vivid Documentation

Release 1.2

Nicholas Marton

January 21, 2016

CHAPTER

ONE

BASIC COMPONENTS

1.1 The Interval object

interval module.

class interval.**Interval**(*inf*, *sup*)

Interval class. Intervals are over natural or real values.

Variables

- **infimum** – The infimum of the interval.
- **supremum** – The supremum of the interval.
- **type** – The type of the Interval object (int, float, or long).
- **_is_Interval** – An identifier to use in place of `type` or `isinstance`.

__and__(*other*)

Overloaded `&` operator; return the intersection of two Interval objects.

Raises **ValueError** – Intervals must overlap to take the intersection.

__contains__(*key*)

Determine if the calling Interval contains an int, float, long, or another Interval.

Parameters **key** (*int | float | long | Interval*) – The value to check for membership in the calling Interval.

__deepcopy__(*memo*)

Deepcopy an Interval object via the `copy.deepcopy` method.

__eq__(*other*)

Determine if two Interval objects are equal via the == operator.

__ge__(*other*)

Overloaded >= operator for Interval. Determine if the calling Interval is greater than another Interval; that is, the infimum of the Interval in `other` parameter is strictly less than the calling Interval's infimum, and the supremum of the interval in `other` parameter is less than the calling Interval's supremum: $(o_{inf}, (s_{inf}, o_{sup}), s_{sup})$.

__getitem__(*index*)

Retrieve the infimum or supremum of the calling Interval via indexing (e.g. `Interval[0]`).

Raises

- **IndexError** – Index must be either "`0`" or "`1`".
- **TypeError** – Index must be an `int`.

__gt__(*other*)
Overloaded > operator for Interval. Determine if the calling Interval is strictly greater than another interval; that is, the infimum of the calling Interval is strictly greater than the supremum of the Interval in `other` parameter: $(o_{inf}, o_{sup})(s_{inf}, s_{sup})$.

__hash__()
Hash implementation for set functionality of Interval objects.

__init__(*inf*, *sup*)
Construct an Interval object.

Parameters

- **inf** (*int* | *float* | *long*) – The value to use as the infimum of the Interval.
- **sup** (*int* | *float* | *long*) – The value to use as the supremum of the Interval.

Raises

- **ValueError** – The infimum must be strictly less than the supremum.
- **TypeError** – The infimum and supremum provided must be ints, floats, or longs and their types must match.

__le__(*other*)
Overloaded <= operator for Interval. Determine if the calling Interval is less than another Interval; that is, the supremum of the calling Interval is greater than the Interval in `other`'s infimum, less than the Interval in `other`'s supremum and the infimum of the calling Interval is strictly less than the infimum of the Interval in `other` parameter: $(s_{inf}, (o_{inf}, s_{sup}), o_{sup})$.

__lt__(*other*)
Overloaded < operator for Interval. Determine if the calling Interval is strictly less than another interval; that is, the supremum of the calling Interval is strictly less than the infimum of the Interval in `other` parameter: $(s_{inf}, s_{sup})(o_{inf}, o_{sup})$.

__ne__(*other*)
Determine if two Interval objects are not equal via the != operator.

__or__(*other*)
Overloaded | operator; return the union of two Interval objects.

Raises **ValueError** – Intervals must overlap to take the union.

__repr__()
Return a string representation of the Interval object.

__str__()
Return a readable string representation of the Interval object.

_key()
Private key function for hashing.

Returns 2-tuple consisting of (infimum, supremum).

Return type `tuple`

static **collapse_intervals**(*intervals*)
Collapse a list of overlapping intervals.

Parameters **intervals** (*list*) – A list of intervals to collapse.

Returns A new list of totally disjoint, collapsed Intervals.

Return type `list`

Raises **TypeError** – `interval` parameter must be a `list` containing only Interval objects.

discretize(*jump=None*)
Return all values within the range of the calling interval.

Parameters **jump** (*None* | *int* | *float* | *long*) – The jump to use after each value. Defaults to `1`, `1.0` and `1` for int, float, and long Intervals respectively.

Returns A list of discrete values contained in the calling Interval with a step size of `jump`.

Return type `list`

Raises **TypeError** – If a jump is provided, it must be an `int`, `float`, or `long` and match the type of the calling Interval.

1.2 The Point object

point module.

class `point.`**Point**(**coordinates*)
Point class. Point objects represent a point of N_d cartesian space. Point objects are immutable.

Variables

- **is_generic** – Whether or not the Point object is generic (i.e., the coordinates have not been defined).
- **coordinate** – The coordinate of the Point object.
- **dimension** – The dimension of space the Point object exists in.
- **_is_Point** – An identifier to use in place of `type` or `isinstance`.

__deepcopy__(*memo*)
Deepcopy a Point object via the `copy.deepcopy` method.

__eq__(*other*)
Determine if two Point objects are equal via the == operator.

__getitem__(*key*)
Retrieve the ith cooridinate from a Point object via indexing (e.g., `Point[i]`).

Raises

- **TypeError** – `key` parameter must be an `int`.
- **IndexError** – `key` parameter must be within the set $\{0, \ldots, d\}$ where d is the dimension of the Point object.

__hash__()
Hash implementation for set functionality of Point objects.

__init__(**coordinates*)
Construct a Point object.

Parameters **coordinates** (*strs* | *floats*) – The values to use as the coordinates of the Point object. At least one coordinate must be provided; to create a generic point object, pass values of `"x"`.

Raises

- **ValueError** – At least one coordinate must be provided.
- **TypeError** – All cooridnates must be either strings (equal to `"x"`) or floats.

__ne__(*other*)
Determine if two Point objects are not equal via the != operator.

__repr__()
Return a string representation of the Point object.

__str__()
Return a readable string representation of the Point object.

_key()
Private key function for hashing.

Returns *d*-tuple consisting of coordinates.

Return type tuple

can_observe(*spacetime_loc*, *worldline_start*, *worldline_end*)
Determine if the calling Point object can observe the spacetime location represented by the Point object in spacetime_loc parameter on the worldline segment determined by the Point objects in worldline_start and worldline_end parameters.

Returns Whether or not the calling Point object can observe spacetime_loc through the worldine defined by the Point objects in the worldline_start and worldline_end parameters.

Return type bool

clocks_unequal(*other*)
Determine if the clocks of two spacetime locations are unequal wherein the last coordinate of each represents time.

Returns Whether or not the calling Point object's last coordinate is equal to the last coordinate of the Point object in other parameter.

Return type bool

Raises **ValueError** – Dimensions of the Point object contained in other parameter and the calling Point must match.

is_on(*endpoint_1*, *endpoint_2*)
Determine if the calling Point object lies on line segment defined by the endpoint Point objects provided in the endpoint_1 and endpoint_2 parameters.

Raises **ValueError** – The calling Point object and the Point objects in the endpoint_1 and endpoint_2 parameters must all be in the same dimension of space and no Point object involved can be generic.

Returns Whether or not the calling Point object lies on the line segment.

Return type bool

meets(*worldline_1_start*, *worldline_1_end*, *worldline_2_start*, *worldline_2_end*)
Determine if the worldline segments defined by the Point objects in the worldline_1_start and worldline_1_end parameters and the worldline_2_start and worldline_2_end parameters meet at the calling Point object's coordinates.

Returns Whether or not worldines meet at the calling Point object's coordinates.

Return type bool

Raises **ValueError** – All Point objects must have the same dimension and cannot be generic.

not_same_point(*other*)
Determine if the calling Point object and the Point object contained in other parameter are not the same.

Returns Whether or not the calling Point object is unequal to the Point object in `other` parameter.

Return type `bool`

static `unstringify`(*point_string*)
Reconstruct a Point object from its string representation.

Returns Point object reconstructed from string representation.

Return type *Point*

Raises **`ValueError`** – The string must match the form given by `str(Point)` or `repr(Point)`, i.e., $P(c_1, \ldots, c_d)$.

1.3 The ValueSet object

valueset module.

Supports any object provided they implement `__deepcopy__`, `__eq__`, `__str__`, `__hash__`, and provide a parser for truth value evaluation. Additionally, `__le__` in ValueSet class must be extended to support the object if the object uses a non-standard form of equality, (e.g. Point objects are subset of generic Point of same dimension).

class `valueset.`**`ValueSet`**(*valueset*)
ValueSet class.

The ValueSet class uses the `total_ordering` decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- **`_base_types`** – The literal types supported by the ValueSet class.
- **`_object_types`** – The object types supported by the ValueSet class.
- **`values`** – The values contained in the ValueSet object.
- **`_is_ValueSet`** – An identifier to use in place of `type` or `isinstance`.

`__add__`(*other*)
Overloaded + operator for ValueSet. Take the union of two ValueSet objects or add a single element to the calling Valueset object. If adding an object, the object must be within `_object_types`.

`__contains__`(*key*)
Overloaded `in` operator for ValueSet. Determine if a value is contained in the calling ValueSet object.

Parameters **`key`** – The item to test for membership in the calling ValueSet object.

`__deepcopy__`(*memo*)
Deepcopy a ValueSet object via the `copy.deepcopy` method.

`__eq__`(*other*)
Determine if two ValueSet objects are equal via the == operator.

`__getitem__`(*key*)
Retrive the value located at the index given by `key` parameter via indexing (e.g. `ValueSet[key]`).

Parameters **`key`** (*int*) – The index to use for retrieval.

Raises

- **`IndexError`** – `key` index must be in $\{0, \ldots, n-1\}$ where n is the number of values contained in the calling ValueSet object.

- **TypeError** – key must be an int.

__iadd__(*other*)
Overloaded += operator for ValueSet. Take the union of two ValueSet objects or add a single element to the calling Valueset object. If adding an object, the object must be within _object_types.

__init__(*valueset*)
Construct a ValueSet object.

Parameters **valueset** (*list | set*) – The values to place in the ValueSet object. These values are passed through the _parse function before being stored.

Raises **TypeError** – valueset parameter must be a list or set.

__iter__()
Provide an iterator for ValueSet objects (e.g. "for value in ValueSet:").

__le__(*other*)
Overloaded <= operator for ValueSet object. Determine if the calling ValueSet object is a subset of the ValueSet object contained in other parameter.

__len__()
Determine the length of a ValueSet object via the len built-in function e.g.(len(ValueSet)).

__ne__(*other*)
Determine if two ValueSet objects are not equal via the != operator.

__nonzero__()
Determine if a ValueSet object is falsy (e.g. if ValueSet or if not ValueSet).

__repr__()
Return a string representation of the ValueSet object.

__setitem__(*key*, *value*)
Assign a value in value parameter to a ValueSet object at the index given by key parameter (e.g. ValueSet[key] = value).

Parameters

- **key** (*int*) – The index to use for assignment.
- **value** – The value to assign at index given by key.

Raises

- **AttributeError** – An invalid/unsupported object is given as value parameter.
- **IndexError** – key index must be in $\{0, \ldots, n-1\}$ where n is the number of values contained in the ValueSet object.
- **TypeError** – key must be an int and value in value parameter must be in ValueSet._base_types or ValueSet._object_types.
- **ValueError** – Duplicate values are not allowed in a ValueSet object.

__str__()
Return a readable string representation of the ValueSet object.

__sub__(*other*)
Overloaded − operator for ValueSet. The − operator functions as the set-theoretic difference.

static _parse(*values*)
Parse a list into the standard format used by ValueSet objects. Any ints, longs, and floats are absorbed into an Interval object if they are contained by that Interval and the base types contained in values are sorted.

Returns Filtered, sorted values in the ValueSet standard format.

Return type `list`

Raises **`TypeError`** – `values` parameter must be either a `list` or `set`.

static `_split_by_types`(*values*)

Split an iterable object by the types of elements within it and return a defaultdict where keys are the types composing the iterable object and the values are lists of the values of the iterable object falling into those types.

Parameters **`values`** (*`list`*|*`set`*|*`ValueSet`*|*`...`*) – An iterable object to split.

Returns A defaultdict of lists where keys correspond to `ValueSet._base_type` and `ValueSet._object_types` present in `values` parameter.

Return type `defaultdict(list)`

Raises

- **`AttributeError`** – Only objects contianing a single identifier in `_object_types` are supported.
- **`TypeError`** – An invalid type exists in the iterable object.

classmethod `add_object_type`(*object_identifier*)

Add compatibility for an object to the ValueSet class. This is part of the vivid object extension protocol.

