Calculational

The calculational package provides a syntax, similar to the one used in calculational mathematics (Dijkstra, Backhouse, Gries), for defining quantifiers, lists, sets, and bags. A novelty in Calculational is that such objects can be executed!

Usage with ghci

For an interactive session use ghci with QuasiQuotes flag:

```
$ ghci -XQuasiQuotes
```

The `[calc| ... |]` syntax is used for the Disjktra-like notation:

```
ghci> :m + Calculational
ghci> [calc| (+ x <- [-100 .. 100] | 0 <= x < 10 : x*x) |]
285
```

Usage in a program

```
{-# LANGUAGE QuasiQuotes #-}
...
import Calculational
...
sum' :: Num a -> [a] -> a
sum' xs = [calc| (+ x <- xs | : x) |]

product' :: Num a -> [a] -> a
product' xs = [calc| (* x <- xs | : x) |]
```

Boolean Expressions

The following operators are defined:

Operator	Description
`~` ¬	Negation
`/\` ∧	Conjunction
`\/` ∨	Disjunction
`==>` ⇒	Implication
`<==` ⇐	Consequence
`===` ≡	Equivalence (Double implication)
`=/=` ≢	Exclusive or (XOR)

Both ∧ and ∨ have the same precedence as in Gries; therefore, they cannot be mixed without parenthesis.

The unary # operator converts a boolean to a number; `#True = 1` and `#False = 0`.

Quantifier

A general quantifier notation has the form:

```
(* x <- S | R : E)
```

Where `x` is a dummy (i.e., bound variable), `S` is a collection of elements (`x <- S` is a generator), `R` is the range, and `E` is the body expression. In the executable interpretation, the variable `x` takes values over each element of `S`, and `E` is evalutaed using the `*` operator for those values of `x` satisfying `R`. The operator `*` and its type must be a monoid.

For example, consider the operational interpretation of the following quantifier:

```
ghci> [calc| (+x <- [-100 .. 100] : 0 <= x < 10 : x^2) |]
285
```

In this example, variable `x` is assigned each value in the set `[-100 .. 100]`. The evaluation corresponds to summing up the square `x̂ 2` for those values of `x` satisfying `0 <= x < 10`.

The expression in this example is equivalent to the following Haskell expression:

```
ghci> sum [ x^2 | x <- [-100 .. 100], 0 <= x && x < 10]
285
```

Consider another example. The following definition:

```
ghci> let sorted xs = [calc| (/\ (x,y) <- zip xs (tail xs) : : x <= y) |]
```

tests if a list is sorted in ascending order.

```
ghci> sorted ['a' .. 'z']
True
```

As explained above, operator `/\` represents the and operator. If the range `R` is empty, like in the last example, it is equivalent to `True`.

Alternatively, in a quantifier, the and operator `/\` can be interchanged with `forall` and the or binary operator `\/` with `exists`:

```
ghci> let s = [-100 .. 100]
ghci> [calc| (exists x <- s : 0 <= x <= 10 : even x) |]
True
ghci> [calc| (forall x <- s : 0 <= x <= 10 /\ x `mod` 2 == 0 :  even x) |]
True
```

The following is a table with the operators that could be used in quantifiers:

Operator	Alternative	Description
+ ∑	sum	Sumation
* ∏	product	Product
#		Apply the # operator to the body and sums it
/\ ∧ ∀	forall	Forall quantifier
\/ ∨ ∃	exists	Exists quantifier
=== ≡		Generalized equivalence
=/= ≢		Generalized inequivalence
++	concat	Generalized concatenation
⋃	union	Generalized union
⋂	intersection	Generalized intersection
↑	max	Maximum
↓	min	Minimum

Multiple quantifier variables

In a quantifier, there can be more than one generator and they are separated by commas.

```
ghci> [calc| (+ x<-[0 .. 5],y<-[0 .. 5] : 0 <= x < y <= 3 : x+y) |]
18
```

In this example for each `x` value between `0` and `5`, `y` takes the values from `0` to `5`.

