GReQL-Reference Card

Daniel Bildhauer, Tassilo Horn and Eckhard Großmann

April 2, 2014

1 Data types

Name	Signature sign	Description
Boolean	*BOOL*	Holds a boolean value.
Integer	*INT*	Holds a 32 bit signed integer value.
Long	*LONG*	Holds a 64 bit signed integer value.
Double	*DOUBLE*	Holds a 64 bit floating point value.
Object	*OBJECT*	Is the super type for all types.
String	*STRING*	Holds a string value.
Enum	*ENUM*	Holds an enum constant.
Collection	*COLLECTION<E>*	Represents the abstract super type of *List* and *Set*.
List	*LIST<E>*	Represents an ordered list of elements of type *E*.
Set	*SET<E>*	Represents an ordered set of values of type *E*.
Bag	*BAG<E>*	Represents a bag of values of type *E*. Multiple occurrences are counted.
Map	*MAP<Key,Value>*	Represents a map from domain *Key* to range *Value*.
Table	*TABLE*	Represents a table with named columns. Every element in one column belongs to the same type.
Tuple	*TUPLE*	Represents a tuple, where every element can be of a different type.
Record	*RECORD*	Represents a tuple with named elements. It is similar to a struct in C.
AttributedElement	*ATTRELEM*	Abstract super type of *Graph*, *Vertex* and *Edge*.
Vertex	*VERTEX*	Represents a vertex (node) in a *Graph*.
Edge	*EDGE*	Represents an edge between two *Vertex* objects in a *Graph*.
Graph	*GRAPH*	Represents a graph.
SubGraph	*SUBGRAPH*	Represents a part of a *Graph*.
Path	*PATH*	Describes a path through a graph as a list of vertices and their connecting edges. $v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} v_3$
PathSystem	*PATHSYSTEM*	Represents a tree-like set of paths with a common start vertex, which is the root of the pathSystem. For every leaf vertex, there is exactly one path.
Slice	*SLICE*	Similar to pathSystem, but there may be more than one path to a vertex stored in a slice. Only one edge is considered if multiple edges of the same type connect the same two vertices.
AttributedElementClass	*ATTRELEMCLASS*	Represents a type of the schema.
TypeCollection	*TYPECOLLECTION*	Stands for a TypeDescription, see TypeDescription in section 2.3.

2 Literals & Expressions

2.1 Literals

Name	Description	Example
BooleanLiteral	Boolean value with two values: `true` and `false`.	`true` `false`
IntegerLiteral, **LongLiteral**	A signed integer value. Can be written as octal, decimal or hexadecimal value. The type of the literal is adjusted to best fit the value (Integer, Long, Double). Numeric literals must start with a digit or a hyphen followed by a digit (negative values)	`0, -23` (Decimal notation) `-051, 022` (Octal notation) `0x2f, 0x65` (Hexadecimal notation)
DoubleLiteral	A 64 bit floating point decimal value in scientific notation. Numeric literals must start with a digit or a hyphen followed by a digit (negative values).	`0.0, -2.3` (Decimal notation) `23e-7` (Exponent notation)
StringLiteral	A character sequence enclosed in single or double quotes. Quotes inside a string literal must be escaped with a backslash.	`"A string"` `"This is a double quote: \"."`

2.2 Regular path expressions

For a better understanding an example schema and an instance graph is illustrated in figure 1.

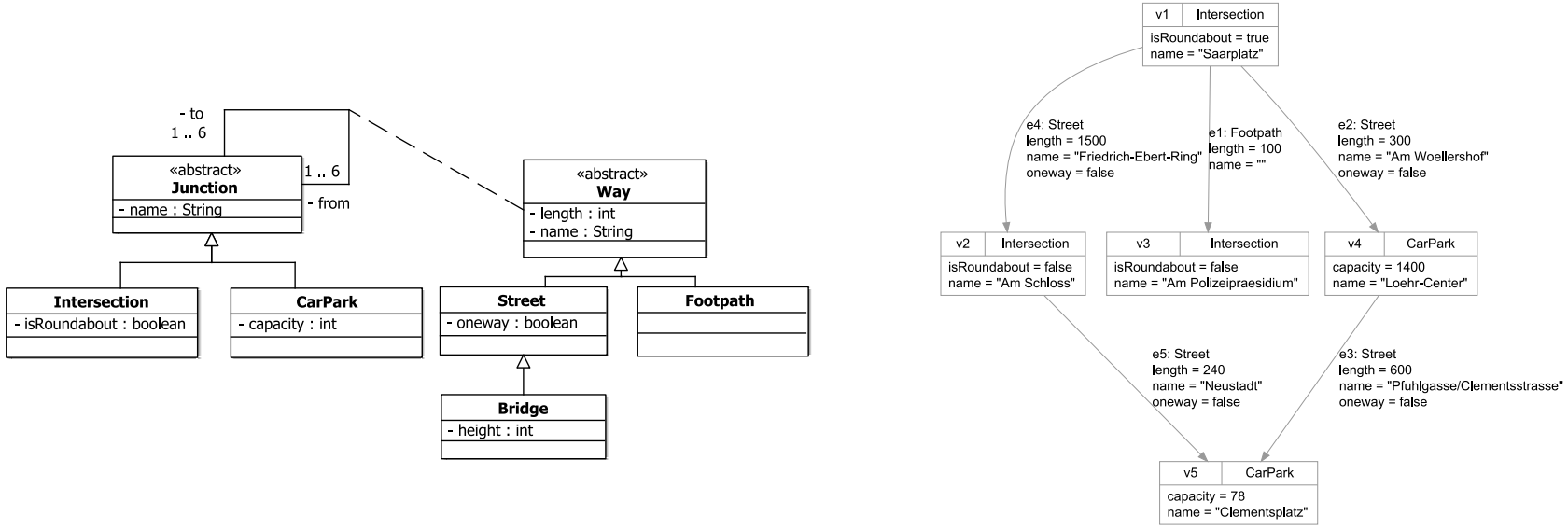

Figure 1: *left*: A schema for a graph. *right*: An instance graph for the given schema.

In the following *exp* is an expression. p, p_1 and p_2 are path descriptions.