Parameters **`object_identifier`** (*`str`*) – The identifier used by each instance of the new class. This must be of the form: `"_is_Object"` (e.g. `"_is_Point"` or `"_is_Interval"`).

CHAPTER

TWO

ATTRIBUTES AND RELATIONS

2.1 The Attribute object

attribute module.

class attribute.**Attribute**(*label*, *value_set*)

Attribute Class. An Attribute is a finite set A with an associated label l.

Variables

- **label** – The associated label l of the Attribute A.
- **value_set** – A ValueSet object functioning as the set of values that the attribute can take on (e.g {small,large}).
- **_is_Attribute** – An identifier to use in place of `type` or `isinstance`.

__add__(*other*)

Combine an Attribute object with another Attribute object, a Relation object, an AttributeStructure object or an AttributeSystem object via the + operator.

Parameters **other** (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the Attribute. If an Attribute, Relation, or AttributeStructure object is provided, an AttributeStructure object is returned; if an AttributeSystem object is provided, an AttributeSystem object is returned.

Raises **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.

__deepcopy__(*memo*)

Deepcopy an Attribute object via the `copy.deepcopy` method.

__eq__(*other*)

Determine if two Attribute objects are equal via the == operator.

__hash__()

Hash implementation for set functionality of Attribute objects.

__init__(*label*, *value_set*)

Construct an Attribute object.

Parameters

- **label** (*str*) – The label *l* to associate with the Attribute object.
- **value_set** (*list|ValueSet*) – The set of values the Attribute object can take on.

Raises **TypeError** – `label` parameter must be a string and `value_set` parameter must be either a ValueSet object or a list.

__ne__(*other*)
Determine if two Attribute objects are not equal via the != operator.

__repr__()
Return a string representation of the Attribute object.

__str__()
Return a readable string representation of the Attribute object.

_key()
Private key function for hashing.

Returns 2-tuple consisting of (label, valueset)

Return type tuple

2.2 The Relation object

relation module.

class relation.**Relation**(*definition*, *D_of_r*, *subscript*)
Relation class. Relation objects represent logical relations used in AttributeStructure objects.

Variables

- **definition** – A string representation of the Relation object's definition with form Rn(a,...)<=>... where n is a positive integer; whitespace is ignored.
- **DR** – DR represents $D(R) \subseteq \{A_1, \ldots, A_n\}$; held as a list of strings corresponding to the labels of some set of Attributes objects; no assumptions are made on the labels of the attributes.
- **subscript** – The subscript of the relation.
- **_is_Relation** – An identifier to use in place of type or isinstance.

__add__(*other*)
Combine a Relation object with an Attribute object, an AttributeStructure object or an AttributeSystem object via the + operator.

Parameters **other** (*Attribute|AttributeStructure|AttributeSystem*) – The object to combine with the Attribute. If an Attribute or AttributeStructure object is provided, an AttributeStructure object is returned; if an AttributeSystem object is provided, an AttributeSystem is returned.

Raises **TypeError** – other parameter must be an Attribute, AttributeStructure, or AttributeSystem object.

__deepcopy__(*memo*)
Deepcopy a Relation object via the copy.deepcopy method.

__eq__(*other*)
Determine if two Relation objects are equal via the == operator.

__init__(*definition*, *D_of_r*, *subscript*)
Construct a Relation object.

Parameters

- **definition** (*str*) – The definition of the logical relation; valid definitions have the form: Rn(a,...) <=> ... where all whitespace is ignored.

- **D_of_r** (*list*) – A list of strings representing $D(R) \subseteq \{A_1, \ldots, A_n\}$.
- **subscript** (*int*) – The subscript of the relation; must match subscript in definition.

Raises

- **TypeError** – `definition` parameter must be a `str`, `D_of_r` parameter must be a `list` containing only `str`s and `subscript` parameter must be an `int`.
- **ValueError** – `definition` parameter must be correctly formatted, the number of parameters provided in `definition` must match the length of `D_of_r` and `subscript` parameter must match subscript provided in `definition` parameter.

__ne__(*other*)
Determine if two Relation objects are not equal via the `!=` operator.

__repr__()
Return a string representation of the Relation object.

__str__()
Return a readable string representation of the Relation object.

get_DR(*string=False*)
Return $D(R)$ of the calling Relation object. If `string` parameter is set to True, return a `str` representation of $D(R)$.

Parameters **string** (*boolean*) – A boolean value for whether or not to return a string representation of $D(R)$.

Returns A representation of $D(R)$.

Return type `str|list`

get_arity()
Return the arity of the calling Relation object.

Returns The length of $D(R)$.

Return type `int`

static is_valid_definition(*definition*)
Determine if a given definition in `definition` parameter is valid. A definition is valid when it is of the form `Rs(x1,...,xn)` <=> <expression>.

The important thing here is the left hand side and the marker `"<=>"`. Everything on the right hand side of `"<=>"` is ignored as far as the definition's validity is concerned; whether or not it is evaluatable is left to `Formula.assign_truth_value()` as it is only during the assignment of a truth value that the expression comes into play. All whitespace is trimmed immediately so arbitrary spacing is allowed.

Parameters **definition** (*str*) – The definition to verify.

Returns Whether or not `definition` is valid.

Return type `bool`

set_DR(*DR*)
Set $D(R)$ to `DR` parameter.

Parameters **DR** (`list`) – The list of strings to set the calling Relation object's $D(R)$ to.

Raises

- **TypeError** – $D(R)$ is not a `list` of `str`s.
- **ValueError** – The cardinality of $D(R)$ must match argument cardinality in Relation object's definition member.

set_definition(*definition*)

Set the definition of the Relation object to `definition` parameter after ensuring that it conforms to required format.

Parameters **definition** (*str*) – The new definition of the Relation object.

Raises

- **TypeError** – `definition` parameter must be a `str`.
- **ValueError** – `definition` must conform to valid definition rules.

CHAPTER

THREE

ATTRIBUTE STRUCTURES

3.1 The AttributeStructure object

attribute_structure module.

class attribute_structure.**AttributeStructure**(**args*)

AttributeStructure class. An AttributeStructure object consists of a finite set of Attribute objects $A_1, \ldots, A_k$; and a countable collection $\mathcal{R}$ of computable Relation objects with $D(R) \subseteq \{A_1, \ldots, A_k\}$ for each $R \subseteq \mathcal{R}$. i.e.,

$$\mathcal{A} = (\{A_1, \ldots, A_k\}; \mathcal{R})$$

The AttributeStructure class uses the `total_ordering` decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- **attributes** – A list of Attribute objects (i.e., $A_1, \ldots, A_k$); always maintained as a list.
- **relations** – A dictionary of relations (i.e., $\mathcal{R}$).
- **_is_AttributeStructure** – An identifier to use in place of `type` or `isinstance`.

__add__(*other*)

Add an Attribute, Relation, AttributeStructure, or AttributeSystem object via the + operator.

Parameters **other** (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the AttributeStructure. If an Attribute, Relation, or AttributeStructure object is provided, an AttributeStructure object is returned; if an AttributeSystem object is provided, an AttributeSystem object is returned.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Duplicate Attribute labels are not permitted, duplicate subscripts are not permitted and every Relation's $D(R)$ must be a subset of Attribute labels in the AttributeStructure.

__contains__(*key*)

Determine if Attribute, Relation, Attribute corresponding to a label in the `key` parameter, or Relation corresponding to a subscript in the `key` parameter is contained by AttributeStructure via `in` operator.

Parameters **key** (*Attribute|Relation|str|int*) – The key to use when checking for membership.

Raises **TypeError** – `key` must be an Attribute object, Relation object, `str`, or `int`.

__deepcopy__(*memo*)
Deepcopy an AttributeStructure object via the `copy.deepcopy` method.

__eq__(*other*)
Determine if two AttributeStructure objects are equal via the == operator.

__getitem__(*key*)
Retrieve a reference to the Attribute object or Relation object in the AttributeStructure via the key provided in the `key` parameter provided.

Parameters **key** (*Attribute|Relation|str|int*) – The Attribute object, Relation object, label, or subscript to use when attempting to find the corresponding Attribute object or Relation object.

Raises

- **KeyError** – Attribute object or Relation object provided in the `key` parameter not found in the AttributeStructure, no Attribute object with label provided in the `key` parameter found in the AttributeStructure, or no Relation object with subscript provided in the `key` parameter found in the AttributeStructure.
- **TypeError** – `key` is not an Attribute object, Relation object, `int`, or `str`.

__iadd__(*other*)
AAdd an Attribute, Relation, AttributeStructure, or AttributeSystem object via the += operator.

Parameters **other** (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the AttributeStructure. If an Attribute, Relation, or AttributeStructure object is provided, an AttributeStructure object is returned; if an AttributeSystem object is provided, an AttributeSystem object is returned.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Duplicate Attribute labels are not permitted, duplicate subscripts are not permitted and every Relation's $D(R)$ must be a subset of Attribute labels in the AttributeStructure.

__init__(**args*)
Construct an AttributeStructure object.

Parameters **args** (*Attribute|Relation*) – Any amount of Attribute and Relation objects.

Raises

- **TypeError** – all optional positional arguments provided must be Attribute or Relation objects.
- **ValueError** – Duplicate Attribute labels are not permitted, Duplicate Relation subscripts are not permitted, and each Relation object's $D(R)$ must be a subset of the cartesian product of some combination of the labels of the Attributes provided.

__isub__(*other*)
Remove Attribute's or Relation's via − operator. If an AttributeStructure object is provided, all Attribute objects and Relation objects within that AttributeStructure object will be removed from the calling AttributeStructure.

Parameters **other** (*Attribute|Relation|AttributeStructure*) – The Attribute, Relation, or AttributeStructure object to remove.

Raises

- **KeyError** – Invalid Attribute or Relation object provided in `other` parameter.
- **TypeError** – Only Attribute, Relation, or AttributeStructure objects can be removed.
- **ValueError** – Some Attribute or Relation object provided in AttributeStructure object in `other` parameter not found in calling AttributeStructure object or some Relation object's $D(R)$ is invalid after an Attribute object is removed.

__le__(*other*)
Overloaded <= operator. Determine if the calling AttributeStructure object is a subset of the AttributeStructure object contained in `other` parameter.

__ne__(*other*)
Determine if two AttributeStructure objects are not equal via the != operator.

__repr__()
Return a string representation of the AttributeStructure object.

__str__()
Return a readable string representation of the AttributeStructure object.

__sub__(*other*)
Remove Attribute's or Relation's via − operator. If an AttributeStructure object is provided, all Attribute objects and Relation objects within that AttributeStructure object will be removed from the calling AttributeStructure.

Parameters **other** (*Attribute|Relation|AttributeStructure*) – The Attribute, Relation, or AttributeStructure object to remove.

Raises

- **KeyError** – Invalid Attribute or Relation object provided in `other` parameter.
- **TypeError** – Only Attribute, Relation, or AttributeStructure objects can be removed.
- **ValueError** – Some Attribute or Relation object provided in AttributeStructure object in `other` parameter not found in calling AttributeStructure object or some Relation object's $D(R)$ is invalid after an Attribute object is removed.

get_cardinality()
Return the cardinality of the calling AttributeStructure object.

Returns The cardinality of the AttributeStructure object, i.e., the amount of Attribute objects contained therein.

Return type `int`

get_labels()
Return the labels of the Attribute objects within the calling AttributeStructure object.

Returns A list of the labels of the Attribute objects in the calling AttributeStructure object.

Return type `list`

get_subscripts()
Return the subscripts of the Relation objects within the calling AttributeStructure object.

Returns A list of the subscripts of the Relation objects in this AttributeStructure object.

Return type `list`

CHAPTER

FOUR

ATTRIBUTE SYSTEMS

4.1 The AttributeSystem object

attribute_system module.

class attribute_system.**AttributeSystem**(*attribute_structure*, *objects*)

AttributeSystem class. An AttributeSystem object, based on the AttributeStructure object $\mathcal{A}$ is a pair

$$\mathcal{S} = (\{s_1, \ldots, s_n\}; \mathcal{A})$$

consisting of a finite number $n > 0$ of objects $s_1, \ldots, s_n$ (represented as strs) and $\mathcal{A}$.

The AttributeSystem class uses the total_ordering decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- ***attribute_structure*** – The AttributeStructure of the AttributeSystem.
- **objects** – The objects of the AttributeSystem; held as a list of strs.
- **_is_AttributeSystem** – An identifier to use in place of type or isinstance.