The bounded variables can be used on later expressions, including other generators:

```
ghci> [calc| (+ x<-[0 .. 2],y<-[x+1 .. 3] : : x+y) |]
18
```

Also, tuples and some patterns for the left generator part can be used:

```
ghci> [calc| (+ (x,y) <- zip [0 .. 9] [1 .. 10] : : x+y) |]
100
```

Alternative bar syntax

While Dijkstra uses : between dummy variables and the range, Gries uses the bar | symbol. The Calculational package supports both notations:

```
ghci> let s = [-100 .. 100]
ghci> [calc| (+ x<-s,y<-s : 0 <= x < y <= 10 /\ even x /\ even y: x+y) |]
150
ghci> [calc| (+ x<-s,y<-s | 0 <= x < y <= 10 /\ even x /\ even y: x+y) |]
150
```

Maximum and minimum

The maximum and minimun operators have the same precedence:

```
ghci> [calc| 5 ↓ (3 ↑ 1) |]
3
```

In order to use the maximun and minimun in quantifiers, it is necesary to have identities. In this purpose, constants `minbound` and `maxbound` of the Bounded class are used: the corresponding identity is returned in the case of a quantifier with empty range.

```
ghci> [calc| (max x <- [-10 .. 10] | False : x^2 ) |] :: Int
-9223372036854775808
ghci> [calc| (max x <- [-10 .. 10] | : x^2 ) |] :: Int
100
```

The data type `Infty` is used to extend an ordered unbounded data type with a lower (NegInfty) and upper (PosInfty) bounds, and they are the identities of the max and min operators.

```
data Infty a = NegInfty
             | Value { getValue :: a }
             | PosInfty
             deriving (Eq, Ord, Read, Show)
```

```
ghci> [calc| (max x <- [-10 .. 10] | False : Value(x^2) ) |]
NegInfty

ghci> [calc| (min x <- [-10 .. 10] | False : Value(x^2) ) |]
PosInfty

ghci> [calc| (max x <- [-10 .. 10] | : Value(x^2) ) |]
```

```
Value {getValue = 100}

ghci> [calc| (min x <- [-10 .. 10] | : Value(x^2) ) |]
Value {getValue = 0}
```

Sets

A set is defined between braces, listing its elements separated by commas:

```
ghci> [calc| { 1,2,3 } |]
fromList [1,2,3]
```

Alternatively, the `..` syntax could be used:

```
ghci> [calc| { 1 .. 3 } |]
fromList [1,2,3]
```

The spaces around the `..` are necessary in some cases to avoid the parser fails.

Set comprehensions

With a syntax similar to that of quantifiers, a set comprehension has the same parts but enclosed with braces and without operator:

```
{ x <- S | R : E }
```

Some set comprehension examples:

```
ghci> [calc| { x <- [-10 .. 10] | 0 <= x < 4 : x^2 } |]
fromList [0,1,4,9]

ghci> let s = [-100 .. 100]
ghci> [calc| { x <- s | -4 <= x <= 4 : x^2 } |]
fromList [0,1,4,9,16]

ghci> [calc| { x <- s,y <- s | 1 <= x <= y <= 3 : (x,y) } |]
fromList [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]

ghci> [calc| { x <- s,y <- s | 1 <= x <= y <= 3 /\ even x : x*y } |]
fromList [4,6]
```

Optional body

If there is only one generator, and the body is ommited, it is equivalent to the left part of the generator.

```
ghci> [calc| { x <- [0 .. 10] : x^2 < 6 } |]
fromList [0,1,2]
```

Membership

The `member` function is used to test if an element is a member of a set.

```
ghci> [calc| 0 ∊ { x <- [0 .. 10] : x^2 < 6 } |]
True
ghci> [calc| 0 `member` { x <- [0 .. 10] : x^2 < 6 } |]
True
```

Number of elements

The number of elements of a finite set is obtained by using the unary # operator:

```
ghci> [calc| # {1,2,3,4} |]
4
```

Operator # is overloaded.

