# Язык СИНТЕЗ как ядро канонической информационной модели

*Л. А. Калиниченко (leonidk@synth.ipi.ac.ru)*
*С. А. Ступников (sstupnikov@ipiran.ru)*

## Lecture outline

- Objectives of the language
- Frame language: semi-structured capabilities
- Type system of SYNTHESIS
- Classes and metaclasses
- Assertions
- Processes, workflows
- Formulae facilities of the language

## History of the language

- Preliminary draft version published in 1991
- First version published in 1993
- English version prepared in 1995 and extended in 1997
- Current version in English published in 2007

## The SYNTHESIS is a multipurpose language

The language is oriented on the following basic kinds of application:

- serving as a **kernel of the canonical information model**;
- providing facilities for **unifying representation** of heterogeneous information models of different kinds (of data, services, processes, ontologies);
- providing sufficient **modeling facilities for formalized definitions of mediators** for various subject domains;
- supporting **mediation-based design of information systems**;
- providing **modeling facilities for mediator and resource specifications** for the mediation architecture;
- serving for **semantic reconciliation of the mediator and resource specifications** to form compositions of resources refining mediator specifications;
- serving as an **interface for users during problem formulation and solving in various applications**

## Mediation orientation

▪ **Two different approaches** to the problem of integrated representation of multiple information resources for an IS are distinguished:

1. moving from resources to problems (**resource driven approach**)
2. moving from an application to resources (**application driven approach**)

The SYNTHESIS language is oriented on support of **application-driven approach for mediation-based IS development.**

▪ The **mediator's layer is introduced** to provide the users with the metainformation uniformly characterizing **subject definitions in the canonical information model – providing application domain conceptual schema**

▪ Each mediator **supports the process of systematic registration and classification of resources providing the uniform ontological knowledge and metainformation for discovery and composition of information resources relevant to mediator.**

## Basic facilities of the language

**A set of the SYNTHESIS language basic facilities** used for uniform representation of mediator and information resources includes the following:

- *Frame representation facilities*.
- *Functions and collections (including sets).*
- *Class representation*.
- *Multiactivity (process) representation*.
- *Definition of assertions*.
- *Facilities for the logical formulae expressions and program formulation .*

## Hybrid object/semistructured information

Information on the entities and situations observed in a real world is represented in the mediator as a **collection of abstract values (immutable or mutable).**

In this range we can differentiate between:

- **collections of self-defined objects** or collections of frames (worlds);
- **worlds with fixed frame associations**;
- **classes containing partially typed objects** containing their own individual attributes that were not specified in a type of the class instance;
- **strictly typed classes** (a set of instance attributes is strictly fixed);
- **aggregates (associations of objects and frames).**

All objects and frames are assumed to belong to one and the same general type of entities - to a **type of abstract values**.

Objects are considered

- **to be a frame specialization providing for unique immutable identification**. This means that all operations for frames are applicable also to objects (but not vice versa).

- **a frame** at any moment of its life cycle **can be declared to belong to a certain class**. At that moment the frame becomes an object.

Capabilities of a mediator definition are ranged **from purely declarative to purely procedural way of thinking.**

## Example of hybrid information

**An object containing semi-structured information**

```
{objid;
  name: John;
  belongs-to: IBM;
  own-info: {duration-of-service: 21;
              project-involved: {problem solving over distributed
                                  heterogeneous information resources}
             }
}
```

## Mediator specification roles

Mediator specification (a result of the consolidation phase of mediator life cycle) serves for the following:

- **provides a conceptual schema** for problem formulation over the mediator;
- **serves as a conceptual model for identification of resources relevant to** the mediator and **construction of their refining mapping** into the mediator specification;
- serves **as a conceptual specification for design and implementation of a mediator specification part** that is not refined by any of the pre-existing resources.

## Structuring of IR specifications

**A mediator or information resource specification module** (further "module" for short) **is the basic unit of specification, import and compilation** in the SYNTHESIS language.