__add__(*other*)

Add an Attribute, Relation, AttributeStructure, or AttributeSystem object via the + operator.

Parameters **other** (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the AttributeSystem. An AttributeSystem object is always returned regardless of the type of other parameter.

Raises

- **TypeError** – other parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Cannot add AttributeSystems with overlapping objects.

__contains__(*key*)

Determine if the Attribute object, Relation object, AttributeStructure object, or str in the key parameter is contained by the calling AttributeSystem object via in operator.

Parameters **key** (*Attribute|Relation|AttributeStructure|str*) – The key to use when checking for membership.

Raises **TypeError** – key parameter must be an Attribute object, Relation object, AttributeStructure object, or str.

__deepcopy__(*memo*)

Deepcopy an AttributeSystem object via the copy.deepcopy method.

__eq__(*other*)
Determine if two AttributeSystem objects are equal via == operator.

__getitem__(*key*)
Retrieve a reference to the Attribute, Relation, or object in the AttributeSystem by indexing with the key provided in `key` parameter (e.g., `"AttributeSystem[key]"`).

Parameters **key** (*Attribute|Relation|str*) – The Attribute, Relation or name of the object to get the reference of from the calling AttributeSystem object.

Raises **TypeError** – `str` in `key` does not match any object contained in the calling AttributeSystem object.

__iadd__(*other*)
Add an Attribute, Relation, AttributeStructure, or AttributeSystem object via the += operator.

Parameters **other** (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the AttributeSystem. An AttributeSystem object is always returned regardless of the type of `other` parameter.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Cannot add AttributeSystems with overlapping objects.

__init__(*attribute_structure*, *objects*)
Construct AttributeSystem object.

Parameters

- **attribute_structure** (AttributeStructure) – an AttributeStructure object to use as the attribute structure $\mathcal{A}$ of the AttributeSystem object.
- **objects** (*list*) – A list of `str`s denoting the objects of the AttributeSystem.

Raises

- **TypeError** – `objects` parameter must be a `list` and `attribute_structure` parameter must be an AttributeStructure object.
- **ValueError** – all objects provided in `objects` parameter must be unique non-empty `str`s.

__isub__(*other*)
Remove an Attribute, Relation, AttributeStructure, or AttributeSystem object via the −= operator. In the case of AttributeStructure and AttributeSystem objects being provided to `other` parameter, remove all of their consituent parts from the calling AttributeSystem.

Parameters **other** (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to remove from the AttributeSystem. An AttributeSystem object is always returned regardless of the type of `other` parameter.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Cannot remove objects not present in this AttributeSystem.

__le__(*other*)
Determine if the calling AttributeSystem object is a subset of the AttributeSystem object contained in the `other` parameter via <= operator.

__ne__(*other*)
Determine if two AttributeSystem objects are not equal via != operator.

__repr__()
Return a string representation of the AttributeSystem object.

__str__()
Return a readable string representation of the AttributeSystem object.

__sub__(*other*)
Remove an Attribute, Relation, AttributeStructure, or AttributeSystem object via the − operator. In the case of AttributeStructure and AttributeSystem objects being provided to `other` parameter, remove all of their consituent parts from the calling AttributeSystem.

Parameters **other** (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to remove from the AttributeSystem. An AttributeSystem object is always returned regardless of the type of `other` parameter.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Cannot remove objects not present in this AttributeSystem.

get_power()
Get the power of the calling AttributeSystem object, i.e., $n \cdot |\mathcal{A}|$.

Returns The power of the calling AttributeSystem object: $n \cdot |\mathcal{A}|$

Return type `int`

is_automorphic()
Determine if the calling AttributeSystem object is automorphic.

Returns Whether or not the calling AttributeSystem object is automorphic; i.e., some (perhaps all) of the objects $s_1, \ldots, s_n$ are contained by at least one of the ValueSets of the Attribute objects of the underlying AttributeStructure object $\mathcal{A}$.

Return type `bool`

CHAPTER

FIVE

STATES

5.1 The State object

state module.

class `state.`**`State`**(*attribute_system*, *ascriptions={}*)

State class. Each State object is a state of an AttributeSystem; that is a set of functions $\sigma = \{\delta_1, \ldots, \delta_k\}$, where each δ_i is a function from $\{s_1, \ldots, s_n\}$ to the set of all non-empty finite subsets of A_i, i.e.,

$$\delta_i : \{s_1, \ldots, s_n\} \to \mathcal{P}_{fin}(A_i) \setminus \emptyset.$$

The State class uses the `total_ordering` decorator so proper extensions, contravariant extensions and contravariant proper extensions are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- ***`attribute_system`*** – A copy of the AttributeSytem object $\mathcal{S}$ that the State object comes from.
- **`ascriptions`** – The ascriptions of the state (i.e., the set of attribute-object pairs and their corresponding ValueSet objects) $\delta_i,\ i = 1, \ldots, k$.
- **`_is_State`** – An identifier to use in place of `type` or `isinstance`.

`__deepcopy__`(*memo*)

Deepcopy a State object via the `copy.deepcopy` method.

`__eq__`(*other*)

Determine if two State objects are equal via the == operator.

`__getitem__`(*key*)

Retrive the ascription δ_i or ascription of a particular object $\delta_i(s_j)$ (that is, the ValueSet corresponding to the attribute-object pair) given by `key` parameter via indexing (e.g. `State[key]`).

Raises

- **`KeyError`** – `key` parameter must be a valid Attribute label or valid attribute-object pair in the underlying AttributeSystem $\mathcal{S}$ of the State object.
- **`TypeError`** – `key` parameter must be a `str` or `tuple` containing only `str`s.

`__init__`(*attribute_system*, *ascriptions={}*)

Construct a State object.

Parameters

- **`attribute_system`** (`AttributeSystem`) – The AttributeSystem object $\mathcal{S}$ from which the State comes from.

- **ascriptions** (dict) – An optional dictionary of attribute-object pairs $\delta_i(s_j)$ to use as ascriptions; if some attribute-object pair is not provided, the full ValueSet of the Attribute object corresponding to the attribute label in the attribute-object pair is used.

Raises **TypeError** – attribute_system parameter must be an AttributeSystem object and ascriptions parameter must be a dict.

__le__(*other*)
Overloaded <= operator for State; Determine if the calling State object is an extension of the State object in other parameter.

Raises

- **TypeError** – other parameter must be a State object.
- **ValueError** – The state object in the other parameter must share the same underlying AttributeSystem object $\mathcal{S}$ as the calling State object.

__ne__(*other*)
Determine if two State objects are not equal via the != operator.

__repr__()
Return a string representation of the State object.

__str__()
Return a readable string representation of the State object.

add_object(*obj*, *ascriptions=None*)
Add an object s' to the calling State object's underlying AttributeSystem $\mathcal{S}$ and optionally update any ascriptions of the new object $\delta_i(s')$ provided.

Parameters

- **obj** (str) – The new object s' to add to the State.
- **ascriptions** (dict) – The optional ValueSets to assign to the attribute-object pairs corresponding to the new object $\delta_i(s')$.

Raises

- **TypeError** – obj parameter must be a non-empty str and if ascriptions parameter is provided, it must be a dict.
- **ValueError** – Duplicate objects cannot be added and all ascriptions provided must be from an existing label of an Attribute object in the underlying AttributeSystem object $\mathcal{S}$ to s'.

get_alternate_extensions(**states*)
Return all alternate extensions of the calling State object with respect to State objects $\sigma_1, \ldots, \sigma_m$ provided by optional positional arguments of states parameter, i.e., generate $\mathbf{AE}(\{\sigma_1, \ldots, \sigma_m\}, \sigma')$.

Parameters **states** (State) – The states $\{\sigma_1, \ldots, \sigma_m\}$ to use for the derivation of the alternate extensions of the calling State object.

Returns $\mathbf{AE}(\{\sigma_1, \ldots, \sigma_m\}, \sigma')$.

Return type list

Raises

- **TypeError** – all optional positional arguments must be State objects.
- **ValueError** – at least one State object must be provided in optional positional arguments and all provided State objects must be proper extensions of the calling State object.

get_worlds()
Return a list of all possible worlds $(w; \widehat{\rho})$ derivable from the calling State object.

Returns all worlds $(w; \widehat{\rho})$ derivable from this State object.

Return type list

is_alternate_extension(*s_prime*, **states*)
Determine if the State object in s_prime parameter is an alternate extension of the calling State object w.r.t. the State objects $\sigma_1, \ldots, \sigma_m$ provided by optional positional arguments of states parameter, i.e., evaluate *Alt*$(\sigma, \{\sigma_1, \ldots, \sigma_m\}, \sigma')$.

Parameters

- **s_prime** (State) – The State object to verify as the alternate extension, σ'
- **states** (State) – The states $\{\sigma_1, \ldots, \sigma_m\}$ to use for the derivation of the alternate extensions of the calling State object.

Returns The result of the evaluate of *Alt*$(\sigma, \{\sigma_1, \ldots, \sigma_m\}, \sigma')$.

Return type bool

Raises **TypeError** – s_prime parameter must be a State object.

is_disjoint(*other*)
Determine if the calling State object is disjoint from the State object in the other parameter.

Returns Whether or not the calling State object and State object contained in the other parameter are disjoint.

Return type bool

is_valuation(*label*)
Determine if the ascription δ_i corresponding to the label parameter is a valuation in the calling State object; that is $|\delta_i(s_j)| = 1$ for every $j = 1, \ldots, n$.

Parameters **label** (str) – The label corresponding to the ascription δ_i to check for valuation in the calling State object.

Returns Whether or not the ascription δ_i corresponding to label is a valuation in the calling State object.

Return type bool

is_world()
Determine if the calling State object σ is a world; that is every ascription δ_i of σ is a valuation.

Returns Whether or not the calling State object is a world.

Return type bool

static join(*s1*, *s2*)
Join two State objects if possible, i.e., return $\sigma_1 \sqcup \sigma_2$.

Parameters

- **s1** (State) – The left operand σ_1 to use for the join operation.
- **s2** (State) – The right operand σ_2 to use for the join operation.

Raises **ValueError** – s1 and s2 parameters must share the same underlying AttributeSystem $\mathcal{S}$.

set_ascription(*ao_pair*, *new_valueset*)

Set an ascription of an object (that is, $\delta_i(s_j)$) in the calling State object, given by the `ao_pair` parameter, to the ValueSet object provided in the `new_valueset` parameter.

Parameters

- **ao_pair** (`tuple`) – The attribute-object pair $\delta_i(s_j)$ to use as a key for the ascription `dict` member in the calling State object.
- **new_valueset** (`ValueSet`) – The new ValueSet object to assign to the corresponding attribute-object pair $\delta_i(s_j)$ given by the `ao_pair` paramater.

Raises

- **TypeError** – `ao_pair` parameter must be a `tuple` and `new_valueset` parameter must be a `list`, `set`, or ValueSet object.
- **ValueError** – `ao_pair` parameter must be a 2-tuple (`str`,`str`) and `new_valueset` parameter must be a non-empty subset of the ValueSet object of the Attribute object corresponding to the attribute in the `ao_pair` parameter.
- **KeyError** – `ao_pair` must be a key in the `ascriptions` member of this State object.

CHAPTER

SIX

VOCABULARIES

6.1 The RelationSymbol object

relation_symbol module.

class relation_symbol.**RelationSymbol**(*name*, *arity*)

Relation Symbols class. The RelationSymbol class is used entirely in the Vocabulary class.

Variables

- **name** – A str designating the name of the RelationSymbol object.
- **arity** – An int designating the arity of the RelationSymbol object.
- **_is_RelationSymbol** – An identifier to use in place of type or isinstance.

__deepcopy__(*memo*)

Deepcopy a RelationSymbol object via the copy.deepcopy method.

__eq__(*other*)

Determine if two RelationSymbol objects are equal via the == operator.

__hash__()

Hash implementation for set functionality of RelationSymbol objects.

__init__(*name*, *arity*)

Construct a RelationSymbol object.

Parameters

- **name** (str) – The name of the RelationSymbol object.
- **arity** (int) – The arity of the RelationSymbol object.

Raises

- **TypeError** – name parameter must be a str and arity parameter must be an int.
- **ValueError** – arity must be positive.

__ne__(*other*)

Determine if two RelationSymbol objects are not equal via the != operator.

__repr__()

Return a string representation of the RelationSymbol object.

__str__()

Return a readable string representation of the RelationSymbol object.

