SDMX Technical Working Group
VTL Task Force

Validation & Transformation Language

Part 2 - Library of Operators

Version 1.0

February 2015

30

Foreword

The SDMX Technical Working Group is pleased to present the version 1.0 of the Validation and Transformation Language, in short VTL.

The work on VTL was launched at the end of 2012 by the SDMX Secretariat. SDMX already has a package for transformations and expressions which is present in the information model, although a specific language does not exist yet. To make this framework fully operational, a standard "language" for defining validation and transformation rules (set of operators, their syntax and semantics) should be adopted, appropriate IT formats for exchanging such rules and related metadata should be introduced, and the web services to store and retrieve them should be designed.

A task force was put in place, composed of members of SDMX, DDI and GSIM communities and the work started in summer 2013. The intention was to provide a language which is usable by statisticians to express logical validation rules and transformations on data, whether described as dimensional tables or as unit-record data. The assumption is that this logical formalization of validation and transformation rules would be converted into specific programming languages for execution (SAS, R, Java, SQL, etc.) but would provide a "neutral" expression at business level of the processing taking place, against which various implementations can be mapped. Experience with existing examples suggests that this goal would be attainable.

An important point that emerged is that several standards are interested in such a language. However, each standard operates on its model artefacts and produces artefacts within the same model (property of closure). To cope with this, VTL has been built upon a very basic information model, taking the common parts of GSIM, SDMX and DDI, mainly using artefacts from GSIM 1.1, somewhat simplified and with some additional detail. This way the existing standards (SDMX, DDI, others) may adopt VTL by mapping their information model against the VTL one. Therefore, although a work-product of SDMX, the VTL language will be usable also with other standards.

The VTL 1.0 package includes:

a) Part 1, highlighting the main characteristics of VTL, its core assumptions and the information model the language is based on;

b) Part 2, containing the full library of operators ordered by category, including examples;

c) BNF notation (Backus-Naur Form) which is the technical notation to be used as a test bed for all the examples throughout the document.

The present document (VTL part 2) contains the full list of VTL Operators and describes their main characteristics. This first version can support validation and basic compilation needs. Future versions will include more features related to transformation of data.

The latest version of the VTL is freely available online at www.sdmx.org.

Acknowledgements

This publication has been prepared thanks to the collective input of experts from Bank of Italy, Bank for International Settlements (BIS), European Central Bank (ECB), Eurostat, ILO, ISTAT (Italy), OECD, Statistics Netherlands, and UNESCO. Other experts from the SDMX Technical Working Group, the SDMX Statistical Working Group and the DDI initiative were consulted and participated in reviewing the documentation.

The list of authors, contributors and reviewers includes the following experts: Sami Airo, Foteini Andrikopoulou, David Barraclough, Luigi Bellomarini, Marc Bouffard, Angel Simon Delgado, Vincenzo Del Vecchio, Fabio Di Giovanni, Jens Dossé, Heinrich Ehrmann, Bryan Fitzpatrick, Arofan Gregory, Edgardo Greising, Angelo Linardi, Chris Nelson, Stratos Nikoloutsos, Marco Pellegrino, Michele Romanelli, Juan Alberto Sanchez, Daniel Suranyi, Olav ten Bosch, Laura Vignola, Nikolaos Zisimos.

Feedback and suggestions for improvement are encouraged and can be sent to the SDMX Technical Working Group (twg@sdmx.org).

Table of contents

VTL library general picture

The VTL library of the Operators is described hereinafter. This is the first version of the library, oriented mainly to data validation purposes. The operators included in this version of the VTL are summarized in the diagrams and tables below.

Diagram of the Operators

List of the Operators

The following table lists the VTL Operators and describes their main characteristics. They are ordered by category except the clauses, which are the operators having a postfix syntax, that are shown all together in the end.

VTL includes operators that may operate both on Data Sets and on Structure Components of the Data Sets. The last column shows if the Operator has a version that operates on Components (besides the version that operates on Data Sets) or if it operates only on Data Sets. The Component version takes as input and returns in output Component expressions. They are not typically used in assignments, since Components cannot be stored directly into the system. Instead, they are part of the syntax of other operators or clauses, where Components are needed. Component expressions are intended to be processed row-wise.

Operator	Category	Syntax	Description	Operand Data Sets	Component version
Round parenthesis ()	General purpose	Functional	Specifies the evaluation precedence	1	YES
assignment :=	General purpose	Infix	Assigns an Expression to a model artefact	2	NO
Membership #	General purpose	Infix	Identifies a Component within a Data Set	1	NO
get	General Purpose	Functional	Retrieves a Data Set	1..N	NO
put	General Purpose	Functional	Stores a Data Set	1	NO
eval	General Purpose	Functional	Evaluates an external routine	1	NO
length	String	Functional	Returns the length of a string	1	YES
concatenation +	String	Functional	Concatenates two strings	2	YES
trim	String	Functional	Eliminates trailing and leading whitespace from a String	1	YES
upper/lower	String	Functional	Makes a string upper / lower case	1	YES

substr	String	Functional	Extracts a substring from a string	1	YES
indexof	String	Functional	Returns the position of a String in another one	1	YES
unary plus +	Mathematical	Infix	Leaves the sign unaltered	1	YES
unary minus -	Mathematical	Infix	Changes the sign	1	YES
addition +, ++ and subtraction -, --	Mathematical	Infix	Sum or subtract	2	YES
multiplication *, ** and division /, //	Mathematical	Infix	Multiply or divide	2	YES
round	Mathematical	Functional	Rounds the values	1	YES
abs	Mathematical	Infix	Calculates the absolute value	1	YES
trunc	Mathematical	Infix	Truncates the values	1	YES
exp	Mathematical	Infix	Calculates the exponential	1	YES
ln	Mathematical	Infix	Calculates the natural logarithm	1	YES
log	Mathematical	Infix	Calculates the a base b logarithm	1	YES
power	Mathematical	Infix	Calculates the power	1	YES
nroot	Mathematical	Infix	Calculates the n-th root	1	YES
mod	Mathematical	Infix	Calculates the m modulo	1	YES
and	Boolean	Infix	Calculates the logical AND	2	YES
or	Boolean	Infix	Calculates the logical OR	2	YES
xor	Boolean	Infix	Calculates the logical XOR	2	YES
not	Boolean	Infix	Calculates the logical NOT	1	YES

union	Relational	Infix	Makes the set union	2	NO
intersect	Relational	Infix	Makes the set intersection	2	NO
symdiff	Relational	Functional	Makes the set symmetric difference	2	NO
setdiff	Relational	Infix	Makes the set difference	2	NO
merge	Relational	Infix	Carries out the join	2..N	NO
min/max	Statistical	Infix	Extracts the Data Point having minimum / maximum value	1	NO
hierarchy	Statistical	Infix	Aggregates following hierarchical links	1	NO
equal to =	Validation	Infix	Compares the values	2	YES
not equal to <>	Validation	Infix	Compares the values	2	YES
Greater than >,>=	Validation	Infix	Compares the values	2	YES
Less than <, <=	Validation	Infix	Compares the values	2	YES
in, not in	Validation	Infix	Verify if a value belongs to a set of values	2	YES
between	Validation	Infix	Verify if a value belongs to a set of values	2	YES
isnull	Validation	Functional	Compares the values with the NULL literal	1	YES
exists_in, not_exists_in, exists_in_all, not_exists_in_all	Validation	Functional	Checks the Identifiers and the foreign keys	2	NO
check	Validation	Functional	Checks if an expression verifies a condition	1	NO

match_characters	Validation	Functional	Checks for the presence of some characters	1	NO
match_values	Validation	Functional	Checks if the Value Domain Subset is respected	1	NO
if-then-else	Conditional	Functional	Makes different calculations according to a condition	1	YES
nvl	Conditional	Functional	Alters NULL values	1	YES
rename	General purpose	Clause (Postfix Operator)	Renames the Structure Components	1	YES (ONLY)
filter	Relational	Clause (Postfix Operator)	Filters the Data Points	1	YES (ONLY)
keep	Relational	Clause (Postfix Operator)	Alters the Data Structure	1	YES (ONLY)
calc	Relational	Clause (Postfix Operator)	Calcuates the values of a Structure Component	1	YES (ONLY)
attrcalc	Relational	Clause (Postfix Operator)	Calculates the values of an Attribute	2..N	YES (ONLY)
aggregate	Statistical	Clause (Postfix Operator)	Calculates summary values for groups of Data Points	1	YES (ONLY)

184

Evaluation order of the Operators

Within a single expression, the operators are applied in sequence, according to the precedence order. Operators with the same precedence level are applied according to associativity rules. Precedence and associativity orders are reported in the following table.

Order	Operator	Description	Associativity
I	()	Round parenthesis. To alter the default order.	Left-to-right
II	get put eval length concatenation trim upper/lower substr indexof round abs trunc exp ln log power nRoot mod union intersect symdiff setdiff merge min/max hierarchy isnull check match_characters match_values nvl	The majority of the operators of the VTL	Left-to-right
III	# rename filter keep calc attrcalc aggregate	Membership & Clauses	Left-to-right

IV	unary plus unary minus not	Unary minus Unary plus Logical negation	Right-to-left
V	*, ** /, //	Multiplication Division	Left-to-right
VI	+, ++ -, --	Addition Subtraction	Left-to-right
VII	> >= < <= in, not in between	Greater than Less than In (not in) a value list In a range	Left-to-right
VIII	exists_in not_exists_in exists_in_all not_exists_in_all	Identifiers matching	Left-to-right
IX	= <>	Equal-to Not-equal-to	Left-to-right
X	and	Logical AND	Left-to-right
XI	or xor	Logical OR Logical XOR	Left-to-right
XII	if-then-else	Conditional (if-then-else)	Right-to-left
XIII	:=	Assignment	Right-to-left

190

Writing conventions for operators and types

In the remainder of the document, the following writing conventions (which are not part of the language) are used when describing assignments, commands and operators.

For denoting the type of Parameter:

`ParameterTypeName<TypeName>`

For denoting Parameter types with more than one data type is:

`ParameterTypeName<TypeName1, TypeName2, …, TypeNameN>`

Or, for brevity, with a regular expression formalism:

`ParameterTypeName<TypeName1+>` (one or more)

`ParameterTypeName<TypeName1*>` (zero or more)

When the data type is unspecified the following syntax is adopted:

`ParameterTypeName<?>`

To indicate a multiplicity of unspecified data types:

`ParameterTypeName<?+>` (one or more)

`ParameterTypeName<?*>` (zero or more)

When the data type in a Parameter type is unspecified, it can be templatized (assigned a name) in order to be cited elsewhere:

`ParameterTypeName<T>`

To indicate a multiplicity of templatized data types:

`ParameterTypeName<T+>` (one or more)

`ParameterTypeName<T*>` (zero or more)

To assert that an artefact is of a certain data type is:

`typeName artefact`

Operator names and parameters are **case sensitive**.

In general, some operators have *infix style*, others have *functional style* and the clauses have *postfix style*. Their syntax is described with **templates**, which are textual descriptions with the following conventions:

{optionalPart}: an optional part is delimited by curly braces

{one-or-more}+: a syntactical part that is repeated from 1 to many occurrences

{zero-or-more}*: a syntactical part that is repeated from 0 to many occurrences

[part1|part2|part3]: alternative syntactical parts

[part1|part2|part3]+: alternative syntactical parts, from 1 to many occurrences

[part1|part2|part3]*: alternative syntactical parts, from 0 to many occurrences

Typically, in the description of operators there are type constraints that cannot be expressed with this regular expression-like formalism. They are textually described in the "Constraints" sections.

Example 1

```
Dataset<?+, MeasureComponent<Numeric>+> ds := abs
({dataset=}Dataset<?+, MeasureComponent<Numeric>+> ds_1)
```

The template above synthesizes:

- "**:=**", **abs**, "**(**" , "**)**" and "**dataset=**" are reserved keywords;
- **abs** takes in input an expression `ds_1`, whose type is constrained to be a *Dataset* with one or more Numeric *MeasureComponent*s. "`?+`" indicates that other Identifier or *AttributeComponent*s can be present (at least one *IdentifierComponent*).
- There can optionally be the "**dataset=**" keyword.
- It returns an expression `ds`, whose type is a *Dataset* with one or more Numeric *MeasureComponent*s. "`?+`" indicates that other Identifier or *AttributeComponent*s can be present (at least one *IdentifierComponent*).

From this template, it is possible to infer some valid instances of the abs operator:

```
ds_1 := abs(ds_2)
ds_1 := abs(dataset=ds_3)
```

Example 2

```
Dataset<?+, MeasureComponent<Numeric>+> ds := trunc
({dataset=}Dataset<?+, MeasureComponent<Numeric>+> ds_1,
{decimals=}Constant<Integer> d)
```

The template above synthesizes:

- "**:=**", **trunc**, "**(**" , "**)**", "**dataset=**" and "**decimals=**" are reserved keywords
- **trunc** takes in input an expression `ds_1`, whose type is constrained to be a *Dataset* with one or more Numeric *MeasureComponent*s. "`?+`" indicates that other Identifier or *AttributeComponent*s can be present (at least one *IdentifierComponent*).
- `trunc` takes in input an expression d, whose type is constrained to be a *Constant*<Integer>
- There can optionally be the "**dataset=**" and "**decimals=**" keywords
- It returns an expression `ds`, whose type is a *Dataset* with one or more Numeric *MeasureComponent*s. "`?+`" indicates that other *Identifier* or *AttributeComponent*s can be present (at least one *IdentifierComponent*).

From this template, it is possible to infer some valid instances of the **trunc** operator:

```
ds_1 := trunc(ds_2,3)
ds_1 := trunc(dataset=ds_3,3)
ds_1 := trunc(dataset=ds_3,decimals=3)
ds_1 := trunc(ds_3,decimals=3)
```

Example 3

```
Dataset<?+, MeasureComponent<Numeric>+> ds := Dataset<?+,
MeasureComponent<Numeric>+> ds_1 + Dataset<?+,
MeasureComponent<Numeric>+> ds_2
```

The template above synthesizes:

- "**:=**", "**+**" are reserved keywords of the operator
- "**+**" takes in input two expressions `ds_1` and `ds_2`, whose type is constrained to be a *Dataset* with one or more Numeric *MeasureComponent*s and at least one other (*Identifier*) *Component*.
- It returns an expression `ds`, whose type is a *Dataset* with one or more Numeric *MeasureComponent*s and at least one other (*Identifier*) *Component*.

From this template, it is possible to infer some valid instances of the + operator, like the following one:

```
ds_1 := ds_2 + ds_3
```

When no constraints on types need to be enforced, artefact names can be simply used. The following assignment is an example:

```
artefact := expression
```

General purpose operators

round parentheses ( )

Syntax

```
( expression )
```

Constraints

No specific constraints.

Semantics

The round parenthesis specify explicitly the order of evaluation of the operators

assignment :=

Syntax

```
variable parameter := expression
```

Constraints

No specific constraints.

Semantics

The assignment is the basic building block of VTL statements.

The ":=" symbol allows to assign the value of an expression to a variable parameter. An expression may evaluate to any Composite or Collection data type (Datasets, Components, Constants and Collections).

Examples

Assignment of a Constant<Numeric> value to a parameter:

```
numpi := 3.14
```

Assignment of a String value to a parameter:

```
str := "hello world"
```

Assignment of an expression to a parameter:

```
popA := populationDS + 1
```

Assignment of a Dataset expression to a parameter:

```
ds_1 := get("NAMESPACE/DF_NAME/2000.USD.M.F.A.BOP.ANN.STO.EABL")
```

Assignment of a Constant<Boolean> value to a parameter:

```
bool_var := true
```

membership

Syntax

```
Dataset<?*,Component<?> c> ds
#
Component<?> c
```

Constraints

- `c` must be a Component of `ds` (static)

Semantics

The membership operator allows to select a single Component within the application of another operator:

- `c` is a *Component*
- All the other *Component*s are not considered in the application of the operator and are preserved with unaltered roles. In case of duplications of names of Components, the added ones will be suffixed with "_1", though a renaming is advisable.

Parameters

- `ds` – the Dataset to be altered
- `c` – the Component that has been selected

Returns

Identifies a Component of the Dataset

Examples

The membership operator is particularly useful to work with operators that have specific constraints in terms of the types of the *MeasureComponent*s or have more than one *MeasureComponent*.

Example 1

Suppose `ds_1` is a multi-measure Dataset, where `M1` is a *MeasureComponent*<Numeric> and `M2` is a *MeasureComponent*<String>; let `ds_2` be a mono-measure Dataset with a single *MeasureComponent*<Numeric> `M1`. `ds_1` and `ds_2` have the same *IdentifierComponent*s. Let us supposed the sum `ds_1 + ds_2` is desired.

The Datasets cannot be directly summed, as `ds_1` has a *MeasureComponent* that is not present in `ds_2` besides being non-numeric.

The following syntax: `ds_1#M1 + ds_2` makes this assignment possible, as `ds_1` is temporarily considered mono-measure.

Example 2

Suppose the comparison operator ("=") needs to be applied on the *IdentifierComponent* `COUNTRY` of the Dataset `ds_1`. The comparison operator, as defined, acts on *MeasureComponent*s.

```
ds_2 := ds_1#COUNTRY="Luxembourg"
```

In this expression, the membership operator specifies that the *IdentifierComponent* `COUNTRY` is temporarily considered as the only *MeasureComponent* to be used in the comparison.

Example 3

Suppose it is needed to round an *IdentifierComponent*. The round operator acts on *MeasureComponent*s, which must be all Numeric. Suppose we have a Dataset `ds_1` with a *MeasureComponent*<String> `DESCRIPTION` and an *IdentifierComponent*<Numeric> `AVERAGE_AGE`, which needs to be rounded to the 3rd decimal. The expression:

```
ds_1 := round(ds_1#AVERAGE_AGE,3)
```

performs this task.

`ds_1#AVERAGE_AGE` temporarily considers `AVERAGE_AGE` the only *MeasureComponent*<Numeric> of `ds_1`. The round is then normally applied and the `DESCRIPTION` propagated.

Example 4

Let us suppose we have two multi-measure Datasets `ds_1` and `ds_2`, having the same *IdentifierComponent*s `K1` and `K2`, and the same *MeasureComponent*s `M1` (which is a Numeric), `M2` which is a String.

The expression

```
ds_3 := ds_1#M1 + ds_2#M1
```

sums only the *MeasureComponent* `M1`. The outcome Dataset has three *MeasureComponent*s: `M1`, which is obtained as the sum of `M1` in `ds_1` and in `ds_2`, `M2` originating from `ds_1` and `M2_1` originating from `ds_2`.

get

Syntax

```
Dataset<?> ds_o := get
      (PersistentDataset<?> ds_id {, PersistentDataset<?> ds_id}*
      {,keep(keepPart Component<?> {, keepClause Component}*)}
      {,filter(Component <Boolean> filterPart)}
      {,aggregate(
            [sum|avg|median|count|count_distinct|min|max](
            {include NULLS} MeasureComponent<Numeric> aggrPart)
            {,[sum|avg|median|count|count_distinct|min|max](
            {include NULLS} MeasureComponent<Numeric> aggrPart)}*
      )}
      )
```

381

382 Constraints

383 • ds_*id* is a PersistentDataset, whose name responds to the syntax explained in Section
384 "Data Identification Scheme" (static).
385 • All the input Datasets *ds_id* must have the same Logical Data Structure, which is the
386 same Components in number, name and type (static).
387 • *keepPart* must be a Component expression containing exactly the name of a
388 Component of any *ds* (complex Component expressions, combining more than one
389 Component are not allowed) (static).
390 • *aggrPart* must be a Component expression containing exactly the name of a
391 MeasureComponent present in any *ds* (no complex Component expressions, combining
392 more than one Component is allowed). If there is at least one *aggrPart*, there must be
393 one for each MeasureComponent that is present in a *keepPart*. If *keepPart* is omitted,
394 all MeasureComponents must be in the aggregate. This means that there cannot be
395 MeasureComponents, kept that are not used in aggregations (static).

396 Semantics

397 It is the data retrieval command. It allows to fetch all the instances of a PersistentDataset from
398 the system and returns a Dataset expression containing them. It takes in input a number (at
399 least one) of PersistentDatasets ds. Together with *put*, it is the only operator in VTL where a
400 PersistentDataset can be mentioned.

401 All the Datasets must have the same Logical Data Structure, which is the same Components in
402 number, name and type.

403 The command operates as follows: considers all the instances of the identified
404 PersistentDataset (selected according to the semantics of the identifier); builds a union with
405 duplicates (like SQL UNION ALL); keeps in the result only the Components that are present in
406 the *keepPart* (like SQL SELECT). If the keep part is omitted, all the Components are preserved
407 in the result; selects the only instances returning *true* for the *filterPart* Component<Boolean>
408 expression. For the *filterPart*, any complex Component <Boolean> expression over all the
409 Datasets Components (not only the ones mentioned in the *keepPart* can be used) and it is
410 evaluated row-wise (like SQL WHERE).

411 Finally, the command aggregates (like SQL aggregations and GROUP BY) applying an
412 aggregation function (**sum**: sum, **avg**: average, median: the statistical median, count: count of
413 occurrences, considering duplicates, **count_distinct**: count of occurrences, without
414 duplicates, **min**: minimum occurrence, **max**: maximum occurrence) over the
415 MeasureComponent specified in *aggrPart* grouing by the IdentifierCompoenents that are kept
416 in the *keepPart* (or all if there is no *keepPart*).

417

418 NULL values are considered in aggregations only if the *include NULLS* part is present.
419 Specifically, they propagate as usual resulting in a NULL sum, average or median if at least one
420 NULL is present among the values; in a NULL minimum or maximum if the only value to
421 aggregate coincides with NULL; they are considered as always distinct in both count and
422 cound_distinct.

423 Viceversa, if *include* NULLS part is absent, NULL values are not considered in aggregations.

Parameters

PersistentDataset<?> *ds_id* – the PersistentDataset to be fetched

Component<?> *keepPart* - a Component name

Component<Boolean> *filterPart* – a Component<Boolean> expression which is evaluated row-wise and states if a row is to be kept (if evaluates to *true)* or removed (if it does not evaluate to true) from the result.

MeasureComponent<Numeric> *aggrPart* – the numeric MeasureComponent to aggregate.

Returns

DataSet<?> *ds_o* – a Dataset expression obtained as the union with duplicates of all the Datasets specified by the identifiers *ds*, keeping only the columns specified in the *keepParts* and the rows in the *filterParts*, aggregating over the all the MeasureComponents in *aggrParts* grouping

Example 1

The expression:

```
ds_1 := get("DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", keep(K1, K2, M1))
```

Retrieves Dataset identified by

```
DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO
```

from the system, keeping the Identifier Components K1 and K2 and the Measure Component M1.

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO		
K1	**K2**	**M1**
1	**A**	**5**
2	**B**	**7**

ds_1		
K1	**K2**	**M1**
1	**A**	**5**
2	**B**	**7**

Example 2

The expression:

```
Ds_1 := get("DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO",
("DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO", keep(K1, K2, K3,
M1)))
```

Retrieves the union of datasets identified by

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO

and

DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO

keeping the Identifier Components K1, K2 and K3 and the Measure Component M1 for all of them.

***DF_NAME*/2000-2010.USD.M.F.A.BOP.ANN.STO**			
K1	K2	K3	M1
1	A	X	5
2	B	Y	7

***DF_NAME*/2011-2012.USD.M.F.A.BOP.ANN.STO**			
K1	K2	K3	M1
1	A	X	6
2	B	Y	7
3	C	Z	9

ds_1			
K1	K2	K3	M1
1	A	X	5
2	B	Y	7
2	B	Y	7
1	A	X	6
3	C	Z	9

Example 3

The expression:

```
ds_1 := get("DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO",
("DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO", keep(K1, K2, K3,
M1), filter(K3="X")))
```

retrieves the union of datasets identified by

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO
and
DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO

keeping the Identifier Components K1, K2 and K3 and the Measure Component M1 for all of them and selecting only the rows where the value of the Component K3 equals to the Constant<String> "X".