Name	Description	Example

Name	Description	Example

EdgeRestriction	Describes a comma separated set of edge types and roles and an optional predicate prefixed with `@`, in which `thisEdge` denotes the edge itself. The restriction matches all edges which have one of the types or roles and for which the predicate holds.
	The operators `!` and `^` are also valid. See TypeDescription in section 2.3.
		`{Street}  // e2, e3, e4 and e5 are selected`
		`{Way, to @ thisEdge.length <= 300}`
		`  // e1, e2 and e5`

SimplePathDescription	A simple path description *p* consists of an edge symbol `-->` (outgoing) , `<--` (incoming), `<->` (direction doesn't matter), `<>--` (parent to child) or `--<>` (child to parent) and optionally an EdgeRestriction in curly braces.	`-->{Street @ thisEdge.name == ""}`
		`  // e1`
		`<--`
		`--<>{@ true}`

EdgePathDescription	A edge path description matches exactly one edge, given as expression `exp`. The form is `--exp->`, `<-exp->` or `<-exp--`.	`-- getEdge(1) ->`
SequentialPathDescription	Sequential use of path descriptions is supported: `p1 p2`.	`--> --<>`
OptionalPathDescription	A path description can be marked as optional by surrounding it with brackets: `[p]`	`[-->]`

IteratedPathDescription	Iteration of path description with the use of Kleene operators * and + where `p*` means, `p` is executed 0 or many times and `p+` means, `p` is executed at least once. Similar, `p^n` denotes a fixed number `n` of iterations.	`-->+`
		`-->*`
		`-->^2`

AlternativePathDescription	Marking paths as alternative is possible by separating them with a pipe: `p1 | p2`.	`--> | --<>`
GroupPathDescription	To group multiple path descriptions, simply surround them with two braces: `(p)`	`--> (--> | --<>)`

StartVertexRestriction or **GoalVertexRestriction**	The start and end vertices of a path description can be restricted. Therefore, the restriction is separated from the path description with a `&`. Its syntax is similar to an EdgeRestriction, but only Vertex types and no role names are allowed. `thisEdge` is replaced by `thisVertex`.
	`{VertexType} & p //` Start vertex is restricted
	`p & {VertexType} //` Goal vertex is restricted
		`{CarPark} & -->`
		`--> & {CarPark`
		`  @ thisVertex.capacity > 100}`

2.3 Expressions

The root node of any GReQL query is a `GReQLExpression`. Every example is given in one row. Continuing examples are indented.

Name	Description	Example
GReQLExpression	Every GReQL query is a GReQLExpression and contains one arbitrary other expression as child to be evaluated. Storing and reuse of query results is possible by the use of `using` (prefix) and `store as` (suffix) clause.	`list(1..10) store as myList` `using myList :  isUnique(myList)`

ValueConstruction	Collections, tuples, maps and records can be constructed using a value construction which specifies the data type and its elements. Types can be `list` (*List*), `set` (*Set*), `tup` (*Tuple*), `map` (*Map*) and `rec` (*Record*). The member elements are denoted by a comma separated list of expressions. For Set, List and Map, the expression's results must be of the same type. In case of a *Map* each argument has an unique key assigned. In case of a *Record* each argument recives has a unique name assigned.
	Additionally, instead of arguments a range $[a, b]$ can be defined in a list construction: `list(a..b)`
		`list(1..10)`
		`list(1,2,3,4,5,6,7,8,9,10)`
		`set(1,1,1,2)`
		`tup("Hello","World", 42)`
		`map("a" -> 1, "b" -> 2)`
		`rec(name:  "Max", alter:  18)`

Variable	Variables declared in iteration expressions or denoting results of other expressions.	`x`
		`thisVertex`
		`thisEdge`

LetExpression	Definition of variables to be used in a query.	`let x := 10, y:= 12 in x+y //` Returns 22.

Name	Description	Example
WhereExpression	Definition of variables to be used in a query.	`x+y where x := 10, y:= 12 //` Returns 22.
AttributeAccess	Access to attributes of elements and records.	`street.length //` 'street' is of type *Street*
ElementAccess	Access to an element of a list or tuple.	`list(1..10)[5] //` Returns 6.
UnaryOperator	Application of an unary operator on an expression.	`- 2, - (3 + 4)` `not true`

BinaryOperator	Application of a binary operator on two expressions.	`1 + 2`
		`true <> false`
		`17 * 4, 23 >= 5`

FunctionApplication	Application of a GReQL function, optionally with some expressions as parameters and type restriction.	`degree{Street}(getVertex(1)) //` 2
		`isAcyclic() //` true
		`contains(list(1..9), 1) //` true

TypeDescription	Describes a set of valid types from the schema and can be used as a *TypeCollection* as input of a GReQL function. In general, the type description is a comma separated list of types. A type marked with `!` (suffix) means, only this type and no subtypes. A type marked with `^` (prefix) means, not this type or subtypes.	`{Street!} //` Only *Street* and not *Bridge* is selected. `{^Street} //` *Street* and *Bridge* are not selected.
EdgeSetExpression	Selects all edges from the current graph and returns them as a *Set*. Optionally, the selection can be restricted by a TypeDescription.	`E //` All edges. `E{Street!} //` Only edges of type *Street*.
VertexSetExpression	Selects all vertices from the current graph and returns them as a *Set*. Optionally, the selection can be restricted by a TypeDescription.	`V //` All vertices. `V{^Way} //` All vertices except Way and all subtypes.

FWRExpression	Allows an iteration over collections and reporting of entries as a *Table*, *Set*, *Map* or *Bag*, whose contained elements are defined by expressions. The first part of FWR is `from`, which is followed by a comma-separated list of declarations. A declaration consists of a variable name `v` and a Collection `C` and is written as `v:C`. The optional second part is with an `with` and an expression, which has to have a *Boolean* as a result. The last part begins with `report` and is followed by comma-separated expressions. Optionally, the column of the resulting table for an expression `e` can be named with a string `s: e as s`. The whole expression is completed by `end`. The result is a table. Additionally, the resulting type can be changed, by using `reportSet` (*Set*), `reportMap` (*Map*) or `reportBag` (*Bag*) instead of `report`.	`from n:list(1..6)`
		`  with n % 2 == 0`
		`  reportSet n`
		`end //` Returns even numbers: {2,4,6}.
		