**Module specification structure**

- **Header**
- **Import**
- **Rename**
- **Frame section = frame list**
- **Type section = type specification list**
- **Function section = function declarator list**
- **Resource component specification section**

# Module structure example

{personnel; in: **module, mediator**;
  **import**: companies, projects, cities;

  **type**:{Person; …}, {Spouse; …}, {Employee; …} …;

  **function:** {salary; … }, {profit; …}, … ;

  **class_specification**: {entity-class; in: metaclass; …},
    {activity-class; in: metaclass}, {person; in: class; …}, {spouse; in: class; …}, {employee; in: class; … } … :

  **singleton_specification**: {service-A; …}, {service-B; …}, …;

}

## Frame language

## Basic elements of the frame language

**A frame is considered to be a symbolic model of a certain entity or a concept.**

- **a frame having no slots :**

{ < frame identifier >;}

- **a frame having no name:**

{< slot1 >:< list of slot1 values >;
< slot2 >:< list of slot2 values >;
...
< slotM >:< list of slotI values >;}

- **a frame having no values :**

{< frame identifier >;
< slot1 >:;
< slot2 >:;
...
< slotM >:;}

- **a frame containing values without any slots :**

{[< frame identifier >];
< value >; < value >; < value >; < value >}

## Metaframes

**Any frame component** (a frame itself, a slot or a value) **can have additional metainformation associated to it.**

**metaframe:**

```
{a;
metaframe b ; end
s : v;
};
```

**metaslot:**

```
{a;
s : v;
metaslot b; end
};
```

**metavalue:**

```
{a;
s : v metavalue b; end;
}
```

## Examples

```
{ Person;
    metaframe in: CIDOCOnto.Person end
  in: type;
  name: Name;
    metaslot in: CIDOCOnto.Person_Appelation end
  nationality: string;
    metaslot in: CIDOCOnto.Group end
  date_of_birth: time;
    metaslot in: CIDOCOnto.Birth end
  date_of_death: time;
    metaslot in: CIDOCOnto.Death end
  residence: Address;
     metaslot in: CIDOCOnto.Address end
};
{ Creator;
    metaframe in: CIDOCOnto.Person end
  in: type;
  supertype: Person;
  …
};
```

# Temporal category of the language

The frame base stores an information that is associated with the application domain states **in different discrete moments of time**. An information for each of such moment should be consistent.

With a frame or a frame component (a slot or a value) **a temporal information can be associated that is kept in a specific history slot of a metaframe, metaslot or a metavalue.** A value of the history slot is a temporal frame or generally a collection of temporal frames. **The structure of a temporal frame follows**:

{**valid** :< constant of a time slice >;
  **belief** :< constant of a time slice }

A temporal frame contains two temporal constants: ***valid*** and ***belief***, that define respectively **a time slice of existence** (during this time an ap-plication entity or its attribute corresponding to a frame or its component according to the user beliefs preserves its value) and **a time slice of belief** (during which the system "believes" into the existence of an entity or of its attribute).

# Temporal category of the language (2)

```
{ name: John;
  address:  Bigtown; {valid: 3-Sep-2000 / 1-Apr-2001; belief: 1-Apr-2001,  inf}
  metaslot
     history:
        Bigtown {valid: 3-Sep-2000 / inf; belief:  2-Feb-2001 /  1-Apr-2001},
        Beachy {valid:  1-Jun-1995 / 3-Sep-2000; belief:  2-Feb-2001 / inf)},
        Bigtown {valid: 26-Aug-1994 / 1-Jun-1995; belief:  2-Feb-2001 / inf},
        Bigtown {valid: 26-Aug-1994 / inf; belief: 27-Dec-1994 / 2-Feb-2001},
        Smallwille {valid: 3-Apr-1975 / 26-Aug-1994; belief: 27-Dec-1994 / inf },
        Smallwille {valid: 3-Apr-1975 / inf; belief: 4-Apr-1975 /  27-Dec-1994 }
  end
}
```

## Temporal category of the language (3)

**Temporal assignment of a frame:**

*{*[< frame identifier >; ][< **(at < constant of a time slice >)** >; ]
[< metaframe >][< frame body >] *}*

**Temporal associations**

Assume that x and y provide for unique identification of temporal frames (that can be associated to a frame, a slot or a value)