NAMESPACE/DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO			
K1	**K2**	**K3**	**M1**
1	**A**	**X**	**5**
2	**B**	**Y**	**7**

NAMESPACE/DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO			
K1	**K2**	**K3**	**M1**
1	**A**	**X**	**6**
2	**B**	**Y**	**7**
3	**C**	**Z**	**9**

ds_1			
K1	**K2**	**K3**	**M1**
1	**A**	**X**	**5**
1	**A**	**X**	**6**

Example 4

The expression:

```
ds_1 := get(
"DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", "DF_NAME/2011-
2012.USD.M.F.A.BOP.ANN.STO",
keep(K1, K2, M1), aggregate(sum(M1)))
```

retrieves the union of datasets identified by

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO
and *DF_NAME*/2011-2012.USD.M.F.A.BOP.ANN.STO

keeping the Identifier Components K1 and K2 and the Measure Component M1 for all of them. It aggregates over the MeasureComponent M1, grouping by the IdentifierComponents K1 and K2.

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO				
K1	K2	K3	M1	M2
1	A	X	5	2
2	B	Y	7	3

DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO				
K1	K2	K3	M1	M2
1	A	Y	6	5
2	B	Y	7	7
3	C	Z	9	11

ds_1		
K1	K2	M1
1	A	11
2	B	14
3	C	9

Example 5

The expression:

```
ds_1 := get(
"DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", "DF_NAME/2011-
2012.USD.M.F.A.BOP.ANN.STO",
keep(K1, K2, M1,M2) filter(K3>5 or K3=1),
aggregate(sum(M1),max(M2)))
```

retrieves the union of datasets identified by

*DF_NAME/*2000-2010.USD.M.F.A.BOP.ANN.STO

and *DF_NAME/*2011-2012.USD.M.F.A.BOP.ANN.STO

keeping the Identifier Components K1 and K2 and the Measure Components M1 and M2 for all of them. It selects only the rows where K3 is greater than 5 or exactly 1. It aggregates over the MeasureComponent M1 by sum, over the MeasureComponent M2 by max, grouping by the IdentifierComponents K1 and K2.

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO				
K1	K2	K3	M1	M2
1	A	10	5	2
2	B	1	7	3

DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO				
K1	K2	K3	M1	M2
1	A	25	6	5
2	B	1	7	7
3	C	3	9	11

ds_1			
K1	K2	M1	M2
1	A	11	5
2	B	14	7

put

Syntax

```
Dataset<?> ds_1 :=
put
(Dataset<?> ds, PersistentDataset<?> ds_id)
```

Constraints

- ds_*id* is a PersistentDataset, whose name responds to the syntax explained in Section "Data Identification Scheme"(static).
- The Dataset *ds* must not contain duplicates, i.e. rows with the same values for all the IdentifierComponents (static).
- The Logical Data Structure of *ds* must conform to the one of the Dataset in the system that is identified by *ds_id* (static).

Semantics

Stores the content of a Dataset expression *ds* into the PersistentDataset *ds_id*. It returns a copy of the input Dataset.

As long as a Dataset is only memorized in a parameter *ds*, it is not persistent. **Put** allows to make it persistent, by mapping its content to a specific PersistentDataset in the system. As explained in the specific section, *ds_id* can also select a particular portion of a Dataset. It

follows that in case *ds* has any Data Points whose Identifier Components are not conform to the ones specified in *ds_id* or they are filtered out by filters possibly used in *ds_id*, they will be discarded and not stored into the system.

Together with *get*, it is the only operator in VTL where a PersistentDataset can be mentioned.

Parameters

ds – the Dataset to be stored into the system.

ds_id – the PersistentDataset

Returns

ds_1 – the output Dataset, which is a copy of the input one.

Example

```
ds := put(ds_1," DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO")
```

ds_1 is stored.

```
ds := put(log((ds_1 + ds_2),10),"DF_NAME/2000-
2010.USD.M.F.A.BOP.ANN.STO")
```

The result of logarithm is stored, while the sum is not persistent.

```
ds := put(log(put(ds_1 + ds_2, "DF_NAME/2000-
2011.USD.M.F.A.BOP.ANN.STO"),10)," DF_NAME/2000-
2010.USD.M.F.A.BOP.ANN.STO")
```

Both the results of the sum and the logarithm are stored into the system. The fact that put outputs the input expression allows for this kind of use.

eval

Syntax

```
Dataset<?> ds_o :=
eval
(Constant<String> language,
[{script=}Constant<String> script | Constant<String> programPath],
{,{params=}ConstantList<?> parameterList}
, {dataset=}PersistentDataset ds_id)
```

Constraints

- *language* must be the name of a programming language, meaningful and executable in the target system (such as a SQL stored-procedure language, R, STATA, etc.) (dynamic).
- *script* must be the code of a program, valid with respect to the specified language. The program can perform whatever internal logic, but is forced to calculate and autonomously store exactly one PersistentDataset, *ds_id* (dynamic).

- *programPath* must be a valid path in the target system to a program file, compliant with the specified language. It does not necessarily correspond to a filesystem file, but can also be the identifier of a DBMS stored procedure, and so forth (dynamic).
- *parameterList* must be compatible in order and type with the input parameters of the script (dynamic).

Semantics

Eval operator allows executing a user-defined program to calculate a Persistent Dataset.

The program is user-defined and can perform any internal logic, however it has to adhere to some conventions:

- it can take as input only String or Numeric parameters, which are directly bound to *parameterList*;
- it must autonomously store its results into a single PersistentDataset *ds_id*. Indeed, the operator fetches the saved Dataset (like a common **get** operation) and returns it as output, which can be handled within other VTL expressions;
- it must calculate exactly one Dataset;
- it cannot refer to a parameter variable, but can only work with physical objects, such as relational tables (for SQL), data frames (for R), which are loaded autonomously by the program with the appropriate commands. Therefore, if a Dataset that has been calculated in a previous step needs to be used within a user-defined program, it must be stored (with a **put**) into the system and loaded appropriately by the program logic afterwards;
- it must return 0 if it has terminated correctly, a negative number otherwise.

Parameters

Constant<String> *language* – the programming language of the script

Constant<String> *script* - the code of the script

Constant<String> *programPath* – a path to a script file

ConstantList<?> *parameterList* – the List of input parameters for the script

PersistentDataset<?> *ds_id* – the PersistentDataset the program saves into

Returns

DataSet<?> *ds_o* – the Dataset whose content has been calculated by the user-defined script.

Example

The following example shows the usage of a user-defined function that calculates the n-th power of a Dataset. Indeed, this would be meaningless as VTL already provides a native operator for such operation, however it shows the potentialities of **eval**.

```
ds_o := eval("PL/SQL","programs.dsToN",[5],
"NAMESPACE/DF_NAME/2010.USD.M.F.A.BOP.ANN.STO")
```

where `programs.dsToN` contains the following code:

```
create function programs.dsToN(n number)
```

```
    return number is

    begin
    insert into 2010.USD.M.F.A.BOP.ANN.STO
        select dim1, dim2, …, power(measure,n)
        from 2010.USD.M.F.A.BOP.ANN.STO2;
    except
        when others then
            return -1;
    return 0;
    end;
```

The reported stored function takes in input a number n and elevates every Data Point within PersistentDataset `2010.USD.M.F.A.BOP.ANN.STO2` to the n-th power and stores the result into PersistentDataset `2010.USD.M.F.A.BOP.ANN.STO`.

It returns 0 if the operation succeeded, -1 otherwise.

At the end of a correct execution, *ds_o* will contain the fetched content of the PersistentDataset `2010.USD.M.F.A.BOP.ANN.STO`.

String operators

length

Component version

Syntax

```
Component<Integer> c :=
length
({component=}Component<String> c_1)
```

Constraints

None

Semantics

Returns the length of c_1.

Example

If A = “Hello, World!”:

```
B := length(A) // B = 13
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Integer>+> ds_2 :=
length
({dataset=}Dataset<?+, MeasureComponent<String>+> ds_1)
```

Constraints

- *ds_1* only has MeasureComponents<String> (static).

Semantics

Get length of a string. It takes in input a Dataset *ds_1* and returns another Dataset *ds_2* with the same IdentifierComponents, AttributeComponents (according to attribute propagation rules) and a MeasureComponent<Integer> for each MeasureComponent<String> in *ds_1* with the same name. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are obtained by calculating the length of the String in the original MeasureComponent.

Parameters

ds_1 – the input Dataset

Returns

ds_2 – the output Dataset with the lengths

Example

```
ds_r := length(ds_1)
```

ds_1		
K1	K2	M1
1	A	“hello”
2	B	“”

ds_r		
K1	K2	M1
1	A	5
2	B	0

concatenation +

Component version

Syntax

```
Component<String> c :=
Component<String> c_1
+
Component<String> c_2
```

Constraints

None

Semantics

Returns the concatenation of *c_1* and *c_2*.

Example

If A = “Hello,”, B = “ world!”:

```
C := A + B // C = “Hello, world!”
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<String>+> ds_3 :=
[Dataset<?+, MeasureComponent<String>+>|Constant<String>] ds_1
+
[Dataset<?+, MeasureComponent<String>+>|Constant<String>] ds_2
```

Constraints

- At least one between *ds_1* and *ds_2* must be a Dataset (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same IdentifierComponents<?>, which must be equal in name and type or, at least, the IdentifierComponents of one must be a subset of the IdentifierComponents of the other (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents (static).
- If *ds_1* is a Dataset, it must have only String MeasureComponents (static).
- If *ds_2* is a Dataset, it must have only String MeasureComponents (static).

Semantics

Takes in input two Datasets *ds_1* and *ds_2,* with at least one String MeasureComponent, and such that the IdentifierComponents of *ds_1* are a subset of the IdentifierComponents of *ds_2* or vice-versa and *ds_1* and *ds*_2 have the same MeasureComponents, which equal in names and types.

Alternatively, takes in input one Dataset and a Constant<String>, without other constraints.

Returns a third Dataset *ds_3* with the all the IdentifierComponents that are in *ds_1* or in *ds_2* (without duplicates) all the MeasureComponents (without duplicates) and AttributeComponents (according to the attribute propagation rules) containing for each pair of rows in *ds_1* and *ds_2* with the same values for all the common IdentifierComponents, a row with those values for the common IdentifierComponents and the ones of the most comprehensive Dataset for the other IdentifierComponents and the concatenation of the values in the corresponding MeasureComponents.

If ds*_2* (or *ds_1*) is a Constant<String>, the operator returns a second Dataset *ds_3* with all the IdentifierComponents in *ds_1* (*ds*_2) with the same values, all the MeasureComponents in *ds_1* (*ds_2*), whose values are the concatenation with the value of the Constant<String> *ds_2* (*ds_1*).

Parameters

ds_1 – a Dataset with at least one MeasureComponent<String> or a Constant<String>.

ds_2 - a Dataset with at least one MeasureComponent<String> or a Constant<String>.

Returns

ds_3 – a Dataset that is the concatenation two according to the described semantics.

Example

```
ds_r := ds_1 + " " + ds_2
```

ds_1		
K1	K2	M1
1	A	“hello”
2	B	“hi”

ds_2		
K1	K2	M1
1	A	“world”
2	B	“there”

ds_r		
K1	K2	M1
1	A	“hello world”
2	B	“hi there”

trim

Component version

Syntax

```
Component<String> c :=
trim
({component=}Component<String> c_1)
```

Constraints

None

Semantics

Trims whitespace from left and right of string.

Example

If A = “ Hello, world! “:

```
B := trim(A) // B = “Hello, world!”
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<String>+> ds_2
trim
({dataset=}Dataset<?+, MeasureComponent<String>+> ds_1)
```

Constrains

- *ds_1* only has MeasureComponents<String> (static).

Semantics

It takes in input a Dataset *ds_1* and returns another Dataset *ds_2* with the same IdentifierComponents , AttributeComponents (according to attribute propagation rules) and a MeasureComponent<String> for each MeasureComponent<String> in *ds_1* with the same name. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are obtained by eliminating leading and trailing whitespace from the String in the original MeasureComponent.

Parameters

ds_1 – the input Dataset.

Returns

ds_2 – the output Dataset

Example

```
r := trim(ds_1)
```

C1		
K1	K2	M1
1	A	“hello world”
2	B	“hi”
3	C	“help!”

R		
K1	K2	M1
1	A	“hello world”
2	B	“hi”
3	C	“help!”

upper/lower

Component version

Syntax

```
Component<String> c :=
[upper|lower]
({component=}Component<String> c_1)
```

Constraints

None

Semantics

Makes a string uppercase/ lowercase.

Example

If A = "Hello, World!":

```
B := upper(A) // B = "HELLO, WORLD!"
B := lower(A) // B = "hello, world!"
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<String>+> ds_2 :=
[upper|lower]
({dataset=}Dataset<?+, MeasureComponent<String>+> ds_1)
```

Constraints

- *ds_1* only has MeasureComponents<String> (static).

Semantics

It takes in input a Dataset *ds_1* and returns another Dataset *ds_2* with the same IdentifierComponents, AttributeComponents (according to attribute propagation rules) and a MeasureComponent<String> for each MeasureComponent<String> in *ds_1* with the same name. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are obtained by making uppercase (lowercase) the String in the original MeasureComponent. Non-alphabetic characters are simply copied.

Parameters

ds_1 – the input Dataset.

Returns

ds_2 - the output Dataset

Example

```
ds_r := upper(ds_1)
```

ds_1		
K1	K2	M1
1	A	“hello world”
2	B	“hi”
3	C	“help”

ds_r		
K1	K2	M1
1	A	“HELLO WORLD”
2	B	“HI”
3	C	“HELP”

substr

Component version

Syntax

```
Component<String> c :=
substr
({component=}Component<String> c_1,
{startPosition=}Constant<Integer> startPosition {,
{length=}Constant<Integer> length})
```

Constraints

0 ≤ startPosition < length(c_1); startPosition + length ≤ length(c_1) (dynamic)

Notice that if startPosition + length > length(c_1), then length is ignored.

Semantics

Returns the substring of c_1 starting from position startPosition and extracting length characters. 0 is the initial position of the Strings. If length is omitted, all the remaining characters in the String are considered.

Examples

Assuming that A = “Hello, world!“:

```
B := substr(A, 2) // B = “llo, world!”
B := substr(A, 2, 5) // B = “llo, ”
B := substr(A, 0, 4) // B = “Hell”
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<String>+> ds_2 :=
substr(
 {dataset=}Dataset<?+, MeasureComponent<String>+> ds_1,
 {startPosition=}Constant<Integer> startPosition,
 {, {length=} Constant<Integer> length }
)
```

Constraints

- *ds_1* only has MeasureComponents<String> (static).
- 0 ≤ startPosition < length(c_1); startPosition + length ≤ length(c_1) (dynamic)
  Notice that if startPosition + length > length(c_1), then length is ignored.

Semantics

It takes in input a Dataset *ds_1* and returns another Dataset *ds_2* with the same IdentifierComponents , AttributeComponents (according to attribute propagation rules) and a MeasureComponent<String> for each MeasureComponent<String> in *ds_1* with the same name.

For each row in *ds_1* there is a row in *ds_2* where the Strings in the MeasureComponents are obtained from the original ones in *ds_1* by taking the substring starting from position *startPosition* and extracting *length* characters. 0 is the initial position. If length is omitted, all the remaining characters in the String are considered. If *startPosition* exceeds the String length, NULL is returned. If *length* exceeds the String length, all the characters of the String from *startPosition* are returned.

Parameters

- *ds_1* – the input Dataset
- *startPosition* – the position to start extracting from. 0 is the beginning of the String
- *length* – the number of characters to extract from *startPosition*

Returns

ds_2 - the output Dataset

Example 1

```
ds_r := substr(ds_1,7)
```

ds_1		
K1	K2	M1
1	A	“hello world”

ds_r		
K1	K2	M1
1	A	“orld”

Example 2

```
ds_r := substr(ds_1,0,5)
```

ds_1		
K1	K2	M1
1	A	“hello world”

ds_r		
K1	K2	M1
1	A	“hello”

indexof

Component version

Syntax

```
Component<Numeric> c :=
indexof
({component=}Component<String> c_1, {string=}Constant<String>
stringToSearch)
```

Constraints

None

Semantics

It takes in input a *Component*<String> `c_1` and a *Constant*<String> `stringToSearch` and returns a *Component*<Numeric>, whose value is the starting position of `stringToSearch` in `c_1` (starting from 0).

If `stringToSearch` is not found, -1 is returned. If `stringToSearch` is empty or NULL, -1 is returned.

Example

If A = "Hello, world!":

```
B := indexof(A, "Hello") // B = 0
B := indexof(A, "world") // B = 7
B := indexof(A, "cat") // B = -1
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Numeric>+> ds_2 :=
indexof
({dataset=}Dataset<?+, MeasureComponent<String>+> ds_1,
 {string=}Constant<String> stringToSearch)
```

Constraints

- `ds_1` only has *MeasureComponent*s<String> (static).

Semantics

It takes in input a *Dataset* `ds_1` and a *Constant*<String> `stringToSearch` and returns another *Dataset* `ds_2` with the same *IdentifierComponent*s, *AttributeComponent*s (according to attribute propagation rules) and a *MeasureComponent*<Numeric> for each *MeasureComponent*<String> in `ds_1` with the same name.

For each row in `ds_1` there is a row in `ds_2` where the numeric values in the *MeasureComponent*s are the starting position of `stringToSearch` in the respective *MeasureComponent*<String> in `ds_1` (starting from 0).

If `stringToSearch` is not found, -1 is returned. If `stringToSearch` is empty or NULL, -1 is returned.

Parameters

- `ds_1` – the input *Dataset*.
- `stringToSearch` – the *Constant*<String> to search.

Returns

`ds_2` - the output *Dataset*

Example

```
ds_2 := indexof(ds_1,”hello”)
```

ds_1		
K1	K2	M1
1	A	“hello world”
2	A	“say hello”
3	A	“he”
4	A	“hi, hello!”

ds_2		
K1	K2	M1
1	A	0
2	A	4
3	A	-1
4	A	4

Mathematical operators

unary plus +

Component version

Syntax

```
Component<Numeric> c :=
+
Component<Numeric> c_1
```

Constraints

None

Semantics

No effect. Returns c_1 without altering its sign.

Example

```
A := +B     //if B = 5, then A = 5
```

Dataset version

Syntax

```
ds_o :=
+
Dataset<?+, MeasureComponent<Numeric>+> ds
```

Constraints

None

Semantics

Returns the Dataset *ds* without changing any sign.

Parameters

ds – a Dataset with at least a Numeric MeasureComponent.

Returns

ds_o – A copy of *ds* with no alterations.

Example

```
ds_2 := +ds_1
```

ds_1		
K1	K2	M1
1	A	11
2	B	14
3	C	9

ds_2		
K1	K2	M1
1	A	11
2	B	14
3	C	9

unary minus -

Component version

Syntax

```
Component<Numeric> c :=
-
Component<Numeric> c_1
```

Constraints

None

Semantics

Returns c changing its sign.

Examples

```
A := -B //if B = 5, then A = -5
C := -A //A = 5
```

Dataset version

Syntax

```
ds_o :=
-
Dataset<?+, MeasureComponent<Numeric>+> ds
```

Constraints

None

Semantics

Returns the Dataset *ds* where the signs of all the MeasureComponent<Numeric> are inverted. Differently typed MeasureComponents (MeasureComponent<String>, MeasureComponent<Boolean>) are ignored and kept unchanged in the result.

Parameters

ds – a Dataset expression with at least a Numeric MeasureComponent.

Returns

ds_o – A copy of *ds* where the signs of all the numeric Measure Components are inverted.

Example 1

```
ds_2 := -ds_1
```

ds_1		
K1	K2	M1
1	A	11
2	B	14
3	C	9

ds_2		
K1	K2	M1
1	A	-11
2	B	-14
3	C	-9

Example 2

```
ds_2 := -ds_1
```

ds_1				
K1	K2	M1	M2	M3
1	A	11	12	“A”
2	B	14	13	“B”
3	C	9	14	“C”

ds_2				
K1	K2	M1	M2	M3
1	A	-11	-12	“A”
2	B	-14	-13	“B”
3	C	-9	-14	“C”

addition and subtraction + ++ - --

Component version

Syntax

```
Component<Numeric> c := Component<Numeric> c_1
[+|-]
Component<Numeric> c_2
```

Constraints

None

Semantics

Calculates the algebraic sum, of c_1 and c_2, or c_1 and –c_2.

Example

For A = 5 and B = 4:

```
C := A + B // C = 9
C := A - B // C = 1
```

Dataset version (implicit and explicit zero)

Syntax

Explicit zero:

```
Dataset<?+, MeasureComponent<Numeric>+> ds_3 :=
[Dataset<?+, MeasureComponent<Numeric>+>|Constant<Numeric>] ds_1
[++|--]
[Dataset<?+, MeasureComponent<Numeric>+>|Constant<Numeric>] ds_2
```

Implicit zero:

```
Dataset<?+, MeasureComponent<Numeric>+> ds_3 :=
[Dataset<?+, MeasureComponent<Numeric>+>|Constant<Numeric>] ds_1
[+|-]
```

```
[Dataset<?+, MeasureComponent<Numeric>+>|Constant<Numeric>] ds_2
```

Constraints

- At least one between *ds_1* and *ds_2* must be a Dataset (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same IdentifierComponents<?>, which must be equal in name and type or, at least, the IdentifierComponents of one must be a subset of the IdentifierComponents of the other (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents (static).
- If *ds_1* is a Dataset ,it must have only numeric MeasureComponents (static).
- If *ds_2* is a Dataset, it must have only numeric MeasureComponents (static).

Semantics

Takes in input two Datasets *ds_1* and *ds_2,* with at least one Numeric MeasureComponent, and such that the IdentifierComponents of *ds_1* are a subset of the IdentifierComponents of *ds_2* or vice-versa and *ds_1* and *ds_*2 have the same MeasureComponents, which equal in names and types.

Alternatively, takes in input one Dataset and a Constant<Numeric>, without other constraints.

Explicit zero

```
[++|--]
```

Returns a third Dataset *ds_3* with the all the IdentifierComponents that are in *ds_1* or in *ds_2* (without duplicates) all the MeasureComponents (without duplicates) and AttributeComponents (according to the attribute propagation rules) containing for each pair of rows in *ds_1* and *ds_2* with the same values for all the common IdentifierComponents, a row with those values for the common IdentifierComponents and the ones of the most comprehensive Dataset for the other IdentifierComponents, the algebraic sum (difference) of the values in the corresponding MeasureComponents.

Explicit zero → implicit missing

As this version of the operator returns a row in *ds_3* only for each pair of matching rows in *ds_1* and *ds_2*, rows that are present only in one between *ds_1* and *ds_2* are assumed to be implicitly missing in the other (since the zero is represented explicitly) and then, not produced in the output.

Implicit zero

```
[+|-]
```

Returns a third Dataset *ds_3* with the all the common IdentifierComponents that are in *ds_1* or in *ds_2*, all the MeasureComponents (one for each homonym pair and with that name and type) and AttributeComponents (according to the attribute propagation rules) containing for each row in *ds_1* and *ds_2* a row with those values for the common IdentifierComponents and the ones of the most comprehensive Dataset for the other IdentifierComponents and the algebraic sum (difference) of the values in the corresponding MeasureComponents. Moreover *ds_3* contains all the rows in *ds_1* without corresponding ones in *ds_2* and viceversa.

Implicit zero → Explicit missing

As this version of the operator returns a row in *ds_3* for each row in *ds_1* or *ds_2,* independently of any matching of the IdentifierComponents, rows that are present only in one between *ds_1* and *ds_2* are assumed to be present in the other with value 0 for all the MeasureComponents. Therefore, missing values should be explicitly represented.