		`from s:E{Street}`
		`  with s.length <= 0`
		`  report s as "illegalStreet",`
		`  s.length as "length"`
		`end //` Returns streets, which can't exist!

QuantifiedExpression	Checks, if all (`forall`), at least one (`exists`) or exactly one (`exists!`) element(s) of a collection fulfill a given expression. After one of the mentioned quantifiers, one or more variables are declared using a declaration (see FWRExpression). Separated by an `@`, the expression to be tested for is specified. The respective *Boolean* value is returned.	`forall v:V @ getId(v) > 0`
		`  //` Should always be true.
		`exist!  n:list(1..9) @ n % 2 == 0`
		`  //` Is always false.

3 Functions

A function can be called by writing the function name followed by its parameter list enclosed in braces. For example the function *and* as `and(a, b)`. Some functions can be used as infix or prefix operators. The function *and* for example can also be called with its infix notation: `a and b`. The function *not* is an example for a prefix operator and is called as `not true`.

Some functions have a *TYPECOLLECTION* in their signature. This means, they can be restricted by a type description, which muss be written between the function name and its parameter list in curly braces. Also see TypeDescription and FunctionApplication in section 2.3.

3.1 Arithmetics

abs. Calculates the absolute value of the given number.
$abs\colon\ Number\,a \longrightarrow Number$

add. Adds the given two numbers with the usual Java overflow semantics. Can be used as operator: $a+b$.
$add\colon\ Number\,a \times Number\,b \longrightarrow Number$

bitAnd. Calculates the bitwise AND of the given two numbers.
$bitAnd\colon\ Integer\,a \times Long\,b \longrightarrow Long$
$bitAnd\colon\ Long\,a \times Integer\,b \longrightarrow Long$
$bitAnd\colon\ Long\,a \times Long\,b \longrightarrow Long$
$bitAnd\colon\ Integer\,a \times Integer\,b \longrightarrow Integer$

bitNot. Calculates the bitwise negation of the given number.
$bitNot\colon\ Integer\,a \longrightarrow Integer$
$bitNot\colon\ Long\,a \longrightarrow Long$

bitOr. Calculates the bitwise OR of the given two numbers.
$bitOr\colon\ Integer\,a \times Long\,b \longrightarrow Long$
$bitOr\colon\ Long\,a \times Integer\,b \longrightarrow Long$
$bitOr\colon\ Long\,a \times Long\,b \longrightarrow Long$
$bitOr\colon\ Integer\,a \times Integer\,b \longrightarrow Integer$

bitShl. Shifts the first number by the second argument's number of bits to the left.
$bitShl\colon\ Integer\,a \times Integer\,b \longrightarrow Integer$
$bitShl\colon\ Long\,a \times Integer\,b \longrightarrow Long$

bitShr. Shifts the first number by the second argument's number of bits to the right.
$bitShr\colon\ Integer\,a \times Integer\,n \longrightarrow Integer$
$bitShr\colon\ Long\,a \times Integer\,n \longrightarrow Long$

bitUnsignedShr. Shifts the first number by the second argument's number of bits to the right (unsigned).
$bitUnsignedShr\colon\ Integer\,a \times Integer\,n \longrightarrow Integer$
$bitUnsignedShr\colon\ Long\,a \times Integer\,n \longrightarrow Long$

bitXor. Calculates the bitwise XOR of the given two numbers.
$bitXor\colon\ Integer\,a \times Long\,b \longrightarrow Long$
$bitXor\colon\ Long\,a \times Integer\,b \longrightarrow Long$
$bitXor\colon\ Long\,a \times Long\,b \longrightarrow Long$
$bitXor\colon\ Integer\,a \times Integer\,b \longrightarrow Integer$

ceil. Returns the ceiling of the given number.
$ceil\colon\ Number\,a \longrightarrow Number$

cos. Returns the cosinus of the given number.
$cos\colon\ Number\,a \longrightarrow Double$

div. Returns the quotient of dividing the first by the second number.
$div\colon\ Number\,a \times Number\,b \longrightarrow Number$

exp. Returns Euler's number e raised to the power of the given number.
$exp\colon\ Number\,a \longrightarrow Double$

floor. Returns the floor of the given number.
$floor\colon\ Number\,a \longrightarrow Number$

ln. Returns the natural logarithm of the given number.
$ln\colon\ Number\,a \longrightarrow Double$

mod. Calculates the remainder of the division a/b. Alternative usage: a % b.
$mod\colon\ Number\,a \times Number\,b \longrightarrow Number$

mul. Multiplies the given two numbers with the usual Java overflow semantics. Can be used as operator: a * b.
$mul\colon\ Number\,a \times Number\,b \longrightarrow Number$

neg. Negates the given number. Can be used as unary operator: -x.
$neg\colon\ Number\,a \longrightarrow Number$

round. Rounds the given number.
$round\colon\ Number\,a \longrightarrow Long$

sin. Returns the sinus of the given number.
$sin\colon\ Number\,a \longrightarrow Double$

sqrt. Returns the square root of the given number.
$sqrt\colon\ Number\,a \longrightarrow Double$

sub. Substracts the second number from the first number with the usual Java overflow semantics. Can be used as operator: a - b.
$sub\colon\ Number\,a \times Number\,b \longrightarrow Number$

tan. Returns the tangens of the given number.
$tan\colon\ Number\,a \longrightarrow Double$

toDouble. Converts a Number into a Double.
$toDouble\colon\ Number\,a \longrightarrow Double$

toInteger. Converts the given number into an Integer.
$toInteger\colon\ Number\,a \longrightarrow Integer$

toLong. Converts the given number into a Long.
$toLong\colon\ Number\,a \longrightarrow Long$

3.2 Collections and maps

concat.
$concat\colon\ Collection\,a \times Collection\,b \longrightarrow List$
Concatenates collections. Can be used as infix operator: a ++ b.