**before(x,y), equals(x,y), after(x,y), during(x,y), starts(x,y), finishes(x,y), meets(x,y), overlaps(x,y), before_starts(x,y), before_ends(x,y)**

*in*(*x* ; *y*) :- *starts*(*x* ; *y*) | *during*(*x* ; *y*) | *finishes*(*x* ; *y*)
*ge*(*x* ; *y*) :- *after* (*x* ; *y*) | *equals*(*x* ; *y*)
*le*(*x* ; *y*) :- *before*(*x* ; *y*) | *equals*(*x* ; *y*)
*ne*(*x* ; *y*) :- *after* (*x* ; *y*) | *before*(*x* ; *y*)

# Worlds and contexts

Frames that are contained in the frame base **are subdivided into the sub-collections (worlds).**

***{**< world identifier >;**
**in : world;**
**< collection of frames >**
***}***

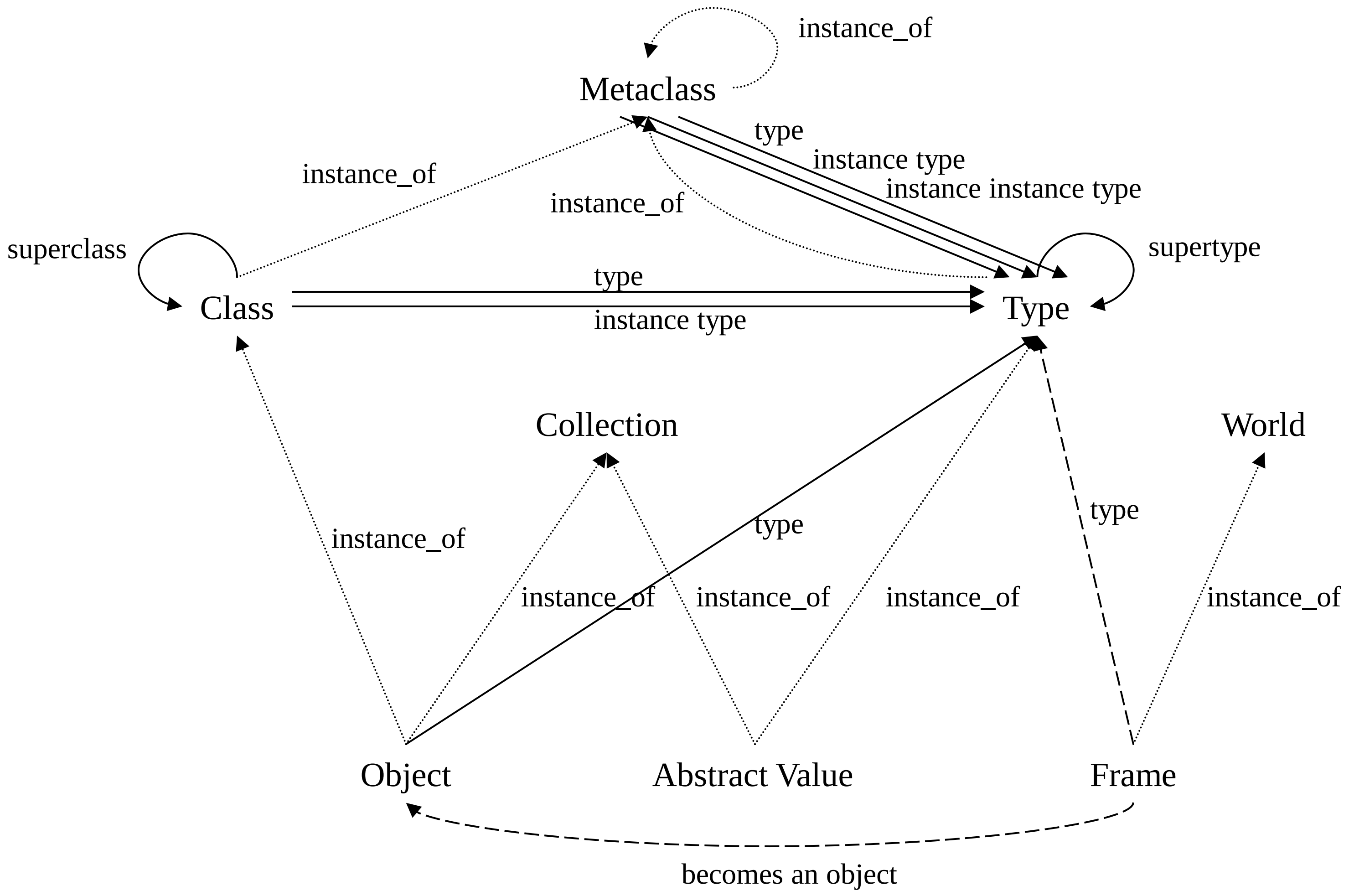
**Collections of worlds can form contexts**

