` a boolean expression, `p` a pattern, `l` a list and `decls` a sequence of declarations.

1. Empty list of qualifiers (only internally)

```
[ e | ] = [ e ]
```

2. Guard

```
[ e | b, Q ] = if b then [ e | Q ] else []
```

3. Generator (let `ok` be a fresh variable)

```
[ e | p <- l, Q ] = let ok p = [ e | Q ]
                        ok _ = []
                    in concat (map ok l)
```

4. Local definition

```
[ e | let decls, Q ] = let decls in [ e | Q ]
```

Polymorphism, Overloading and Type Classes

- Polymorphism: type-independence

  Ex.: `length :: [a] -> Int` independent of the type of `a`

- Overloading: one name for several functions ($+^{\mathbb{Z}}$, $+^{\mathbb{Q}}$)

  Ex: `(+) :: (Num a) => a -> a -> a`

  `(+)` only defined for elements of type class `Num`

- Type class: set of types, for which a common set of overloaded functions has been declared

  Bsp.: `Num` has overloaded operators `(+)`, `(-)` and `(*)`;

  elements of `Num`: `Int`, `Integer`, `Float`, `Double`, `Rational` ...

Polymorphic Function

- One or more, but only unrestricted type variables.
- No type-specific operations on elements with polymorphic type.
- Sometimes the function is fully determined by the type definition:

type	`a->a`	`(a,b)->a`	`a->b->a`
only total function	`id`	`fst`	`const`

- Implementation only governs polymorphic data, but does not compose them. Ex.:

```
my_replicate :: Int -> a -> [a]
my_replicate n x = [ x | _ <- [1..n] ]
```

Each element of the result list is a reference to the common data object `x`.