contains.
$contains: Collection\, c \times Object\, el \longrightarrow Boolean$
Returns true, iff c contains el.

containsKey. Returns true, iff the map contains the key.
$containsKey: Map\, map \times Object\, key \longrightarrow Boolean$

containsValue. Returns true, iff the given map contains value.
$containsValue: Map\, map \times Object\, value \longrightarrow Boolean$

count.
$count: Collection\, l \longrightarrow Integer$
Returns the number of items in the given collection.
$count: Map\, m \longrightarrow Integer$
Returns the number of items in the given map.

difference.
$difference: Set\, a \times Set\, b \longrightarrow Set$
Returns the set-difference a-b.
$difference: Map\, a \times Map\, b \longrightarrow Map$
Returns the map-difference a-b, w.r.t. the keyset of the maps.

entrySet. Returns the set of entries of the map.
$entrySet: Map\, map \longrightarrow Set$

get.
$get: Tuple\, t \times Integer\, i \longrightarrow Object$
Returns the i-th of tuple t. Short notation: t[i]
$get: List\, v \times Integer\, i \longrightarrow Object$
Returns the value stored in v at index i. Short notation: v[i]
$get: Table\, t \times Integer\, i \longrightarrow Object$
Returns the value stored in t at index i. Short notation: t[i]
$get: Set\, s \times Integer\, i \longrightarrow Object$
Returns the value stored in s at index i. Short notation: s[i].
$get: Map\, map \times Object\, key \longrightarrow Object$
Returns the map value associated with key. Short notation: map[key]

indexOf.
$indexOf: Object\, el \times Set\, s \longrightarrow Integer$
Returns the index of the first occurence of el in s, or -1 if el is not in s.
$indexOf: Object\, el \times List\, v \longrightarrow Integer$
Returns the index of the first occurence of el in v, or -1 if el is not in v.

intersection. Returns the intersection of a and b.
$intersection: Set\, a \times Set\, b \longrightarrow Set$

isSubSet. Returns true, iff the sub is subset of s.
$isSubSet: Set\, sub \times Set\, s \longrightarrow Boolean$

keySet. Returns the set of keys of the map.
$keySet: Map\, map \longrightarrow Set$

max.
$max: Collection\, l \longrightarrow Comparable$
Returns the maximum of a collection of comparable things.

min.
$min: Collection\, l \longrightarrow Comparable$
Returns the minimum of a collection of comparable things.

pos. Returns the position of the first occurence of the given element in the given collection, or -1, if the element is not contained in the collection.
$pos: List\, l \times Object\, x \longrightarrow Integer$
$pos: Set\, l \times Object\, x \longrightarrow Integer$

sort. Sorts the given collection according to natural ordering.
$sort: Tuple\, l \longrightarrow List$
$sort: Collection\, l \longrightarrow List$

sortByColumn.
$sortByColumn: Integer\, column \times Table\, t \longrightarrow Table$
Sorts a table of tuples by one column.
$sortByColumn: List\, columns \times Table\, t \longrightarrow Table$
Sorts a table of tuples by many columns.
$sortByColumn: Integer\, column \times Collection\, l \longrightarrow List$
Sorts a collection of tuples by one column.
$sortByColumn: List\, columns \times Collection\, l \longrightarrow List$
Sorts a collection of tuples by many columns.

subCollection.
$subCollection: Set\, coll \times Integer\, startIndex \longrightarrow Set$
Returns a sub PSet starting at the given start index (including).
$subCollection: Set\, coll \times Integer\, startIndex \times Integer\, endIndex \longrightarrow Set$
Returns a sub PSet starting at the given start index (including), and ending at the given end index (excluding).
$subCollection: List\, coll \times Integer\, startIndex \longrightarrow List$
Returns a sub PVector starting at the given start index (including).
$subCollection: List\, coll \times Integer\, startIndex \times Integer\, endIndex \longrightarrow List$
Returns a sub PVector starting at the given start index (including), and ending at the given end index (excluding).

theElement. Returns the only element in the given collection. If the collection is empty or contains more than one element, an exception is thrown.
$theElement: List\, c \longrightarrow Object$
$theElement: Set\, c \longrightarrow Object$

toList. Converts a collection into a list.
toList: $Tuple\, l \longrightarrow List$
toList: $Collection\, l \longrightarrow List$

toSet. Converts a collection into a set (removes duplicates).
toSet: $Tuple\, c \longrightarrow Set$
toSet: $Collection\, c \longrightarrow Set$

union.
union: $Set\, a \times Set\, b \longrightarrow Set$
Computes the union of the given two sets.
union: $Map\, a \times Map\, b \longrightarrow Map$
Computes the union of the given maps. In case of common keys in maps, the entries of the second one override the first one's entries.

values. Returns the collection of values of the given map.
values: $Map\, map \longrightarrow List$

3.3 Graph

alpha. Returns the start vertex of an edge.
alpha: $Edge\, e \longrightarrow Vertex$

alphaIncidenceIndex.
alphaIncidenceIndex: $Edge\, e \longrightarrow Integer$
Returns the index of e in the incidence sequence of its alpha vertex.
alphaIncidenceIndex: $Edge\, e \times Vertex\, v \longrightarrow Integer$
Returns the index of e in the incidence sequence of v. Returns -1 if e is not in v's incidence sequence.

degree.
degree: $Vertex\, v \times Path\, p \longrightarrow Integer$
Returns the degree of vertex v. The scope is limited by a path, a path system.
degree: $Vertex\, v \times TypeCollection\, c \longrightarrow Integer$
Returns the degree of vertex v. The scope is limited by a type collection.
degree: $Vertex\, v \longrightarrow Integer$
Returns the degree vertex v.

describe. Returns a human-readable description of the given element.
describe: $AttributedElement\, el \longrightarrow Map$

edgeSetSubgraph. Returns the subgraph induced by the edge set, i.e. the egdes in edgeSet together with their alpha and omega vertices.
edgeSetSubgraph: $Graph\, graph \times Collection\, edgeSet \longrightarrow SubGraphMarker$

edgeTypeSubgraph. Returns the subgraph induced by the edge types in typeCollection, i.e. all edges specified by typeCollection together with their alpha and omega vertices.
edgeTypeSubgraph: $Graph\, graph \times TypeCollection\, typeCollection \longrightarrow SubGraphMarker$

edges.
edges: $PathSystem\, p \longrightarrow Set$
Returns the set of edges in the given path system.