***{**< context identifier >;**
**in : context;**
**member :< list of world or context identifiers >**
***}***

The association **member** ( **member of** is its inverse association) is used for establishing of **a membership of frames (or contexts) in contexts.** Contexts in their turn can participate in various associations to **form hierarchical or network structures.**

## Type system

# Canonical Model Entities



# ADT example

```
{Entity;
 in: type;
     title: Text;
    created_by: Creator;
    date: time;
    narrative: Text;
    identity: string;
    value:{in:function;
          params: {+e/Entity[title, created_by, date], -v/real}};
    language:  string;
    subject: Text;
}
```

## Attributes definition in ADT

```
< attribute identifier >:< attribute type >;
   [metaslot
     [card_numb : <unsigned integer >; ]
     [init :< constant >; ]
     [in :< attribute category name list >; ]
     [< assertion list >; ]
     [inverse :< type identifier >. < attribute identifier >; ]
   end]
```

```
{Employee; in: type;
 supertype: Person;
 belongs_to: company;
 metaslot
  b: invariant, {{obligatory;}}
 end
}
```

```
{Proj_ worker;
  in: type;
   supertype: Employee;
 works on: {set; type of element:Project};
   metaslot
     c: invariant, {{obligatory;}}
    workinv: invariant, {{this.works on <=
                this.belongs to.engaged_ in}}
   end
   }
```

# Attribute category definition

```
{mandatory_automatic;
in: metaclass, association;
inverse: automatic_mandatory;
    instance section:
        {association_type:{{0, inf}, {1,1}};
              domain: class;
              range: class; }}
```

*Automatic*: a transaction is created only as a member of *c.transactions_of_customer* of some customer *c*

*Mandatory:* if a transaction is removed from *c.transactions_of_customer* then it is automatically deleted

**Attribute categories are defined by association metaclasses**.
Specifications of classes *c1* and *c2* :

```
{transactions;
 in: class;
 instance section:
   {customer: customers.inst;
     metaslot
       in: automatic_mandatory;
       inverse:
        customers. transactions_of_customer ;
      end }
  }
```

```
{customers;
  in: class;
  instance section:
  { transactions_of_customer:
   {set; type of element: transactions.inst; };
    metaslot
     in: mandatory_automatic;
     inverse: transactions.customer;
    end }
  }
```

# Function Specification

<function section>::= function: <function declarator list>

<function declarator>::= {<identifier>; <function type>; <implementation> }

<function  type>::= in:<function kind>[,<metaclass name list>];
    [params: {<formal parameter list>};]
    [<function specification>;]

<function kind>::= function | predicate

<formal parameter identifier>::= <parameter kind symbol><typed variable>

<parameter kind symbol>::= - | + | <empty>

<function specification> ::= {predicative: {<formula>}}  | {{  <rule> }}

```
search_for_experts: { in: function;
 params: {-number_of_experts/integer};
 { predicative:
   {ex exps/({set; type_of_element: Expert;})(
      exps = { exp/Expert | expert(exp)  &
                       exp.research_area = ’computer science’} &
      this.available_experts’ = exps &
      number_of_experts = card(exps)) } };
};
```

## Classes and Metaclasses

## Sets and classes

**Mutable and immutable values**. Mutable types are subtypes of immutable.

**Sets in the language** (alongside with collections, bags, sequences) are considered to be a mechanism of grouping of abstract values (object if mutable)

**A class combines properties of a type and of a set.** A class supports a set of objects of a given type that constitutes an extent of the class

**Sets resemble classes**: collections of ADT values (immutable) or sets of objects. One and the same object can belong at the same time to several sets and to change dynamically its participation in them. **A set is distinguished from a class in the following:**

1. an object  cannot be created by sets;
2. for a set there is no difference between **the own and total** extent as for a class

At the same time in the SYNTHESIS language **a class is considered as a subtype of a set type.** Due to that a class can be used everywhere where a set can be used.