_key()

Private key function for hashing.

Returns 2-tuple consisting of (`name`, `arity`)

Return type `tuple`

6.2 The Vocabulary object

vocabulary module.

class `vocabulary.`**Vocabulary**(*C*, *R*, *V*)

Vocabulary class. The Vocabulary class represents a first-order vocabulary $\Sigma = (\mathrm{C},\mathrm{R},\mathrm{V})$ consisting of a set of constant symbols C; a set of relation symbols R; and a set of variables V.

Variables

- **C** – The constants C of the vocabulary.
- **R** – The relation symbols R of the vocabulary.
- **V** – The variables V of the vocabulary.
- **_is_Vocabulary** – An identifier to use in place of `type` or `isinstance`.

__contains__(*key*)

Determine if the calling Vocabulary object contains the `str` or RelationSymbol object in the `key` parameter.

Parameters **key** (*RelationSymbol | str*) – The `str` or RelationSymbol object to test for membership in the calling Vocabulary object.

__deepcopy__(*memo*)

Deepcopy a Vocabulary object via the `copy.deepcopy` method.

__eq__(*other*)

Determine if two Vocabulary objects are equal via == operator.

__hash__()

Hash implementation for set functionality of Vocabulary objects.

__init__(*C*, *R*, *V*)

Construct a Vocabulary object. Each parameter `C`, `R` and `V` are sorted before being stored.

Parameters

- **C** (`list`) – The constants C of the Vocabulary object; held as a `list` of `str`s.
- **R** (`list`) – The relation symbols R of the Vocabulary object; held as a `list` of RelationSymbol objects.
- **V** (`list`) – The variables V of the Vocabulary object; held as a `list` of `str`s.

Raises

- **TypeError** – `C`, `R`, and `V` parameters must all be lists, `C` and `V` must be contain only `str`s and `R` must contain only RelationSymbol objects.
- **ValueError** – `C` and `V` cannot overlap and duplicates are not permitted in any of the lists.

__ne__(*other*)

Determine if two Vocabulary objects are not equal via `!=` operator.

__repr__()

Return a string representation of the Vocabulary object.

__str__()
Return a readable string representation of the Vocabulary object.

_key()
Private key function for hashing.

Returns 3-tuple consisting of (C, R, V)

Return type tuple

add_constant(*constant*)
Add a constant to this Vocabulary object's constants C.

Parameters **constant** (str) – The new constant to add to the Vocabulary's constants C.

Raises

- **TypeError** – constant parameter must be a str.
- **ValueError** – Duplicate symbols are not permitted.

add_variable(*variable*)
Add a variable to this Vocabulary object's variables V.

Parameters **variable** (str) – The new variable to add to the Vocabulary's variables V.

Raises

- **TypeError** – variable parameter must be a str.
- **ValueError** – Duplicate symbols are not permitted.

CHAPTER

SEVEN

CONSTANT AND VARIABLE ASSIGNMENTS

7.1 The Assignment base class

assignment module.

class assignment.**Assignment**(*vocabulary*, *attribute_system*)

Assignment class. The Assignment class functions as a superclass for the ConstantAssignment and VariableAssignment classes.

Variables

- ***vocabulary*** – A reference to the Vocabulary object Σ that the Assignment is defined over.
- ***attribute_system*** – A copy of the AttributeSystem the Assignment originates from.
- **_is_Assignment** – An identifier to use in place of `type` or `isinstance`.

__eq__(*other*)

Determine if two Assignment objects are equal via the == operator.

__init__(*vocabulary*, *attribute_system*)

Construct a base Assignment.

Parameters

- **vocabulary** (Vocabulary) – The Vocabulary Σ the Assignment is defined over.
- **attribute_system** (AttributeSystem) – The AttributeSystem from which the objects in the Assignment come from.

Raises **TypeError** – `vocabulary` parameter must be a Vocabulary object and `attribute_system` parameter must be an AttributeSystem object.

__ne__(*other*)

Determine if two Assignment objects are not equal via the != operator.

7.2 The ConstantAssignment object

constant_assignemnt module.

class constant_assignment.**ConstantAssignment**(*vocabulary*, *attribute_system*, *mapping*)

Bases: *assignment.Assignment*

ConstantAssignment class. A ConstantAssignment is a partial function ρ from the constants C of some Vocabulary object Σ, to the objects $\{s_1, \ldots, s_n\}$ of some AttributeSystem $\mathcal{S}$.

The ConstantAssignment class uses the `total_ordering` decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- ***vocabulary*** – A reference to the Vocabulary object Σ the ConstantAssignment is defined over.
- ***attribute_system*** – A copy of the AttributeSystem object $\mathcal{S}$ the ConstantAssignment originates from.
- **mapping** – The mapping $\mathrm{C} \longmapsto \{s_1, \ldots, s_n\}$.
- **source** – The constants of Σ used in the partial mapping ρ.
- **target** – The objects of $\mathcal{S}$ used in the partial mapping ρ.
- **_is_ConstantAssignment** – An identifier to use in place of `type` or `isinstance`.

__deepcopy__(*memo*)
Deepcopy a ConstantAssignment object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

__eq__(*other*)
Determine if two ConstantAssignment objects are equal via the == operator.

__getitem__(*key*)
Retrive the object s_i mapped to the constant c_i given by `key` parameter via indexing (e.g. `ConstantAssignment[key]`).

Parameters **key** (*str*) – The constant c_i to use for retrieval.

Raises

- **KeyError** – The constant c_i given by the `key` parameter is not in this ConstantAssignment's `source` member.
- **TypeError** – `key` parameter must be a `str`.

__init__(*vocabulary*, *attribute_system*, *mapping*)
Construct a ConstantAssignment object.

Parameters

- **vocabulary** (`Vocabulary`) – The Vocabulary object Σ the ConstantAssignment is defined over.
- **attribute_system** (`AttributeSystem`) – The AttributeSystem object $\mathcal{S}$ from which the objects $\{s_1, \ldots, s_n\}$ in the ConstantAssignment come from.
- **mapping** (`dict`) – The mapping ρ from the constants C of the Vocabulary object Σ in the `vocabulary` parameter to the objects $\{s_1, \ldots, s_n\}$ of the AttributeSystem object $\mathcal{A}$ in the `attribute_system` parameter.

Raises

- **TypeError** – `vocabulary` parameter must be a Vocabulary object, `attribute_system` parameter must be an AttributeSystem object and `mapping` parameter must be a `dict` with `str` keys and values.
- **ValueError** – All keys in the `mapping` parameter must be in the Vocabulary object in the `vocabulary` parameter's `C` member and all values in the `mapping` parameter must be unique and match some object in the `object` member of the AttributeSystem object in the `attribute_system` parameter.

__lt__(*other*)
Overloaded < operator for ConstantAssignment. Determine if the calling ConstantAssignment object is a subset of the ConstantAssignment object in the `other` parameter.

Raises **TypeError** – `other` parameter must be a ConstantAssignment object.

__ne__(*other*)
Determine if two ConstantAssignment objects are not equal via the != operator.

__repr__()
Return a string representation of the ConstantAssignment object.

__str__()
Return a readable string representation of the ConstantAssignment object.

add_mapping(*constant_symbol*, *obj*)
Extend the calling ConstantAssignment object by adding a new mapping from the constant c' in the `constant_symbol` parameter to the object o' in the `obj` parameter.

Raises

- **TypeError** – Both `constant_symbol` and `obj` parameters must be `str`s.
- **ValueError** – The constant c' in the `constant_symbol` parameter must be in the `C` member of the underlying Vocabulary object Σ, the object o' in the `obj` parameter must be in the objects of the `objects` member of the underlying AttributeSystem object $\mathcal{S}$ and neither the constant c' nor the object o' may be a duplicate.

get_domain()
Get the set of all and only those constant symbols for which the calling ConstantAssignment object ρ is defined w.r.t. the `C` member of the `vocabulary` member of ρ.

Returns The list of constants for which ρ is defined.

Return type `list`

in_conflict(*other*)
Check if the calling ConstantAssignment object ρ_1 is in conflict with the ConstantAssignment object ρ_2 provided in the `other` parameter.

Returns Whether or not ρ_1 and ρ_2 are in conflict, that is, if there is some $c \in Dom(\rho_1) \cap Dom(\rho_2)$, such that $\rho_1(c) \neq \rho_2(c)$.

Return type `bool`

Raises **TypeError** – `other` parameter must be a ConstantAssignment object.

is_total()
Determine if the calling ConstantAssignment object ρ is a total function $\widehat{\rho}$ from C $\longrightarrow \{s_1, \ldots, s_n\}$.

Returns Whether or not the source of ρ spans the `C` member of Σ.

Return type `bool`

remove_mapping(*constant_symbol*, *obj*)
Extend the calling ConstantAssignment object by removing an existing mapping from the constant c' in the `constant_symbol` parameter to the object o' in the `obj` parameter.

Raises

- **TypeError** – both `constant_symbol` and `obj` parameters must be `str`s.
- **ValueError** – The constant c' in the `constant_symbol` parameter must be in the `C` member of the underlying Vocabulary object Σ, the object o' in the `obj` parameter must be in the objects of the `objects` member of the underlying AttributeSystem object $\mathcal{S}$ and

the constant c' and the object o' must already be in the `source` and `target` members of the calling ConstantAssignment object respectively.

7.3 The VariableAssignment object

variable_assignment module.

class `variable_assignment.`**VariableAssignment**(*vocabulary*, *attribute_system*, *mapping*, *dummy=False*)

VariableAssignment class. A VariableAssignment is a total function χ from the variables V of some Vocabulary object Σ, to the objects $\{s_1, \ldots, s_n\}$ of some AttributeSystem $\mathcal{S}$.

Variables

- ***vocabulary*** – A reference to the Vocabulary object Σ the ConstantAssignment is defined over.
- ***attribute_system*** – A copy of the AttributeSystem object $\mathcal{S}$ the ConstantAssignment originates from.
- **mapping** – The mapping V $\longrightarrow \{s_1, \ldots, s_n\}$.
- **source** – The variables of Σ used in the total mapping χ.
- **target** – The objects of $\mathcal{S}$ used in the total mapping χ.
- **_is_VariableAssignment** – An identifier to use in place of type or isinstance.

The VariableAssignment class uses the `total_ordering` decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

__deepcopy__(*memo*)

Deepcopy a VariableAssignment object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

__eq__(*other*)

Determine if two VariableAssignment objects are equal via the == operator.

__getitem__(*key*)

Retrive the object s_i mapped to the variable v_i given by `key` parameter via indexing (e.g. `VariableAssignment[key]`).

Parameters **key** (*str*) – The variable v_i to use for retrieval.

Raises

- **KeyError** – The variable v_i given by the `key` parameter is not in this VariableAssignment's `source` member.
- **TypeError** – `key` parameter must be a `str`.

__init__(*vocabulary*, *attribute_system*, *mapping*, *dummy=False*)

Construct a VariableAssignment object.

Parameters

- **vocabulary** (Vocabulary) – The Vocabulary object Σ the VariableAssignment is defined over.
- **attribute_system** (AttributeSystem) – The AttributeSystem object $\mathcal{S}$ from which the objects $\{s_1, \ldots, s_n\}$ in the VariableAssignment come from.

- **mapping** (dict) – The mapping χ from the variables V of the Vocabulary object Σ in the `vocabulary` parameter to the objects $\{s_1, \ldots, s_n\}$ of the AttributeSystem object $\mathcal{A}$ in the `attribute_system` parameter.
- **dummy** (bool) – A flag for creating a dummy (i.e., empty) VariableAssignment object χ_{dummy}.

Raises

- **TypeError** – `vocabulary` parameter must be a Vocabulary object, `attribute_system` parameter must be an AttributeSystem object and `mapping` parameter must be a `dict` with `str` keys and values.
- **ValueError** – All keys in the `mapping` parameter must be in the Vocabulary object in the `vocabulary` parameter's `V` member and all values in the `mapping` parameter must be unique and match some object in the `object` member of the AttributeSystem object in the `attribute_system` parameter.

__ne__(*other*)
Determine if two VariableAssignment objects are not equal via the != operator.

__repr__()
Return a string representation of the VariableAssignment object.

__str__()
Return a readable string representation of the VariableAssignment object.

CHAPTER

EIGHT

NAMED STATES

8.1 The NamedState object

named_state module.

class named_state.**NamedState**(*attribute_system*, *p*, *ascriptions={}*)

Bases: *state.State*

NamedState class. Each state is a pair $(\sigma; \rho)$ consisting of a state σ and a constant assignment ρ.