In both explicit and implicit cases, if ds_*2* (or *ds_1*) is a Constant<Numeric>, the operator returns a second Dataset *ds_3* with all the IdentifierComponents in *ds_1* (*ds*_2) with the same values, all the MeasureComponents in *ds_1* (*ds_2*), whose values are the algebraic sum (difference) with the value of the Constant<Numeric> *ds_2* (*ds_1*).

Parameters

ds_1 – a Dataset with at least one MeasureComponent<Numeric> or a Constant<Numeric>

ds_2 – a Dataset with at least one MeasureComponent<Numeric> or a Constant<Numeric>

Returns

ds_3 – a Dataset that is the algebraic sum of the two according to the described semantics.

Example

In this example, we calculate the total population of a set of countries given two Datasets: one of the male population, and another, of the female population. They contain one measure each. Thus, the result will contain a single measure with the results of the addition.

ds_1

TIME	GEO	POPULATION
2013	Belgium	5
2013	Denmark	2
2013	France	3
2013	Spain	4

ds_2

TIME	GEO	AGE	POPULATION
2013	Belgium	Total	10
2013	Greece	Total	11
2013	Belgium	Y15-24	3
2013	Greece	Y15-24	2
2013	Spain	Y15-24	6

The result of the expression `ds_3 := ds_1 + ds_2` (an implicit zero addition) will be:

ds_3			
TIME	GEO	AGE	POPULATION
2013	Belgium	Total	15
2013	Belgium	Y15-24	8
2013	Denmark	NULL	2
2013	France	NULL	3
2013	Greece	Total	11
2013	Greece	Y15-24	2
2013	Spain	Y15-24	10

The result of the expression `ds_3 := ds_1 ++ ds_2` (explicit zero addition) will be:

ds_3			
TIME	GEO	AGE	POPULATION
2013	Belgium	Total	15
2013	Belgium	Y15-24	8
2013	Spain	Y15-24	10

Multiplication and division `* ** / //`

Component version

Syntax

```
Component<Numeric> c := Component<Numeric> c_1
[*|/]
Component<Numeric> c_2
```

Constraints

For division (/) c_2 <> 0 (dynamic).

Semantics

Calculates the product of c_1 and c_2.

Calculates the ratio between c_1 and c_2. The case of c_2 = 0, should result into a runtime exception .

Example

For A = 5 and B = 4:

```
C := A * B // C = 20
C := A / B // C = 1.25
```

Dataset version (implicit zero and explicit zero)

Syntax

Explicit zero:

```
Dataset<?+, MeasureComponent<Numeric>+> ds_3 :=
[Dataset<?+, MeasureComponent<Numeric>+>|Constant<Numeric>] ds_1
[**|//]
[Dataset<?+, MeasureComponent<Numeric>+>|Constant<Numeric>] ds_2
```

Implicit zero:

```
Dataset<?+, MeasureComponent<Numeric>+> ds_3 :=
[Dataset<?+, MeasureComponent<Numeric>+>|Constant<Numeric>] ds_1
[*|/]
[Dataset<?+, MeasureComponent<Numeric>+>|Constant<Numeric>] ds_2
```

Constraints

- At least one between *ds_1* and *ds_2* must be a Dataset (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same IdentifierComponents<?>, which must be equal in name and type or, at least, the IdentifierComponents of one must be a subset of the IdentifierComponents of the other (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents (static).
- If *ds_1* is a Dataset ,it must have only numeric MeasureComponents (static).
- If *ds_2* is a Dataset, it must have only numeric MeasureComponents (static).
- If *ds_2* is a Constant<Numeric>, it cannot be zero (dynamic).

Semantics

Takes in input two Datasets *ds_1* and *ds_2,* with at least one Numeric MeasureComponent, and such that the IdentifierComponents of *ds_1* are a subset of the IdentifierComponents of *ds_2* or viceversa and *ds_1* and *ds*_2 have the same MeasureComponents, which equal in names and types.

Alternatively, takes in input one Dataset and a Constant<Numeric>, without other constraints.

Explicit zero

[**|//]

Returns a third Dataset *ds_3* with the all the IdentifierComponents that are in *ds_1* or in *ds_2* (without duplicates) all the MeasureComponents (without duplicates) and AttributeComponents (according to the attribute propagation rules) containing for each pair of rows in *ds_1* and *ds_2* with the same values for all the common IdentifierComponents, a row with those values for the common IdentifierComponents and the ones of the most comprehensive Dataset for the other IdentifierComponents and the product (division) of the values in the corresponding MeasureComponents.

Explicit zero → implicit missing

As this version of the operator returns a row in *ds_3* only for each pair of matching rows in *ds_1* and *ds_2*, rows that are present only in one between *ds_1* and *ds_2* are assumed to be implicitly missing in the other (since the zero is represented explicitly) and then, not produced in the output.

Implicit zero

[*|/]

Returns a third Dataset *ds_3* with the all the common IdentifierComponents that are in *ds_1* or in *ds_2*, all the MeasureComponents (one for each homonym pair and with that name and type) and AttributeComponents (according to the attribute propagation rules) containing for each row in *ds_1* and *ds_2* a row with those values for the common IdentifierComponents and the ones of the most comprehensive Dataset for the other IdentifierComponents and the product (division) of the values in the corresponding MeasureComponents. Moreover *ds_3* contains all the rows in *ds_1* without corresponding ones in *ds_2* and viceversa.

Implicit zero → Explicit missing

As this version of the operator returns a row in *ds_3* for each row in *ds_1* or *ds_2,* independently of any matching of the IdentifierComponents, rows that are present only in one between *ds_1* and *ds_2* are assumed to be present in the other with value 0 for all the MeasureComponents. Therefore, missing values should be explicitly represented.

In both explicit and implicit cases, if ds*_2* (or *ds_1*) is a Constant<Numeric>, the operator returns a second Dataset *ds_3* with all the IdentifierComponents in *ds_1* (*ds*_2) with the same values, all the MeasureComponents in *ds_1* (*ds_2*), whose values are the product (division) involving the value of the Constant<Numeric> *ds_2* (*ds_1*) as numerator or denominator.

Parameters

ds_1 – a Dataset with at least one MeasureComponent<Numeric> or a Constant<Numeric>.

ds_2 – a Dataset with at least one MeasureComponent<Numeric> or a Constant<Numeric>.

Returns

ds_3 – a Dataset that is the sum (division) of the two according to the described semantics.

Example

total_population				
TIME	GEO	AGE	POPULATION	UNEMPLOYMENT_RATE
2012	Belgium	Total	100	7.6
2012	Greece	Total	10	24.3
2012	Spain	Total	20	25
2012	Belgium	Y15-24	30	3.6
2012	Greece	Y15-24	5	18.3
2012	Switzerland	Y15-24	2	20

Overcrowding_rate_urbanization		
TIME	GEO	PERCENTAGE
2012	Belgium	0.01
2012	Greece	0.1
2012	Spain	0.2
2012	Malta	0.3
2012	Finland	0.4
2012	France	0.5

The result of the expression

```
total_population[rename POPULATION as PERCENTAGE]#PERCENTAGE *
Overcrowding_rate_urbanization#PERCENTAGE
```

is the following dataset:

TIME	GEO	AGE	PERCENTAGE
2012	Belgium	Total	1
2012	Greece	Total	1
2012	Spain	Total	4
2012	Belgium	Y15-24	0.3
2012	Greece	Y15-24	0.5

round

Component version

Syntax

```
Component<Numeric> c :=
round
({component=}Component<Numeric> c_1, {decimals=}Constant<Integer>
decimals)
```

Constraints

None

Semantics

Rounds c_1 to the number of decimal digits specified by decimals.

Examples

Let us assume P = 3.14159

```
A := round(P, 2) // A = 3.14
B := round(P, 4) // B = 3.1416
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Numeric>+> ds_2 :=
round
({dataset=}Dataset<?+, MeasureComponent<Numeric>+> ds_1,
{decimals=}Constant<Integer> decimals)
```

Constraints

- *ds_1* only has Numeric MeasureComponents (static).
- decimals >0 (static).

Semantics

It takes in input a Dataset *ds_1* and returns another Dataset *ds_2* with the same IdentifierComponents , AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are rounded to *decimals* decimal positions.

Parameters

ds_1 – the Dataset to round

decimals – the decimal positions to round to

Returns

ds_2 – the rounded Dataset

Example

Let us start from the the dataset:

unemployment					
AGE	TIME	GEO	SEX	YOUTH_UNEMPLOYMENT	UNEMPLOYMENT
From 20 to 29 years	2011	Germany	Total	7.5	5.9
From 20 to 29 years	2012	Germany	Total	7.1	5.5
From 20 to 29 years	2011	Greece	Total	33.7	17.7
From 20 to 29 years	2012	Greece	Total	42.5	24.3

The result of the expression:

`ds_1 := round(unemployment,0)` or `round(dataset=unemployment,decimals=0)`

will be:

ds_1					
AGE	TIME	GEO	SEX	YOUTH_UNEMPLOYMENT	UNEMPLOYMENT
From 20 to 29 years	2011	Germany	Total	8	6
From 20 to 29 years	2012	Germany	Total	7	6
From 20 to 29 years	2011	Greece	Total	34	18
From 20 to 29 years	2012	Greece	Total	43	24

As explained, the operator has been applied to all the MeasureComponents.

Instead, the result of the expression:

```
ds_1 := round(unemployment#YOUTH_UNEMPLOYMENT, 0)
```

will be:

AGE	TIME	GEO	SEX	YOUTH_UNEMPLOYMENT	UNEMPLOYMENT
From 20 to 29 years	2011	Germany	Total	8	5.9
From 20 to 29 years	2012	Germany	Total	7	5.5
From 20 to 29 years	2011	Greece	Total	34	17.7
From 20 to 29 years	2012	Greece	Total	43	24.3

abs

Component version

Syntax

```
Component<Numeric> c :=
abs
({component=}Component<Numeric> c_1)
```

Constraints

None

Semantics

Returns the absolute value of *c_1*.

Examples

Let us assume A = -5:

```
B := abs(A) // B = 5
C := abs(B) // C = 5
```

Dataset version

Syntax

```
Dataset<?+, Component<Numeric>+> ds_1 :=
abs
({dataset=}Dataset<?+, Component<Numeric>+> ds_2)
```

Constraints

- *ds_1* only has Numeric MeasureComponents (static).

Semantics

It takes in input a Dataset *ds_1* and returns another Dataset *ds_2* with the same IdentifierComponents, AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents contain the absolute values of the corresponding ones in *ds_1*.

Parameters

ds_1 – the input Dataset

Returns

ds_2 – the output Dataset

Example

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	0.484183
FR	Females	2011	-0.515817
FR	Total	2011	-1.000000

The following expression

```
DatasetB := abs(DatasetA)
```

applied on DatasetA, will return the result:

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	0.484183
FR	Females	2011	0.515817
FR	Total	2011	1.000000

trunc

Component version

Syntax

```
Component<Numeric> c :=
trunc
({component=}Component<Numeric>  c_1,  {decimals=}Constant<Integer>
decimals)
```

Constraints

None

Semantics

Discards the decimal digits of *c_1*, beyond the number of digits specified by decimals.

Examples

Let us assume P = 3.14159

```
A := trunc(P, 2) // A = 3.14
B := trunc(P, 4) // B = 3.1415
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Numeric>+> ds_2 :=
trunc
({dataset=}Dataset<?+, MeasureComponent<Numeric>+> ds_1,
{decimals=}Constant<Integer> decimals)
```

Constraints

- *ds_1* only has Numeric MeasureComponents (static).
- decimals >0 (dynamic).

Semantics

It takes in input a Dataset *ds_1* and returns another Dataset *ds_2* with the same IdentifierComponents , AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are obtained by discarding the decimal digits after the *decimals* position.

Parameters

ds_1 – the input Dataset

Returns

ds_2 – the output Dataset

Example

DatasetA			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	0.484183
FR	Females	2011	0.515817
FR	Total	2011	1.000000

The result of the expression:

```
ds_1 := trunc(DatasetA, 2)
```

will be:

COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	0.48
FR	Females	2011	0.51
FR	Total	2011	1.00

exp

Component version

Syntax

```
Component<Numeric> c :=
exp
({component=}Component<Numeric> c_1)
```

Constraints

None

Semantics

Returns *e* (Napier's – or Euler's – constant) raised to *c_1*.

Examples

If B = 5:

```
A := exp(B) // A = 148.413
```

If B = -1:

```
A := exp(B) // A = 0.368
```

Dataset version

Syntax

```
Dataset<?+, Component<Numeric>+> ds_2 :=
exp
({dataset=}Dataset<?+, Component<Numeric>+> ds_1)
```

Constraints

- *ds_1* only has Numeric MeasureComponents (static).

Semantics

It takes in input a Dataset *ds_1* and returns another Dataset *ds_2* with the same IdentifierComponents , AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are obtained by elevating *e* (Nepero's number) to the value in the original MeasureComponent.

Parameters

ds_1 – the input Dataset

Returns

ds_2 – the output Dataset

Example

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	5
FR	Females	2011	8
FR	Total	2011	2

The following expression

```
DatasetB := exp(DatasetA)
```

applied on DatasetA, will return the result:

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	148.41
FR	Females	2011	2980.95
FR	Total	2011	7.389

ln

Component version

Syntax

```
Component<Numeric> c :=
ln
({component=}Component<Numeric> c_1)
```

Constraints

c_1 > 0 (dynamic).

Semantics

Returns the natural logarithm (base *e*) of c_1.

Examples

If B = 1:

```
A := ln(B) // A = 0
```

If B = 148:

```
A := ln(B) // A = 4.997
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Numeric>+> ds_2 :=
ln({dataset=}Dataset<?+, MeasureComponent<Numeric>+> ds_1)
```

Constraints

- *ds_1* only has Numeric MeasureComponents (static).

Semantics

It takes in input a Dataset *ds_1* and returns another Dataset *ds_2* with the same IdentifierComponents, AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are obtained by calculating the natural logarithm (in base *e,* Nepero's number) of the value in the original MeasureComponent.

Parameters

ds_1 – the input Dataset

Returns

ds_2 – the output Dataset

Example

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	148.41
FR	Females	2011	2980.95
FR	Total	2011	7.389

The following expression

```
DatasetB := ln(DatasetA)
```

applied to DatasetA, will return the result:

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	5
FR	Females	2011	8
FR	Total	2011	2

log

Component version

Syntax

```
Component<Numeric> c  :=
log
({component=}Component<Numeric> c_1, {base=}Constant<Integer> base)
```

Constraints

c_1 > 0 (dynamic).

base > 0 (static).

Semantics

Returns the base b logarithm of c_1.

Examples

If B = 1024:

```
A := log(B, 2) // A = 10
```

```
A := log(B, 10) // A = 3.01
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Numeric>+> ds_2 :=
log({dataset=}Dataset<?+, MeasureComponent<Numeric>+> ds_1,
{base=}Constant<Numeric> base)
```

Constraints

- *ds_1* only has Numeric MeasureComponents (static).
- base > 0 (static)
- *ds_1* Measure values > 0 (dynamic)

Semantics

It takes in input a Dataset *ds_1* and a Numeric Constant and returns another Dataset *ds_2* with the same IdentifierComponents, AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are obtained by calculating the logarithm in base *base* of the value in the original MeasureComponent.

Parameters

ds_1 – the input Dataset

base – the logarithm base

Returns

ds_2 – the output Dataset

Example

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	1024
FR	Females	2011	64
FR	Total	2011	32

The following expression

```
DatasetB := log(2,DatasetA)
```

applied to DatasetA, will return the result:

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	10
FR	Females	2011	6
FR	Total	2011	5

power

Component version

Syntax

```
Component<Numeric> c :=
power
({component=}Component<Numeric> base, {exponent=}Component<Integer>
exponent)
```

Constraints

None

Semantics

Returns base raised to the exponent power.

Example

If A = 2, B = 5:

```
C := power(B, A) // C = 25
```

Dataset version

Syntax

```
ds_2 :=
power({dataset=}Dataset<?+, MeasureComponent<Numeric>+> ds_1,
{exponent=}Constant<Numeric> exponent)
```

Constraints

- *ds_1* only has Numeric MeasureComponents (static).

Semantics

It takes in input a Dataset *ds_1* and a Numeric Constant and returns another Dataset *ds_2* with the same IdentifierComponents, AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are obtained by elevating the original MeasureComponent to the exponent-th power.

Parameters

ds_1 – the input Dataset

exponent – the power exponent

Returns

ds_2 – the output Dataset

Example

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	3
FR	Females	2011	4
FR	Total	2011	5

The following expression

```
DatasetB := power(DatasetA,2)
```

applied to DatasetA, will return the result:

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	9
FR	Females	2011	16
FR	Total	2011	25

nroot

Component version

Syntax

```
Component<Numeric> c :=
nroot
({component=}Component<Numeric>   c_1,   {index=}Component<Integer>
index)
```

Constraints

c_1 ≥ 0 when index even (dynamic).

Semantics

Returns the n-th root of c_1

Example

If A = 2, B = 25:

```
C := nroot(B, A) // C = 5
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Numeric>+> ds_2 :=
nroot
({dataset=}Dataset<?+, MeasureComponent<Numeric>+> ds_1,
{index=}Constant<Numeric> n)
```

Constraints

- *ds_1* only has Numeric MeasureComponents (static).
- *ds_1* Measure value is positive when n is even (dynamic).

Semantics

It takes in input a Dataset *ds_1* and a Numeric Constant and returns another Dataset *ds_2* with the same IdentifierComponents, AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* where

the MeasureComponents are obtained by calculating the n-th root of the original MeasureComponent.

Parameters

ds_1 – the input Dataset

n – the index of the root

Returns

ds_2 – the output Dataset

Example

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	8
FR	Females	2011	27
FR	Total	2011	64

The following expression

```
DatasetB := nroot(DatasetA,3)
```

applied to DatasetA, will return the result:

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	2
FR	Females	2011	3
FR	Total	2011	4

mod

Component version

Syntax

```
Component<Integer> c :=
```

```
mod
({component=}Component<Integer>   c_1,   {modulo=}Component<Integer>
c_2)
```

Constraints

None

Semantics

Returns the remainder of the division of c_1 by c_2.

Example

If A = 5, B = 2:

```
C := mod(A, B) // C = 1
```

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Integer>+> ds_2 :=
mod
({dataset=}Dataset<?+, MeasureComponent<Integer>+> ds_1,
{modulo=}Constant<Integer> d)
```

Constraints

- *ds_1* only has Numeric MeasureComponents (static).
- d >0 (static).

Semantics

It takes in input a Dataset *ds_1* and a Numeric Constant *d* and returns another Dataset *ds_2* with the same IdentifierComponents, AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents are obtained by calculating the remainder of the division of the original MeasureComponent by *d*.

Parameters

ds_1 – the input Dataset

d – the divisor

Returns

ds_2 – the output Dataset

Example

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	7
FR	Females	2011	10
FR	Total	2011	12

The following expression

```
DatasetB := mod(DatasetA,3)
```

applied on DatasetA, will return the result:

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	1
FR	Females	2011	1
FR	Total	2011	0

Boolean operators

VTL provides the three basic Boolean operators that allow calculations on Boolean Components.

and

Component version

Syntax

```
Component<Boolean> c :=
Component<Boolean> c_1
and
Component<Boolean> c_2
```

Constraints

None

Semantics

Returns the logical and between *c_1* and *c_2*.

Dataset version

Syntax

```
Dataset<?+,MeasureComponent<Boolean>+> ds_3 :=
[Dataset<?+,MeasureComponent<Boolean>+>|Constant<Boolean>] ds_1
and
[Dataset<?+,MeasureComponent<Boolean>+>|Constant<Boolean>] ds_2
```

Constraints

- at least one between *ds_1* and *ds_2* must be a Dataset (static).
- if *ds_1* (*ds_2*) is a Dataset, it must have only Boolean MeasureComponents (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same IdentifierComponents in name and type or, at least, the IdentifierComponents of *ds_1* must be a subset of the ones of *ds_2* (or viceversa) (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents (static).
- In case *ds_1* (*ds_2*) is a Constant<Boolean>, *ds_2* (*ds_1*) must have only one MeasureComponent<Boolean> (static).

Semantics

The operator performs a logical conjunction on the operand Datasets or constants.

It takes in input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents, or at least the identifier components of one being a subset of the IdentifierComponents of the other, and

the same MeasureComponents (in name and with Boolean type). It returns a third Dataset *ds_3*, having the superset of the IdentifierComponents of *ds_1* and *ds_2*, some AttributeComponents (according to the attribute propagation rules) and, for each pair of homonym MeasureComponents of *ds_1* and *ds_2* a MeasureComponent<Boolean> with the same name.

For each pair of rows in *ds_1* and *ds_2* having the same values for all the common IdentifierComponents, the MeasureComponent<Boolean> in *ds_*3 will contain the result of the logical *and* of the original values.

If *ds_1* (or *ds_2*) is a Constant, *ds_3* has a single MeasureComponent<Boolean>, named as the MeasureComponent<Boolean> of *ds_2* (*ds_1*), containing the logical *and* of the original value with the Constant.

Parameters

ds_1 – the first operand, Dataset or Constant of the logical operator.

ds_2 – the second operand, Dataset or Constant of the logical operator.

Returns

ds_3 – the Dataset resulting from the application of the logical operator.

Example 1

In this example we examine whether there are rows for a particular sex and age group.

population				
SEX	AGE	GEO	TIME	VALUE
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

The expression

```
population#sex="M" and population#age_group="Y15-64"
```

will yield:

SEX	AGE	GEO	TIME	CONDITION[1]
M	Y_LT15	BE	2013	false
M	Y15-64	BE	2013	true
M	Y_GE65	BE	2013	false
F	Y_LT15	BE	2013	false
F	Y15-64	BE	2013	false
F	Y_GE65	BE	2013	false
M	Y_LT15	UK	2013	false
M	Y15-64	UK	2013	true
M	Y_GE65	UK	2013	false
F	Y_LT15	UK	2013	false
F	Y15-64	UK	2013	false
F	Y_GE65	UK	2013	false

Each of the expressions `population#sex="M"` and `population#age_group="Y15-64"` specifies implicitly a Boolean measure that shows whether the condition stated in the expression holds. The and operator creates the conjunction of these two measures.

or

Component version

Syntax

```
Component<Boolean> c :=
Component<Boolean> c_1
or
Component<Boolean> c_2
```

Constraints

None

[1] This column is named CONDITION, since it is essentially a result of a Boolean operation on two comparison operations.

Semantics

Returns the logical or between *c_1* and *c_2*.

Dataset version

Syntax

```
Dataset<?+,MeasureComponent<Boolean>+> ds_3 :=
[Dataset<?+,MeasureComponent<Boolean>+>|Constant<Boolean>] ds_1
or
[Dataset<?+,MeasureComponent<Boolean>+>|Constant<Boolean>] ds_2
```

Constraints

- at least one between *ds_1* and *ds_2* must be a Dataset (static).
- if *ds_1* (*ds_2*) is a Dataset, it must have only Boolean MeasureComponents (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same IdentifierComponents in name and type or, at least, the IdentifierComponents of *ds_1* must be a subset of the ones of *ds_2* (or vice-versa) (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents (static).
- In case *ds_1* (*ds_2*) is a Constant<Boolean>, *ds_2* (*ds_1*) must have only one MeasureComponent<Boolean> (static).

Semantics

The operator performs a logical disjunction on the operand Datasets or constants.

It takes in input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents, or at least the identifier components of one being a subset of the IdentifierComponents of the other, and the same MeasureComponents (in name and with Boolean type). It returns a third Dataset *ds_3*, having the superset of the IdentifierComponents of *ds_1* and *ds_2*, some AttributeComponents (according to the attribute propagation rules) and, for each pair of homonym MeasureComponents of *ds_1* and *ds_2* a MeasureComponent<Boolean> with the same name.

For each pair of rows in *ds_1* and *ds_2* having the same values for all the common IdentifierComponents, the MeasureComponent<Boolean> in *ds*_3 will contain the result of the logical *or* of the original values.

If *ds_1* (or *ds_2*) is a Constant, *ds_3* has a single MeasureComponent<Boolean>, named as the MeasureComponent<Boolean> of *ds_2* (*ds_1*), containing the logical *or* of the original value with the Constant.

Parameters

ds_1 – the first operand, Dataset or Constant of the logical operator.

ds_2 – the second operand, Dataset or Constant of the logical operator.

Returns

ds_3 – the Dataset resulting from the application of the logical operator.

Example

In this example we examine whether there are rows for a particular sex or age group.

Consider the following Dataset of population.

population				
SEX	AGE	GEO	TIME	VALUE
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

The expression