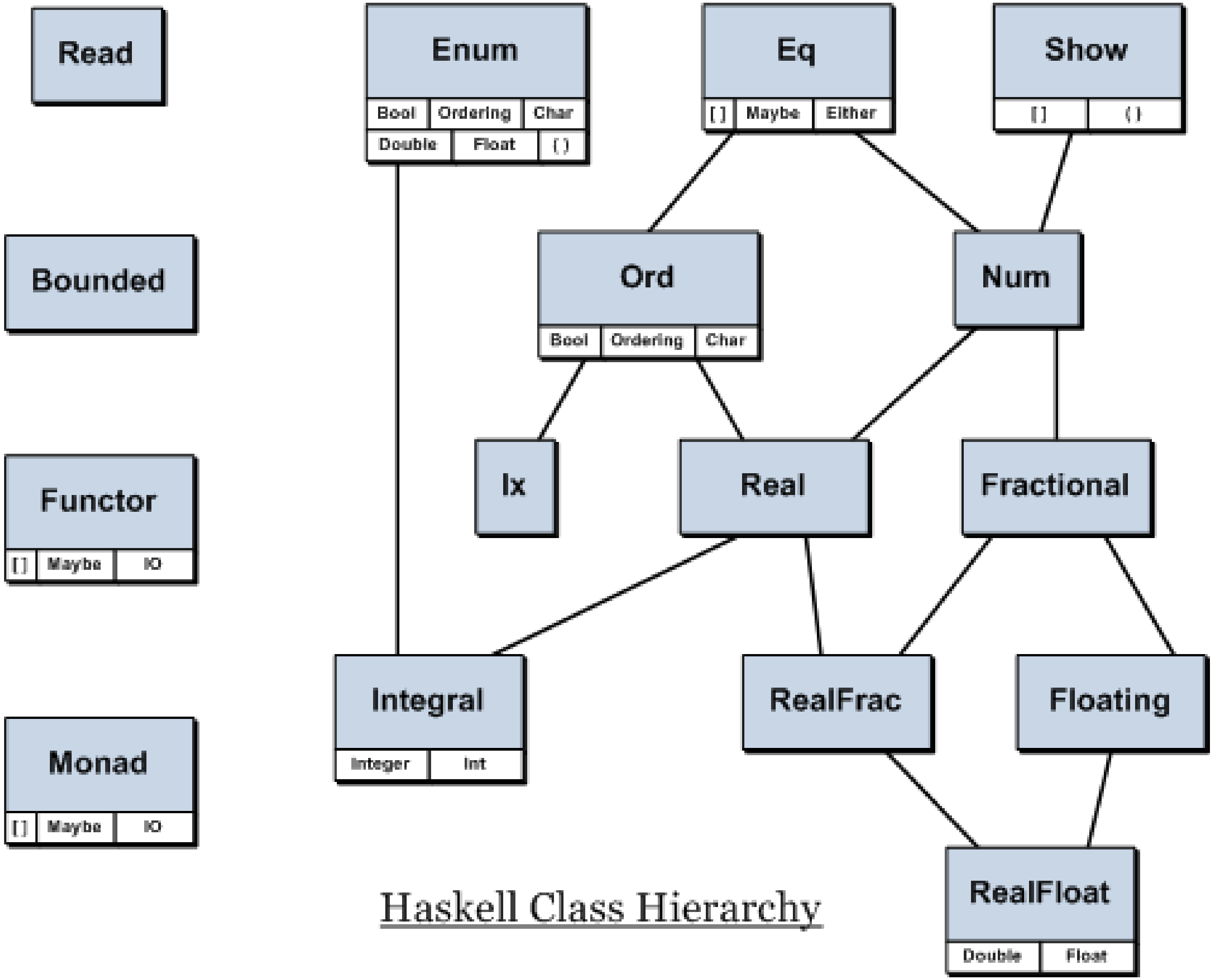
Overloaded Function

- At least one restricted type variable.
- Operations on the elements of the restricted type variables:
  1. directly via use of an operation at a lower software layers, or a hardware operation; implementation by specialization
     - `(+)::Int`, e.g., hardware instruction for the ALU
     - `(+)::Float`, e.g., hardware instruction for the floating-point unit
     - `(+)::Integer`, e.g., function of the GNU Multiple Precision Library
  2. indirectly via use of other overloaded functions;
     - implementation by type information in run-time data and case distinctions by transition to (1)
     - no specialization in order to avoid explosive code duplication

Some Predefined Type Classes in Haskell

name	el. property	ex. fct.	element
`Show`	printable	`show`	all but `IO`, `->`
`Read`	readable	`read`	all but `IO`, `->`
`Eq`	comparable	`(==)`	all but `IO`, `->`
`Ord`	ordered	`(<)`	all but `IO`, `->`
`Num`	numbers	`(+)`	`Int`, `Float`, ...
`Enum`	"enumerable"	`[..]`	`Bool`, `Int`, `Char`, ...
`Integral`	whole numbers	`mod`, `div`	`Int`, `Integer`
`Fractional`	invertible	`(/)`	`Float`, `Double`, `Rational`
`Real`	rational	`toRational`	`Int`, `Integer`, `Float`, `Double`, `Rational`
`Floating`	floating-point	`sin`	`Float`, `Double`

Predefined Type Class System in Haskell



Haskell Class Hierarchy

Type Synonyms (`type`)

- An additional name for the same type.
- Ex.: `type IndexValue = (Int,Int)` for index-value pairs.
- Benefit: the type information reflects an implicit commentary.
- Drawback: logic errors are not automatically recognized.
  - `type Point = (Int,Int)`

    `Point` can be used in place of `IndexValue`.

    Remedy: algebraic data types (`data`)
  - Index and value can be exchanged unnoticed.

    Remedy: names for components (labelled fields)
- Useful as shorthand without special meaning:

  `type DoubleV4 = (Double,Double,Double,Double)`

Algebraic Data Types (`data`)

construction of a new type, different from all existing ones

Ex.: predefined type `Bool`

```
data Bool           type constructor
   = False          first data constructor
   | True           sedond data constructor
   deriving         automatic deduction of type class instances
   (Eq, Ord, Enum, Read, Show, Bounded)
```

• `Eq`	`False==True` ⇝ `False`
• `Ord`	`False<True` ⇝ `True`
• `Enum`	`fromEnum True` ⇝ `1`
• `Read`	`read "False"` ⇝ `False`
• `Show`	`show True` ⇝ `"True"`
• `Bounded`	`maxBound::Bool` ⇝ `True`

data: Data Constructors with Type Arguments

```
data Temperature  name of new type
   = Celsius    Double    data constructor Celsius, one argument
   | Kelvin     Double    data constructor Kelvin,  one argument
   | Fahrenheit Double    data constructor Fahrenheit, one argument
   deriving Show   deduction of function show

normalize :: Temperature -> Temperature
normalize (Celsius    cel) = Kelvin (cel+273.16)
normalize (Kelvin     kel) = Kelvin kel
normalize (Fahrenheit fah) = normalize (Celsius ((fah-32)*5/9))
```

Function `normalize` can be applied only to elements of type `Temperature`; values must carry the tag (`Celsius`/`Kelvin`/`Fahrenheit`)
$\Longrightarrow$ the unit corresponding to the value is directly transparent.

data: Type Constructors with Type Parameters

Ex.: predefined “lifting” type constructor Maybe

```
data Maybe a = Nothing | Just a
               deriving (Eq,Ord,Read,Show)
```

```
[("Bill",Nothing),("Peter",Just 2.0),
 ("Tom",Just 1.2)]  :: [(String, Maybe Double)]
```

A data constructor is a kind of function
(its call is the normal form, it cannot be computed):

```
Nothing :: Maybe a
Just    :: a -> Maybe a
```

data: Use of a Context

Problem: directory as a list of key-value pairs.