edgesConnected.
edgesConnected: $Vertex\, v \longrightarrow List$
(deprecated, use incidences) Returns the list of edges of the given vertex.
edgesConnected: $Vertex\, v \times TypeCollection\, tc \longrightarrow List$
(deprecated, use incidences) Returns the list of edges of the given vertex restricted by a type collection.

edgesFrom.
edgesFrom: $Vertex\, v \longrightarrow List$
(deprecated, use outIncidences) Returns the list of outgoing edges of the given vertex.
edgesFrom: $Vertex\, v \times TypeCollection\, tc \longrightarrow List$
(deprecated, use outIncidences) Returns the list of outgoing edges of the given vertex restricted by a type collection.

edgesTo.
edgesTo: $Vertex\, v \longrightarrow List$
(deprecated, use inIncidences) Returns the list of incoming edges of the given vertex.
edgesTo: $Vertex\, v \times TypeCollection\, tc \longrightarrow List$
(deprecated, use inIncidences) Returns the list of incoming edges of the given vertex restricted by a type collection.

elementSetSubgraph. Returns the subgraph consisting of all vertices in vset and all edges in eset that connect vertices in vset.
elementSetSubgraph: $Graph\, g \times Collection\, vset \times Collection\, eset \longrightarrow SubGraphMarker$

endVertex.
endVertex: $Edge\, e \longrightarrow Vertex$
Returns the end vertex of the given edge.

extractPaths.
extractPaths: $PathSystem\, p \longrightarrow Set$
Returns the set of Paths in the PathSystem p.

first.
first: $Vertex\, v \longrightarrow Edge$
Returns the first incident edge of vertex v.
first: $Vertex\, v \times TypeCollection\, c \longrightarrow Edge$
Returns the first incident edge of vertex v. The scope is limited by a type collection.

firstEdge.
firstEdge: $Graph\, g \longrightarrow Edge$
Returns the first edge of the graph g.
firstEdge: $Graph\, g \times TypeCollection\, c \longrightarrow Edge$
Returns the first edge of the graph g. The scope is limited by a type collection.

firstIn.
firstIn: $Vertex\, v \longrightarrow Edge$
Returns the first incoming edge of vertex v.
firstIn: $Vertex\, v \times TypeCollection\, c \longrightarrow Edge$
Returns the first incoming edge of vertex v. The scope is limited by a type collection.

firstOut.
firstOut: $Vertex\, v \longrightarrow Edge$
Returns the first outgoing edge of vertex v.
firstOut: $Vertex\, v \times TypeCollection\, c \longrightarrow Edge$
Returns the first outgoing edge of vertex v. The scope is limited by a type collection.

firstVertex.
firstVertex: $Graph\, g \longrightarrow Vertex$
Returns the first vertex of the graph g.
firstVertex: $Graph\, g \times TypeCollection\, c \longrightarrow Vertex$
Returns the first vertex of the graph g. The scope is limited by a type collection.

getEdge. Returns the edge with the given id.
getEdge: $Graph\, graph \times Integer\, id \longrightarrow Edge$

getValue.
getValue: $AttributedElement\, el \times String\, name \longrightarrow Object$
Returns the value of the given element's attribute specified by its name. Can be used using the dot-operator: myElement.attrName.
getValue: $Record\, rec \times String\, name \longrightarrow Object$
Returns the value of the given record's component specified by its name. Can be used using the dot-operator: myRecord.compName.

getVertex. Returns the vertex with the given id.
getVertex: $Graph\, graph \times Integer\, id \longrightarrow Vertex$

id. Returns the id of the given graph element.
id: $GraphElement\, el \longrightarrow Integer$

inDegree.
inDegree: $Vertex\, v \times Path\, p \longrightarrow Integer$
Returns the in-degree of the given vertex. The scope is limited by a path, a path system.
inDegree: $Vertex\, v \times TypeCollection\, c \longrightarrow Integer$
Returns the in-degree of the given vertex. The scope is limited by a type collection.
inDegree: $Vertex\, v \longrightarrow Integer$
Returns the in-degree of the given vertex.

inIncidences.
inIncidences: $Vertex\, v \longrightarrow List$
Returns the incoming edges of vertex v.
inIncidences: $Vertex\, v \times TypeCollection\, c \longrightarrow List$
Returns the incoming edges of vertex v. The scope is limited by a type collection.

incidenceIndex.
incidenceIndex: $Edge\, e \times Vertex\, v \longrightarrow Integer$
Returns the index of e in the incidence sequence of v. Returns -1 if e is not in v's incidence sequence.

incidences.
incidences: $Vertex\, v \longrightarrow List$
Returns the incident edges of vertex v.
incidences: $Vertex\, v \times TypeCollection\, c \longrightarrow List$
Returns the incident edges of vertex v. The scope is limited by a type collection.

inverseEdge.
inverseEdge: $Edge\, e \longrightarrow Edge$
Returns the inverse-oriented edge of the given edge e. I.e., if e is a normal (forward-oriented) edge, returns the reversed (backward-oriented) edge and vice versa.

isAcyclic. Returns true, iff the graph is acyclic.
isAcyclic: $Graph\, g \longrightarrow Boolean$

isLoop. Returns true, iff the given edge is a loop, i.e. it starts and ends at the same vertex.
isLoop: $Edge\, e \longrightarrow Boolean$

isReachable. Returns true, iff there is a path from vertex given as first argument to vertex given as second argument that matches the path description given as second argument. Usually invoked like so: myVertex (–> | <>–)+ myOtherVertex.
isReachable: $Vertex\, u \times Vertex\, v \times DFA\, dfa \longrightarrow Boolean$

last.
last: $Vertex\, v \longrightarrow Edge$
Returns the last incident edge of vertex v.
last: $Vertex\, v \times TypeCollection\, c \longrightarrow Edge$
Returns the last incident edge of vertex v. The scope is limited by a type collection.