```
population#sex="M" OR  population#age_group="Y15-64"
```

will yield:

SEX	AGE	GEO	TIME	VALUE
M	Y_LT15	BE	2013	true
M	Y15-64	BE	2013	true
M	Y_GE65	BE	2013	true
F	Y_LT15	BE	2013	false
F	Y15-64	BE	2013	true
F	Y_GE65	BE	2013	false
M	Y_LT15	UK	2013	true
M	Y15-64	UK	2013	true
M	Y_GE65	UK	2013	true
F	Y_LT15	UK	2013	false
F	Y15-64	UK	2013	true
F	Y_GE65	UK	2013	false

xor

Component version

Syntax

```
Component<Boolean> c :=
Component<Boolean> c_1
xor
Component<Boolean> c_2
```

Constraints

None

Semantics

Returns the logical xor between *c_1* and *c_2*.

Dataset version

Syntax

```
Dataset<?+,MeasureComponent<Boolean>+> ds_3 :=
[Dataset<?+,MeasureComponent<Boolean>+>|Constant<Boolean>] ds_1
xor
[Dataset<?+,MeasureComponent<Boolean>+>|Constant<Boolean>] ds_2
```

Constraints

- at least one between *ds_1* and *ds_2* must be a Dataset (static).
- if *ds_1* (*ds_2*) is a Dataset, it must have only Boolean MeasureComponents (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same IdentifierComponents in name and type or, at least, the IdentifierComponents of *ds_1* must be a subset of the ones of *ds_2* (or vice-versa) (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents (static).
- In case *ds_1* (*ds_2*) is a Constant<Boolean>, *ds_2* (*ds_1*) must have only one MeasureComponent<Boolean> (static).

Semantics

The operator performs a logical exclusive disjunction on the operand Datasets or constants.

It takes in input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents, or at least the identifier components of one being a subset of the IdentifierComponents of the other, and the same MeasureComponents (in name and with Boolean type).

It returns a third Dataset *ds_3*, having the superset of the IdentifierComponents of *ds_1* and *ds_2*, some AttributeComponents (according to the attribute propagation rules) and, for each pair of homonym MeasureComponents of *ds_1* and *ds_2* a MeasureComponent<Boolean> with the same name.

For each pair of rows in *ds_1* and *ds_2* having the same values for all the common IdentifierComponents, the MeasureComponent<Boolean> in *ds_*3 will contain the result of the logical *xor* of the original values.

If *ds_1* (or *ds_2*) is a Constant, *ds_3* has a single MeasureComponent<Boolean>, named as the MeasureComponent<Boolean> of *ds_2* (*ds_1*), containing the logical *xor* of the original value with the Constant.

Parameters

ds_1 – the first operand, Dataset or Constant of the logical operator.

ds_2 – the second operand, Dataset or Constant of the logical operator.

Returns

ds_3 – the Dataset resulting from the application of the logical operator.

Example

In this example we examine whether there are rows that correspond to only one of a particular sex and a particular age group.

Consider the following Dataset of population.

population				
SEX	AGE	GEO	TIME	VALUE
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

The expression

```
population#sex="M" xor  population#age_group="Y15-64"
```

will yield:

SEX	AGE	GEO	TIME	VALUE
M	Y_LT15	BE	2013	true
M	Y15-64	BE	2013	false
M	Y_GE65	BE	2013	true
F	Y_LT15	BE	2013	false
F	Y15-64	BE	2013	true
F	Y_GE65	BE	2013	false
M	Y_LT15	UK	2013	true
M	Y15-64	UK	2013	false
M	Y_GE65	UK	2013	true
F	Y_LT15	UK	2013	false
F	Y15-64	UK	2013	true
F	Y_GE65	UK	2013	false

not

Component version

Syntax

```
Component<Boolean> c :=
not
Component<Boolean> c_1
```

Constraints

None

Semantics

Returns the logical negation of *c_1*.

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Boolean>+> ds_2 :=
not
Dataset<?+, MeasureComponent<Boolean>+> ds_1
```

Constraints

- if *ds_1* must have only Boolean MeasureComponents (static).

Semantics

Takes in input a Dataset *ds_1* and returns a new Dataset *ds_2* with the same IdentifierComponents, the AttributeComponents (according to attribute propagation rules) and the same MeasureComponents. For each row in *ds_1* there is a row in *ds_2* with the logical negation (*not*) of the original Boolean value.

Parameters

ds_1 – the input Dataset .

Returns

ds_2 – the output Dataset.

Example

In this example we examine whether there are rows that do not correspond to a particular sex.

Consider the following Dataset of population.

population				
SEX	AGE	GEO	TIME	VALUE
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

The expression

```
not population#sex="M"
```

1790 will yield:

1791

SEX	AGE	GEO	TIME	VALUE
M	Y_LT15	BE	2013	false
M	Y15-64	BE	2013	false
M	Y_GE65	BE	2013	false
F	Y_LT15	BE	2013	true
F	Y15-64	BE	2013	true
F	Y_GE65	BE	2013	true
M	Y_LT15	UK	2013	false
M	Y15-64	UK	2013	false
M	Y_GE65	UK	2013	false
F	Y_LT15	UK	2013	true
F	Y15-64	UK	2013	true
F	Y_GE65	UK	2013	true

union (records)

Syntax

```
Dataset<?> ds_2 :=
union
(Dataset<?> ds_1 {, Dataset<?> ds_1}*)
```

Constraints

- all the Datasets *ds_1* must have the same IdentifierComponents and MeasureComponents, in name and type (static).

Semantics

Performs row-wise union. Takes in input one or more Datasets *ds_1* with the same IdentifierComponents and MeasureComponents and returns a new Dataset *ds_2* with the same IdentifierComponents and MeasureComponents and AttributeComponents (according to attribute propagation rules).

The result contains all the rows from every *ds_1* without duplicates, which is their union. Therefore, in the output there cannot be two rows with the same values for all the IdentifierComponents and MeasureComponents. If only one argument is given, it is returned unchanged.

It is worth underlying that, on the other hand, it is possible to have in *ds_2* two rows with the same values for the IdentifierComponents. However, if this happens, this will result in a runtime error if *ds_2* is actually assigned in an expression as Datasets with duplicates are only admitted in will not be allowed to be stored into the right-hand side.

Parameters

`ds_1` – all the input Datasets to consider in the union.

Returns

`ds_2` – the output Dataset

Example 1

In this example we perform a union of two Datasets, with the same IdentifierComponents and MeasureComponents.

Both Datasets contain information about the total population of a set of countries.

total_population1

TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	5
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	7
2012	Finland	Total	Total	9
2012	Switzerland	Total	Total	12

1825

Total_population2				
TIME	GEO	AGE	SEX	POPULATION
2012	Netherlands	Total	Total	23
2012	Greece	Total	Total	2
2012	Spain	Total	Total	5
2012	Iceland	Total	Total	1

1826

1827 The result of the expression

1828 `ds_r := union(total_population1, total_population2)`

1829 will be:

ds_r				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	5
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	7
2012	Finland	Total	Total	9
2012	Switzerland	Total	Total	12
2012	Netherlands	Total	Total	23
2012	Spain	Total	Total	5
2012	Iceland	Total	Total	1

1830

1831 Example 2

1832 In this example we perform a union of two Datasets that contain population data for a group
1833 of countries for two different reference years.

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6

total_population2				
TIME	GEO	AGE	SEX	POPULATION
2011	Belgium	Total	Total	10
2011	Greece	Total	Total	20
2011	France	Total	Total	30
2011	Malta	Total	Total	40
2011	Finland	Total	Total	50
2011	Switzerland	Total	Total	60

The result of the expression

```
ds_r := union(total_population1, total_population2)
```

will be:

ds_r				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6
2011	Belgium	Total	Total	10
2011	Greece	Total	Total	20
2011	France	Total	Total	30
2011	Malta	Total	Total	40

2011	Finland	Total	Total	50
2011	Switzerland	Total	Total	60

intersect

Syntax

```
Dataset<?> ds_2 :=
intersect
(Dataset<?> ds_1 {, Dataset<?> ds_1}*)
```

Constraints

- all the Datasets *ds_1* must have the same IdentifierComponents and MeasureComponents, in name and type (static).

Semantics

Performs row-wise intersection. Takes in input one or more Datasets *ds_1* with the same IdentifierComponents and MeasureComponents and returns a new Dataset *ds_2* with the same IdentifierComponents and MeasureComponents and AttributeComponents (according to attribute propagation rules).

The result only contains the rows that are present in every *ds_1*, which is their intersection. If only one argument is given, it is returned unchanged. For the comparison, both the IdentifierComponents and the MeasureComponents values are considered.

Parameters

`ds_1` – all the input Datasets to consider in the intersection.

Returns

`ds_2` – the output Dataset

Example

In this example, we perform an intersection of the Datasets total_population1, total_population2.

The result will contain information of the total population of the set of countries that are common in the initial datasets.

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2012	Greece	Total	Total	2

2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6

total_population2				
TIME	GEO	AGE	SEX	POPULATION
2011	Belgium	Total	Total	10
2012	Greece	Total	Total	2
2011	France	Total	Total	30
2011	Malta	Total	Total	40
2011	Finland	Total	Total	50
2011	Switzerland	Total	Total	60

The expression

```
d_r := intersect(total_population1, total_population2)
```

will yield the result:

d_r				
TIME	GEO	AGE	SEX	POPULATION
2012	Greece	Total	Total	2

symdiff

Syntax

```
Dataset<?> ds_3 :=
symdiff
({dataset1=}Dataset<?> ds_1, {dataset2=}Dataset<?> ds_2)
```

Constraints

- *ds_1* and *ds_2* must have the same IdentifierComponents and MeasureComponents, in name and type (static).

Semantics

Performs row-wise symmetric difference. It takes in input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents and MeasureComponents and returns a new Dataset *ds_3* with the same IdentifierComponents and MeasureComponents and AttributeComponents (according to attribute propagation rules). The result contains only the rows that are present

either *ds_1* or in *ds_2* but not in both, which is their symmetric difference. For the comparison, both the IdentifierComponents and the MeasureComponents values are considered.

Parameters

`ds_1` – the first input Dataset to consider in the symmetric difference.

`ds_2` – the second input Dataset to consider in the symmetric difference.

Returns

`ds_3` – the output Dataset

Examples

In this example we perform a symmetric difference of the Datasets total_population1, total_population2. The result will contain information of the total population of the countries that are not common in the initial datasets.

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6

total_population2				
TIME	GEO	AGE	SEX	POPULATION
2011	Belgium	Total	Total	1
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6

The expression:

```
d_r := symdiff(total_population1, total_population2)
```

will yield the result:

d_r				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2011	Belgium	Total	Total	1

setdiff

Syntax

```
Dataset<?> ds_3 :=
Dataset<?> ds_1
setdiff
Dataset<?> ds_2
```

Constraints

- ds_*1* and *ds_2* must have the same IdentifierComponents and MeasureComponents, in name and type (static).

Semantics

Performs row-wise difference. It takes as input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents and MeasureComponents and returns a new Dataset *ds_3* with the same IdentifierComponents and MeasureComponents and AttributeComponents (according to attribute propagation rules).

The result only contains the rows that are present in *ds_1* but not in *ds_2*, which is their difference. For the comparison, both the IdentifierComponents and the MeasureComponents values are considered.

Parameters

`ds_1` – the first input Dataset to consider in the difference.

`ds_2` – the second input Dataset to consider in the difference.

Returns

`ds_3` – the output Dataset

Example 1

In this example we perform a difference between two Datasets that contain population data for a group of countries for two different reference years.

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	10
2012	Greece	Total	Total	20
2012	France	Total	Total	30
2012	Malta	Total	Total	40
2012	Finland	Total	Total	50
2012	Switzerland	Total	Total	60

1932

total_population2				
TIME	GEO	AGE	SEX	POPULATION
2011	Belgium	Total	Total	10
2012	Greece	Total	Total	20
2012	France	Total	Total	30
2012	Malta	Total	Total	40
2012	Finland	Total	Total	50
2012	Switzerland	Total	Total	60

1933 The statement

1934 `d_r := total_population1 setdiff total_population2`

1935 will yield the result:

d_r				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	10

1936

1937 Example 2

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	7
FR	Females	2011	10
FR	Total	2011	12

1938

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	7
FR	Females	2011	10

The following expression

```
DatasetC := DatasetA setdiff DatasetB
```

applied on DatasetA and DatasetB will return the result:

Dataset C			
COUNTRY	SEX	YEAR	VALUE
FR	Total	2011	12

merge

Syntax

```
Dataset<?> ds_3 :=
merge
(Dataset<?> ds_1 {as} Constant<String> alias_1,
Dataset<?> ds_2 {as} Constant<String> alias_2
{,Dataset<?> ds_2 {as} Constant<String> alias_2}*
on(Component<Boolean> cond)
return(
{Component<?> comp {as} Constant<String> alias_3}
{, Component<?> comp {as} Constant<String>}* alias_3))
```

Constraints

- *cond* is a Component expression evaluating to a Boolean. Literals (name of Components) must be qualified by prefixing them with the alias name (*alias_1* or *alias_2*) (static).
- *aliases* must be alphanumeric Strings corresponding to valid names for Components (static).
- *Comp* is a Component expression where only the names of Components from *ds_1* or *ds_2* are allowed. They must be prefixed with the alias name (*alias_1* or *alias_2*) as it guarantees no ambiguities (static).
- *alias_1* and {*alias_2*}+ must be different (static).
- all the {alias_3}+ must be different as there cannot be duplicated Component names in the output (static).

Semantics

Takes in input two or more Datasets *ds_1*, *ds_2* and returns a third Dataset *ds_3*, which is obtained as follows:

- *ds_1* and *ds_2* are given an alias *ds_1* and *ds_2*, respectively
- the Cartesian product of all the rows from *ds_1* and *ds_2* is temporarily calculated
- from the Cartesian products, the rows for which the Boolean Component expression evaluates to *true* are kept, while the others are filtered out. The Component<Boolean> is a qualified expression involving *ds_1* and *ds_2* components (e.g. *alias_1*#k1 = *alias_2*#k2 and *alias_1#k3* > alias1#k4)
- from the kept rows in the Cartesian products, the output rows are produced by keeping the only Components specified in *comp* Component expressions. For each of them, the role (i.e. if they are IdentifierComponent or MeasureComponent) is directly derived from the role they have in *ds_1* or *ds_2* and an *alias_3*, which is a new name, must be assigned to each of them.
- AttributeComponents are added according to the attribute propagation rules.

Parameters

ds_1 – the first Dataset to merge

ds_2 – all the following Dataset to merge

alias_1 – an alias for *ds_1* to be used in *cond* and *comp*

alias_2 – an alias for all the *ds_2* to be used in *cond* and *comp*

cond – a qualified Component expression, specifying the merge conditions

comp – a qualified Component expression (only literal Component names are allowed) specifying the desired Components in the output

alias_3 – an alias for a desired Component in the output

Returns

ds_3 – the returned Dataset

Example 1

In this example we report the multi-measure Dataset produced by the following expression.

```
ds_r := merge(ds_1 "A",ds_2 "B",on(A#country =
B#country),return(A#country as "Country" ,A#value as
"Value",B#population as "Population"))
```

DS_1		
COUNTRY	CITY	VALUE
Belgium	Brussels	34
Greece	Athens	0
France	Paris	21

DS_2		
COUNTRY	CITY	POPULATION
Belgium	Brussels	45
Greece	Athens	46
France	Paris	47

DS_R		
COUNTRY	VALUE	POPULATION
Belgium	34	45
Greece	0	46
France	21	47

Example 2

In this example we report the mono-measure Dataset produced by the following expression.

```
ds_r := merge(ds_1 "A",ds_2 "B",on(A#country = B#country and A#city
= B#city), return(A#country as "Country",B#city as "City", A#value
as "Value"))
```

DS_R		
COUNTRY	CITY	VALUE
Belgium	Brussels	34
Greece	Athens	0
France	Paris	21

min/max

Syntax

```
Dataset<?> ds_2 :=
[min|max]
({dataset=}Dataset<?> ds_1)
```

Constraints

- ds_1 must have only one MeasureComponent (static).

Semantics

These operators find the minimum or maximum row in a Dataset according to the minimum or maximum value of the MeasureComponent (based on numerical order for numeric values, or lexicographical order, for String values). In case of ambiguity, the one to be returned is up to the implementation. They take in input a Dataset *ds*_1 with a single ComponentMeasure and return a new Dataset *ds_2* with the same IdentifierComponent, MeasureComponents and AttributeComponents (according to attribute propagation rules).

NULL values are not considered and rows with NULLS are never returned.

Example

In this example we find the country with the largest (smallest) population.

DatasetA

GEO	TIME	VALUE
Sweden	2012	9.500
Denmark	2012	5.590
Finland	2012	5.400
Norway	2012	4.960
Iceland	2012	0.328
Greenland	2012	0.057
Faroe Islands	2012	0.049

The expression:

```
max (DatasetA)
```

returns:

GEO	TIME	VALUE
Sweden	2012	9.500

`min (DatasetA)` returns:

GEO	TIME	VALUE
Faroe Islands	2012	0.049

hierarchy

VTL foresees the existence of set relations among Code Items in Code List.

Many Enumerated Value Domains have an intrinsic Boolean algebraic structure, in the sense that a Boolean algebra can be defined on the respective Code Items.

In general, a Boolean algebra is an algebraic structure (elements and operators having some properties) that summarizes the properties of set operators (*union*, *intersection*, *complement*) and logical operators (or, and, not).

Elements of a Boolean Value Domain can be combined with the elementary set operators[2]; e.g. the item C is the union of the items D and E; the item K is the complement of S with respect to J (so the elements in J that are not in S), the item A is the intersection of B and C and so on.

Only considering the set union, there are two possible organizations for hierarchical Code Lists: classifications and free hierarchies.

In **classifications**, every element is uniquely classified by a single partition, so that the overall structure is a tree and, consequently, every Code Item can be given a specific level. An example is the usual geographical classification of the world into continents, each partitioned into nations.

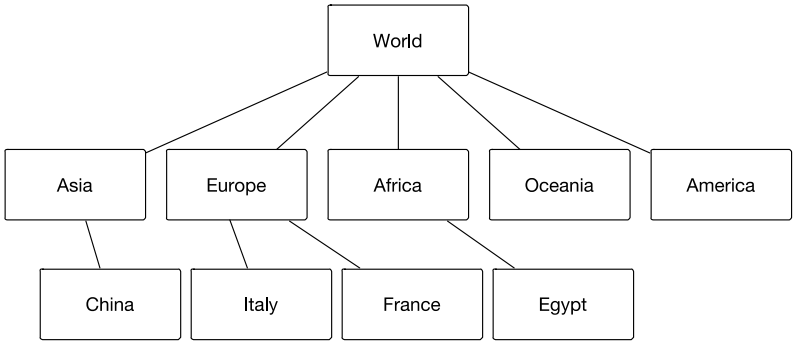

Figure 1 - example of hierarchical Code List

[2] Here we refer to elementary set operations and not to any operator of the language.

In **free hierarchies** each item can be partitioned according to multiple criteria; in turn, every element can be used to compose multiple other elements. The overall structure is not a tree and isolated Items can be present. An example is the hierarchy of European countries, where Belgium, Holland and Luxembourg contribute to the "European Countries" Code Item and to the "Benelux" Code Item.

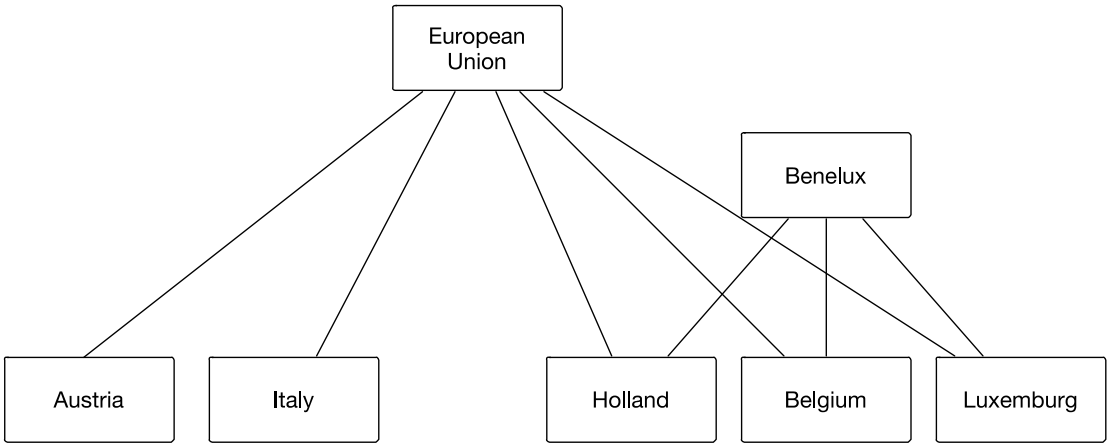

Figure 2 - Example of free hierarchy

More sophisticated hierarchical organizations can indeed exist if intersection and complement set operators are also considered.

For example the element "Benelux" could also be defined as the complement of the European Union with respect to all the countries except Holland, Belgium and Luxembourg.

Therefore we would have:

BENELUX = EU – (ITALY ∪ AUSTRIA ∪ …)

In order to support multiple classifications of Code Items in Code Lists and allow for the adoption of all the set operators, we introduce two concepts *hierarchical aggregations* and *mappings* (that will be used in the **hierarchy** operator).

A *hierarchical aggregation* is a set of *mappings*, each transforming a Code Item into another Code Item. All the *mappings* within a *hierarchical aggregation* associate Code Items of the same Code List with Code Items of a single Code List.

Hierarchical aggregations and mappings can be easily expressed in tabular form and referred to in the language (in **hierarchy** operator) by an identifier (*hierarchyName*). However, there is also an inline syntactical form.

Suppose we want to express the hierarchical relationship BENELUX = Belgium ∪ Holland ∪ Luxembourg. There will be a hierarchical aggregation **Benelux_aggr**, with the following mappings:

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
Benelux_aggr	Belgium	Benelux	….	…	+	1	true
Benelux_aggr	Holland	Benelux	…	…	+	1	true
Benelux_aggr	Luxembourg	Benelux	…	…	+	1	true

It maps each component item into the compound one.

Each mapping has a *Sign*. It specifies if the contribution of the MAPS FROM Code Item in the composition is positive (UNION) or negative (COMPLEMENT). Notice that there is not a particular convention to represent intersection, as it can be obtained with an appropriate composition of UNION and COMPLEMENT.

For instance, if we want to define in the element **EuropeWithoutItaly**, a possible definition could be the one complementing Italy, with respect to the entire Europe (i.e. subtracting Italy from Europe), as shown in the following table:

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
Eu_no_Italy_aggr	Europe	EuropeWithoutItaly	….	…	+	1	true
Eu_no_Italy_aggr	Italy	EuropeWithoutItaly	…	…	-	1	true

Moreover, mappings are divided into levels, in the sense that a complete tree can be embedded into one hierarchy. The OUTPUT property, for each MAPS TO, indicates if the value must be preserved in the output or is only to be used for aggregations at a higher level.

For instance, suppose we want to express the hierarchical relationship in Figure 2, there will be a hierarchy **World_aggr**, with the following correspondences:

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
World_aggr	Italy	Europe	….	…	+	1	false
World_aggr	France	Europe	…	…	+	1	false
World_aggr	Luxembourg	Europe	…	…	+	1	false
World_aggr	…	Europe			+	1	false
World_aggr	Asia	World			+	2	true
World_aggr	Europe	World			+	2	true
World_aggr	America	World			+	2	true
World_aggr	Oceania	World			+	2	true
World_aggr	Africa	World			+	2	true

Output false for LEVEL 1, indicates that the aggregations into Europe (and into the other countries) are only functional to LEVEL 2 and the MAPS TO values with false will not be in the output.