## Class relationships

**A class combines properties of a type and of a set.**

***Generalization/specialization*** relationship provides for the definition of an application domain structure as a hierarchy of interrelated classes.

Subclass objects are considered to be a specialized *category* of the superclass objects. The same real world objects are modelled by a subclass establishing for them an additional information w.r.t. superclass objects.

**The categorization is disjoint with respect to the criterion of categorization. E.g., a part belongs to a simple or to a complex part.**

An instance type of a subclass is a subtype of  an instance type of the superclass. Total extent of subclass is a subset of superclass extent.

**Class relationships**:

- **class - class instance relationship**;
- ***class - metaclass classification*** relationship that creates a classication hierarchy orthogonal to the generalization hierarchy of classes

## Instances of classes

Objects can be **completely typed** (in this case all their attributes are defined by a class instance type specification), **partially typed** (part of their attributes are defined by a class instance type specification and other attributes can be arbitrarily defined for different instances of the class) **or can be completely autonomous** (not associated to any class). Autonomous objects exist isolated or can be related to some worlds and are self-defined.

It is essential that an object can be an **instance of several classes** so that in each of them a structure and behavior of an object correspond to the class or are a specialization of a structure and behavior specified by a class.

**At the extreme object database can be treated as a classified set of frames with object identities.** At the same time autonomous objects are allowed that may not belong to any classes. Generally, the behaviors of objects belonging to classes and autonomous objects are similar. Autonomous object at any moment of its life cycle can be defined as belonging to a class using suitable instance type definition.

## Mediator or IR specification

**The IR component specifications is given by singleton (interface) specifications and/or by class specifications**. - by single objects as well as by collections of objects (classes).

A class specification combines information about **two kinds of objects**:
about **a class as an object** itself and **about objects** - instances of the class.

**Classes as objects themselves can have specific properties** that are defined as attribute specifications of a type corresponding to a class as an object that is defined in a class section of the given class specification or defined in instance section or in an instance-instance section of a metaclass to which the class belongs as its direct or transitive instance.

## Example

```
class_specification:

{companies; in: class;
   instance_section: {
      name: string;
      metaslot
        a: invariant, {{unique;}};
        b: invariant, {{constant;}};
      end
      engaged_in: {set; type_of_element: projects.inst;};
}},

{employees;  in: class;
    instance_section: {
        name: string;  metaslot   c: invariant, {{constant;}}  end
        belongs_to: companies.inst;
       works_on: {set; type_of_element: projects.inst;};
       on_emp_proj: { in: predicate, invariant;
         { predicative:
            {all e (employees(e) -> e.works_on <= e.belongs_to.engaged_in)} }
       }
}}
```

## Metaclasses

A classification "dimension" that **is orthogonal to a generalization / specialization dimension** is formed with the following levels:

- **the zero level includes all objects and all frames** that are not classes and do not belong to any specific class;
- **the first level includes all classes** having zero level objects as their instances;
- **the second level includes all metaclasses** having classes of the first level as their instances;
- **the third level includes all metaclasses having metaclasses** of the second level as their instances, and so on.

**The mixed level is also allowed having objects, classes and metaclasses of any levels as its instances**. The set of built-in classes of this level includes:

- **the class frame** that is the universal class to which all frames and objects belong as its instances;
- **the metaclass class** containing all classes;
- **the metaclass metaclass** containing all metaclasses.

# Association metaclasses

A specific feature of metaclasses (comparing to classes) is **an ability to declare attributes specified in a metaclass as association metaclasses**. Thus a metaclass can introduce attribute categories that can be used in classes being instances of this metaclass and in their instances. **To declare an attribute specified in a metaclass as an association metaclass** it is necessary **to indicate *in: association* in its metaslot.**

In classes being instances of a given metaclass in metaslots of the respective attributes **to indicate belonging of an attribute to an attribute category specified in the metaclass** the construct in **:< attribute category; in: metaclass attribute identifer >** is to be used.