lastIn.
lastIn: $Vertex\, v \longrightarrow Edge$
Returns the last incoming edge of vertex v.
lastIn: $Vertex\, v \times TypeCollection\, c \longrightarrow Edge$
Returns the last incoming edge of vertex v. The scope is limited by a type collection.

lastOut.
lastOut: $Vertex\, v \longrightarrow Edge$
Returns the last outgoing edge of vertex v.
lastOut: $Vertex\, v \times TypeCollection\, c \longrightarrow Edge$
Returns the last outgoing edge of vertex v. The scope is limited by a type collection.

leaves.
leaves: $PathSystem\, p \longrightarrow Set$
Returns the set of leaf vertices in the given path system.

next.
next: $Edge\, e \longrightarrow Edge$
Returns the next edge following e in incidence order.
next: $Edge\, e \times TypeCollection\, c \longrightarrow Edge$
Returns the next edge following e in incidence order. The scope is limited by a type collection.

nextGraphElement.
nextGraphElement: $Edge\, e \times TypeCollection\, tc \longrightarrow Edge$
Returns the next edge for a given element, restricted by a type collection.
nextGraphElement: $Vertex\, v \longrightarrow Vertex$
Returns the next vertex for a given element.
nextGraphElement: $Vertex\, v \times TypeCollection\, tc \longrightarrow Vertex$
Returns the next vertex for a given element, restricted by a type collection.
nextGraphElement: $Edge\, e \times Boolean\, global \times TypeCollection\, tc \longrightarrow Edge$
Returns the next edge for a given element, restricted by a type collection. The boolean parameter *global* decides if successor is taken from the global edge sequence (true), or from the incidence sequence (false).
nextGraphElement: $Edge\, e \times Boolean\, global \longrightarrow Edge$
Returns the next edge for a given element. The boolean parameter *global* decides if successor is taken from the global edge sequence (true), or from the incidence sequence (false).
nextGraphElement: $Edge\, e \longrightarrow Edge$
Returns the next edge for a given element from the incidence sequence.

nextIn.
nextIn: $Edge\, e \longrightarrow Edge$
Returns the next incoming edge following e in incidence order.
nextIn: $Edge\, e \times TypeCollection\, c \longrightarrow Edge$
Returns the next incoming edge following e in incidence order. The scope is limited by a type collection.

nextOut.
nextOut: $Edge\, e \longrightarrow Edge$
Returns the next outgoing edge following e in incidence order.
nextOut: $Edge\, e \times TypeCollection\, c \longrightarrow Edge$
Returns the next outgoing edge following e in incidence order. The scope is limited by a type collection.

normalEdge.
normalEdge: $Edge\, e \longrightarrow Edge$
Returns the forward-oriented edge of the given edge e. If e is already forward-oriented simply returns e.

omega. Returns the end vertex of an edge.
omega: $Edge\, e \longrightarrow Vertex$

omegaIncidenceIndex.
omegaIncidenceIndex: $Edge\, e \longrightarrow Integer$
Returns the index of e in the incidence sequence of its omega vertex.
omegaIncidenceIndex: $Edge\, e \times Vertex\, v \longrightarrow Integer$
Returns the index of e in the incidence sequence of v. Returns -1 if e is not in v's incidence sequence.

outDegree.
outDegree: $Vertex\, v \times Path\, p \longrightarrow Integer$
Returns the out-degree of the given vertex. The scope is limited by a path, a path system.
outDegree: $Vertex\, v \times TypeCollection\, c \longrightarrow Integer$
Returns the out-degree of the given vertex. The scope is limited by a type collection.
outDegree: $Vertex\, v \longrightarrow Integer$
Returns the out-degree of the given vertex.

outIncidences.
outIncidences: $Vertex\, v \longrightarrow List$
Returns the outgoing edges of vertex v.
outIncidences: $Vertex\, v \times TypeCollection\, c \longrightarrow List$
Returns the outgoing edges of vertex v. The scope is limited by a type collection.

path. Returns the shortest path between v1 and v2 matching the path description pd.
path: $Vertex\, v1 \times DFA\, pd \times Vertex\, v2 \longrightarrow Path$

pathLength. Returns the length of the given Path.
pathLength: $Path\, p \longrightarrow Integer$

reachableVertices. Returns all vertices that are reachable from the given vertex by a path matching the the given path description.
reachableVertices: $Vertex\, v \times DFA\, d\, fa \longrightarrow Set$

reversedEdge.
reversedEdge: $Edge\, e \longrightarrow Edge$
Returns the backward-oriented edge of the given edge e. If e is already backward-oriented simply returns e.

slice.
slice: $Vertex\, v \times DFA\, d\, fa \longrightarrow SubGraphMarker$
Returns a SubGraphMarker, starting at the given root vertex and being structured according to the given path description.
slice: $Set\, roots \times DFA\, d\, fa \longrightarrow SubGraphMarker$
Returns a SubGraphMarker, starting at the given root vertices and being structured according to the given path description.

startVertex.
startVertex: $Edge\, e \longrightarrow Vertex$
Returns the start vertex of a given edge.

that. Returns the far vertex of an oriented edge.
that: $Edge\, e \longrightarrow Vertex$

thatIncidenceIndex.
thatIncidenceIndex: $Edge\, e \longrightarrow Integer$
Returns the index of e in the incidence sequence of its that-vertex.
thatIncidenceIndex: $Edge\, e \times Vertex\, v \longrightarrow Integer$
Returns the index of e in the incidence sequence of v. Returns -1 if e is not in v's incidence sequence.

this. Returns the near vertex of an oriented edge.
this: $Edge\, e \longrightarrow Vertex$

thisIncidenceIndex.
thisIncidenceIndex: $Edge\, e \longrightarrow Integer$
Returns the index of e in the incidence sequence of its this-vertex.
thisIncidenceIndex: $Edge\, e \times Vertex\, v \longrightarrow Integer$
Returns the index of e in the incidence sequence of v. Returns -1 if e is not in v's incidence sequence.