Summarizing, the set of mappings within a hierarchy has to be interpreted as follows.

For every level, from the lowest to the highest, each MAPS TO Code Item is the set difference between the UNION of all the corresponding MAPS FROM Code Items with positive SIGN and the UNION of all the corresponding MAPS FROM Code Items with the negative SIGN.

Rules inherently represent hierarchies in Code Lists, but, at the same time, only refer to Code Items. Thus, can be applied on different *Identifier Components* referring to different Code Lists.

Syntax ("hierarchy" Operator)

```
Dataset ds_2 :=
hierarchy
(
{dataset=}Dataset<?,MeasureComponent<Numeric>+> ds_1,
{component=}IdentifierComponent<?> comp,
 [
{hierarchy_name=}Constant<String> hierarchyname
|(
{
( ({from=}Constant<T> maps_from, {level=}Constant<Integer> level,
{sign=} [+|-])
{,({from=}Constant<T> maps_from, {level=}Constant<Integer> level,
{sign=} [+|-])}*)

to {to=}Constant<T> maps_to}+

) as Constant<String> hierarchyname
], {isFilter=}Constant<Boolean> isFilter
{, {aggregation=} [sum|prod] }
)
```

Constraints

- `hierarchyname` denotes a hierarchical aggregation either externally configured in a table or embedded in the operator with the inline notation (static).
- `level > 0` (static).
- All the *MeasureComponent*s of `ds_1` must be Numeric (static).

Semantics

It applies a hierarchical aggregation with name `hierarchyname` on an *IdentifierComponent* `comp` in a Dataset `ds_1`, aggregating all the *MeasureComponent*s according to an aggregation function (algebraic sum or product).

`hierarchyname` can be the identifier of an aggregation that is externally configured in a table, with an associated set of mappings, each with a sign and a level.

Alternatively, a hierarchical aggregation can be expressed in an inline fashion, where `maps_from` constants are mapped into `maps_to` ones in the specific `level` and `sign`.

For the given aggregation, for each level, all the mappings are considered and orderly applied with the following logic.

For each value of the *IdentifierComponent* of all the records of the considered Dataset, if the value is present in `maps_from` for any mapping, it is turned into the respective `maps_to` value; if the value is not present in `maps_from` for any correspondence, the entire record is discarded if `isFilter` is true, the original value is preserved if `isFilter` is false.

Aggregations are typically hierarchical, in the sense that they map many `maps_from` values into fewer `maps_to` values: often, multiple component Code Items collapse into the same compound Code Item.

Therefore, at this stage, there may be multiple records having the same values for all the *IdentifierComponent*s, as the differentiating ones have been aggregated into the same one.

The records having the same value for all the *IdentifierComponent*s are aggregated by algebraic sum (**sum**) or product (**prod**) of their *MeasureComponent*s. If the aggregation function is omitted, sum is implied.

Sum implies that the *MeasureComponent*s have to be algebraically summed, considered with positive or negative sign depending on the Sign of the used mapping.

Prod implies that the *MeasureComponent*s have to be multiplied, considered with -1 exponent when the negative Sign is used in the mapping.

Notice that the use of prod as aggregation function is meaningful only when the *IdentifierComponent* is a measure dimension.

Hierarchies and measure dimensions

As it is well known, an *IdentifierComponent* in a Dataset can play the role of *measure dimension*, meaning that the Dataset is indeed multi-measure, but represented as mono-measure with a further, measure-qualifying dimension.

*IdentifierComponent*s in `hierarchy` can also be a measure dimension, although in this case the aggregation inherently assumes a different meaning.

In facts, Data Points with all coinciding *IdentifierComponent*s, except for the measure dimension, are nothing but an expression of different measures for the same data point. An aggregation over the *measure dimension* is conceptually an operation involving the measures of the same Data Point (algebraic sum or multiplication).

Hierarchies as transcodings

The presented mapping method can be used intuitively to express a transcoding in a synthetic way. In this case, the involved Component value is mapped into another one, which is not hierarchically related with the first, but simply represents the same value expressed in another coding standard.

Hierarchies as filters

`isFilter` parameter in `hierarchy` allows choosing whether an input Data Point is to be kept in the output even if there are not mappings having the `maps_from` value corresponding to the *IdentifierComponent* of that Data Point.

Indeed, this mechanism lends itself to the construction of reusable filters, independent of the specific Dataset and *IdentifierComponent* they are applied on.

Such rules comprise a set of (X→X, Y→, …, Z→Z) correspondences preserving the values X, Y and Z of the *IdentifierComponent*s and filtering out the Data Points having a value for the *IdentifierComponent* different from X, Y or Z.

Parameters

`ds_1` – the Dataset to be aggregated

`comp` – the *IdentifierComponent* to aggregate upon

`hierarchyname` – the name of the hierarchical aggregation of the information model, which can be optionally replaced by an inline specification of the rule

`maps_from` – an input value in an inline aggregation rule

`level` – the level of hierarchy of a correspondence (which represents the order of application) in an aggregation rule

`sign` – the sign of the contribution of a `maps_from` value in an aggregation rule

`maps_to` – an output value in an inline aggregation rule

`isFilter` – if the aggregation must be interpreted as a filter (excluding non matching records)

Returns

`ds_2` – the aggregated Dataset

Example 1

The expression:

```
Income_by_country_and_nace_ISO :=
hierarchy("Income_by_country_and_nace_UN", GEO, un_to_ISO_aggr,
false)
```

or its equivalent inline form:

```
Income_by_country_and_nace_ISO :=
hierarchy("Income_by_country_and_nace_UN", GEO,
(("ITA",1,+)) to "IT", (("ALB",1,+)) to AL, (("BEL",1,+)) to "BE")
as un_to_ISO_aggr, false)
```

Converts the values of the GEO Identifier Component from the United Nations 3-letter standard into the ISO 2-letter one.

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
un_to_ISO_aggr	ITA	IT	….	…	+	1	true
un_to_ISO_aggr	ALB	AL	…	…	+	1	true
un_to_ISO_aggr	BEL	BE	…	…	+	1	true

Example 2

The expression:

```
Income_by_continent_and_nace :=
hierarchy("Income_by_state_and_nace", GEO, Continent_aggr, false)
```

Takes as input the Dataset of the income, broken down by states and NACE and aggregates to continent level.

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
Continent_aggr	Italy	Europe	….	…	+	1	true
Continent_aggr	France	Europe	…	…	+	1	true
Continent_aggr	Luxembourg	Europe	…	…	+	1	true
Continent_aggr	…	Europe			+	1	true
Continent_aggr	China	Asia			+	1	true
Continent_aggr	India	Asia			+	1	true
Continent_aggr	USA	America			+	1	true
Continent_aggr	…	…			+	1	true

Income_by_state_and_nace

GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20
France	IND	31
France	TECH	50
Spain	IND	30
Spain	TECH	15
China	IND	250
China	TECH	250
India	IND	30
India	TECH	100
Luxembourg	IND	10
Luxembourg	TECH	12

Income_by_continent_and_nace

GEO	NACE	VALUE
Europe	IND	81
Europe	TECH	97
Asia	IND	280
Asia	TECH	350

Example 3

The expression:

```
Income_by_world_and_nace := hierarchy("Income_by_state_and_nace",
GEO, World_aggr, false)
```

Takes as input the Dataset of the income, broken down by states and NACE and aggregates to the world level. World_aggr is a 2 levels Rule. LEVEL 1 is only a temporary output and its MAPS TO Code Items do not appear in the final result.

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
World_aggr	Italy	Europe	….	…	+	1	false
World_aggr	France	Europe	…	…	+	1	false
World_aggr	Luxembourg	Europe	…	…	+	1	false
World_aggr	…	Europe			+	1	false
World_aggr	…	Europe			+	1	false
World_aggr	China	Asia			+	1	false
World_aggr	India	Asia			+	1	false
World_aggr	USA	America			+	1	false
World_aggr	Asia	World			+	2	true
World_aggr	Europe	World			+	2	true
World_aggr	America	World			+	2	true
World_aggr	Oceania	World			+	2	true
World_aggr	Africa	World			+	2	true

Income_by_state_and_nace

GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20
France	IND	31

France	TECH	50
Spain	IND	30
Spain	TECH	15
China	IND	250
China	TECH	250
India	IND	30
India	TECH	100
Luxembourg	IND	10
Luxembourg	TECH	12

Income_world

GEO	NACE	VALUE
World	IND	361
World	TECH	447

Example 4

The expression:

```
Income_world_and_nace_par := hierarchy("Income_by_state_and_nace",
World_aggr_par, false)
```

Or its equivalent inline version:

```
Income_world_and_nace_par := aggrrule(
Income_by_state_and_nace, GEO,
(("Italy",1,+),("France",1,+),("Luxembourg",1,+) to "Europe,
(("China",1,+),("India",1,+)) to "Asia",
(("USA",1,+)) to "America,
(("Asia",2,+),("Europe",2,+),("America",2,+),
("Oceania",2,+),("Africa",2,+)) to "World")
as World_aggr_par,
false)
```

Takes as input the Dataset of the income, broken down by states and NACE and aggregates to the World level. Differently from example 2, it preserves the first level in the output.

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
World_aggr_par	Italy	Europe	….	…	+	1	true
World_aggr_par	France	Europe	…	…	+	1	true
World_aggr_par	Luxembourg	Europe	…	…	+	1	true
World_aggr_par	…	Europe			+	1	true

World_aggr_par	…	Europe			+	1	true
World_aggr_par	China	Asia			+	1	true
World_aggr_par	India	Asia			+	1	true
World_aggr_par	USA	America			+	1	true
World_aggr_par	Asia	World			+	2	true
World_aggr_par	Europe	World			+	2	true
World_aggr_par	America	World			+	2	true
World_aggr_par	Oceania	World			+	2	true
World_aggr_par	Africa	World			+	2	true

2258

2259 **Income_by_state_and_nace**

GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20
France	IND	31
France	TECH	50
Spain	IND	30
Spain	TECH	15
China	IND	250
China	TECH	250
India	IND	30
India	TECH	100
Luxembourg	IND	10
Luxembourg	TECH	12

2260

2261 **Income_world_and_nace_par**

GEO	NACE	VALUE
World	IND	361
World	TECH	447
Europe	IND	81
Europe	TECH	97
Asia	IND	280
Asia	TECH	350

Example 5

The expression:

```
Italian_income_by_nace:= hierarchy("Income_by_state_and_nace",
Italy_filter, true)
```

Takes as input the Dataset of the income, broken down by states and NACE and filters out all the incomes that do not refer to Italy.

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
Italy_filter	Italy	Italy	….	…	+	1	true

Income_by_state_and_nace

GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20
France	IND	31
France	TECH	50
Spain	IND	30
Spain	TECH	15
China	IND	250
China	TECH	250
India	IND	30
India	TECH	100
Luxembourg	IND	10
Luxembourg	TECH	12

Italian_income_by_nace

GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20

Example 6

The expression:

```
income_prod_2:=    hierarchy("Income_by_state_and_nace_mm",    GEO,
mult_rule, false, prod)
```

Takes as input the Dataset of the income, broken down by states and NACE in a measure dimension form and aggregates by calculating INC2 / INC1 into INC.

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
Mult_measure	INC1	INC	….	…	-	1	true
Mult_measure	INC2	INC	….	…	+	1	true

Income_by_state_and_nace

GEO	NACE	MEASURE	VALUE
Italy	IND	INC1	10
Italy	IND	INC2	20
Italy	TECH	INC1	20
Italy	TECH	INC2	40
France	IND	INC1	31
France	IND	INC2	61
France	TECH	INC1	50
France	TECH	INC2	100
China	IND	INC1	250
China	IND	INC2	500
China	TECH	INC1	250
China	TECH	INC2	500
India	IND	INC1	30
India	IND	INC2	60
India	TECH	INC1	100
India	TECH	INC2	200

Income_prod2

GEO	NACE	MEASURE	VALUE
Italy	IND	INC	2
Italy	TECH	INC	2
France	IND	INC	2
France	TECH	INC	2
China	IND	INC	2
China	TECH	INC	2
India	IND	INC	2
India	TECH	INC	2

Validation operators

VTL provides operators for **comparing** Datasets with one another and Datasets with Constants.

For NULL values, the following holds for all relational operators: Since NULL does not represent a value, but instead is a marker indicating undefined, comparisons with NULL can never result in either True or False, but always in a third logical result, which is undefined and therefore best represented as NULL. This approach follows in principle the three-valued logic of Kleene and Łukasiewicz.

Syntax

```
Component<Boolean> c :=
Component<T> c_1
[>|<|>=|<=|=|<>]
Component<T> c_2
```

Constraints

T is not Boolean (static).

Semantics

Returns true if c_1 is greater-than, less-than, greater-or-equal, less-or-equal, equal, not equal c_2; otherwise false. For Numeric Component the meaning is straightforward, whereas for Strings, the comparison operators must be interpreted with respect to the lexicographical order.

Examples

If A = 5, B = 9, C = 5:

```
D := A > B // D = false
D := A < B // D = true
D := A = B // D = false
D := A >= C // D = true
D := A <= C // D = true
D := A = C // D = true
D := A <> C // D = false
```

If A = “hello”, B = “hi”, C = “Hi”:

```
D := A > B // D = false
D := C < B // D = false
D := A = B // D = false
```

equal to =

Syntax

```
Dataset<?+, MeasureComponent<Boolean>+> ds_3  :=
[Dataset<T> | Constant<T>] ds_1
=
[Dataset<T> | Constant<T>] ds_2
```

Constraints

- at least one between *ds_1* and *ds_2* must be a Dataset (static).
- if both *ds_1* and *ds_2* are Datasets they must have the same IdentifierComponents (in name and type) or, at least, the IdentifierComponents of *ds_1* may be a subset of the ones of *ds_2* (or viceversa) (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents, in name and number (static).
- If *ds_1* (*ds_2*) is a Constant<T>, *ds_2* (*ds*_1) must have a single MeasureComponent<T> (static).

Semantics

It takes in input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents, or at least the identifier components of one being a subset of the IdentifierComponents of the other, and the same MeasureComponents (in name and type). It returns a third Dataset *ds_3*, having the superset of the IdentifierComponents of *ds_1* and *ds_2*, some AttributeComponents (according to the attribute propagation rules) and, for each pair of homonym (and same type) MeasureComponents of *ds_1* and *ds_2* a MeasureComponent<Boolean>. For each pair of homonym MeasureComponents, the corresponding MeasureComponent<Boolean> in *ds_3* is named as the concatenation, separated by "_", of their names with the suffix "CONDITION", (e.g. if the compared MeasureComponents are named "VALUE", the MeasureComponent<Boolean> will be named "VALUE_CONDITION").

For each pair of rows in *ds_1* and *ds_2* having the same values for all the common IdentifierComponents, the MeasureComponent<Boolean> in *ds*_3 will contain the result of the comparison. Whenever a NULL is present in a comparison, NULL is returned as a result.

If *ds_1* (or *ds_2*) is a Constant, *ds_3* has a single MeasureComponent<Boolean>, named "CONDITION", containing the result of the comparison of the Constant literal with the single MeasureComponent of all the rows.

Parameters

ds_1 – a Dataset or a Constant

ds_2 – a Dataset or a Constant.

Returns

ds_3 – a Dataset containing the result of the comparison.

Example

In this example we check whether the values of a dataset measure are equal to a specific numeric value. The result dataset has the same classification structure as the initial dataset and it contains a single measure.

overcrowding_rate_urbanization				
TIME	GEO	AGE	SEX	VALUE
2012	Belgium	Total	Total	NULL
2012	Greece	Total	Total	0.286
2012	Spain	Total	Total	0.064
2012	Malta	Total	Total	0.043
2012	Finland	Total	Total	0.08
2012	Switzerland	Total	Total	0.08

The expression:

```
Overcrowding_rate_urbanization = 0.08
```

will yield the following result dataset:

TIME	GEO	AGE	SEX	CONDITION
2012	Belgium	Total	Total	NULL
2012	Greece	Total	Total	false
2012	Spain	Total	Total	false
2012	Malta	Total	Total	false
2012	Finland	Total	Total	true
2012	Switzerland	Total	Total	true

not equal to <>

Syntax

```
Dataset<?+, MeasureComponent<Boolean>+> ds_3 :=
[Dataset<T> | Constant<T>] ds_1
<>
[Dataset<T> | Constant<T>]
```

Constraints

- at least one between *ds_1* and *ds_2* must be a Dataset (static).

- if both *ds_1* and *ds_2* are Datasets they must have the same IdentifierComponents (in name and type) or, at least, the IdentifierComponents of *ds_1* may be a subset of the ones of *ds_2* (or viceversa) (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents, in name and number (static).
- If *ds_1* (*ds_2*) is a Constant<T>, *ds_2* (*ds*_1) must have a single MeasureComponent<T> (static).

Semantics

It takes in input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents, or at least the identifier components of one being a subset of the IdentifierComponents of the other, and the same MeasureComponents (in name and type). It returns a third Dataset *ds_3*, having the superset of the IdentifierComponents of *ds_1* and *ds_2*, some AttributeComponents (according to the attribute propagation rules) and, for each pair of homonym (and same type) MeasureComponents of *ds_1* and *ds_2* a MeasureComponent<Boolean For each pair of homonym MeasureComponents, the corresponding MeasureComponent<Boolean> in *ds_3* is named as the concatenation, separated by "_", of their names with the suffix "CONDITION", (e.g. if the compared MeasureComponents are named "VALUE", the MeasureComponent<Boolean> will be named "VALUE_CONDITION").

For each pair of rows in *ds_1* and *ds_2* having the same values for all the common IdentifierComponents, the MeasureComponent<Boolean> in *ds*_3 will contain the result of the comparison not equal. Whenever a NULL is present in a comparison, NULL is returned as a result.

If *ds_1* (or *ds_2*) is a Constant, *ds_3* has a single MeasureComponent<Boolean>, named "CONDITION", containing the result of the comparison of the Constant literal with the single MeasureComponent of all the rows.

Parameters

ds_1 – a Dataset or a Constant

ds_2 – a Dataset or a Constant.

Returns

ds_3 – a Dataset containing the result of the comparison.

Example

Y_unemployment_2012				
GEO	SEX	UNIT	C_BIRTH	VALUE
Germany	Total	Percentage	Total	7.1
Greece	Total	Percentage	Total	NULL

y_unemployment_2011				
GEO	SEX	UNIT	C_BIRTH	VALUE
Germany	Total	Percentage	Total	7.5
Greece	Total	Percentage	Total	3

The expression:

```
compare_ds := y_unemployment_2012 <> y_unemployment_2011
```

will yield the result:

GEO	SEX	UNIT	C_BIRTH	CONDITION_VALUE
Germany	Total	Percentage	Total	true
Greece	Total	Percentage	Total	NULL

If VALUE for Greece in the second operand had also been NULL, then the result would still be NULL for Greece. This reveals the shortcoming of our approach, namely that (NULL<>NULL) = (NULL==NULL), which necessitates the introduction of an isnull operator in the language.

greater than > >=

Syntax

```
Dataset<?+, MeasureComponent<Boolean>+> ds_3 :=
[Dataset<T> | Constant<T>] ds_1
>{=}
[Dataset<T> | Constant<T>] ds_2
```

Constraints

- at least one between *ds_1* and *ds_2* must be a Dataset (static).
- if both *ds_1* and *ds_2* are Datasets they must have the same IdentifierComponents (in name and type) or, at least, the IdentifierComponents of *ds_1* may be a subset of the ones of *ds_2* (or viceversa) (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents, in name and number (static).
- If *ds_1* (*ds_2*) is a Constant<T>, *ds_2* (*ds_*1) must have a single MeasureComponent<T>.
- T must not be Boolean (static).

Semantics

It takes in input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents, or at least the identifier components of one being a subset of the IdentifierComponents of the other, and the same MeasureComponents (in name and type). It returns a third Dataset *ds_3*, having the superset of the IdentifierComponents of *ds_1* and *ds_2*, some AttributeComponents (according to the attribute propagation rules) and, for each pair of homonym (and same type)

MeasureComponents of *ds_1* and *ds_2* a MeasureComponent<Boolean For each pair of homonym MeasureComponents, the corresponding MeasureComponent<Boolean> in *ds_3* is named as the concatenation, separated by "_", of their names with the suffix "CONDITION", (e.g. if the compared MeasureComponents are named "VALUE", the MeasureComponent<Boolean> will be named "VALUE_CONDITION").

For each pair of rows in *ds_1* and *ds_2* having the same values for all the common IdentifierComponents, the MeasureComponent<Boolean> in *ds*_3 will contain the result of the comparison greather (or equal) than. For Numeric MeasureComponents, the meaning is straightforward, whereas for Strings, the comparison operators must be interpreted with respect to the lexicographical order.

If *ds_1* (or *ds_2*) is a Constant, *ds_3* has a single MeasureComponent<Boolean>, named "CONDITION", containing the result of the comparison of the Constant literal with the single MeasureComponent of all the rows.

Parameters

ds_1 – a Dataset or a Constant

ds_2 – a Dataset or a Constant.

Returns

ds_3 – a Dataset containing the result of the comparison.

Example 1

In this example, we check whether the values of a dataset measure are greater than a Constant value. The result dataset has the same classification structure as the initial dataset and it contains a single measure.

foreign_languages_known					
N_LANG	GEO	TIME	AGE	UNIT	VALUE
2	Germany	2011	Total	Percentage	NULL
2	Greece	2011	Total	Percentage	12.2
2	Finland	2011	Total	Percentage	29.5

The expression:

```
compare_ds := foreign_languages_known > 20
```

will yield the following result dataset:

compare_ds					
N_LANG	GEO	TIME	AGE	UNIT	CONDITION
2	Germany	2011	Total	Percentage	NULL
2	Greece	2011	Total	Percentage	false
2	Finland	2011	Total	Percentage	true

Example 2

In the following example we check whether the values of a dataset measure are greater than the values of another dataset measure. We use data about the youth unemployment rate of two consecutive years.

y_unemployment_2012				
GEO	SEX	UNIT	C_BIRTH	VALUE
Germany	Total	Percentage	Total	7.1
Greece	Total	Percentage	Total	42.5

y_unemployment_2011				
GEO	SEX	UNIT	C_BIRTH	VALUE
Germany	Total	Percentage	Total	7.5
Greece	Total	Percentage	Total	33.7

The expression

```
compare_ds := y_unemployment_2012 > y_unemployment_2011
```

will yield the result:

compare_ds				
GEO	SEX	UNIT	C_BIRTH	CONDITION_VALUE
Germany	Total	Percentage	Total	false
Greece	Total	Percentage	Total	true

If the VALUE column for Germany in the y_unemployment_2012 dataset had a NULL value the result would be:

GEO	SEX	UNIT	C_BIRTH	CONDITION_VALUE
Germany	Total	Percentage	Total	NULL
Greece	Total	Percentage	Total	true

less than `< <=`

Syntax

```
Dataset<?+, MeasureComponent<Boolean>+> ds_3 :=
[Dataset<T> | Constant<T>] ds_1
<{=}
[Dataset<T> | Constant<T>] ds_2
```

Constraints

- at least one between *ds_1* and *ds_2* must be a Dataset (static).
- if both *ds_1* and *ds_2* are Datasets they must have the same IdentifierComponents (in name and type) or, at least, the IdentifierComponents of *ds_1* may be a subset of the ones of *ds_2* (or viceversa) (static).
- If both *ds_1* and *ds_2* are Datasets, they must have the same MeasureComponents, in name and number (static).
- If *ds_1* (*ds_2*) is a Constant<T>, *ds_2* (*ds*_1) must have a single MeasureComponent<T> (static).