Generally metaclasses provide for introducing of **generic concepts and of common attributes (or of their categories) for similar classes, for introducing of common consistency constraints and deductive rules for such classes and their attributes**. Metaclasses provide for **proper grouping** of an application domain information and for **proper differentiation** of various application domains.

## Formulae facilities of the language

## Objectives of the formulae language

Logic formulae (or simply formulae) in the SYNTHESIS language are used

- **to define rules in logic programs**,
- **to formulate assertions** (e.g., consistency constraints of a mediator or an information resource),
- **to express programs** (queries) over a mediator,
- **to form predicative specifications.**

To specify formulae a variant of a **typed (multisorted) first order predicate logic language** is used. Every predicate, function, constant and variable in formulae is typed.

**Predicates in formulae correspond to classes, worlds, collections and functions** that are specified in resource component specification modules and have respective types.

**Rules are represented as closed formulae of the form: *a :- w*** where a is an atom and w is a formula. **a** is a head and **w** is a body of a rule.

## General Rules of Formulae Construction

|  |  |
|---|---|
|  | **Atomic Formulae** |
| *Predicate-collection* | creator (c/Creator[name, culture]) |
| *Condition* | c.date_of_origin >= 1550 |
|  | e.value() < 20000 |
| *Built-in predicate* | in(c, creator) |
|  | **Compound Formulae** |
| *Conjunction* | $w_1$ & $w_2$ |
| *Disjunction* | $w_1$ \| $w_2$ |
| *Implication* | $w_1$ -> $w_2$ |
| *Negation* | ^w |
| *Existential Quantifier* | exists x/T (w) |
| *Universal Quantifier* | all x/T (w) |
| *Group by* | group_by ({$a_1$, $a_2$}, {w}) |
| *Order by* | order_by ({asc $a_1$, desc $a_2$}, {w}) |

# Formulae without predicate-collections

```
{ Secretary; in: type;
available_experts: {set; type_of_element: Expert;};
search_for_experts: { in: function;
  params: {-number_of_experts/integer};
    { predicative:
        {ex exps/({set; type_of_element: Expert;})(
               exps = { exp/Expert | expert(exp) &
                              exp.research_area = 'computer science'} &
              this.available_experts' = exps &
              number_of_experts = cardinal(exps)) } };
};
dispatch: { in: function; params: { +revi/Review };
{ predicative:
   { ^isempty(this.available_experts) &
       ex exp/Expert (in(exp, this.available_experts) &
               exp.area_of_expertise = revi.for_proposal.area &
               revi.by_expert' = exp) } };
};
}
```