Requirements:

- keys must be comparable ⇒ type class Eq
- keys and values should be printable ⇒ type class Show

```
data Dictionary a b = Dict [(a,b)]


lookupDict :: (Eq a, Show a, Show b)           context
           => a -> Dictionary a b -> String
lookupDict key (Dict xs)
 = case lookup key xs of
    Nothing -> "key " ++ show key ++ " not found"
    Just x  -> show x
```

In Haskell predefined: `lookup :: (Eq a) => a -> [(a, b)] -> Maybe b`

data: Labelled Fields

- Definition of a data type via its components:

```
data Person = Customer { name::String, address::String }
            | Employee { name::String, salary::Float   }
```

- Generation of a new element of the data type:

```
employee = Employee { salary=5000.00, name="Schmidt"}
```

- Retrieval of a component's value: `salary employee` ⇝`5000.00`

- Pattern match with component assignment to local variables `n` and `s`

```
printPerson (Employee {salary=s, name=n})
 = "Person: " ++ n ++ ", Pay: " ++ show s
```

- Copy with modification of an individual component (`@`: as-pattern)

```
changeAddress :: Person -> String -> Person
changeAddress person@(Customer {}) newaddress
 = person {address=newaddress}
```

Declaration of new Type Classes

```
class [ Context =>] Classname where
   Type declaration of the functions
   Default definitions
```

Ex.: predefined type class `Eq`

```
class Eq a where                      type a belongs to class Eq  <=>
     (==), (/=) :: a -> a -> Bool        equality is defined on a

     x /= y   = not (x==y)            default definitions
     x == y   = not (x/=y)
```

Declaration of new Instances

```
instance [ Context  => ] Classname Instancetype where
   Implementation of functions
```

Ex.: let `data Color = Red | Yellow | Green`

```
instance Eq Color where         let Color be an instance of Eq
 Red    == Red     = True
 Yellow == Yellow  = True
 Green  == Green   = True
 _      == _       = False
```

In simple cases automatically with deriving:

```
data Color = Red | Yellow | Green deriving Eq
```

Predefined Class Num

```
class  (Eq a, Show a)                     Context (superclasses)
                      => Num a  where     new class for type a

    (+), (-), (*)    :: a -> a -> a       signatures
    negate           :: a -> a
    abs, signum      :: a -> a
    fromInteger      :: Integer -> a

        -- Minimal complete definition:
        --      All, except negate or (-)
    x - y            =  x + negate y      default definitions
    negate x         =  0 - x
```

Complex as Instance of Num (Module Complex)

Syntax of a complex number: <real>:+<imaginary>

```
data Complex a  =  a :+ a

instance  (RealFloat a)              a can be Float or Double
 => Num (Complex a)  where           (class_name, element_type)
    (x:+y) + (x':+y') =  (x+x') :+ (y+y')
    (x:+y) * (x':+y') =  (x*x'-y*y') :+ (x*y'+y*x')
    negate (x:+y) =  negate x :+ negate y
    abs z =  magnitude z :+ 0
    signum 0 =  0
    signum z@(x:+y) =  x/r :+ y/r  where r = magnitude z
    fromInteger n =  fromInteger n :+ 0
```

Ratio as Instance of Num (Module Ratio)

Syntax of a rational number: <numerator>%<denominator>[a]
(in contrast to constructor :+ of Complex a,
module Ratio does not export infix constructor :%)

```
instance  (Integral a)                   context, a must be a whole number
 => Num (Ratio a)  where                 (class_name, element_type)
    (x:%y) + (x':%y')    =  reduce (x*y' + x'*y) (y*y')
    (x:%y) * (x':%y')    =  reduce (x * x') (y * y')
    negate (x:%y)        =  negate x :% y
    abs (x:%y)           =  abs x :% y
    signum (x:%y)        =  signum x :% 1
    fromInteger x        =  fromInteger x :% 1
```