The NamedState class uses the `total_ordering` decorator so proper extensions, contravariant extensions and contravariant proper extensions are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- ***attribute_system*** – A copy of the AttributeSytem object $\mathcal{S}$ that the NamedState object comes from.
- **ascriptions** – The ascriptions of the named state (i.e., the set of attribute-object pairs and their corresponding ValueSet objects) $\delta_i,\ i = 1, \ldots, k$.
- **p** – The ConstantAssignment object ρ of the named state.
- **_is_NamedState** – An identifier to use in place of `type` or `isinstance`.

__deepcopy__(*memo*)

Deepcopy a NamedState object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

__eq__(*other*)

Determine if two NamedState objects are equal via the == operator.

__init__(*attribute_system*, *p*, *ascriptions={}*)

Construct a NamedState object.

Parameters

- **attribute_system** (AttributeSystem) – The AttributeSystem object $\mathcal{S}$ from which the NamedState object comes from.
- **p** (ConstantAssignment) – The ConstantAssignment object ρ of the NamedState object.
- **ascriptions** (`dict`) – An optional dictionary of attribute-object pairs $\delta_i(s_j)$ to use as ascriptions; if some attribute-object pair is not provided, the full ValueSet of the Attribute object corresponding to the attribute label in the attribute-object pair is used.

Raises

- **TypeError** – p parameter must be a ConstantAssignment object.
- **ValueError** – The AttributeSystem object provided in the attribute_system parameter and the AttributeSystem object of the ConstantAssignment object in the p parameter must match.

__le__(*other*)
Overloaded <= operator for NamedState; Determine if the calling NamedState object $(\sigma; \rho)$ is an extension of the NamedState object $(\sigma'; \rho')$ in the other parameter; that is, if $(\sigma; \rho) \sqsubseteq (\sigma'; \rho')$.

Raises **TypeError** – other parameter must be a NamedState object.

__ne__(*other*)
Determine if two NamedState objects are not equal via the != operator.

__repr__()
Return a string representation of the NamedState object.

__str__()
Return a readable string representation of the NamedState object.

_generate_variable_assignments()
Generate all possible VariableAssignment objects χ derivable from the calling NamedState object i.e., find all combinations of objects not in the calling NamedState object's ConstantAssignment ρ and variables in the underlying Vocabulary Σ object (a reference to it is held by ρ). If no VariableAssignments can be created, a dummy VariableAssignment χ_{dummy} is returned.

Returns A generator for all derivable VariableAssignment objects χ.

Return type generator

add_object(*obj*, *ascriptions=None*, *constant_symbol=None*)
Add an object s' to the calling NamedState object's underlying AttributeSystem $\mathcal{S}$, optionally update any ascriptions of the new object $\delta_i(s')$ provided and optionally bind it to a constant c' given by the constant_symbol parameter (if c' does not exist in the underlying Vocabulary object Σ, it will be added to Σ and furthermore all objects holding a reference to Σ will receive the update).

Parameters

- **obj** (str) – The new object s' to add to the NamedState.
- **ascriptions** (dict) – The optional ValueSets to assign to the attribute-object pairs corresponding to the new object $\delta_i(s')$.
- **constant_symbol** (str) – The optional constant c' to bind to the new object s'.

Raises

- **TypeError** – obj parameter must be a non-empty str, if ascriptions parameter is provided, it must be a dict and if constant_symbol parameter is provided, it must be a str.
- **ValueError** – Duplicate objects cannot be added, c' cannot be bound already and all ascriptions provided must be from an existing label of an Attribute object in the underlying AttributeSystem object $\mathcal{S}$ to s'.

get_named_alternate_extensions(**named_states*)
Obtain all alternate extensions of the calling NamedState object w.r.t. the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ provided in the named_states parameter, i.e., compute **AE**(Σ_i, σ) for the various applicable *i* according to algorithm.

Returns all alternative extensions of this NamedState object $(\sigma; \rho)$ w.r.t. the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ provided as optional positional arguments in the `named_states` parameter.

Return type `list`

Raises

- **`TypeError`** – `ns_prime` and all arguments provided as optional positional arguments to the `named_states` parameter must be NamedState objects.
- **`ValueError`** – At least one NamedState object must be provided in `named_states` parameter and all NamedState objects in `ns_prime` and `named_states` parameters must be proper extensions of this NamedState.

`get_worlds`()

Return a generator for the generation of all possible worlds $(w; \widehat{\rho})$ derivable from the calling NamedState object.

Returns A generator for the generation of all possible worlds $(w; \widehat{\rho})$ derivable from this NamedState object.

Return type `generator`

`is_exhaustive`(*basis*, **named_states*)

Determine if on some basis (i.e., a set of attribute-object pairs $\delta_i(s_j)$), a set of NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$ is exhaustive w.r.t the calling NamedState object $(\sigma; \rho)$, that is, if the ascriptions (ob objects in the basis) of the states $\sigma_1, \ldots, \sigma_n$ span the ascriptions of σ.

Parameters

- **`basis`** (`list`) – A list of attribute-object pairs $\delta_i(s_j)$.
- **`named_states`** (`NamedState`) – Any positive amount of NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$.

Returns Whether or not the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$ provided as optional positional arguments are exhaustive w.r.t. the calling NamedState object $(\sigma; \rho)$ on the basis provided in the `basis` parameter.

Return type `bool`

Raises **`ValueError`** – `basis` parameter cannot be empty and at least one NamedState object must be provided to `named_states` parameter.

`is_named_alternate_extension`(*ns_prime*, **named_states*)

Determine if the NamedState object in the `ns_prime` parameter $(\sigma';\ \rho')$ is an alternate extension of the calling NamedState object $(\sigma; \rho)$ w.r.t. the NamedState objects provided in the `named_states` parameter $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$, i.e., evaluate $\mathit{Alt}((\sigma; \rho), \{(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)\}, (\sigma'; \rho'))$

Returns The result of the evaluation of $\mathit{Alt}((\sigma; \rho), \{(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)\}, (\sigma'; \rho'))$

Return type `bool`

Raises

- **`TypeError`** – `ns_prime` and all arguments provided as optional positional arguments to `named_states` parameter must be NamedState objects.
- **`ValueError`** – At least one NamedState object must be provided to `named_states` parameter and all NamedState objects in `ns_prime` and `named_states` parameters must be proper extensions of the calling NamedState object.

is_named_entailment(*assumption_base*, *attribute_interpretation*, **named_states*)

Determine if the calling NamedState object $(\sigma; \rho)$ entails the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ provided as optional positional arguments to the `named_states` parameter w.r.t. the AssumptionBase object β in the `assumption_base` parameter, using the AttributeInterpretation object I provided in the `attribute_interpretation` parameter to resolve truth values of the Formula objects contained therein, i.e., evaluate $(\sigma; \rho) \Vdash_\beta \{(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)\}$.

Parameters

- **assumption_base** (AssumptionBase) – The AssumptionBase object β to use when evaluating $I_{(\sigma';\rho')/\chi}\left(\bigwedge_{F \in \beta} F\right) =$ **false** for all χ.
- **attribute_interpretation** (AttributeInterpretation) – The AttributeInterpretation object I to use to resolve the truth values of Formula objects in the AssumptionBase object β in `assumption_base` parameter.
- **named_states** (NamedState) – Any amount of NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ to check for entailment.

Returns Whether or not $(\sigma; \rho) \Vdash_\beta \{(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)\}$

Return type `bool`

Raises

- **TypeError** – `assumption_base` parameter must be an AssumptionBase object, `attribute_interpretation` parameter must be an AttributeInterpretation object, and all optional positional arguments in `named_states` parameter must be NamedState objects.
- **ValueError** – All NamedState objects provided as optional positional arguments to the `named_states` parameter $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ must share the same underlying Vocabulary object Σ, have equivalent AttributeSystem objects $\mathcal{S}$ and be proper extensions of the calling NamedState object $(\sigma; \rho)$, that is, $(\sigma_i; \rho_i) \sqsubset (\sigma; \rho)$ for $i = 1, \ldots, m$.

is_world()

Determine if the calling NamedState object $(\sigma; \rho)$ is a world; that is every ascription δ_i of σ is a valuation and ρ is total (that is, $\rho \to \widehat{\rho}$ holds).

Returns Whether or not the calling NamedState object is a world.

Return type `bool`

satisfies_context(*context*, *X*, *attribute_interpretation*)

Determine if this NamedState object $(w; \widehat{\rho})$ (which must be a world) satisfies the given Context object in the `context` parameter $\gamma = (\beta; (\sigma; \rho))$ w.r.t. a given VariableAssignment object χ and given AttributeInterpretation object I, i.e., $(w; \widehat{\rho}) \models_\chi \gamma$ (the calling NamedState object $(w; \widehat{\rho})$ satisfies every Formula object in the AssumptionBase object of the Context object β and the calling NamedState object satisfies the NamedState object of the Context object $(\sigma; \rho)$, that is, $(w; \widehat{\rho}) \models_\chi F$ for every $F \in \beta$ and $(w; \widehat{\rho}) \models (\sigma; \rho)$).

Parameters

- **context** (Context) – The context γ to check for satisfaction.
- **X** (VariableAssignment) – The variable assignment χ
- **attribute_interpretation** (AttributeInterpretation) – The fixed attribute interpretation I to use for interpreting which constants and variables are substituted into the formula for evaluation.

Returns Whether or not $(w; \widehat{\rho}) \models_\chi \gamma$.

Return type `bool`

Raises

- **TypeError** – `context` parameter must be a Context object, `X` parameter must be a VariableAssignment object and `attribute_interpretation` must be an AttributeInterpretation object.
- **ValueError** – The calling NamedState object must be a world.

satisfies_formula(*formula*, *X*, *attribute_interpretation*)

Determine if the calling NamedState object $(w; \widehat{\rho})$ (which must be a world) satisfies the given Formula object F in the `formula` parameter w.r.t. the VariableAssignment object χ in the `X` parameter and given AttributeInterpretation I in the `attribute_interpretation` parameter, i.e., $(w; \widehat{\rho}) \models_{\chi} F$.

Parameters

- **formula** (Formula) – The formula F to check for satisfaction.
- **X** (VariableAssignment) – The variable assignment χ
- **attribute_interpretation** (AttributeInterpretation) – The fixed attribute interpretation I to use for interpreting which constants and variables are substituted into the formula F for evaluation.

Returns Whether or not $(w; \widehat{\rho}) \models_{\chi} F$.

Return type `bool`

Raises

- **TypeError** – `formula` parameter must be a Formula object, `X` parameter must be a VariableAssignment object and `attribute_interpretation` parameter must be an AttributeInterpretation object.
- **ValueError** – The calling NamedState object must be a world.

satisfies_named_state(*named_state*)

Determine if the calling NamedState object $(w; \widehat{\rho})$ (which must be a world) satisfies the given NamedState object $(\sigma; \rho)$ i.e., $(w; \widehat{\rho}) \models (\sigma; \rho)$.

Parameters **named_state** (NamedState) – The named state $(\sigma; \rho)$ to check for satisfaction.

Returns Whether or not $(w; \widehat{\rho}) \models (\sigma; \rho)$.

Return type `bool`

Raises

- **TypeError** – `named_state` parameter must be a NamedState object.
- **ValueError** – The calling NamedState object must be a world.

CHAPTER

NINE

ATTRIBUTE INTERPRETATIONS

9.1 The AttributeInterpretation object

attribute_interpretation module.

class attribute_interpretation.**AttributeInterpretation**(*vocabulary*, *attribute_structure*, *mapping*, *profiles*)

AttributeInterpretation class. Build an interpretation table; that is, a mapping I that assigns, to each RelationSymbol object of a Vocabulary object Σ, $R \in$ R of arity n:

1. a relation $R^I \in \mathcal{R}$ of some arity *m*, called the **realization** of *R*:

$$R^I \subset A_{i_1} \times \cdots \times A_{i_m}$$

(where we might have $m \neq n$); and

2. a list of *m* pairs

$$[(l_{i_1}; j_1) \cdots (l_{i_m}; j_m)]$$

called the **profile** of *R* and denoted by $Prof(R)$, with $1 \leq j_x \leq n$ for $x = 1, \ldots, m$

Variables

- ***vocabulary*** – The Vocabulary object Σ of the interpretation.
- ***attribute_structure*** – The AttributeStructure object the interpretation is into.
- **mapping** – The mapping R $\to \mathcal{R}$.
- **profiles** – A list of profiles $[(l_{i_1}; j_1) \cdots (l_{i_m}; j_m)]$; one for each realization.
- **table** – The interpretation table of the attribute interpretation.
- **relation_symbols** – A copy of the RelationSymbol objects from Σ (for convenient access).
- **is_AttributeInterpretation** – An identifier to use in place of `type` or `isinstance`.