## Formulae with predicate-collections

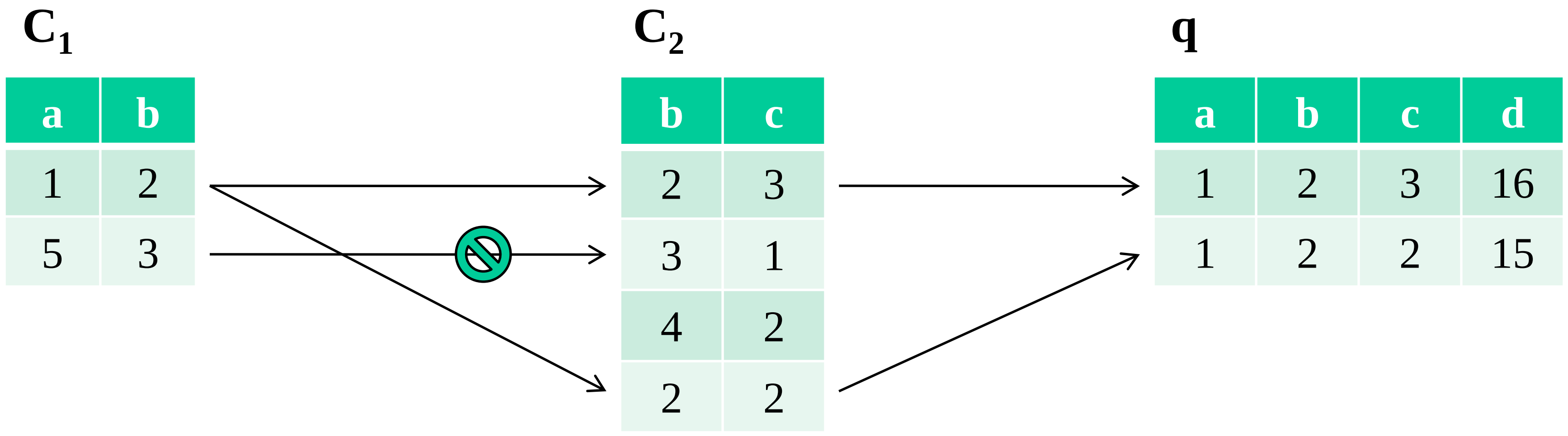
```
canvas(p/Canvas[title, name, culture, place_of_origin, r_name]) :-
   painting (p/Painting[title, name: created_by.name, place_of_origin,
                     date_of_origin, r_name: in_collection.in_repository.name])  &
    creator (c/Creator[name, culture])  &
    r_name = 'Uffizi‘   &
   date_of_origin >= 1550  & date_of_origin < 1700,

entities(e/HeritageEntity) :-
   canvas(p/Canvas)  |
   antiquities(a/Antiquities)
```

# Rule Semantics

q(x/T) :- $C_1$(y/$T_1$[a, b]) & $C_2$(z/$T_2$[b, c]) & F(a+b+c, d) & a <= b+c

```
{ F; in: fucntion;
   params: {+m/real, -n/real};
   { predicative: {  n = m + 10 } };
}
```

T = $T_1$[a, b]  | $T_2$[b, c] = { in: type; a: real; b: real; c: real; d: real; }



**$C_1$**

| a | b |
|---|---|
| 1 | 2 |
| 5 | 3 |

**$C_2$**

| b | c |
|---|---|
| 2 | 3 |
| 3 | 1 |
| 4 | 2 |
| 2 | 2 |

**q**

| a | b | c | d |
|---|---|---|---|
| 1 | 2 | 3 | 16 |
| 1 | 2 | 2 | 15 |

# Flattening

**Type T reduct** has the following syntactic form: ***T*[*e*1, … , *en*]**
***a/V* : *p* is a general form of reduct element** for attribute *a* typed by type V reduced to **the type of a last element in a path *p* = *c1*.*c2*. … .*ck***

The path in a reduct element **can contain an attribute of a set type**. **A *flattening*** takes place **when an attribute in a reduct element path is declared as a set type** *{set*; *type of element: E;*} and is typed in the path with type *E*.

```
repository(r/Repository[name, author/Creator[name, generalInfo]) :-
museum(m/Repository[name,
   author/Creator[name, generalInfo]:
      collections/Collection.contains/Painting.created_by
) & m.name = 'Uffizi'
```

## Programs

In the simplest case a program is represented as a list of rules separated by periods. A function is one of the program control facilities.

Explicit control facilities:

< program >::=< rule list >|< block >|< program sequence >
< block >::= **begin** < program > **end**
< program sequence >::=< program >; < program sequence >
< conditional rule body >::= **if** < condition > **then** < block >
< iterative rule body >::= **while** < condition > **do** < block >

```
iter : - while
            ex emp/Employee(in(emp, dsal) & emp.dep = ‘db’ &
            emp.name = ‘Ivanov’ & emp.sal < 10000 )
            do raise(‘db’).
```

# SYNTHESIS Formulae Language Subset (Syfs)

- Logic program is a *rule list*
- Syfs rule:
- $q(x/R)$ :- $p_1$ & … & $p_n$ & $f_1$ & … & $f_m$ & $C$
  - $p_i$ – class-predicates
  - $f_i$ – function predicates
  - $C$ – constraint
- Class-predicate looks as follows: $P(v/T\text{: } a_1.a_2. \ldots a_k)$
- Constraint can contain predicates <, <=, =, >, >=
- Arithmetic expressions can contain variables of types *integer*, *real*, *Boolean*, *string*, *time*