[a] Cond.: % generates only positive denominators

Recursive Instance Declarations

Ex.: equality on lists

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

Ex.: equality on pairs

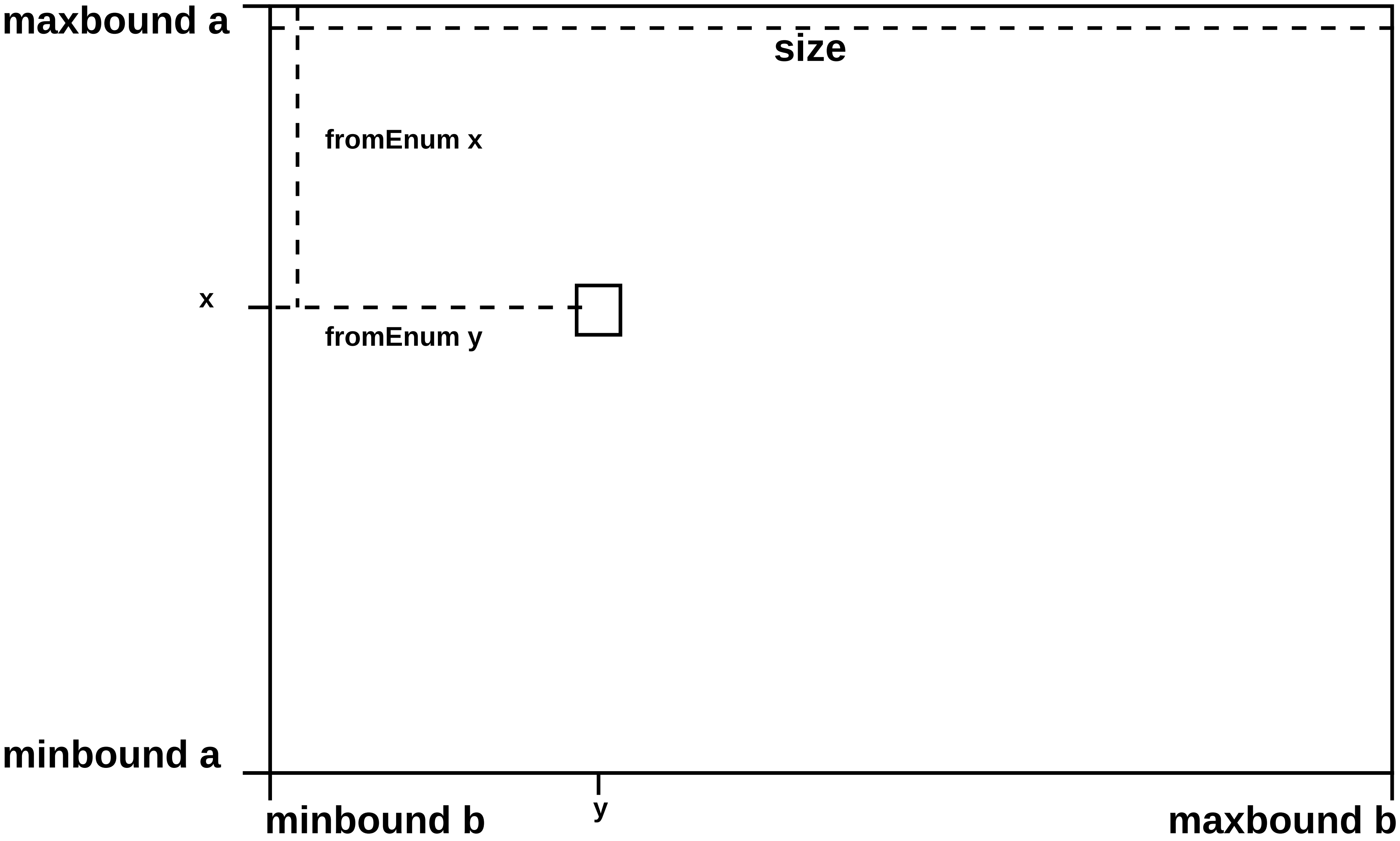
```
instance (Eq a, Eq b) => Eq (a,b) where
    (x,y) == (x',y')  =  x==x' && y==y'
```

Arithmetic Sequences of Boolean Tuples

```
instance (Bounded a, Bounded b, Enum a, Enum b)
      => Enum ((,) a b) where
 fromEnum (x,y) = fromEnum x * (1+fromEnum (maxBound::b)
                                -fromEnum (minBound::b))
                + fromEnum y
 toEnum i = let size = 1 + fromEnum (maxBound::b)
                         - fromEnum (minBound::b)
              in (toEnum (i‘div‘size), toEnum (i‘mod‘size))
```

Test: `[(False,(False,False))..(True,(True,True))]` ⇝

```
[(False,(False,False)),(False,(False,True)),
 (False,(True,False)),(False,(True,True)),
 (True,(False,False)),(True,(False,True)),
 (True,(True,False)),(True,(True,True))]
```


maxbound a
size
fromEnum x
x
fromEnum y
minbound a
minbound b
y
maxbound b