__deepcopy__(*memo*)

Deepcopy an AttributeInterpretation object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

__eq__(*other*)

Determine if two AttributeInterpretation objects are equal via the == operator.

__init__(*vocabulary*, *attribute_structure*, *mapping*, *profiles*)

Construct an AttributeInterpretation object.

Parameters

- **vocabulary** (Vocabulary) – The Vocabulary object Σ to define the AttributeInterpretation over.
- **attribute_structure** (AttributeStructure) – The AttributeStructure object for which to define the AttributeInterpretation into.
- **mapping** (dict) – The mapping from the RelationSymbol objects of the Vocabulary object Σ in the vocabulary parameter to the Relation objects of the AttributeStructure object in the attribute_structure parameter; the keys being RelationSymbol objects and values being ints corresponding to the subscripts of the AttributeStructure Relation objects.
- **profiles** (list) – A list of realizations corresponding to the mapping wherein each element is a list where the first element is the RelationSymbol and the following elements are 2-tuples $(l_{i_k}; j_k)$.

Raises

- **TypeError** – vocabulary parameter must be a Vocabulary object, attribute_structure parameter must be an AttributeStructure object, mapping parameter myst be a dict and profile parameter must be a list.
- **ValueError** – All keys in the mapping parameter must be RelationSymbol objects and all values must be unique ints, duplicate profiles are not permitted (determined by repeated RelationSymbol objects), the set of RelationSymbol's provided in the mapping parameter and the set of RelationSymbol's in the vocabulary parameter and the set of RelationSymbol's provided in the profile parameter must all be equal, all subscripts provided in the mapping parameter keys must be valid subscripts of Relation objects in the attribute_structure parameter, all l_{i_k} in $(l_{i_k}; j_k)$ 2-tuples provided in each realization of the profile parameter must be a label of some Attribute object in the attribute_structure parameter, and all j_k in $(l_{i_k}; j_k)$ 2-tuples provided in each realization of the profile parameter must be between 1 and the arity of the RelationSymbol object of the realization.

__iter__()
Provide an iterator for the interpretation table of AttributeInterpretation objects.

(e.g. "for entry in attribute_interpretation:")

__ne__(*other*)
Determine if two AttributeInterpretation objects are not equal via the != operator.

__repr__()
Return a string representation of the AttributeInterpretation object.

__str__()
Return a readable string representation of the AttributeInterpretation object.

CHAPTER

TEN

FORMULAE AND ASSUMPTION BASES

10.1 The Formula object

formula module.

class formula.**Formula**(*vocabulary*, *name*, **terms*)

Formula class. Formula objects are defined over some Vocabulary Σ. Formula objects are immutable.

Variables

- ***vocabulary*** – The underlying Vocabulary object Σ the Formula is defined over.
- **name** – The name of the Formula object.
- **terms** – The terms of the Formula object.
- **is_Formula** – An identifier to use in place of type or isinstance.

__add__(*other*)

Combine a Formula object and another Formula object or an AssumptionBase object into an AssumptionBase object via the + operator.

Raises

- **TypeError** – Only a Formula object or AssumptionBase object can be combined with a Formula object.
- **ValueError** – This Formula and other parameter must share the same underlying Vocabulary object Σ and duplicate Formula objects are not permitted (determined by name of the Formula object only).

__deepcopy__(*memo*)

Deepcopy a Formula object via the copy.deepcopy method. This does not break the reference to the underlying Vocabulary object Σ.

__eq__(*other*)

Determine if two Formula objects are equal via the == operator.

__hash__()

Hash implementation for set functionality of Formula objects.

__init__(*vocabulary*, *name*, **terms*)

Construct a Formula object.

Parameters

- **vocabulary** (Vocabulary) – The underlying Vocabulary object Σ the Formula is defined over.
- **name** (str) – The name (identifier) of the formula.

- **terms** (str) – Any amount of str constants and variables representing the terms of the formula.

Raises

- **TypeError** – vocabulary parameter must be a Vocabulary object.
- **ValueError** – name parameter must match some RelationSymbol object in the vocabulary parameter, at least one term must be provided and all terms provided must be in either the constants or variables of the vocabulary parameter Σ.

__ne__(*other*)
Determine if two Formula objects are not equal via the != operator.

__repr__()
Return a string representation of the NamedState object.

__str__()
Return a readable string representation of the NamedState object.

_key()
Private key function for hashing.

Returns tuple consisting of (name, $t_1, \ldots, t_n$)

Return type tuple

assign_truth_value(*attribute_interpretation*, *named_state*, *X*)
Assign a truth value in $\{$**true**, **false**, **unknown**$\}$ to the calling Formula object F given an arbitrary NamedState object $(\sigma; \rho)$ in the named_state parameter and VariableAssignment object χ in the X parameter w.r.t. an AttributeInterpretation object I.

This function makes use of the ParserSet object; the ParserSet object is a key part in the vivid object extension protocol.

The assign_truth_value function works as follows:

1. Find the entry in the interpretation table of the AttributeInterpretation object I in the attribute_interpretation parameter and extract the corresponding profile and Relation object (the 3rd element of the corresponding row of the table is the identifier for the Relation object; e.g. $R_{subscript}$).

2. Substitute the terms of the Formula object F into the profile (the 2nd element of each pair in the profile corresponds to the index of the term in the F to use, shifted down by 1).

3. Using the ConstantAssignment object of the named_state parameter ρ and the VariableAssignment in the X parameter χ, substitute for each term now in the profile, the object corresponding to that term given by the mapping in ρ or χ (if the term is in neither ρ nor χ, "unknown" is returned as the truth value).

4. The profile now consists of the attribute-object pairs $(\delta_i(s_j)$ for some set of the possible values of i and j) to use in the Relation object's definition when creating the evaluatable expression. Now, all worlds $(w; \widehat{\rho})$ derivable from the NamedState are generated and the ValueSets of the attribute-object pairs in the profile (consisting of single elements) are extracted from the ascriptions of these worlds.

5. The single element ValueSets are zipped together with the arguments in the Relation object definition (the ith attribute-object pair of the profile is zipped with the ith argument of the definition) and these new argument-ValueSet pairs are used to substitute every occurance of each argument in the definition with the corresponding single element ValueSet creating a (hopefully) evaluatable expression (the RHS of the substituted definition) for each world $(w; \widehat{\rho})$.

6. Each parser in the ParserSet object will then try to evaluate the expression and save the truth value for each $(w; \widehat{\rho})$. If some expression is unevaluatable for all parsers in the ParserSet a ValueError is raised.

7. If the expression of every world $(w; \widehat{\rho})$ evaluates to True, the truth value returned is **true**, if the expression of every world evaluates to False, the truth value returned is **false** and if the expressions of any two worlds evaluate to different values, the truth value returned is **unknown**.

Returns A truth value in the set $\{$**true**, **false**, **unknown**$\}$

Return type `bool | str`

Raises

- **`TypeError`** – `attribute_interpretation` parameter must be an AttributeInterpretation object, `named_state` parameter must be a NamedState object and `X` parameter must be a VariableAssignment object.
- **`ValueError`** – This Formula object, the AttributeInterpretation object in the `attribute_interpretation` parameter, the NamedState object in the `named_state` parameter and the VariableAssignment object in the `X` parameter must all share the same underlying Vocabulary object (that is F, I, $(\sigma; \rho)$ and χ must all share the same Σ), the Formula object must match an entry in the interpretation table of the AttributeInterpretation I in the

`attribute_interpretation` parameter, the number of attribute-object pairs in the profile corresponding to the Formula must match the arity of the corresponding Relation object found in the table (where the Relation object is found in the AttributeStructure object in the AttributeSystem member of the `named_state` parameter), $1 \leq j_x \leq n$ for each j_x in the profile (where n is the arity of the RelationSymbol corresponding to the RelationSymbol object matching the Formula in the interpretation table, or equivalently, the number of terms in the Formula object) and a parser in the ParserSet object must be able to evaluate the expression obtained after substituting the objects of the AttributeSystem in the `named_state` parameter, corresponding to the terms of the Formula, into the Relation object's definition.

static `get_basis` (*constant_assignment*, *variable_assignment*, *attribute_interpretation*, **formulae*)

Get the basis of the Formula objects $F_1, \ldots, F_k$ provided as optional positional arguments in the `formulae` parameter w.r.t. the ConstantAssignment object ρ provided in the `constant_assignment` parameter, VariableAssignment object χ provided in the `variable_assignment` parameter, and the AttributeInterpretation object I provided in the `attribute_interpretation` parameter, i.e., compute $\mathcal{B}(F_1, \rho, \chi) \cup \cdots \cup \mathcal{B}(F_k, \rho, \chi)$.

Parameters

- **`constant_assignment`** (`ConstantAssignment`) – The ConstantAssignment object ρ to use to compile the profile corresponding to each Formula object $F_i, i = 1, \ldots, k$ into attribute-object pairs to consider for the basis.
- **`variable_assignment`** (VariableAssignment | `None`) – The VariableAssignment object χ to use to compile the profile corresponding to each Formula object $F_i, i = 1, \ldots, k$ into attribute-object pairs to consider for the basis or `None`.
- **`attribute_interpretation`** (`AttributeInterpretation`) – The AttributeInterpretation object I to use to determine the profiles corresponding to the Formula objects $F_1, \ldots, F_k$ provided (the profile is extracted from the interpretation table when the RelationSymbol matching the Formula object's name is found).
- **`formulae`** (`Formula`) – Any positive amount of Formula objects $F_1, \ldots, F_k$ to consider in the basis.

Returns A list of attribute-object pairs comprising the basis of the Formula objects $F_1, \ldots, F_k$ provided w.r.t. ρ and χ.

Return type `list`

Raises

- **TypeError** – `constant_assignment` parameter must be a ConstantAssignment object, and all optional positional arguments provided in the `formulae` parameter must be Formula objects.
- **ValueError** – At least one Formula object must be provided and all Formula objects provided must match some entry in the interpretation table of the AttributeInterpretation object I.

10.2 The AssumptionBase object

assumption_base module.

class `assumption_base.`**AssumptionBase**(**formulae*)

AssumptionBase class.

An AssumptionBase object functions as a container for a finite set of Formula objects $F_1, \ldots, F_k$ over a single underlying Vocabulary Σ, denoted β.

Variables

- **formulae** – The set of Formula objects $F_1, \ldots, F_k$ contained in the AssumptionBase object β.
- ***vocabulary*** – The underlying Vocabulary object Σ the AssumptionBase object β is defined over.
- **_is_AssumptionBase** – An identifier to use in place of `type` or `isinstance`.

__add__(*other*)

Add all Formula objects in another AssumptionBase object or a single Formula object to the calling AssumptionBase object via the + operator.

Raises

- **TypeError** – Only Formula objects or AssumptionBase objects can be added to the calling AssumptionBase object.
- **ValueError** – Cannot add objects with different underlying Vocabulary objects and duplicate Formula objects are not permitted.

__contains__(*item*)

Overloaded `in` operator for AssumptionBase. Determine if a Formula object is contained in the calling AssumptionBase object.

Parameters **key** (Formula | `str`) – The Formula object or the name of a Formula object to test for membership in the calling AssumptionBase object.

__deepcopy__(*memo*)

Deepcopy an AssumptionBase object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

__eq__(*other*)

Determine if two AssumptionBase objects are equal via the == operator.

__getitem__(*key*)

Retrive the Formula object corresponding to the key given by the `key` parameter via indexing (e.g. `AssumptionBase[key]`).

Parameters **key** (int | str | Formula) – The key to use for indexing in the calling AssumptionBase object.

Raises

- **IndexError** – int key is out of range.
- **KeyError** – key parameter does not correspond to any Formula object in the AssumptionBase.
- **TypeError** – key parameter must be an int, str, or Formula object.

__iadd__(*other*)

Add all Formula objects in another AssumptionBase object or a single Formula object to the calling AssumptionBase object via the + operator.