topologicalSort. Returns a list of vertices in topological order, iff the graph g is acyclic. Otherwise, the result is undefined.
topologicalSort: $Graph\, g \longrightarrow List$

vertexSetSubgraph. Returns the subgraph induced by the vertex set, i.e. the vertices in vertexSet together with all edges between vertices in vertexSet.
vertexSetSubgraph: $Graph\, graph \times Collection\, vertexSet \longrightarrow SubGraphMarker$

vertexTypeSubgraph. Returns the subgraph induced by the vertex types in typeCollection, i.e. all vertices specified by typeCollection together with all edges between those vertices.
vertexTypeSubgraph: $Graph\, graph \times TypeCollection\, typeCollection \longrightarrow SubGraphMarker$

3.4 Logics

and. Logical AND. Can be used as infix operator: a and b.
and: $Boolean\, a \times Boolean\, b \longrightarrow Boolean$

not. Logical NOT. Can be used as unary operator: not a.
not: $Boolean\, a \longrightarrow Boolean$

or. Logical OR. Can be used as infix operator: a or b.
or: $Boolean\, a \times Boolean\, b \longrightarrow Boolean$

xor. Logical XOR, i.e., $(a \wedge \neg b) \vee (\neg a \wedge b)$.
xor: $Boolean\, a \times Boolean\, b \longrightarrow Boolean$

3.5 Paths and pathsystems and slices

contains.
contains: $PathSystem\, p \times GraphElement\, el \longrightarrow Boolean$
Returns true, iff p contains el.
contains: $Path\, p \times GraphElement\, el \longrightarrow Boolean$
Returns true, iff p contains el.

degree.
degree: $Vertex\, v \times Path\, p \longrightarrow Integer$
Returns the degree of vertex v. The scope is limited by a path, a path system.

depth. Returns the depth of the given path system.
depth: $PathSystem\, p \longrightarrow Integer$

distance. Returns the distance from the root to the given vertex in the given path system.
distance: $PathSystem\, ps \times Vertex\, v \longrightarrow Integer$

edgeTrace. Returns the edge trace of a Path p.
edgeTrace: $Path\, p \longrightarrow List$

edges.
edges: $Path\, p \longrightarrow List$
Returns the list of edges in the Path p.
edges: $SubGraphMarker\, s \longrightarrow Set$
Returns the set of edges in the given slice.

endVertex.
endVertex: $Path\, p \longrightarrow Vertex$
Returns the end vertex of the given path.

inDegree.
inDegree: $Vertex\, v \times Path\, p \longrightarrow Integer$
Returns the in-degree of the given vertex. The scope is limited by a path, a path system.

isReachable. Returns true, iff there is a path from vertex given as first argument to vertex given as second argument that matches the path description given as second argument. Usually invoked like so: myVertex (–> | <>–)+ myOtherVertex.
isReachable: $Vertex\, u \times Vertex\, v \times DFA\, dfa \longrightarrow Boolean$

leaves.
leaves: $PathSystem\, p \longrightarrow Set$
Returns the set of leaf vertices in the given path system.

outDegree.
outDegree: $Vertex\, v \times Path\, p \longrightarrow Integer$
Returns the out-degree of the given vertex. The scope is limited by a path, a path system.

pathSystem. Returns a path system with the given root vertex, which is structured according to the given path description.
pathSystem: $Vertex\, startVertex \times DFA\, fa \longrightarrow PathSystem$

reachableVertices. Returns all vertices that are reachable from the given vertex by a path matching the the given path description.
reachableVertices: $Vertex\, v \times DFA\, dfa \longrightarrow Set$

slice.
slice: $Vertex\, v \times DFA\, dfa \longrightarrow SubGraphMarker$
Returns a SubGraphMarker, starting at the given root vertex and being structured according to the given path description.
slice: $Set\, roots \times DFA\, dfa \longrightarrow SubGraphMarker$
Returns a SubGraphMarker, starting at the given root vertices and being structured according to the given path description.

startVertex.
startVertex: $Path\, p \longrightarrow Vertex$
Returns the start vertex of a given path.

vertexTrace. Returns the vertex trace of the given path.
vertexTrace: $Path\ p \longrightarrow List$

vertices.
vertices: $Path\ p \longrightarrow List$
Returns the list of vertices in the Path p.
vertices: $SubGraphMarker\ s \longrightarrow Set$
Returns the set of vertices in the given slice.
vertices: $PathSystem\ p \longrightarrow Set$
Returns the set of vertices in the given path system.

3.6 Reflection

valueType. Returns a String denoting the value type of the given object.
valueType: $Object\ val \longrightarrow String$

3.7 Relations

equals. Determines if *a* and *b* are equal. Alternative: a = b
equals: $Number\ a \times Number\ b \longrightarrow Boolean$
equals: $Enum\ a \times String\ b \longrightarrow Boolean$
equals: $String\ a \times Enum\ b \longrightarrow Boolean$
equals: $Object\ a \times Object\ b \longrightarrow Boolean$

grEqual. Determines if $a \geq b$. Alternative: a >= b
grEqual: $Number\ a \times Number\ b \longrightarrow Boolean$
grEqual: $Comparable\ a \times Comparable\ b \longrightarrow Boolean$

grThan. Determines if $a > b$. Alternative: a > b
grThan: $Number\ a \times Number\ b \longrightarrow Boolean$
grThan: $Comparable\ a \times Comparable\ b \longrightarrow Boolean$

leEqual. Determines if $a \leq b$. Alternative: a <= b
leEqual: $Number\ a \times Number\ b \longrightarrow Boolean$
leEqual: $Comparable\ a \times Comparable\ b \longrightarrow Boolean$

leThan. Determines if $a < b$. Alternative: a < b
leThan: $Number\ a \times Number\ b \longrightarrow Boolean$
leThan: $Comparable\ a \times Comparable\ b \longrightarrow Boolean$

nequals. Determines if *a* and *b* are different. Alternative: a <> b
nequals: $Number\ a \times Number\ b \longrightarrow Boolean$
nequals: $Enum\ a \times String\ b \longrightarrow Boolean$
nequals: $String\ a \times Enum\ b \longrightarrow Boolean$
nequals: $Object\ a \times Object\ b \longrightarrow Boolean$