Subset, Superset

The subset and superset definitions:

```
ghci> [calc| {1,2} ⊆ {1,2,3,4} |]
True
ghci> [calc| {1,2} ⊂ {1,2,3,4} |]
True
ghci> [calc| {1..4} ⊂ {1,2,3,4} |]
False
ghci> [calc| {1..4} ⊇ {1,2} |]
True
ghci> [calc| {1..4} ⊃ {1,2,3,4} |]
False
```

Set difference

The notation for set difference is \

```
ghci> let s = [-100 .. 100]
ghci> [calc| { x <- s | -4 <= x <= 4 : x^2 } \ { x <- s | 0 <= x <= 10 } |]
fromList [16]
```

Set Union and Intersection

The union and intersection functions are defined for sets:

```
ghci> let s = [-100 .. 100]
ghci> [calc| { x <- s | -4 <= x <= 4 : x^2 } ∪ { x <- s | 0 <= x <= 10 } |]
fromList [0,1,2,3,4,5,6,7,8,9,10,16]
ghci> [calc| { x <- s | -4 <= x <= 4 : x^2 } `union` { x <- s | 0 <= x <= 10 } |
fromList [0,1,2,3,4,5,6,7,8,9,10,16]
ghci> [calc| { 1 .. 4 } ∩ { 3 .. 5 } |]
fromList [3,4]
ghci> [calc| { 1 .. 4 } `intersection` { 3 .. 5 } |]
fromList [3,4]
```

Generalized union and intersection

The set union and intersection can be used as quantifier operators.

The generalized union is the union multiple sets:

```
ghci> [calc| (∪ s <- {{1},{2,3},{4}} : : s) |]
fromList [1,2,3,4]
ghci> [calc| (union s <- {{1},{2,3},{4}} : : s) |]
fromList [1,2,3,4]

ghci> [calc| (union s <- {{1},{2,3},{4}} : False : s) |]
fromList []

ghci> [calc| (union s <- {{1},{2,3},{4}} : : Data.Set.map (\x -> x^2) s) |]
fromList [1,4,9,16]
```

A wrapper data type `Universe` gives an upper bound to ordered monoids which lower bound is mempty.

```
data Universe m a = Container { getContainer :: m a }
                  | Universe
                  deriving (Eq, Ord, Read, Show)
```

```
ghci> [calc| (intersection s <- {{1,2},{2,3},{2}} : False : Container s) |]
Universe
```

```
ghci> [calc| (intersection s <- {{1,2},{2,3},{2}} : : Container s) |]
Container {getContainer = fromList [2]}
```

Lists

List comprehension is defined similar to set comprehensions:

```
ghci> [calc| [ x <- [0 .. 10] : 1 <= x <= 3 /\ even x : x^2 ] |]
[4]
```

Preppend and append

The prepend operator <| prepends an element to a list.

```
ghci> [calc| 1 <| [ x <- [0 .. 5] : 2 <= x <= 3 : x^2 ] |]
[1,4,9]
ghci> [calc| 1 ◂ [ x <- [0 .. 5] : 2 <= x <= 3 : x^2 ] |]
[1,4,9]
```

The append operator |> appends an element to a list.

```
ghci> [calc| [ x <- [0 .. 5] : 1 <= x <= 2 : x^2 ]  |> 9 |]
[1,4,9]
ghci> [calc| [ x <- [0 .. 5] : 1 <= x <= 2 : x^2 ]  ▸ 9 |]
[1,4,9]
```

As is the case for sets, operators like member (∈), union (∪), intersection (∩), difference (\), sublist (⊆,⊂), superlist (⊇,⊃), number of elements of finite lists # are also defined for lists.

Bags or multisets

```
ghci>  [calc| {| x <- [-3 .. 3] | : abs x |} |]
fromOccurList [(0,1),(1,2),(2,2),(3,2)]
```

Occurs

The number of ocurrences of an element in a bag could be obtained using the binary `#` operator:

```
ghci> [calc| 2 # {| 1,1,2,2,2 |} |]
3
```

As is the case for sets and lists, operators member (∈), union (∪), intersection (∩), subbag (⊆,⊂), superbag (⊇,⊃), difference \ number of elements of finite bag # are defined for lists.