Raises

- **TypeError** – Only Formula objects or AssumptionBase objects can be added to the calling AssumptionBase object.
- **ValueError** – Cannot add objects with different underlying Vocabulary objects and duplicate Formula objects are not permitted.

__init__(**formulae*)

Construct an AssumptionBase object.

Parameters **formulae** (*Formula* | *Vocabulary*) – Any amount of Formula objects $F_1, \ldots, F_k$ or a single Vocabulary object Σ no Formula objects are provided.

Raises

- **TypeError** – All optional positional arguments provided must be Formula objects or a single Vocabulary object Σ.
- **ValueError** – All Formula objects provided as optional positional arguments must share the same Vocabulary object Σ.

__iter__()

Provide an iterator for AssumptionBase objects (e.g. "for formula in AssumptionBase:").

__len__()

Determine the length of an AssumptionBase object β (i.e., the amount of Formula objects contained in β) via the len built-in function e.g.(len(AssumptionBase)).

__ne__(*other*)

Determine if two AssumptionBase objects are not equal via the != operator.

__repr__()

Return a string representation of the AssumptionBase object.

__str__()

Return a readable string representation of the AssumptionBase object.

CHAPTER

ELEVEN

CONTEXTS

11.1 The Context object

context module.

class context.**Context**(*assumption_base*, *named_state*)

Context class. A Context is a pair composed of an AssumptionBase object β and a NamedState object $(\sigma; \rho)$, i.e., $\gamma = (\beta; (\sigma; \rho))$.

Variables

- ***assumption_base*** – The AssumptionBase object β of the Context object.
- ***named_state*** – The NamedState object $(\sigma; \rho)$ of the Context object.
- **is_Context** – An identifier to use in place of `type` or `isinstance`.

__deepcopy__(*memo*)

Deepcopy a Context object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

__eq__(*other*)

Determine if two Context objects are equal via the == operator.

__init__(*assumption_base*, *named_state*)

Construct a Context object.

Parameters

- **assumption_base** (AssumptionBase) – The AssumptionBase object β to use in the Context object.
- **named_state** (NamedState) – The NamedState object $(\sigma; \rho)$ to use in the Context object.

Raises

- **TypeError** – `assumption_base` parameter must be an AssumptionBase object and `named_state` parameter must be a NamedState object.
- **ValueError** – The underlying Vocabulary objects of the `assumption_base` and `named_state` parameters must be the same Vocabulary object Σ.

__ne__(*other*)

Determine if two Context objects are not equal via the != operator.

__repr__()

Return a string representation of the Context object.

__str__()
Return a readable string representation of the Context object.

entails_formula(*formula*, *attribute_interpretation*)
Determine if the calling Context object $\gamma = (\beta; (\sigma; \rho))$ entails the Formula object F provided in the `formula` parameter, using the AttributeInterpretation object I provided in the `attribute_interpretation` parameter to interpret truth values, i.e., determine if $\gamma \models F$.

Parameters

- **formula** (Formula) – The Formula object F to check for entailment.
- **attribute_interpretation** (AttributeInterpretation) – The AttributeInterpretation object I to use for the interpretation of truth values during the evauation of $\gamma \models F$.

Returns Whether or not $\gamma \models F$, that is, whether or not $(w; \widehat{\rho}) \models_\chi \gamma$ implies $(w; \widehat{\rho}) \models_\chi F$ for all worlds $(w; \widehat{\rho})$ and variable assignments χ.

Return type bool

Raises

- **TypeError** – `formula` parameter must be a Formula object and `attribute_interpretation` parameter must be an AttributeInterpretation object.
- **ValueError** – The calling Context object and the Formula object F provided in the `formula` parameter must share the same underlying Vocabulary object Σ.

entails_named_state(*named_state*, *attribute_interpretation*)
Determine if the calling Context object $\gamma = (\beta; (\sigma; \rho))$ entails the NamedState object $(\sigma'; \rho')$ provided in the `named_state` parameter, using the AttributeInterpretation object I provided in the `attribute_interpretation` parameter to interpret truth values, i.e., determine if $\gamma \models (\sigma'; \rho')$.

Parameters

- **named_state** (NamedState) – The NamedState object $(\sigma'; \rho')$ to check for entailment.
- **attribute_interpretation** (AttributeInterpretation) – The AttributeInterpretation object I to use for the interpretation of truth values during the evauation of $\gamma \models (\sigma'; \rho')$.

Returns Whether or not $\gamma \models (\sigma'; \rho')$, that is for all worlds $(w; \widehat{\rho})$ and variable assignments χ, $(w; \widehat{\rho}) \models (\sigma'; \rho')$ whenever $(w; \widehat{\rho}) \models_\chi \gamma$.

Return type bool

Raises

- **TypeError** – `named_state` parameter must be a NamedState object and `attribute_interpretation` parameter must be an AttributeInterpretation object.
- **ValueError** – The calling Context object and the NamedState object $(\sigma'; \rho')$ provided in the `named_state` parameter must share the same underlying Vocabulary object Σ.

CHAPTER

TWELVE

RULES OF INFERENCE FOR DIAGRAMMATIC DEDUCTIONS

12.1 The [Thinning] rule

inference_rules.**thinning**(*context*, *named_state*, *assumption_base=None*, *attribute_interpretation=None*)

Verify that the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter can be obtained by thinning from the NamedState object $(\sigma; \rho)$ contained in the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter w.r.t. the AssumptionBase object $\{F_1, \ldots, F_n\}$ given by the `assumption_base` parameter, using the AttributeInterpretation object I in the `attribute_interpretation` parameter to interpret truth values.

By Corollary 26, if $(\sigma; \rho) \Vdash_{\{F_1,\ldots,F_n\}} (\sigma'; \rho')$ then $(\{F_1, \ldots, F_n\}; (\sigma; \rho)) \models (\sigma'; \rho')$.

Then, by weakening, $(\beta \cup \{F_1, \ldots, F_n\}; (\sigma; \rho)) \models (\sigma'; \rho')$ and thinning holds, thus it suffices to show that a call to `entails_named_state` with context $(\{F_1, \ldots, F_n\}; (\sigma; \rho))$ and named state $(\sigma'; \rho')$, that is $(\sigma; \rho) \Vdash_{\{F_1,\ldots,F_n\}} (\sigma'; \rho')$, holds to show that thinning holds.

Parameters

- **context** (Context) – The Context object $(\beta; (\sigma; \rho))$.
- **named_state** (NamedState) – The NamedState object $(\sigma'; \rho')$
- **assumption_base** (AssumptionBase | None) – The set of Formula objects to thin with $\{F_1, \ldots, F_n\}$ if thinning is to be done with any Formula (i.e., $n > 0$), otherwise None.
- **attribute_interpretation** (AttributeInterpretation | None) – The AttributeInterpretation object I to use to interpret truth values if $n > 0$, otherwise None.

Returns Whether or not thinning holds, i.e., the result of $(\sigma; \rho) \Vdash_{\{F_1,\ldots,F_n\}} (\sigma'; \rho')$

Return type bool

Raises **TypeError** – `context` parameter must be a Context object and `named_state` parameter must be a NamedState object.

12.2 The [Widening] rule

inference_rules.**widening**(*context*, *named_state*, *attribute_interpretation=None*)

Verify that the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter can be obtained from the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter by widening, using the AttributeInterpretation object I in the `attribute_interpretation` parameter to interpret truth values.

Parameters

- **context** (Context) – The Context object $(\beta; (\sigma; \rho))$.

- **named_state** (NamedState) – The NamedState object $(\sigma'; \rho')$
- **attribute_interpretation** (AttributeInterpretation | None) – The AttributeInterpretation object I to use to interpret truth values if widening should consider the AssumptionBase object of the `context` parameter, otherwise `None`.

Returns Whether or not the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter can be obtained from the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter by widening, i.e., whether or not $(\beta; (\sigma; \rho)) \models (\sigma'; \rho')$

Return type bool

Raises **TypeError** – `context` parameter must be a Context object and `named_state` parameter must be a NamedState object.

12.3 The Observe rule

inference_rules.**observe**(*context*, *formula*, *attribute_interpretation*)

Determine if the Formula object F given by the `formula` parameter can be observed in the Context object $(\beta; (\sigma; \rho))$ given by the `context` parameter, using the AttributeInterpretation object I in the `attribute_interpretation` parameter to interpret truth values, i.e., determine if **observe** F holds in $(\beta; (\sigma; \rho))$.

Parameters

- **context** (Context) – The Context object $(\beta; (\sigma; \rho))$ in which the Formula object F can potentially be observed.
- **formula** (Formula) – The (potentially) observable Formula object F.
- **attribute_interpretation** (AttributeInterpretation) – The AttributeInterpretation object I to use to interpet truth values in the `context` and `formula` parameters.

Returns Whether or not **observe** F holds in $(\beta; (\sigma; \rho))$, that is, whether or not $(\beta; (\sigma; \rho)) \models F$.

Return type bool

12.4 The [Absurdity] rule

inference_rules.**diagrammatic_absurdity**(*context*, *named_state*, *attribute_interpretation*)

Verify that the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter can be obtained from the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter by absurdity, using the AttributeInterpretation object I provided in the `attribute_interpretation` parameter to interpet truth values.

To show $(\sigma'; \rho')$ **by absurdity**, we must show $(\beta \cup \{\textbf{false}\}; (\sigma; \rho)) \models (\sigma'; \rho')$.

By lemma 20, $(\beta \cup \{\textbf{false}\}; (\sigma; \rho)) \models (\sigma'; \rho')$, thus it suffices to show that a call to `entails_named_state` with context $(\beta; (\sigma; \rho))$ and named state $(\sigma'; \rho')$, that is, $(\beta; (\sigma; \rho)) \models (\sigma'; \rho')$ holds, implicitly assuming that some $F \in \beta$ evaulates to **false** (as then no world can satisify the context, i.e., for any world $(w; \widehat{\rho})$ derivable from the context $(\beta; (\sigma; \rho))$, $(w; \widehat{\rho}) \not\models_\chi (\beta; (\sigma; \rho))$ and thus `entails_named_state` will always hold yielding $(\sigma'; \rho')$ **by absurdity** regardless of the NamedState object $(\sigma'; \rho')$ provided in the `named_state` parameter)

Parameters

- **context** (Context) – The Context object $(\beta; (\sigma; \rho))$.
- **named_state** (NamedState) – The NamedState object $(\sigma'; \rho')$.

- **attribute_interpretation** (AttributeInterpretation) –

Returns Whether or not $(\sigma'; \rho')$ **by absurdity**, that is, whether or not $(\beta; (\sigma; \rho)) \models (\sigma'; \rho')$ holds.

Return type bool

Raises **TypeError** – context parameter must be a Context object, named_state parameter must be a NamedState object and attribute_interpretation parameter must be an AttributeInterpretation object.

12.5 The [Diagram-Reiteration] rule

inference_rules.**diagram_reiteration**(*context*)

Perform $[Diagram - Reiteration]$ to retrieve the current diagram of the Context object $(\beta; (\sigma; \rho))$ provided in the context parameter, i.e., from lemma 19: $(\beta; (\sigma; \rho)) \models (\sigma; \rho)$.

Parameters **context** (Context) – The Context object $(\beta; (\sigma; \rho))$ from which to retrieve the current NamedState object $(\sigma; \rho)$.

Returns The NamedState object $(\sigma; \rho)$ of the Context object $(\beta; (\sigma; \rho))$ in context parameter.

Return type *NamedState*

12.6 The Sentential-to-Sentential rule

inference_rules.**sentential_to_sentential**(*context*, *F1*, *F2*, *G*, *attribute_interpretation*, *variable_assignment=None*)

Verify that a disjunction $F_1 \vee F_2$ holds in the Context object $(\beta; (\sigma; \rho))$ in the context parameter and that the Formula object G in the G parameter follows in either case, using the AttributeInterpretation object I in the attribute_interpretation parameter to interpret truth values.

To perform the **sentential-to-sentential** inference, first the disjunction $F_1 \vee F_2$ is verified. Then the truth values of $F_1 \Rightarrow G$ and $F_1 \Rightarrow G$ are determined. If either $F_1 \Rightarrow G$ or $F_1 \Rightarrow G$ do not hold, then **sentential-to-sentential** does not hold, otherwise, **sentential-to-sentential** holds.