Semantics

It takes in input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents, or at least the identifier components of one being a subset of the IdentifierComponents of the other, and the same MeasureComponents (in name and type). It returns a third Dataset *ds_3*, having the superset of the IdentifierComponents of *ds_1* and *ds_2*, some AttributeComponents (according to the attribute propagation rules) and, for each pair of homonym (and same type) MeasureComponents of *ds_1* and *ds_2* a MeasureComponent<Boolean>. For each pair of homonym MeasureComponents, the corresponding MeasureComponent<Boolean> in *ds_3* is named as the concatenation, separated by "_", of their names with the suffix "CONDITION", (e.g. if the compared MeasureComponents are named "VALUE", the MeasureComponent<Boolean> will be named "VALUE_CONDITION").

For each pair of rows in *ds_1* and *ds_2* having the same values for all the common IdentifierComponents, the MeasureComponent<Boolean> in *ds*_3 will contain the result of the comparison less (or equal) than. Whenever a NULL is present in a comparison, NULL is returned as a result.

If *ds_1* (or *ds_2*) is a Constant, *ds_3* has a single MeasureComponent<Boolean>, named "CONDITION", containing the result of the comparison of the Constant literal with the single MeasureComponent of all the rows.

Parameters

ds_1 – a Dataset or a Constant

ds_2 – a Dataset or a Constant.

Returns

- *ds_3* – a Dataset containing the result of the comparison.

Example

In this example, we check whether the values of a Dataset are smaller than a specific Constant numeric value. The result dataset has the same classification structure as the initial dataset and it contains a single measure.

total_population				
TIME	GEO	AGE	SEX	VALUE
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	46818219
2012	Malta	Total	Total	NULL
2012	Finland	Total	Total	5401267
2012	Switzerland	Total	Total	7954662

The expression:

```
compare_ds := total_population < 15000000
```

will yield the following result dataset:

compare_ds				
TIME	GEO	AGE	SEX	CONDITION
2012	Belgium	Total	Total	true
2012	Greece	Total	Total	true
2012	Spain	Total	Total	false
2012	Malta	Total	Total	NULL
2012	Finland	Total	Total	true
2012	Switzerland	Total	Total	true

in, not in

Component version

Syntax

```
Component<T> c_1 {not}
in
{Set|List}<T> c_2
```

Constraints

None

Semantics

Returns true if c_1 is (not) in (set or list) c_2.

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Boolean>+> ds_2 := Dataset<?+,
MeasureComponent<T>+> ds_1
{not} in
{CollectionSet|CollectionList}<T> s
```

Constraints

- all the MeasureComponents of *ds_1* must have the same type T (which is also the type of the Set or List) (static).

Semantics

Checks whether the MeasureComponents of the Dataset *ds_1* take values in the Set or List *s*. It returns a second Dataset *ds_2*, with all the IdentifierComponents in *ds_1*, all the AttributeComponents in *ds_1* (according to the attribute propagation rule) and a MeasureComponent<Boolean> for each MeasureComponent<T> of *ds*_1, named "CONDITION_" concatenated to the name of the original MeasureComponent. For each row in *ds_1* there is a row in *ds_2* where the MeasureComponents states if the values of *ds_1* are (not) in *s*.

Parameters

ds_1 – a Dataset

s – *a Collection*Set or CollectionList of values

Returns

ds_2 – a Dataset containing the result of the comparison.

Example

In this example we apply the {**not**} **in** operator on the GEO dimension of the total_population Dataset.

total_population				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034

2012	Spain	Total	Total	46818219
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	NULL	Total	Total	7954662

The result yielded by the expression

```
ds_1 := total_population in (11094850, 46818219, 222, 111)
```

will have a single measure column that will take value *true* for records for which the value in the POPULATION ComponentMeasure is contained in the set, *false* otherwise.

TIME	GEO	AGE	SEX	CONDITION_POPULATION
2012	Belgium	Total	Total	true
2012	Greece	Total	Total	false
2012	Spain	Total	Total	true
2012	Malta	Total	Total	false
2012	Finland	Total	Total	false
2012	NULL	Total	Total	false

between

Component version

Syntax

```
Component<Boolean> c :=
Component<T> c_1
between
Component<T> c_2
and
Component<T> c_3
```

Constraints

None

Semantics

Returns True if c_1 is greater or equal than c_2 and less or equal than c_3; otherwise false.

Dataset version

Syntax

```
Dataset<?+, MeasureComponent<Boolean>+> ds_4 :=
```

```
[Dataset<T> | Constant<T>] ds_1
{not} between
[Dataset<T> | Constant<T>] ds_2
and
[Dataset<T> | Constant<T>] ds_3
```

Constraints

- at least one between *ds_1*, *ds_2* and *ds_3* must be a Dataset (static).
- if two (or three) among *ds_1*, *ds_2* and *ds_3* are Datasets, for every pair of Datasets, it must hold that either they have the same IdentifierComponents or the ones of the former is a subset of the ones of the latter (static).
- If two (or three) among *ds_1*, *ds_2* and *ds_3* are Datasets, they must have the same MeasureComponents, in name and number (static).
- If at least one among *ds_1*, *ds_2* or *ds_3* is a Constant, the others must have a single MeasureComponent<T> (static).

Semantics

It takes in input three Datasets *ds_1*, *ds_2* and *ds_3* with the described constraints and returns a fourth Dataset *ds_4* with the most comprehensive set of IdentifierComponents out of the ones of *ds_1*, *ds_2* and *ds_3*, all the AttributeComponents (according to attribute propagation rules) and, for each triple of homonym (and same type) MeasureComponent, a MeasureComponent<Boolean>.

For each triple of homonym MeasureComponents, the corresponding MeasureComponent<Boolean> in *ds_3* is named as the concatenation, separated by "_", of their names (e.g. if the compared MeasureComponents are named "VALUE", the MeasureComponent<Boolean> will be named "VALUE_CONDITION").

For each triple of rows in *ds_1*, *ds_2* and *ds_3* having the same values for all the common IdentifierComponents, the MeasureComponent<Boolean> in *ds*_3 will contain the result of the range comparison: *true* (false) if the MeasureComponent value of *ds_1* is (not) between the values of *ds_2* and *ds_3*.

If at least one among *ds_1*, *ds_2* or *ds_3* is a Constant, *ds_4* has a single MeasureComponent<Boolean>, named "CONDITION", containing the result of the comparison of the Constant literal(s) with the single MeasureComponent of the Dataset(s).

Parameters

ds_1 – a Dataset or a Constant

ds_2 – a Dataset or a Constant

ds_3 – a Dataset or a Constant.

Returns

ds_4 – a Dataset containing the result of the range comparison.

Example 1

In the following example, we filter the years with unemployment rate between 7.5 and 8.

unemployment_rate		
TIME	GEO	UNEMPLOYMENT_RATE
2013	Finland	8.2
2012	Finland	7.7
2011	Finland	7.8
2010	Finland	8.4
2009	Finland	NULL

The expression

```
comparison_ds := unemployment_rate between 7.5 and 8.0
```

will yield:

comparison_ds		
TIME	GEO	UNEMPLOYMENT_RATE
2013	Finland	false
2012	Finland	true
2011	Finland	true
2010	Finland	false
2009	Finland	NULL

Example 2

In the following example we use a Dataset showing the overcrowding rate in densely populated areas.

overcrowding_rate_urbanization_2011	
GEO	VALUE
Belgium	NULL
Greece	0.276
Finland	0.093
Switzerland	0.08
United Kingdom	0.089
France	0.125

overcrowding_rate_urbanization_2010			
GEO	AGE	SEX	VALUE
Belgium	Total	Total	0.06
Greece	Total	Total	0.281
Spain	Total	Total	0.06
Malta	Total	Total	0.041
Switzerland	Total	Total	NULL

2634

overcrowding_rate_urbanization_2012				
TIME	GEO	AGE	SEX	VALUE
2012	Belgium	Total	Total	0.023
2012	Greece	Total	Total	0.286
2012	Spain	Total	Total	0.064
2012	Malta	Total	Total	0.043
2012	Finland	Total	Total	0.08
2012	Switzerland	Total	Total	0.08

2635

2636 The expression:

2637 `comparison_ds := overcrowding_rate_urbanization_2011 between`
2638 `Overcrowding_rate_urbanization_2010 and`
2639 `Overcrowding_rate_urbanization_2012`

2640

2641

2642 will yield the result:

TIME	GEO	AGE	SEX	CONDITION_VALUE
2012	Belgium	Total	Total	NULL
2012	Greece	Total	Total	false
2012	Switzerland	Total	Total	NULL

2643

isnull

Component version

Syntax

```
Component<Boolean> c :=
isnull
({component=}Component<T> c_1)
```

Constraints

None

Semantics

Returns *true* if c_1 is null, *false* otherwise.

Examples

Let us assume component C from a dataset.

If C is null:

```
A := isnull(C) // A = true
```

If C is not null:

```
A := isnull(C) // A = false
```

Dataset version

Syntax

```
Dataset<?+,MeasureComponent<Boolean>+> ds_2 :=
isnull
({dataset=}Dataset<?> ds_1)
```

Constraints

None

Semantics

Checks if the rows of *ds_1* have NULL MeasureComponents. Returns a Dataset *ds_1* with the same IdentifierComponents as *ds_1*, the AttributeComponents (according to the attribute propagation rule) and a MeasureComponent<Boolean>, named "CONDITION_" concatenated to the name of the original MeasureComponent, for each MeasureComponent of *ds_1*. For each row in *ds_1* there is a row in *ds_2* with *true* or *false* as MeasureComponent stating if *ds_1* MeasureComponents are NULL or not NULL.

Parameters

- *ds_1* – the Dataset to be checked

Returns

- *ds_2* – a Dataset with the result of the check

Example

population

TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	NULL
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	NULL	Total	Total	NULL

The expression

```
ds_1 := isnull(population)
```

applied against the previous Dataset, returns:

ds_1

TIME	GEO	AGE	SEX	CONDITION
2012	Belgium	Total	Total	false
2012	Greece	Total	Total	false
2012	Spain	Total	Total	true
2012	Malta	Total	Total	false
2012	Finland	Total	Total	false
2012	NULL	Total	Total	true

exists_in, not_exists_in/in_all

Syntax

```
Dataset<?, MeasureComponent<Boolean>> ds_3 :=
Dataset<?> ds_1
[exists_in|exists_in_all|not_exists_in|not_exists_in_all]
Dataset<?> ds_2
```

Constraints

- *ds_1* and *ds_2* must have the same IdentifierComponents in name and type or, at least, the IdentifierComponents of *ds_1* (*ds_2*) must be a subset of the ones of *ds_2* (*ds_1*) (static).

Semantics

Checks that rows in the first argument have matching rows in the second argument.

Takes in input two Datasets *ds_1* and *ds_2* with the same IdentifierComponents or, at least, the IdentifierComponents of *ds_1* (*ds_2*) being a subset of the ones of *ds_2* (*ds_1*), and returns a third Dataset *ds_3* with all the IdentifierComponents of the two and one MeasureComponent<Boolean> named "CONDITION".

The MeasureComponent value in each row in *ds_*3 indicates whether a row with matching key (not) exists in the second argument for the corresponding row of the first argument.

If **all** versions are used, both the *true* and the *false* rows are kept in the result. Otherwise, only the *true* rows are kept.

Parameters

ds_1 – the operand whose IdentifierComponents have to be looked up in *ds_2*

ds_2 – the reference Dataset in which *ds_1* IdentifierComponents are looked up

Returns

ds_3 – The output Dataset with the result of the comparison

Example 1

Let us assume the following datasets:

population

TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	46818219
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	Switzerland	Total	Total	7954662

urbanization_rate

TIME	GEO	AGE	SEX	RATE
2012	Belgium	Total	Total	0.023
2012	NULL	Total	Total	0.286
2012	Spain	Total	Total	0.064

2012	Malta	Total	M	0.043
2012	Finland	NULL	Total	NULL
2012	Switzerland	Total	Total	0.08

The expression

```
ds_check := population exists_in_all urbanization_rate
```

will result into the following dataset:

TIME	GEO	AGE	SEX	CONDITION
2012	Belgium	Total	Total	true
2012	Greece	Total	Total	false
2012	Spain	Total	Total	true
2012	Malta	Total	Total	false
2012	Finland	Total	Total	false
2012	Switzerland	Total	Total	true

In the following examples, Ki are Identifier Components, Mi are Measure Components

Example 2

```
R := C1 exists_in C2
```

C1		
K1	K2	M1
1	A	100
2	B	200
3	C	700
4	A	550
5	D	120

C2		
K1	K2	M1
1	A	100
2	B	200
5	D	700

R		
K1	K2	CONDITION
1	A	True
2	B	True
5	D	True

Example 3

```
R := C1 exists_in_all C2
```

C1		
K1	K2	M1
1	A	100
2	B	200
3	C	700
4	A	550
5	D	120

C2		
K1	K2	M1
1	A	100
2	B	200
5	D	700

Example 4

```
R := C1 does not_exist_in C2
```

C1		
K1	K2	M1
1	A	100
2	B	200
3	C	700
4	A	550
5	D	120

C2		
K1	K2	M1
1	A	100
2	B	200
5	D	700

R		
K1	K2	CONDITION
1	A	True
2	B	True
3	C	False
4	A	False
5	D	True

R		
K1	K2	CONDITION
3	C	True
4	A	True

Example 5

```
R := C1 not_exists_in_all C2-
```

C1		
K1	K2	M1
1	A	100
2	B	200
3	C	700
4	A	550
5	D	120

C2		
K1	K2	M1
1	A	100
2	B	200
5	D	700

R		
K1	K2	CONDITION
1	A	False
2	B	False
3	C	True
4	A	True
5	D	False

Example 6

```
R := C1 not_exists_in_all C2
```

C1		
K1	K2	M1
1	A	100
2	B	200

3	C	700
4	A	550
5	D	120

C2			
K1	K2	K3	M1
1	A	X	100
2	B	Y	200
5	D	Z	700
5	D	K	1500

R		
K1	K2	CONDITION
1	A	False
2	B	False
3	C	True
4	A	True
5	D	False

check

Syntax

```
Dataset<?+, MeasureComponent<Boolean>> ds_2 :=
check(
{dataset=}Dataset<?+, MeasureComponent<Boolean>> ds_1
  {, imbalance(Dataset<Numeric> imbalance)}
  {, erlevel(Dataset<Numeric> errorlevel)}
  {, errorCode(Constant<String> errorCode)}
  {, threshold(Constant<Numeric> threshold)}
  {, all})
```

Constraints

- *ds_1* must have exactly one MeasureComponent<Boolean> (static).

Semantics

It is the validation operator: it allows to enrich and control a comparison condition, expressed as a Boolean Dataset, with supplementary properties, essential for a validation.

Imbalance, *errorlevel* and *errorCode* are extra indicators for the validation, while *threshold* allows to relax the comparison adding a tolerance.

Specifically, it takes as input a Dataset *ds_1* having exactly one MeasureComponent<Boolean> and returns a Dataset with the same IdentifierComponents, MeasureComponents and AttributeComponents (according to attribute propagation rules) with at most three supplementary MeasureComponents whose values are calculated row-wise:

- *imbalance* is a Dataset<Numeric> expression typically over the Datasets possibly used in calculating *ds_1*, calculating in its single MeasureComponent<Numeric> the imbalance between an expected value for each row in *ds_1*. The MeasureComponent is named IMBALANCE. This value is returned as 0 if the row has *true* in the MeasureComponent, otherwise it is calculated.
- *errorlevel* is Dataset<Numeric> expression typically over the Datasets possibly used in calculating *ds_1,* calculating an in its single MeasureComponent<Numeric> an error level for each row in *ds_1*. This value is returned as 0 if the row has *true* in the MeasureComponent, otherwise it is calculated. The errorlevel is typically used by the following statements to behave accordingly. The MeasureComponent is named ERRORLEVEL.
- *errorCode* the error code of a validation error to be returned when the row has *false* in the MeasureComponent. In the other cases it is NULL. The MeasureComponent storing this value is named ERRORCODE.

The **all** flag allows to specify that both *true* and *false* rows have to be kept in the output. If it is not present, only *true* rows are kept.

Parameters

ds_1 – the Dataset with the check

imbalance – the Dataset<Numeric> calculating the imbalance

errorlevel – the Dataset<Numeric> calculating an error level for the check

errorCode – an error code to be used when the check is false.

threshold - a tolerance threshold to be used when reporting the outcome of the validation. This parameter allows relaxing the check enforced in *ds_1*. For every row where the MeasureComponent is false (hence the validation would fail), if *imbalance* < *threshold*, then such MeasureComponent is turned into true and the validation succeeds.

Returns

ds_2 – the output Dataset, which is a copy of the input one plus the IMBALANCE, ERRORLEVEL, ERRORMESSAGE Components.

Example 1

```
R := check( C1 > C2, imbalance (abs(C1-C2)) , erlevel (8) )
```

C1		
K1	K2	M1
1	A	100
2	B	200

C2		
K1	K2	M1
1	A	50
2	B	260

R				
K1	K2	CONDITION	*IMBALANCE*	*ERRORLEVEL*
1	A	true	0	0

Example 2

```
R := check( C1 > C2, imbalance (abs(C1-C2)) , erlevel (8) , all)
```

C1		
K1	K2	M1
1	A	100
2	B	200

C2		
K1	K2	M1
1	A	50
2	B	260

R				
K1	K2	CONDITION	*IMBALANCE*	*ERRORLEVEL*
1	A	true	0	0
2	B	false	60	8

Example 3

```
R := check(C1 > =C2 * 0.1, imbalance (C1), erlevel (C1 * 0.1), all)
```

C1		
K1	K2	M1
1	A	100
2	B	200

R					
K1	K2	K3	CONDITION	*IMBALANCE*	*ERRORLEVEL*
1	A	X	true	0	0
2	B	Y	true	0	0
2	B	Z	true	0	0

C2			
K1	K2	K3	M1
1	A	X	1000
2	B	Y	200
2	B	Z	350

Example 4

```
R := check(C1 > =C2 * 0.1, imbalance (C1), erlevel (C1 * 0.1),
threshold(210), all)
```

C1		
K1	K2	M1
1	A	100
2	B	200

C2			
K1	K2	K3	M1
1	A	X	1000
2	B	Y	200
2	B	Z	350

R					
K1	K2	K3	CONDITION	*IMBALANCE*	*ERRORLEVEL*
1	A	X	true	0	0
2	B	Y	true	200	20
2	B	Z	true	0	0

match_characters

Syntax

```
Dataset<?+, MeasureComponent<Boolean>+> ds_2 :=
match_characters
({dataset=}Dataset<?+, MeasureComponent<String>> ds_1,
{pattern=}Constant<String> pattern {, all})
```

Constraints

- *ds_1* must have only one MeasureComponents<String> (static).
- *pattern* must be a regular expression according to POSIX extended standard (http://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html) (static).

Semantics

It takes in input a Dataset *ds_1* and a regular expression pattern *pattern* and returns a second Dataset *ds_2*, having the same IdentifierComponents, AttributeComponents (according to attribute propagation rule) and a MeasureComponent<Boolean> for each MeasureComponent<String> in *ds_1*.

These MeasureComponents<Boolean> are named with the name of the MeasureComponent in *ds_1* concatenated with the suffix "_CONDITION".

The rows of *ds_1* are copied into *ds_1*; the MeasureComponent<Boolean> will have *true* if the respective in *ds_1* matches with the *pattern*, *false* otherwise.

The **all** flag allows to specify that both *true* and *false* rows have to be kept in the output. If it is not present, only *true* rows are kept.

Parameters

ds_1 – the Dataset to be checked against the *pattern*

pattern – a regular expression pattern according to POSIX extended standard (http://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html)

Returns

ds_2 – a Dataset with the checked condition

Example

In this validation rule example we check if all values for the Year IdentifierComponent are composed of numeric digits. The result dataset has the same Components as the argument dataset and it contains a single MeasureComponent, holding the check.

```
population
```

TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012A	Greece	Total	Total	11123034
2012	Spain	Total	Total	46818219
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012C	Switzerland	Total	Total	7954662

The expression:

```
ds_r := match_characters(population, TIME,"[123456789,]",all))
```

will yield the result:

TIME	GEO	AGE	SEX	POPULATION_CONDITION
2012	Belgium	Total	Total	true
2012A	Greece	Total	Total	false
2012	Spain	Total	Total	true
2012	Malta	Total	Total	true
2012	Finland	Total	Total	true
2012C	Switzerland	Total	Total	false

match_values

Syntax

```
Dataset<?+,MeasureComponent<Boolean>+> ds_2 :=
match_values
({dataset=}Dataset<?+,MeasureComponent<String>> ds_1,
{values=}
[ValueDomainSubset<?> | ConstantSet< <String>] vds {, all})
```

Constraints

- `ds_1` must have only one *MeasureComponent*<String> (static).

Semantics

It takes in input a *Dataset* `ds_1` and a Value Domain Subset (or a collection of allowed value codes in the form of a Set of Strings) `vds` and returns a second *Dataset* `ds_2`, having the same *IdentifierComponent*s, *AttributeComponent*s (according to attribute propagation rule) and a *MeasureComponent*<Boolean> for each *MeasureComponent*<String> in `ds_1`. These *MeasureComponent*s<Boolean> are named "`CONDITION_`" concatenated to the name of the *MeasureComponent* in `ds_1`.

The rows of `ds_1` are copied into `ds_1`; the *MeasureComponent*<Boolean> will have `true` if the respective in `ds_1` has a value that is contained within the Value Domain Subset, otherwise `false`.

The `all` flag allows specifying that both `true` and `false` rows have to be kept in the output. If it is not present, only `true` rows are kept.

Parameters

`ds_1` – the Dataset to be checked against the *pattern*

`vds` – the Value Domain Subset to be looked up or a collection of allowed codes in the form of a Set of Strings

Returns

`ds_2` – a Dataset with the checked condition

Example

In this validation rule example we check if values in the Dataset *MeasureComponent* "`SEX`" are taken from a specific `Value Domain Subset`.

unemployment

TIME	GEO	SEX	CITIZEN	AGE	*WSTATUS*
2008	IT	T	TOTAL	Y_LT15	UNE
2008	IT	T	TOTAL	Y15-19	UNE
2008	IT	T	TOTAL	Y20-64	UNE
2008	IT	T	TOTAL	Y_GE65	UNE
2008	IT	M	TOTAL	Y15-19	UNE
2008	IT	M	TOTAL	Y20-64	UNE
2008	IT	M	TOTAL	Y_GE65	UNE
2008	IT	F	TOTAL	Y15-19	UNE
2008	IT	F	TOTAL	Y20-64	UNE
2008	IT	F	TOTAL	Y_GE65	UNE
2008	NULL	F	TOTAL	Y_GE65	UNE

The expression:

```
r:= match_values ( unemployment#SEX, ("M","F"), all)
```

will result into the following dataset:

TIME	GEO	SEX	CITIZEN	AGE	*WSTATUS*	**CONDITION_SEX**
2008	IT	T	TOTAL	Y_LT15	UNE	false
2008	IT	T	TOTAL	Y15-19	UNE	false
2008	IT	T	TOTAL	Y20-64	UNE	false
2008	IT	T	TOTAL	Y_GE65	UNE	false
2008	IT	M	TOTAL	Y15-19	UNE	true
2008	IT	M	TOTAL	Y20-64	UNE	true
2008	IT	M	TOTAL	Y_GE65	UNE	true
2008	IT	F	TOTAL	Y15-19	UNE	true
2008	IT	F	TOTAL	Y20-64	UNE	true
2008	IT	F	TOTAL	Y_GE65	UNE	true

2980

Conditional operator

VTL allows conditional evaluation via the if-then-else operator.

if-then-else

Component version

Syntax

```
Component<T> c :=
if Component<Boolean> cond_1
then Component<T> c_1
{elseif Component<Boolean> cond_2
then Component<T> c_2}*
else Component<T> c_3
```

Constraints

None

Semantics

If *cond_1* evaluates to true returns c_1; alternatively, if cond_2 evaluates to true returns *c_2*. Otherwise returns c_3.

Examples

Expressions evaluating to Component types are typically used to calculate Measure Components or evaluate filters.

Some examples follow:

```
K1 + K2 < K3
K1 – K2 > 5.5
K2 + round(K2, 3)
K1 > 3 and k1 < 5
if k1>4 then K2 else K3 + 3
K1 in (1,2,3,4) and K3 not in ('a','b','c')
```

Dataset version

Syntax

```
Dataset<T> ds_4 :=
if Dataset<?+,MeasureComponent<Boolean>> ds_cond_1
then Dataset<T> ds_1
{elseif Dataset<?+,MeasureComponent<Boolean>> ds_cond_2
then Dataset<T> ds_2}*
```

```
else Dataset<T> ds_3
```

Constraints

- *ds_cond_1*, *ds_1*, *ds_cond_2*, *ds_2* and *ds_3* must have the same IdentifierComponents, in name and type
- *ds_cond_1* and *ds_cond_2* must have only one MeasureComponent<Boolean>
- *ds_1*, *ds_2* and *ds_3* must have the same IdentifierComponents and MeasureComponents.

Semantics

It takes in input a number of condition Datasets *ds_cond_1*, having exactly one MeasureComponent<Boolean> and, for each of them a Dataset *ds* to return in case of positive evaluation (**then**) of the condition. Besides it takes in input a default case (**else**) Dataset to be returned if all the previous conditions evaluate to false.

All the involved Datasets must have the same IdentifierComponents, the returned Datasets must have the same MeasureComponents in name and type.

Starting from *ds_cond_1*, for each row, if the MeasureComponent is *true*, it looks up in the corresponding Datasets (*ds_1*) all the rows for the corresponding values of the IdentifierComponents and returns them in *ds_4*. If the MeasureComponent is *false*, it looks up in the following *elseif* Dataset (*ds_cond_2*) for the corresponding values of the IdentifierComponents. If no row is found, the elaboration skips to the next row of *ds_cond_*1. If any rows are found and *ds_cond_2* is *true* for them, the corresponding **then** Dataset (*ds_2*) is returned; otherwise, the evaluation continues likewise, until the else part is reached (in case every previous conditional Datasets evaluate to *false*) and, if any matching rows are found in *ds_3* they are returned. Then the elaboration is repeated for all the rows in *ds_cond_1*.

Parameters

ds_cond_1 – the condition Dataset to be evaluated. It drives the evaluation.

ds_1 - the Dataset containing the rows to be returned where *ds_cond_1* evaluates to *true* (the **then** part)

ds_cond_2 – the following condition Datasets to be evaluated where *ds_cond_1* evaluates to *false* (the **elseif** part).

ds_2 – the Dataset containing the rows to be returned where *ds_cond_2* evaluates to *true* (the **then** part of the **elseif**).

ds_3 – the Dataset containing the rows to be returned where all the previous condition Datasets evaluate to *false* (the **else** part).

Returns

ds_4 – the Dataset to be returned, which is built according to the described semantics.

Example

Consider the following Datasets.

population				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	M	5451780
2012	Belgium	Total	F	5643070
2012	Greece	Total	M	5449803
2012	Greece	Total	F	5673231
2012	Spain	Total	M	23099012
2012	Spain	Total	F	23719207
2012	France	Total	M	31616281
2012	France	Total	F	33671580
2012	Italy	Total	M	28726599
2012	Italy	Total	F	30667608
2012	Austria	Total	M	NULL
2012	Austria	Total	F	NULL

3053

unemp_rates_1				
TIME	GEO	AGE	SEX	RATE
2012	Spain	Total	F	25.8
2012	France	Total	F	NULL
2012	Italy	Total	F	20.9
2012	Austria	Total	M	6.3

3054

unemp_rates_2				
TIME	GEO	AGE	SEX	RATE
2012	Belgium	Total	M	0.12
2012	Greece	Total	M	22.5
2012	Spain	Total	M	23.7
2012	Austria	Total	F	NULL

3055

3056 The expression:

3057 `ds_1 := if (population#SEX="F")#CONDITION`

3058 `then unemp_rates_1`

```
else unemp_rates_2
```

will yield:

TIME	GEO	AGE	SEX	RATE
2012	Belgium	Total	M	0.12
2012	Greece	Total	M	22.5
2012	Spain	Total	M	23.7
2012	Spain	Total	F	25.8
2012	France	Total	F	NULL
2012	Italy	Total	F	20.9

`population#SEX` allows to consider `SEX` into the only *MeasureComponent*, which is compared with "`F`" by the operator "`=`". Correctly, it acts on the only *MeasureComponent* as `POPULATION` is temporarily not considered. The comparison returns `CONDITION` as the only *MeasureComponent*<Boolean>.

Thus, as the `if` operators requires a Dataset with a single *MeasureComponent*<Boolean>, `#` is applied again in order to isolate `CONDITION_SEX` as the only *MeasureComponent*<Boolean>.

nvl

Component version

Syntax

```
Component<T> c :=
nvl
({component=}Component<T> c_1, {constant=}Constant<T> const)
```

Constraints

None

Semantics

Returns c_1 if c_1 is not null, *const* otherwise.

Examples

Let us assume component C from a dataset.

If C is null:

```
A := nvl(C, 5) // A = 5
```

If COMPX is not null and equal to 10:

```
A := nvl(C, 5) // A = 10
```

Dataset version

Syntax

```
Datset<?+, MeasureComponent<T>+> ds_2 :=
nvl
({dataset=}Dataset<?+, MeasureComponent<T>+> ds_1,
{constant=} Constant<T> rep_value)
```

Constraints

All the MeasureComponents of *ds_1* must be of the same type (static).

Semantics

Takes in input a Dataset *ds_1* and a constant *rep_value* of the same type as the MeasureComponents of *ds_1*. It returns a second Dataset *ds_2* with the same IdentifierComponents of *ds_1*, the AttributeComponents (according to attribute propagation rule) and all the MeasureComponents. For each row of *ds_1* returns a row where the NULL values of the MeasureComponents are replaced with *rep_value*.

Parameters

ds_1 – the Dataset to amend from NULL values

rep_value – the value to be used to replace NULL values

Returns

ds_2 a Dataset without NULL values for MeasureComponents

Example

population				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	NULL
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	NULL	Total	Total	NULL

The expression

```
ds_1 := nvl(population,0)
```

applied against the previous Dataset, returns:

TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	0
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	NULL	Total	Total	0

Clauses (postfix operators)

Clauses are operators adopting a postfix syntax, used to define expressions involving only Components. In particular:

rename – allows to rename an existing Component

filter – filters the rows in a Dataset according to a Component<Boolean> expression

keep – chooses which Components to keep in or to drop from a Dataset

calc – allows to calculate and add a new Component, alter the role of an existing one, calculate and modify the value of an existing one

attrcalc – allows to define explicit calculation formulas for attributes

aggregate – allows to perform data aggregations

rename

Syntax

```
Dataset<?> ds_2 := Dataset<?> ds_1
[rename Component<?> k as Constant<String> compName
                          {role=[MEASURE|IDENTIFIER|ATTRIBUTE]}

 {, Component<?> k as Constant<String> compName
                          {role=[MEASURE|IDENTIFIER|ATTRIBUTE]}
}*]
```

Constraints

- k is a Component expression that can have only Component literals of *ds_1* (static).
- *role* can be one of : “MEASURE”, “IDENTIFIER”, “ATTRIBUTE” (static).

Semantics

It allows to change name and role of a Component of a Dataset.

Takes in input a Dataset *ds_1* and returns a new Dataset *ds_2* with the same IdentifierComponents, MeasureComponents and Attributes according to attribute propagation rules.

It renames each Component in *ds_1* that is mentioned in the operator with the new name given in *compName* and the role given in *role* variable. If *role* variable is not specified, the role is left unmodified. All the rows in *ds_1* are copied into *ds_2*.

Parameters

ds_2 – the input Dataset

k – each Component to rename

compName – the new name for each Component

role – the new role for each Component

Returns

ds_1 – the Dataset with renamed Components

Example

```
ds_2 := ds_1[rename M1 as "I1" role IDENTIFIER]
```

The expression above renames MeasureComponent M1 into I1 and alters its role to IdentifierComponent.

filter

Syntax

```
Dataset<?> ds_1 :=
Dataset<?> ds_2
[filter Component<Boolean> f]
```

Constraints

- f is a Component expression over the Components of *ds_1* (static).

Description

Performs row-wise filtering. It takes in input a Dataset *ds*_2 and returns a new one with the same IdentifierComponents, MeasureComponents and AttributeComponents (according to attribute propagation rules).

The Component<Boolean> expression *f* is evaluated for each row of *ds_1*. A row is returned in the result, only if the corresponding evaluation of *f* yields *true*.

Parameters

- *ds_2* – the Dataset to filter
- *f* – the Component expression defining the filter

Returns

- *ds_1* – the filtered Dataset

Example

In this example we perform filtering of a Dataset with population data and will retain only the rows about females.

population1

SEX	AGE	GEO	TIME	POPULATION
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

The expression:

```
population1[filter SEX = "F"]
```

will yield the result:

SEX	AGE	GEO	TIME	POPULATION
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

keep

Syntax

```
Dataset<?> ds_1 :=
Dataset<?> ds_2
[keep Component<Boolean> k {, Component<Boolean> k }*]
```

Constraints

- f is a Component expression over the Components of *ds_1* containing only Component literals (i.e. names of the Components of *ds_1*) (static).

Semantics

Selects a subset of the IdentifierComponent and MeasureComponent, performing a kind of relational projection.

It takes in input a Dataset *ds_2* and returns a new one with the IdentifierComponents and MeasureComponents selected in *k*.

Special care must be paid in relation to how AttributeComponents are propagated. Keep clause overrides virality, so that an AttributeComponent will be present in the output if and only if it is present as a Component expression in keep. In practice, if an AttributeComponent is not mentioned in keep it will be dropped, otherwise it will be kept independently of its virality.

If an AttributeComponent is kept in the result, its virality is not altered.

Parameters

ds_2 – the input Dataset

k – each Component to be kept in the output

Returns

ds_1 – the output Dataset

Example

In this example we apply keep clause on the following Dataset.

population1

SEX	AGE	GEO	TIME	POPULATION
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

Thus, the expression:

```
ds_1 := population1[keep(SEX, GEO, POPULATION)]
```

will yield the result:

SEX	GEO	POPULATION
M	BE	970428
M	BE	3678355
M	BE	838653
F	BE	927644
F	BE	3625561
F	BE	1121001
M	UK	5757444
M	UK	20748657
M	UK	4917238
F	UK	5488356
F	UK	20915924
F	UK	6068452

calc

Syntax

```
Dataset<?> ds_2 :=
Dataset<?> ds_1
[calc Component<?> k as Constant<String> compName}
                              {role [MEASURE|IDENTIFIER|ATTRIBUTE]
                         {viral}}

 {, Component<?> k as Constant<String> compName}
                              {role [MEASURE|IDENTIFIER|ATTRIBUTE]
                         {viral}}
}*]
```

Constraints

- *k* is a Component expression on *ds_1* components (static).
- *role* can be one of : “MEASURE”, “IDENTIFIER”, “ATTRIBUTE”.

Semantics

Takes in input a Dataset *ds_1* and returns a new Dataset *ds_2* with the same IdentifierComponents, MeasureComponents.

It adds to *ds_2* a Component for each Component expression *k* specified in the clause, calculating it row-wise according to the Component expression.

The added Component is named *compName* and is given a role (IdentifierComponent, MeasureComponent or AttributeComponent) according to *role* Constant. If the role is omitted, "MEASURE" is implied. If any *k* coincides with the name of an existing Component in *ds_1* (even with different type), the calculated one replaces the former, in name, value and type.

Special care must be paid to the handling of AttributeComponents. If a Component expression has the same name as an existing AttributeComponent, the previous one is overridden, independently of its virality. In this sense, **calc** clause overrides virality. On the other hand, if no AttributeComponent expressions override an existing Component, it will be kept in the result, only if viral, with unaltered virality. In general, when an AttributeComponent is calculated, its virality can be set by the use of keyword **viral**. If it is omitted, the AttributeComponent is non viral by default. As a special case of this, a **calc** can be also used simply to alter the virality of an AttributeComponent.

Parameters

ds_2 – the input Dataset

k – each Component expression to be calculated

role – each role to be assigned to calculated Components

Returns

ds_1 – the output Dataset

Example

```
ds_2 := ds_1[calc M1*M2/3 as "M4" role MEASURE]
```

The expression above calculates a MeasureComponet by combining the ones of the involved Datasets.

```
ds_2 := ds_1[calc M1-1 as "M1" role MEASURE, M1+M2 as M2, if M2>3
then M2 else M3 as M3]
```

Like the preceeding example, but with a conditional logic.

```
ds_2 := ds_1[calc A1 + A2 as "A3" role ATTRIBUTE viral]
```

The expression above calculates AttributeComponent A3 as a combination of A1 and A2.

attrcalc

The default attribute propagation rule based on weights can be altered with user-defined formulas. It is possible to associate a calculation formula for a specific attribute in an expression with an *attrcalc* clause.

Syntax

```
Dataset<?,AttributeComponent<?>+> ds_2 :=
Dataset<?> ds_1
```

```
[attrcalc AttributeComponent<?> k as Constant<String> compName
{viral}
 {, AttributeComponent<?> k as Constant<String> compName {viral}}*
]
```

Constraints

- k is a Component expression on *ds_1* components or on components used to calculate *ds_1* properly qualified (static).

Semantics

Takes in input a Dataset *ds_1* (which, in general, can be a complex expression of type Dataset) and returns a new Dataset *ds_2* with the same IdentifierComponents, MeasureComponents.

Ds_2 has an Attribute Component named compName for each AttributeComponent<?> expression *k* specified in the clause. *k* is a component expression, evaluated row-wise.

Special care must be paid to the handling of AttributeComponents. If a Component expression has the same name as an existing AttributeComponent, the previous one is overridden, independently of its virality. In this sense, **attrcalc** clause overrides virality. On the other hand, if no AttributeComponent expressions override an existing Component, it will be kept in the result, only if viral, with unaltered virality. In general, when an AttributeComponent is calculated, its virality can be set by the use of keyword **viral**. If it is omitted, the AttributeComponent is non viral by default. As a special case of this, an **attrcalc** can be also used simply to alter the virality of an AttributeComponent.

Parameters

ds_1 – the input Dataset

k – each Component expression to be calculated

compName – the name of the AttributeComponent to be calculated

Returns

ds_1 – the output Dataset with the calculated attribute component

Example 1

```
ds_2 := ds_1[attrcalc QUALITY+1 as QUALITY]
```

Calculates ds_2, keeping the QUALITY Attribute Component in ds_1, but adding 1 to its value.

ds_1

K1	K2	M1	*QUALITY*
1	A	1	1
2	B	3	2
3	C	5	3

ds_2

K1	K2	M1	*QUALITY*

1	A	1	2
2	B	3	3
3	C	5	4

3307

3308 Example 2

3309 `ds_r := (ds_1 + ds_2)[attrcalc ds_1#QUALITY + ds_2#QUALITY as QUALITY]`

3310 The expression calculates ds_r as the sum of Datasets *ds_1* and *ds_2*. Besides, it calculates the
3311 QUALITY attribute as the sum of the two.

3312 **ds_1**

K1	K2	M1	*QUALITY*
1	A	1	1
2	B	3	2
3	C	5	3

3313 **ds_2**

K1	K2	M1	*QUALITY*
1	A	6	2
2	B	7	3
3	C	8	4

3314 **ds_r**

K1	K2	M1	*QUALITY*
1	A	7	3
2	B	10	5
3	C	13	7

3315

3316 Example 3

```
3317 ds_r := (ds_1 + ds_2)
3318 [
3319   attrcalc if ds_1#QUALITY=”A” and ds#2QUALITY=”B” then “C”
3320         elseif ds_1#QUALITY=”K” and ds#2QUALITY=”K” then “M”
3321          else “Z” AS AGGREGATED_QUALITY
3322 ]
```

3323 The expression calculates the AGGREGATED_QUALITY attribute as a combination of QUALITY
3324 Attribute Components of the operands according to a decision rule.

3325 In particular, if *ds_1* quality is “A” and *ds_2* quality is “B”, then the AGGREGATED_QUALITY will
3326 be “C”. Else, if *ds_1* quality is “K” and *ds_2* quality is “K”, then “M” is returned. Otherwise “Z” is
3327 the AGGREGATED_QUALITY value.

ds_1			
K1	K2	M1	QUALITY
1	A	1	A
2	B	3	B
3	C	5	K

ds_2			
K1	K2	M1	QUALITY
1	A	6	B
2	B	7	A
3	C	8	K

ds_r			
K1	K2	M1	QUALITY
1	A	7	C
2	B	10	Z
3	C	13	M

Example 4

```
ds_r := (ds_1 + ds_2)[attrcalc ds_1#QUALITY_1 + ds_2#QUALITY_1 as
QUALITY_1, ds_2#QUALITY_2 as QUALITY_2]
```

The expression sums two multi-measure Datasets and calculates two Attribute Components with different formulas: QUALITY_1 is the sum of *ds_1*#QUALITY_1 and *ds_2*#QUALITY_1, while QUALITY_2 is simply copied from *ds_2.*

ds_1					
K1	K2	M1	M2	QUALITY_1	QUALITY_2
1	A	1	5	1	2
2	B	3	3	2	7
3	C	5	1	3	4

ds_2					
K1	K2	M1	M2	*QUALITY_1*	*QUALITY_2*
1	A	6	1	2	1
2	B	7	1	3	3
3	C	8	1	4	1

ds_r					
K1	K2	M1	M2	*QUALITY_1*	*QUALITY_2*
1	A	7	6	3	1
2	B	10	4	5	3
3	C	13	2	7	1

aggregate

Syntax

```
Dataset<?> ds_2 :=
Dataset ds_1<?+, MeasureComponent<Numeric>+>
[aggregate[sum|avg|median|count|count_distinct|min|max](
{include NULLS} MeasureComponent<Numeric> aggrPart),
{[sum|avg|median|count|count_distinct|min|max](
{include NULLS} MeasureComponent<Numeric> aggrPart )}*]
```

Constraints

- *ds_1* must have only Numeric MeasureComponents (static).
- If the aggregation clause is present, for each MeasureComponent, there must be an aggregation function (static).
- *aggrPart* can contain only Component names (static).
- # clause cannot be used (static).

Description

It takes in input a Dataset *ds_1* and applies an aggregation function to each MeasureComponent, grouping by the values of the IdentifierComponents. It returns a Dataset with the same IdentifierComponents, MeasureComponents and AttributeComponents (according to the attribute propagation rules).

Many different aggregation functions are possible:

- sum: algebraic sum of values (if the clause *include NULLS* is present and a NULL is present among the values, the aggregation evaluates to NULL. Otherwise, NULLs are considered 0 in the sum).

- avg: average of the values (if the clause *include NULLS* is present and a NULL is present among the values, the aggregation evaluates to NULL. Otherwise, NULLs are not considered in the average).
- median: calculates the median of the involved values (if the clause *include NULLS* is present and a NULL is present among the values, the aggregation evaluates to NULL. Otherwise, NULLs are not considered in the average).
- count: counts the values (if the clause *include NULLS* is present, NULLS are considered in the count, otherwise they are excluded).
- count_distinct: counts the distinct values, excluding the duplicates (if the clause *include NULLS* is present, NULLS are considered in the count, otherwise they are excluded. Notice that if NULLS are considered, each NULL contributes to the count).
- min: calculates the minimum value (if the clause *include NULLS* is present, NULL is returned only if it is the only value to aggregate).
- max: calculates the maximum value (if the clause *include NULLS* is present, NULL is returned only if it is the only value to aggregate).

Parameters

ds_1 – the Dataset to aggregate

aggrPart – the MeasureComponents to aggregate on

Returns

ds_2 – the aggregated Dataset

Example 1

Consider the following Dataset:

Population 1

SEX	AGE	GEO	TIME	POPULATION
M	Y_LT15	BE	2013	90
M	Y15-64	BE	2013	100
M	Y_GE65	BE	2013	150
F	Y_LT15	BE	2013	130
F	Y15-64	BE	2013	120
F	Y_GE65	BE	2013	110
M	Y_LT15	UK	2013	70
M	Y15-64	UK	2013	80
M	Y_GE65	UK	2013	120
F	Y_LT15	UK	2013	40
F	Y15-64	UK	2013	400
F	Y_GE65	UK	2013	500

The expression

```
ds_1 := population1[keep[SEX,GEO,TIME,POPULATION][aggregate
sum(POPULATION)]
```

computes the total population by country, sex and reference year, i.e. we aggregate over age classes. It yields the following result:

the result would then be:

SEX	GEO	TIME	POPULATION
M	BE	2013	340
F	BE	2013	360
M	UK	2013	270
F	UK	2013	940

Example 2

In this example we calculate the average in Dataset datasetA, with microdata about the monthly income of individuals.

datasetA

PERS_ID	YEAR	GEO	SEX	INCOME
DE001	2011	DE	M	2340
DE002	2011	DE	M	1850
DE003	2011	DE	M	2170
DE004	2011	DE	M	0
DE005	2011	DE	F	2200
DE006	2011	DE	F	1850
DE007	2011	DE	F	1840
DE008	2011	DE	F	1680
FR001	2011	FR	M	1150
FR002	2011	FR	M	3450
FR003	2011	FR	M	2360
FR004	2011	FR	M	1830
FR005	2011	FR	F	1940
FR006	2011	FR	F	1150
FR007	2011	FR	F	NULL

```
ds_1 := datasetA[keep(GEO,SEX,INCOME)][aggregate avg(INCOME)]
```

ds_1		
GEO	SEX	INCOME
DE	M	1590.0
DE	F	1892.5
FR	M	2197.5
FR	F	1545.0

It is easy to see in this example that rows with NULL values are ignored in the calculation of the mean for consistency with other operators.

Example 3

In this example we count the distinct entries of the DatasetB by YEAR, GEO, SEX.

DatasetB

PERS_ID	YEAR	GEO	SEX	OCCUPATION[3]	INCOME
DE001	2011	DE	M	1	2340
DE002	2011	DE	M	4	1850
DE003	2011	DE	M	2	2170
DE004	2011	DE	M	5	0
DE005	2011	DE	F	NULL	2200
DE006	2011	DE	F	8	1851
DE007	2011	DE	F	5	1840
DE008	2011	DE	F	2	1680
FR001	2011	FR	M	5	1153
FR002	2011	FR	M	1	3450
FR003	2011	FR	M	1	2360
FR004	2011	FR	M	4	1830
FR005	2011	NULL	F	8	1940
FR006	2011	FR	F	5	1150
FR007	2011	FR	F	7	NULL
FR008	2011	NULL	F	NULL	NULL

[3] The occupation column is coded according to ISCO-08 one digit.

The expression

```
DatasetB[keep(YEAR, GEO, SEX, OCCUPATION)][aggregate count
OCCUPATION]
```

would yield the result:

YEAR	GEO	SEX	COUNT
2011	DE	M	4
2011	DE	F	4
2011	FR	M	4
2011	NULL	F	2
2011	FR	F	2

Example 4

In this example we count the distinct entries of the DatasetB by YEAR and GEO.

DatasetB

PERS_ID	YEAR	GEO	SEX	OCCUPATION[4]	INCOME
DE001	2011	DE	M	1	2340
DE002	2011	DE	M	4	1850
DE003	2011	DE	M	2	2170
DE004	2011	DE	M	5	0
DE006	2011	DE	F	8	1851
DE007	2011	DE	F	5	1840
DE008	2011	DE	F	2	1680
FR001	2011	FR	M	5	1153
FR002	2011	FR	M	1	3450
FR003	2011	FR	M	1	2360
FR004	2011	FR	M	4	1830
FR006	2011	FR	F	5	1150
FR007	2011	FR	F	7	NULL

The expression:

```
ds_1 := DatasetB[keep(YEAR, GEO, OCCUPATION)][aggregate
count(OCCUPATION)]
```

[4] The occupation column is coded according to ISCO-08 one digit.

would yield the result:

YEAR	GEO	OCCUPATION
2011	DE	5
2011	FR	4

Note that despite there are 8 rows for DE only 5 are counted in the result because there are five distinct values and the NULL is ignored in the result.

Example 5

Let us start again from DatasetA in example 2.

The expression:

```
ds_1 := datasetA[keep(GEO,SEX,INCOME)][aggregate min(INCOME)]
```

will yield the result:

ds_1

GEO	SEX	INCOME
DE	M	0.0
DE	F	1680.0
FR	M	1150.0
FR	F	1150.0

Equally, the expression:

```
ds_1 := datasetA[keep(GEO,SEX,INCOME)][aggregate max(INCOME)]
```

will yield the result:

ds_1

GEO	SEX	INCOME
DE	M	2340.0
DE	F	2200.0
FR	M	3450.0
FR	F	1940.0

Example 6

Consider the Dataset population1.

population1

SEX	AGE	GEO	TIME	POPULATION
M	Y_LT15	BE	2013	100
M	Y15-64	BE	2013	200
M	Y_GE65	BE	2013	150
F	Y_LT15	BE	2013	320
F	Y15-64	BE	2013	NULL
F	Y_GE65	BE	2013	340
M	Y_LT15	UK	2013	400
M	Y15-64	UK	2013	110
M	Y_GE65	UK	2013	130
F	Y_LT15	UK	2013	320
F	Y15-64	UK	2013	1000
F	Y_GE65	UK	2013	500

The expression

```
ds_1 := population1[keep[SEX,GEO,TIME,POPULATION][aggregate
sum(include NULLS POPULATION)]
```

computes the total population by country, sex and reference year. The "include NULLS" clause specifies that NULLS must be considered in the aggregation. Therefore for the female, Belgian aggregate, POPULATION is NULL.

ds_1

SEX	GEO	TIME	POPULATION
M	BE	2013	450
F	BE	2013	NULL
M	UK	2013	640
F	UK	2013	1820

Use Cases

Labour Force Survey and VTL

Introduction

In this section, the Validation and Transformation Language is used to describe some operations needed in a Labour Force Questionnaire to validate micro-data.

In the remainder of the section, the micro-data Dataset is named **DS** while the Components in the Dataset are named as the name of the question in the questionnaire and referred to with the *DS#DS_FIELD* notation.

For example in the question below, which belongs to Dataset DS:

SG18B. In what year did you come to live in Italy the first time?

the name of the question is SG18B, hence it corresponds to Component SG18B in Dataset DS1 and, as usual, can be referred to with DS#SG18B.

There are three additional fields that belong to Dataset DS, although they are not directly part of the questionnaire: **REG** and **MUN**, which are the region and the municipality where the respondent lives respectively, and **SURVEY_YEAR**, which is the year when the survey takes place. These fields are part of the Dataset but are not part of the questionnaire, in the sense that their values are not provided by the respondent, as they are implicitly defined by the place and the time of the survey.

The remainder of this section is organized as follows:

In the first part, ***incompatibility rules***, some of the rules used to validate the micro-data Dataset resulting from the survey, are defined. In these cases, operators are applied to one or more Components of the same Dataset; this means that there are no comparisons with other Datasets. In order to facilitate the interpretation of the rules, the questions of the questionnaire, used for the definition of the rules, are also reported.

Every rule includes:

- **Operators** - the VTL operators involved in the rule
- **Questions** - the questions of the questionnaire from which the rule is derived
- **Textual rule**- a textual description of rule

- **VTL rule**- the definition of the rule using the VTL language. In order to make rules more clear, they have been divided into sub-conditions. The resulting Dataset, deriving from the comination of such condition is always named **DS_R** in the example. Operators are reported in bold.

The second part, **Domains of variables**, shows some VTL rules to check variable domains. For the sake of clarity, also in this case the questions have been reported.

The third part, **Aggregation,** contains some examples of possible aggregations of micro-data to obtain a simple Dataset of macro-data (i.e. employed by sex). The examples are based on two aggregation rules used to define employed and unemployed people on the basis of micro-data Dataset.

Compatibility rules

1) **Operators**: check, not, LESS THAN, not EQUAL, and

Questions:

SG18B. In what year did you come to live in Italy the first time?

- *Year* |_|_|_|_|
- *Don't know* |0|9|9|7|

SG18F. Since when have you been living in Italy without leaving the Country for one year or more?

- *Year* |_|_|_|_|
- *Don't know* |0|9|9|7|

Textual rule: The year declared in SG18F cannot be less than of the year in which he came to Italy for the first time

The rule above results from the combination of two conditions below:

SG18B_SG18F_NotNull :

the answer to SG18F question is different from "Don't know".

- **SG18F_NotNull** := ds#SG18F <> 997

SG18F_LT_SG18B (SG18F is less than SG18B):

The year declared in SG18F is less than the year declared on SG18B

- **SG18F_LT_SG18B**:= ds[rename SG18F as "CONDITION"]#CONDITION < ds[rename SG18B as "CONDITION"]#CONDITION

VTL rule:

```
ds_r:=check( not(SG18B_SG18F_NotNull and SG18F_LT_SG18B),
        imbalance(DS#SG18B-DS#SG18F),
        erlevel(8),
        errorcode(2001),
        all)
```

where error code 2001 corresponds to 'Compatibility error for questions SG18B , SG18F'

2) **Operators**: check, not, LESS THAN, not EQUAL, SUBTRACTION, ISNULL

Questions:

SG18B. In what year did you come to live in Italy the first time?

- *Year* |_|_|_|_|
- *Don't know* |0|9|9|7|

SG18F. Since when have you been living in Italy without leaving the Country for one year or more?

- *Year* |_|_|_|_|
- *Don't know* |0|9|9|7|

SG20. Would you like to tell me how old you are?

- *Age* |_|_|_|

Textual rule: Age cannot be less than the years of residence in Italy

The rule above results from the combination of three conditions below:

SG18B_SG18F_NotNull:

The answer to the SG18B and SG18F questions is different from "Don't know".

- **SG18B_SG18F_NotNull** := ds#SG18F <> 997 **and** ds#SG18B <> 997

SG20_NotNull:

The answer to the SG20 questions is different from null.

- **SG20_NotNull** := **not isnull**(ds#SG20)

SG20_LT_RSD (SG20 Less Than Residence):

The year answered to the question SG20 is less than the residence year, obtained as difference between the year of the survey less the year of coming in Italy.

- **SG20_LT_RSD** := ds[rename SG20 as "CONDITION"]#CONDITION < ds[rename SURVEY_YEAR as "CONDITION"]#CONDITION - ds[rename SG18F as "CONDITION"]#CONDITION

VTL rule:

```
ds_r:=check(not(SG18B_SG18F_NotNull and SG20_NotNull and
        SG20_LT_RSD),
        imbalance(ds#SG20-( ds#SURVEY_YEAR - DS#SG18F )),
        erlevel(8),
        errorcode(2002),
        all)
```

3) **Operators**: if…then…else, check, not, LESS THAN, not EQUAL, or,and, ADDITION, SUBTRACTION,IN

Questions:

SG18B. In what year did you come to live in Italy for the first time?

- *Year* |_|_|_|_|
- *Don't know* |0|9|9|7|

SG18F. Since when have you been living in Italy without leaving the Country for one year or more?

- *Year* |_|_|_|_|
- *Don't know* |0|9|9|7|

SG19. Would you like to tell me your date of birth?

- *Day* |_|_| • *Don't know* |9|9|7|
- *Month* |_|_| • *Don't know* |9|9|7|
- *Year* |_|_|_|_| • *Don't know* |0|9|9|7|

SG21. Age on Sunday of reference week (calculated by system using SG19 or SG20)

- *Age* |_|_|_|

SG23. In what year did you get married?

- *Year* |_|_|_|_|
- *Don't know* |0|9|9|7|
- *Don't answer* |0|9|9|8|

Textual rule: the respondent cannot be married before the age of 14

The rule above results from the combination of four conditions below:

SG19_NotNull:

The answer to the SG19 is different from "Don't know"

- **SG19_NotNull** := ds#SG19_YEAR <> 997

SG23_NotNull:

The answer SG23 is different from "Don't know" or "Don't Answer"

- **SG23_NotNull:= not** ds#SG23 **in** (997,998)

SG23_ SG19_LT_Y14 (SG20-SG19 Less than 14 years)**:**

The difference between the date of the marriage and the date of birth is less than 14

- **SG23_SG19_LT_Y14:**= ds[rename SG23 as "CONDITION"]#CONDITION **-** ds[rename SG19_YEAR[5]as "CONDITION"]#CONDITION) **<** 14

AGE_LT_Y14 :

The difference between age at the moment of the interview and the age of marriage is less than 14

- **AGE_LT_Y14:**= ds[rename SG23 as "CONDITION"]#CONDITION **-** ds[rename SURVEY_YEAR as "CONDITION"]#CONDITION) **+** ds[rename SG21 as "CONDITION"]#CONDITION **<** 14

VTL rule:

```
ds_r:=check(not((SG19_NotNull and SG23_NotNull and
            SG23_SG19_LT_Y14))
            or
            (not( SG19_Null and SG23_NotNull and not
            AGE_LT_Y14)),
            Imbalance(if (ds#SG19_YEAR=997) then (ds#SG21)
            else (ds#SG19_YEAR),
            erlevel(8),
            errorcode(2003),
            all)
```

4) **Operators**: check, not IN, EQUAL

5 In this case the answer to the question SG19 is stored in three different fields: SG19_YEAR for the year, SG19_MONTH for the month, SG19_DAY for the day.

Questions:

SG24. Which is your highest school qualification?

- *No qualification* 1 |_|
- *Primary school* 2 |_|
- *Diploma of lower secondary education* 3 |_|
- *2-3-year upper secondary school professional qualification not allowing admission to university* 4 |_|
- *Diploma of upper secondary education (4-5-years) allowing admission to University Music conservatory certificate/National dance academy certificate* 5 |_|
- *Academy of Fine Arts Licence, Higher Institute for Applied Arts Diploma, National Academy of Drama Diploma, National Dance Academy (advanced course) Diploma, Music Conservatory (advanced course) Diploma, Institute for Interpreters and Translators qualification, Archivist, Diplomatic and Palaeography School qualification Art and Music degree* 6 |_|
- *University education undergraduate diploma* 7 |_|
- *University first degree (3years programmes)* 8 |_|
- *University second degree (2years programmes)* 9 |_|
- *University degree (4years and more old and new programmes)* 10 |_|

SG24B. Do you have a postgraduate degree or doctorate?

- *Master (university or Art and Music) first level* 1 |_|
- *Master (university or Art and Music) second level* 2 |_|
- *Post graduate degree* 3 |_|
- *Doctorate (PhD)* 4 |_|
- *No postgraduate degree* 5 |_|

Textual rule: The educational qualifications (SG24) equal to 7 or 8 does not allow to access to a post graduate degree.

The rule above results from the combination of two conditions below:

SG24_IN_7_8 (SG24 take values 7 or 8)**:**

The highest school qualification is 7 or 8

- **SG24_IN_7_8** := ds#SG24 **in** (7, 8)

SG24B_EQ_3 (SG24B is equal to 3)**:**

The interviewer has a post graduate degree

- **SG24B_EQ_3** := ds#SG24B = 3

VTL rule:

```
ds_r:=check(SG24_IN__7_8 and SG24B_EQ_3,
            erlevel(8),
            errorcode(2004),
            all)
```

5) **Operators:** substr, and, equals, IN, check

Questions :

C1. Do you work:

- **Read all the answering items**
  - *as an employee* *1* |_|

with an:

- *employer-coordinated freelance work contract (on specific project or not)* *2* |_|
- *occasional work contract* *3* |_|

as self-employed:

- *running a business* *4* |_|
- *as a professional* *5* |_|
- *as an own-account worker* *6* |_|
- *as a family worker* *7* |_|
- *as a member of producers' co-operative* *8* |_|

C12. Occupation code

[ISCO]

_____|_|_|_|_|_|

Textual rule: a self-employed (as running a business, as a professional or as an own-account worker) usually is not involved in an occupation group 4[6] (professions executive office).

The rule above results from the combination of two conditions below:

[6] The occupation group is calculated as the first character of the ISCO08 code

C1_IN_4_6 (C1 take values from 4 to 6)

The type of work is 4 or 5 or 6

- **C1_IN__4_6** := ds#C1 **in** (4, 5, 6)

C12_EQ_4 (The first character of C12 is equal to 4)

The code of the profession has length 4 digit

- **C12_LEN_NEQ_4:=** substr(DS#C12,1)=4

VTL rule:

```
ds_r:= check (not(C1_IN_4_6 and C12_LEN_NEQ_4),
        imbalance(C12),
        errorcode(2005),
        all)
```

6) **Operators:** check, and, not EQUAL,ADDITION, SUBTRACTION, MULTIPLICATION

Questions :

SG19. Would you like to tell me your date of birth?			
• *Day*	__|	• *Don't know*	|9|9|Z|
• *Month*	__|	• *Don't know*	|9|9|Z|
• *Year*	____|	• *Don't know*	|0|9|9|Z|

E2 When was the last time you worked? Please indicate the conclusion year

If s/he did not work in the reference week (B2=1) or s/he is an occasional worker or a seasonal work (B3=994,995)

- *Year* |_|_|_|_|
- *Don't know* *997* |_|

E4. In which month? **(month in which the responder stopped work)**

- *Month* |_|_|
- *Don't know* *997* |_|

E14. What is the main reason why you stopped working?

- *Retirement (contributory, non-contributory pension)* *1* |_|
- *Dismissed or made redundant (also due to bankruptcy or closing down of the company)* *2* |_|
- *Job of limited duration (including occasional and seasonal work)* *3* |_|
- *Illness, personal health problems* *4* |_|
- *Maternity, birth of a child* *5* |_|
- *Looking after children and/or other incapacitated persons* *6* |_|
- *Education or professional training* *7* |_|
- *Compulsory military or community service* *8* |_|
- *Other family reasons (except for maternity, care of children or of other people)* *9* |_|
- *Other reasons* ***(specify)***_____ *996* |_|

Textual rule: The respondent cannot have started the military service before the age of 18.

The rule above results from the combination of five conditions below:

E4_E2_NotNull

The answers E2 and E4 are different from "Don't know"

- **E4_E2_NotNull** := ds#E4<>997 and ds#E2<>997

E14_EQ_8

The answers E14 is equal to 8

- **E14_EQ_8 :=** ds#E14=8

SG19_Y_M_NotNull

The answers SG19_YEAR and SG19_MONTH are different from "Don't know"

- **SG19_Y_M_NotNull:=**ds#SG19_YEAR<>997 and ds#SG19_MONTH<>997

YEARS_MS (Years military service)

The years of the interviewer before he started the military service

- **YEARS_MS := ds[rename E2 as "CONDITION"]#CONDITION - ds[rename SG19_YEAR as "CONDITION"] #CONDITION)**

MONTHS_MS (Months military service)

The difference of months between the birth month and the month when the military service started

- **MONTHS_MS:=** ds[rename SG19_MONTH as "CONDITION"]#CONDITION - ds[rename E4 as "CONDITION"] #CONDITION)

VTL rule:

```
ds_r:=check (not((E4_E2_NotNull and E14_EQ_8 and
      SG19_Y_M_NotNull) and (YEARS_MS *12 + MONTHS_MS)<216),
      errorcode(2006),
      all)
```

7) **Operators:** check, ISNULL, EQUAL, and

Questions

I5. 12 months ago (on REFMONTH of Year before REFYEAR), were you:

- *Employed* *1* |_|
- *Unemployed looking for new employment* *2* |_|
- *Looking for first employment* *3* |_|
- *Fulfilling domestic tasks* *4* |_|
- *Student* *5* |_|
- *Retired* *6* |_|
- *Disabled for work* *7* |_|
- *In other condition* *9* |_|

C1. Do you work:

- *as an employee* *1* |_|

with an:

- *employer-coordinated freelance work contract (on specific project or not)* *2* |_|
- *occasional work contract* *3* |_|

as self-employed:

- *running a business* *4* |_|
- *as a professional* *5* |_|
- *as an own-account worker* *6* |_|
- *as a family worker* *7* |_|
- *as a member of producers' co-operative* *8* |_|

Textual rule: the respondent has declared to be unable to work then he cannot answer question C1.

The rule above results from the combination of two conditions below:

I5_EQ_7

The interviewer is “Disabled for work”

- **I5_EQ_7** := ds#I5=7

C1_IsNotNull
Question C1 is not null
➢ **C1_IsNotNull:=** (DS#C1 not ISNULL)

VTL rule:

ds_r:= **check**(not (I5_EQ_7 **and** C1_IsNotNull), **imbalance**(ds#C1), **errorcode**(2007), **all**)

Domains of the variables

1) **Operators:** check, RANGE

SG4. At first I am going to ask you some information about your household. How many people permanently live in this household, including yourself (except for possible domestic servants or tenants)?

FOR THE INTERVIEWER: Please enter the number of people living in this household on Sunday …..

- *Number of household members* |_|_|

Textual rule: the number of components cannot be more than 15

VTL rule:

ds_r := check(ds#SG4 **between** 0 **and** 15, **all**)

2) **Operators:** check, RANGE

SG20. Would you like to tell me how old you are?

[YEARBIR]

- *Age* |_|_|_|

Textual rule: the age cannot be more than 120 years

VTL rule:

ds_r := check(ds#SG20 **between** 0 **and** 120, **all**)

3) **Operators:** and, or ,equal to, greater than, less than

C31A. How many hours did you work on average in a week? Please consider the last 4 week excluding meal breaks and travel time between home and the place of work.

- *Number of hours* |_|_|_|
- *Don't know* *997* |_|_|_|

Textual rule: the usual working hours of the last 4 weeks for the main activity cannot be more than 150 or they can assume 997

VTL rule:

ds_r := check((ds#C31A>0 **and** ds#C31A<120) **or** ds#C31A=997, **all**)

4) **Operators:** EXISTS_IN

Codelist **ISCO08** represents ISCO 2008 classification.

C12. Occupation code

Coding based on the annex "Classification of Occupations"

- _____|_|_|_|_|

Textual rule: the value of the occupation code must be equal to the ISCO 2008 code

VTL rule:

ds_r := check_codelist(ds#C12 **,** ISCO08#CODE, **all**)

Aggregation

In the Labour Force Survey there are some rules needed to define if the interviewed is occupied or not. These rules refer to a subset of questions reported below:

B1. In the week from Monday ….. to Sunday, did you work for at least one hour? Please consider paid work or the work you expect to be paid for or unpaid work only if in some relative's business

- *Yes* *1*
|_|
- *No* *2*
|_|
- *Permanently disabled for work* *3*
|_|

B3. What is the main reason why you did not work in that week?

- *Lay-off (ordinary or extraordinary)* *1* |_|
- *Slack work for economical and/or technical reasons (except for lay-off)* *2* |_|
- *Labour dispute* *4* |_|
- *Bad weather* *5* |_|
- *Illness, personal health problems, accident* *6* |_|
- *Annual holidays* *7* |_|
- *Bank holidays during the week* *8* |_|
- *Variable working hours or flexitime (e.g. compensation leave)* *9* |_|
- *Vertical part-time* *10* |_|
- *Education or training not organized by the employer* *11* |_|
- *Compulsory maternity leave* *12* |_|
- *Optional maternity/paternity leave until the child's eighth birthday (parental leave)* *13* |_|
- *Family reasons (no including compulsory maternity/paternity leave and parental leave)* *14* |_|
- *No chances for more work* *15* |_|
- *Occasional worker* *994* |_|
- *Seasonal worker as employee (e.g. lifeguard, fruit gatherer, waiter in mountain during the winter, etc.)* *995* |_|
- *Other reasons* ***(specify)***_____ *996* |_|

B4. Do you work:

as an employee *1* |_|

with an:

- *employer-coordinated freelance work contract (on specific project or not)* *2* |_|
- *occasional work contract* *3* |_|

as a self-employed:

- *running a business* *4* |_|
- *as a professional* *5* |_|
- *as an own-account worker* *6* |_|
- *as a family worker* *7* |_|
- *as a member of producers' co-operative* *8* |_|

B4A.α. Do you have an employment contract?

B4A.β. Do you have an employment contract or an employer-coordinated freelance work contract?

- Yes, *an employment contract* *1* |_|
- *Yes, an employer-coordinated freelance work contract* *2* |_|
- *No* *3* |_|

B4bis.α. The employer-coordinated freelance work contract (on a specific project or not) provides the obligation for the employer to pay for the insurance contribution to the INPS? Does your employer pay for this?

B4bis.β. The occasional work contract provides the payment of a withholding tax. Does your employer pay for the withholding tax on your contract?

- *Yes* *1* |_|
- *No* *2* |_|

B6. Will this period of absence last less or more than three months from start to finish?

If s/he is in parental leave (B3=13) consider only the period of parental leave excluding the period of compulsory maternity/paternity leave (if the parental leave comes straight after the period of compulsory absence)

- *Till three months* *1* |_|
- *Three months or more* *2* |_|

B7. Is this period of absence paid at least partially?

- *Yes, 50% or more* *1* |_|
- *Yes, less than 50%* *2* |_|
- *No, it is not paid* *3* |_|

B8. Is it a leave period?

- *Yes* *1* |_|
- *No* *2* |_|
- *Don't know* *997* |_|
- *Don't answer* *998* |_|

B9. Is your job paid as family worker?

- *Yes* *1* |_|
- *No* *2* |_|

B10. In the week from Monday ….. to Sunday ….., was your work contract still in force, i.e. its expiry date was after Monday …?

- *Yes* *1* |_|
- *No* *2* |_|

B11. Is your working activity interrupted at the moment (e.g. for professional updating, for restructuring of premises, seasonal closing) or is it definitively finished?

- *Interrupted at the moment* *1* |_|
- *Finished* *2* |_|

3872

3873 VTL rules to define if the interviewed is employed if one of the following conditions is
3874 respected:

B1_B3

The answer to the question B1 is "yes" and the answer to B3 is from 5 to 9 or equal to 12

- **B1_B3:**=(DS#B1=1) or (DS#B3 IN (3,5,6,7,8,9,12))

B4_B4A_B6_B7

The answer to B4 is 1 or the answer to B4A is 1 and the answer to B6 is 1 or to B7 is 1

- **B4_B4A_B6_B7:=** ((ds#B4=1 or ds#B4A=1) and (ds#B6=1 or ds#B7=1))

B4bis_B10

The answer to the B4bis question is 1 and the answer to the question B10 is 1

- **B4bis_B10:=** (ds#B4bis=1 and ds#B10=1)

B4_B4A_B11

The answer to the B4 is from 4 to 6 or the answer to B4 is 8 and B4A is 3 and B11 is equal to 11

- **B4bis_B10:=** ((ds#B4 in (4,5,6) or (ds#B4=8 and ds#B4A=3)) and ds#B11=1)

B4_B4A_B6_B9

The answer to the B4 is 7 and the answer to B4A is 3 and B6 is 1 or B9 is equal to 1

- **B4bis_B10:=** (ds#B4=7 and ds#B4A=3 and (ds#B6=1 or ds#B9=1))

ds_r:= (B1_B3) **or** (B4_B4A_B6_B7) **or** (B4bis_B10) **or** (B4_B4A_B11) **or** (B4_B4A_B6_B9)

Aggregation: Number of occupied by region

ds_r:= **get**(DS, **Keep**(REG),

filter (Result), **count _distinct**)

1) **Aggregation:** Number of occupied by region and municipality

ds_r:= **get**(DS, **Keep**(REG,MUN),

filter (Result),

count _distinct)

2) **Aggregation:** Number of occupied by region and sex

SG11. Sex of the member

- *Male* *1*|_|
- *Female* *2*|_|

ds_r:= **get**(DS, **Keep**(REG, SG11),

filter (Result), **count _distinct**)