3.8 Schema access

attributeNames.
attributeNames: $AttributedElementClass\ cls \longrightarrow Set$
Returns the set of attribute names of the specified schema class.
attributeNames: $AttributedElement\ el \longrightarrow Set$
Returns the set of attribute names of the specified element.

attributes.
attributes: $AttributedElementClass\ cls \longrightarrow List$
Returns the attribute names and domains of the specified schema class in terms of a vector containing one map per attribute with the keys name and domain.
attributes: $AttributedElement\ el \longrightarrow List$
Returns the attribute names and domains of the specified element in terms of a vector containing one map per attribute with the keys name and domain.

hasAttribute.
hasAttribute: $AttributedElementClass\ aec \times String\ name \longrightarrow Boolean$
Returns true, iff the attribute given by its name is defined for the given attributed element class.
hasAttribute: $AttributedElement\ el \times String\ name \longrightarrow Boolean$
Returns true, iff the attribute given by its name is defined for the given attributed element.

hasComponent. Returns true, iff the given record has a component with the given name.
hasComponent: $Record\ r \times String\ name \longrightarrow Boolean$

hasType.
hasType: $GraphElement\ el \times TypeCollection\ tc \longrightarrow Boolean$
Returns true, iff the given attributed element has an attributed element class accepted by the given type collection.
hasType: $GraphElement\ el \times String\ qn \longrightarrow Boolean$
Returns true, iff the given attributed element has an attributed element class with the given qualified name.

type. Returns the AttributedElementClass of the given element.
type: $AttributedElement\ el \longrightarrow AttributedElementClass$

typeName.
typeName: $AttributedElement\ el \longrightarrow String$
Returns the qualified name of the given element's type.
typeName: $AttributedElement\ el \times String\ kind \longrightarrow String$
Returns the name of the given element's type. If kind is "simple", return the simple name. If kind is "unique", return the unique name. Else, return the qualified name.

3.9 Statistics

count.
count: $Collection\, l \longrightarrow Integer$
Returns the number of items in the given collection.
count: $Map\, m \longrightarrow Integer$
Returns the number of items in the given map.

isEmpty.
isEmpty: $Map\, m \longrightarrow Boolean$
Returns true, iff m is empty.
isEmpty: $Set\, s \longrightarrow Boolean$
Returns true, iff s is empty.
isEmpty: $List\, v \longrightarrow Boolean$
Returns true, iff v is empty.

max.
max: $Number\, a \times Number\, b \longrightarrow Number$
Returns the maximum of the given two numbers.
max: $Collection\, l \longrightarrow Comparable$
Returns the maximum of a collection of comparable things.

mean. Returns the mean value of a collection of numbers.
mean: $Collection\, l \longrightarrow Double$

min.
min: $Number\, a \times Number\, b \longrightarrow Number$
Returns the minimum of the given two numbers.
min: $Collection\, l \longrightarrow Comparable$
Returns the minimum of a collection of comparable things.

sdev. Returns the standard deviation of a collection of numbers. If the collection's size is less than 2, the standard deviation is undefined.
sdev: $Collection\, l \longrightarrow Double$

sum. Returns the sum of the given collection of numbers.
sum: $Collection\, l \longrightarrow Number$

variance. Returns the variance of the given collection of numbers. If the size of the collection is less than 2, the variance is undefined.
variance: $Collection\, l \longrightarrow Double$

3.10 Strings

capitalizeFirst. Returns the given string with the first character made uppercase.
capitalizeFirst: $String\, s \longrightarrow String$

concat.
concat: $String\, a \times Object\, b \longrightarrow String$
Concatenates strings. Can be used as infix operator: a ++ b.
concat: $Object\, a \times String\, b \longrightarrow String$
Concatenates strings. Can be used as infix operator: a ++ b.

contains.
contains: $String\, s \times String\, sub \longrightarrow Boolean$
Returns true, iff s contains sub.

endsWith. Returns true, iff the String s ends with the given suffix.
endsWith: $String\, suffix \times String\, s \longrightarrow Boolean$

indexOf.
indexOf: $String\, sub \times String\, s \longrightarrow Integer$
Returns the index of the first occurence of sub in s, or -1 if sub is not in s.

join. Joins the strings in the given collection by interleaving with the given delimiter.
join: $Collection\, l \times String\, delimiter \longrightarrow String$

length. Returns the length of String s.
length: $String\, s \longrightarrow Integer$

lowerCase.
lowerCase: $String\, s \longrightarrow String$
Returns s in lowercase letters.

reMatch. Returns true, iff the given string matches the given regular expression. Can be used as infix operator: myString =~ myRegexp.
reMatch: $String\, s \times String\, regex \longrightarrow Boolean$

replace.
replace: $String\, s \times String\, old \times String\, new \longrightarrow String$
Replaces all occurences of old in s with new.

split. Splits the given string according to the given regular expression and returns the parts as list.
split: $String\, s \times String\, regex \longrightarrow List$

startsWith.
startsWith: $String\, prefix \times String\, s \longrightarrow Boolean$
Returns true, iff the String s starts with the given prefix.
startsWith: $String\, prefix \times String\, s \times Integer\, offset \longrightarrow Boolean$
Returns true, iff the String s starts with the given prefix, beginning search at the given offset.

substring.
substring: $String\, s \times Integer\, beginIndex \longrightarrow String$
Returns the substring of s starting at beginIndex.
substring: $String\, s \times Integer\, beginIndex \times Integer\, endIndex \longrightarrow String$
Returns the substring of s from beginIndex (incl) to endIndex (excl).

toString. Returns the string representation of the given object.
toString: $Object\ o \longrightarrow String$

upperCase.
upperCase: $String\ s \longrightarrow String$
Returns s in uppercase letters.

3.11 Miscellaneous

isDefined. Returns true, iff the given object is defined.
isDefined: $Object\ val \longrightarrow Boolean$

isUndefined. Returns true, iff the given object is undefined.
isUndefined: $Object\ val \longrightarrow Boolean$

log. Logs a line of the form *s* ++ toString(*o*) to sysout and returns *o*.
log: $String\ s \times Object\ o \longrightarrow Object$