Parameters

- **context** (Context) – The Context object $(\beta; (\sigma; \rho))$ in which the the Formula objects in the parameters F1, F2 apply and in which the Formula object in the G parameter would follow.
- **F1** (Formula) – The left operand of the disjunction F_1.
- **F2** (Formula) – The right operand of the disjunction F_2.
- **G** (Formula) – The Formula object G potentially following the disjunction in either case.
- **attribute_interpretation** (AttributeInterpretation) – The AttributeInterpretation object I to use for interpeting truth values.
- **variable_assignment** (VariableAssignment | None) – The optional VariableAssignment object χ to consider in the interpretation of truth values.

Returns Whether or not **sentential-to-sentential** holds.

Return type bool

Raises **ValueError** – The disjunction $F_1 \vee F_2$ does not hold.

12.7 The [C1] rule

inference_rules.**diagrammatic_to_diagrammatic**(*context*, *inferred_named_state*, *named_states*, *attribute_interpretation*, *variable_assignment*, **formulae*)

Verify that on the basis of the present diagram $(\sigma;\rho)$ of the Context object $(\beta;(\sigma;\rho))$ in the `context` parameter and some set of Formula objects $F_1,\ldots,F_k, k \geq 0$ provided as optional positional arguments in the `formulae` parameter, that for each NamedState object $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n), n > 0$, contained in the `named_states` parameter, a NamedState object $(\sigma';\rho')$ provided in the `inferred_named_state` parameter can be derived in every one of these n cases.

This is rule [C1].

This function works as follows:

1. If $k > 0$ (i.e., if at least one Formula object is provided as an optional positional argument to the `formulae` parameter), compute the basis $\mathcal{B}(F_1,\rho,\chi) \cup \cdots \cup \mathcal{B}(F_k,\rho,\chi)$ of $F_1,\ldots,F_k$ and determine if the NamedState objects $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n)$ provided in the `named_states` parameter form an exhaustive set of possibilities on this basis.

2. Determine if the proviso $(\sigma;\rho) \Vdash_{\{F_1,\ldots,F_k\}} \{(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n)\}$ (where $k \geq 0$) holds.

3. Return the evaluation of $(\beta \cup \{F_1,\ldots,F_k\};(\sigma;\rho)) \models (\sigma';\rho')$.

Parameters

- **context** (Context) – The Context object $(\beta;(\sigma;\rho))$ from which the present diagram $(\sigma;\rho)$ comes from.
- **inferred_named_state** (NamedState) – The NamedState object $(\sigma';\rho')$ derivable in the $n > 0$ cases provided by the NamedState objects $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n)$ in the `named_state` parameter.
- **named_states** (list) – The NamedState objects $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n), n > 0$ functioning as the set of n exhaustive cases from which F can be derived.
- **attribute_interpretation** (AttributeInterpretation) – The AttributeInterpretation object I to use for the interpretation of truth values and the computation of the basis of $F_1,\ldots,F_k$.
- **variable_assignment** (VariableAssignment | None) – The VariableAssignment object χ to consider when computing the basis $\mathcal{B}(F_1,\rho,\chi) \cup \cdots \cup \mathcal{B}(F_k,\rho,\chi)$ of $F_1,\ldots,F_k$ or None if all terms of the $F_1,\ldots,F_k$ are in ρ.
- **formulae** (Formula) – The $k \geq 0$ Formula objects $F_1,\ldots,F_k$ to use in the computation of the basis, computation of the proviso and the evaluation of $(\beta \cup \{F_1,\ldots,F_k\};(\sigma;\rho)) \models (\sigma';\rho')$.

Returns The result of the evaluation of $(\beta \cup \{F_1,\ldots,F_k\};(\sigma;\rho)) \models (\sigma';\rho')$.

Return type bool

Raises **ValueError** – If $k > 0$, the NamedState objects $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n), n > 0$ are not exhaustive on the basis of the Formula objects $F_1,\ldots,F_k$ or the proviso $(\sigma;\rho) \Vdash_{\{F_1,\ldots,F_k\}} \{(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n)\}$ (where $k \geq 0$) does not hold.

12.8 The [C2] rule

inference_rules.**sentential_to_diagrammatic**(*context*, *F1*, *F2*, *named_state*, *attribute_interpretation*, *variable_assignment=None*)

Verify that a disjunction $F_1 \vee F_2$ holds in the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter and that the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter follows in either case, using the AttributeInterpretation object I in the `attribute_interpretation` parameter to interpet truth values.

This is rule [C2].

This function works as follows:

1. Verify that the disjunction $F_1 \cup F_2$ given by `F1` and `F2` parameters holds in the Context object $(\beta; (\sigma; \rho))$ given by `context` parameter.

2. Generate two new Context objects $\gamma_1 = (\beta_1; (\sigma; \rho))$ where $\beta_1 = \beta \cup F_1$ and $\gamma_2 = (\beta_2; (\sigma; \rho))$ where $\beta_2 = \beta \cup F_2$.

3. For all possible worlds $\left(w'; \widehat{\rho'}\right)$ and variable assignments χ of the NamedState object $(\sigma'; \rho')$, determine if the world $\left(w'; \widehat{\rho'}\right)$ satisfies both Context objects γ_1 and γ_2, that is $\left(w'; \widehat{\rho'}\right) \models \gamma_1 \bigwedge \left(w'; \widehat{\rho'}\right) \models \gamma_2$.

4. If any world $\left(w'; \widehat{\rho'}\right)$ and variable assignments χ of the NamedState object $(\sigma'; \rho')$ does not satisify both γ_1 and γ_2, then **sentential-to-diagrammatic** does not hold, otherwise, **sentential-to-diagrammatic** holds.

In this way, we capture the idea that any world $\left(w'; \widehat{\rho'}\right)$ and variable assignments χ of the NamedState object $(\sigma'; \rho')$ (and thus the NamedState object $(\sigma'; \rho')$ itself) follows from the context $(\beta \cup \{F_1 \vee F_2\}; (\sigma; \rho))$ in either case of the disjunction $F_1 \vee F_2$.

Parameters

- **context** (Context) – The Context object $(\beta; (\sigma; \rho))$ in which the Formula objects in the parameters `F1` and `F2` apply and in which the NamedState object $(\sigma'; \rho')$ in `named_state` parameter would follow.
- **F1** (Formula) – The left operand of the disjunction F_1.
- **F2** (Formula) – The right operand of the disjunction F_2.
- **named_state** (NamedState) – The NamedState object $(\sigma'; \rho')$ potentially following the disjunction in either case.
- **attribute_interpretation** (AttributeInterpretation) – The AttributeInterpretation object I to use for the interpretation of truth values.
- **variable_assignment** (VariableAssignment | None) – The optional VariableAssignment object χ to consider in the interpretation of truth values.

Returns Whether or not **sentential-to-diagrammatic** holds.

Return type bool

Raises **ValueError** – The disjunction $F_1 \vee F_2$ does not hold.

12.9 The [C3] rule

inference_rules.**diagrammatic_to_sentential**(*context*, *F*, *named_states*, *attribute_interpretation*, *variable_assignment*, **formulae*)

Verify that on the basis of the present diagram $(\sigma;\rho)$ of the Context object $(\beta;(\sigma;\rho))$ in the `context` parameter and some set of Formula objects $F_1,\ldots,F_k, k \geq 0$ provided as optional positional arguments in the `formulae` parameter, that for each NamedState object $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n), n > 0$, contained in the `named_states` parameter, a Formula object F provided in the `F` parameter can be derived in every one of these n cases.

This is rule [C3].

This function works as follows:

1. If $k > 0$ (i.e., if at least one Formula object is provided as an optional positional argument to the `formulae` parameter), compute the basis $\mathcal{B}(F_1,\rho,\chi) \cup \cdots \cup \mathcal{B}(F_k,\rho,\chi)$ of $F_1,\ldots,F_k$ and determine if the NamedState objects $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n)$ provided in the `named_states` parameter form an exhaustive set of possibilities on this basis.

2. Determine if the proviso $(\sigma;\rho) \Vdash_{\{F_1,\ldots,F_k\}} \{(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n)\}$ (where $k \geq 0$) holds.

3. Return the evaluation of $(\beta \cup \{F_1,\ldots,F_k\};(\sigma;\rho)) \models F$.

Parameters

- **context** (Context) – The Context object $(\beta;(\sigma;\rho))$ from which the present diagram $(\sigma;\rho)$ comes from.
- **F** (Formula) – The Formula object F derivable in the $n > 0$ cases provided by the NamedState objects $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n)$ in the `named_state` parameter.
- **named_states** (list) – The NamedState objects $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n), n > 0$ functioning as the set of n exhaustive cases from which F can be derived.
- **attribute_interpretation** (AttributeInterpretation) – The AttributeInterpretation object I to use for the interpretation of truth values and the computation of the basis of $F_1,\ldots,F_k$.
- **variable_assignment** (VariableAssignment | None) – The VariableAssignment object χ to consider when computing the basis $\mathcal{B}(F_1,\rho,\chi) \cup \cdots \cup \mathcal{B}(F_k,\rho,\chi)$ of $F_1,\ldots,F_k$ or None if all terms of the $F_1,\ldots,F_k$ are in ρ.
- **formulae** (Formula) – The $k \geq 0$ Formula objects $F_1,\ldots,F_k$ to use in the computation of the basis, computation of the proviso and the evaluation of $(\beta \cup \{F_1,\ldots,F_k\};(\sigma;\rho)) \models F$.

Returns The result of the evaluation of $(\beta \cup \{F_1,\ldots,F_k\};(\sigma;\rho)) \models F$.

Return type bool

Raises **ValueError** – If $k > 0$, the NamedState objects $(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n), n > 0$ are not exhaustive on the basis of the Formula objects $F_1,\ldots,F_k$ or the proviso $(\sigma;\rho) \Vdash_{\{F_1,\ldots,F_k\}} \{(\sigma_1;\rho_1),\ldots,(\sigma_n;\rho_n)\}$ (where $k \geq 0$) does not hold.

CHAPTER

THIRTEEN

PARSERS

13.1 The ParserSet object

parser_set module.

class parser_set.**ParserSet**
ParserSet class. The ParserSet object functions as a sequence/collection. The ParserSet class is part of the vivid object extension protocol.

Variables

- **parsers** – The parsers contained in the ParserSet object.
- **_is_ParserSet** – An identifier to use in place of `type` or `isinstance`.

__getitem__(*key*)
Retrive the parser located at the index given by `key` parameter via indexing (e.g. `ParserSet[key]`).

Parameters **key** (`int`) – The index to use for retrieval.

Raises **TypeError** – `key` parameter must be an index.

__init__()
Construct a ParserSet object.

__iter__()
Provides an iterator for ParserSet (e.g. "`for parser in ParserSet:`").

__len__()
Determine the length of the ParserSet object via the `len` built-in function e.g.(`len(ParserSet)`).

13.2 The PointParser object

point_parser module.

class point_parser.**PointParser**
PointParser class. The PointParser class is used for parsing Point object related expressions.

Variables **_is_Parser** – An identifier to use in place of `type` or `isinstance`.

__call__(**args*)
Call PointParser object (e.g., `PointParser(expression)`).

__init__()
Construct a PointParser object.

_eval(*string*)
Try to evaluate given string (e.g., "`is_on(P(2.0,2.0),P(1.0,1.0),P(3.0,3.0))`").

Parameters **string** (`str`) – The expression to evaluate; the PointParser object unstringifies Point objects in `string` parameter and tries to call a function of the Point object (also given by `string` parameter) with unstringified Points as arguments.

Raises **ValueError** – Function provided in `string` parameter is not a function in the Point class, some argument is not a Point after trying to unstringify or the `string` parameter is improperly formatted.

13.3 The TruthValueParser object

truth_value_parser module.

class `truth_value_parser.`**TruthValueParser**
TruthValueParser class. TruthValueParser provides parsing functionality for entirely mathematical/logical strings.

Variables **_is_Parser** – An identifier to use in place of `type` or `isinstance`.

__call__(**args*)
Call TruthValueParser object (e.g., `TruthValueParser(expression)`).

__init__()
Construct a TruthValueParser object.

_eval(*string*)
Try to evaluate given string in `string` parameter. (e.g.,"`(4 < 5 * cos(2 * PI) and 4*e^3 > 3 *(3 + 3))and!(2 < 3)`").

Parameters **string** (`str`) – The expression to evaluate.

a

c

f

i

n

p

r

s

t

v

Symbols

A

C

D

E

F

G