Hybrid ontology-oriented programming for semi-structured data processing

Oleksii Voropai

voropay.o@gmail.com

Abstract. In this work a design of a hybrid ontology-oriented programming language is introduced. It combines declarative style for building a model of domain and imperative style for processing it. The model of domain takes the form of an ontology and consists of a set of interrelated concepts. The syntax of the concept definition allows to create a new concept based on existing ones by linking them with logic relations. It is also possible to create concepts using inheritance, nested concepts, aggregation, higher-order logic and instances generation function.
The imperative style allows to prepare initial data for the model in an object-oriented or functional style, dynamically create its elements, implement some of its elements in the form of a deterministic sequence of calculations, process the results of queries to the model.
The declarative and imperative components are tightly integrated. Concepts combine logical, object-oriented and functional semantics. Definitions of concepts are first-class citizens of the component of computing. They can include variables, operators and functions. This makes it possible to combine the styles in one application.
The key principles of the hybrid programming language implementation are considered. They include the scope of concepts, structures for storing local values of stateful language elements, inference algorithms.
The proposed hybrid programming language simplifies the creation and work with a unified model of domain based on semi-structured data or data from diverse sources. It is suitable for tasks of information extraction, automated testing, process automation, business process modeling, creation of applications based on ontologies.

Keywords: Multi-Paradigm Programming, Logic Programming, Ontology, Semi-Structured Data, Information Extraction.

1 Introduction

Semi-structured data plays important role in software. It's data which structure does not correspond to the formal structure of data tables in relational databases, but nonetheless contains tags or other markers to specify semantic elements and their relations within the data [1].

The sources of such data are very diverse. This can be data obtained from external sources such as the World Wide Web which has no unified data forms. It is often much easier to integrate with an outside service to receive data instead of putting those data into the application and keeping it up to date. Also, the software solutions can be non-monolithic and include parts developed independently and using different technologies, for example, in the case of a service-oriented architecture. Or if the application is only part of the vast information infrastructure. In all these cases, applications must work with data whose structure is defined in external source and cannot be controlled. In other cases, the relational data model is intentionally discarded, for example, if the data structure is fuzzy or changes frequently. Also, data may have some basic structure, which however does not describe the semantic elements within it. It may be the results of preprocessing of unstructured data such as texts, videos and images, event logs, sensor readings, content of WEB pages, etc. As a result, the application has to work with data that comes from diverse sources and doesn't have unified structure, format and terminology. Such data is usually designed for a wide range of consumers, and their structure describes only the minimum necessary set of entities.

But this data contains much more useful information then described by its structure. For example, structure of a HTML document represents only a tree of its element, but we can group them accordingly their location, style, captions and recognize their roles. The result of text parsing is a set of syntax trees, but we can connect them together and give them meaning. Using financial statements one can conduct a financial analysis of profitability, stability, liquidity and many other indicators. This can be done by building a domain model that connects the basic structures into a network of meaningful concepts. It describes segmented data within a unified system of concepts and introduces new concepts needed for solving specific problems. This model can consist of several abstract layers, include rules and mathematical models, complex abstract data types (graphs, trees) that describe the relations between entities.

Such a model can be either high-level conceptual model which describes the basic concepts in a natural language. Or more specific one in one of the formal languages corresponding to the selected implementation approach. In any case the way how it describes the domain should be human friendly. One of the its main requirements is to be clear for everyone who takes part in development process including non-technical stakeholders. On the other hand, the model of domain should be translated to programming code during the implementation phase. Moreover, in the process of the application development, both the model itself and its software implementation can repeatedly change. Unfortunately, the methods for describing the model and its implementation are weakly consistent. This complicates the process of creating program code of the model, testing and adding modifications. Let's overview main approaches to the software implementation of conceptual models.

If the size of the model is small, then it is usually implemented on general purpose programming language. In functional programming languages, the conceptual model is implemented by a set of data structures and functions. As a result, the model components, especially relations between entities, are not introduced explicitly but hidden inside a set of functions and distributed over the program code. This creates a big gap between the declarative description of the model and its software implementation.

Structuring the program in object-oriented style mitigates this problem. Each domain entity is represented by an object whose data fields correspond to entity attributes. And relations between entities are implemented in the form of relations between objects, partly based on the principles of OOP (inheritance, data abstraction and polymorphism), partly using design patterns. But in many cases relations must be implemented in object's methods. In addition to creating classes representing entities programmer should take care about data structures to organize them, algorithms to populate these structures and search for information in them. Such a solutions are cumbersome and includes a large amount of boilerplate code. The implementation of the model is mixed with the auxiliary logic of its storing, search, processing, format conversions, etc.

The implementation of the model in knowledge representation languages [2] is much closer to its description. The examples of such languages are Prolog, Datalog, OWL, Flora and many others. They are usually based on first-order logic or its fragments, e.g., descriptive logic [3]. These languages allow to declaratively describe the specification of the solution to the problem, the structure of the simulated object or phenomenon and the expected result. And the built-in inference mechanisms can automatically find a solution that satisfies the given conditions. The program code of the model is extremely concise and clear. Unfortunately, most of these languages are subject-oriented and are not intended to solve a wide range of problems. Although Prolog [4] is a Turing-complete language, it is not used as a general-purpose language in practice. Its weak point is creating sequences of actions. Their declarative implementation, even in simple cases, can look very unnatural and requires considerable effort and qualification.

In some cases, the task can be solved by a combination of a basic general-purpose programming language and an external knowledge representation system. Unfortunately, this approach complicates the solution, the same entities must be implemented in both languages. It requires configuration of the interaction between them through an API and additional support of the knowledge representation system, etc. Therefore, it makes sense only for models of large size and complexity. But for most tasks of semi-structured data processing this approach is redundant. Also, models over semi-structured data are closely intertwined with the logic of their processing and distributed across different components of the application. Therefore, transferring them to an external system is not always possible.

Ontology-oriented programming [5] brings the external knowledge representation system closer to object-oriented languages. This approach is based on the fact of the similarity between the elements of ontology and the object programming model. Elements of ontology such as classes, entities, and attributes can be automatically mapped to classes, objects, and fields of the object programming model. This mapping can be performed statically at the stage of compilation of the program [6], dynamically during its execution [7], or created manually in one of the specialized programming languages [8]. After that the obtained classes can be used to get access to the elements of the ontology and perform reasoning procedures. Unfortunately, the basic implementation of this idea will have rather limited capabilities. Ontology languages are quite expressive and far from all ontology components can be transformed into classes in a straightforward and natural way. Also, it is not enough just to create a set of classes and objects to implement a full-fledged inference. It needs information on ontology elements in an explicit form, for example, in the form of meta-classes.

An alternative approach to software development is model-driven development [9]. According to it, the main task of development is the creation of domain models, and then program code should be automatically generated on their basis. But in practice, such a radical solution does not always have enough flexibility, especially in matters of performance. A developer of such models has to combine the roles of both a programmer and a business analyst. Therefore, this approach has not yet been able to replace traditional approaches to the implementation of the model in general-purpose programming languages.

A combination of an object-oriented (or functional) and declarative paradigms in a hybrid programming language could be a promising solution. The declarative part of the language (let's name it “component of modeling”) would be responsible for the description of the model and object-oriented or functional one (let's name it “component of computing”) for processing it. It would allow to combine model description and processing in one code base, each task could be solved by appropriate tools. Moreover, a close integration of these paradigms could benefit both. Declarative and computational constructs of the language can include and interact with each other, which would make the language more expressive. This would make the program code more understandable and natural for humans, bring it closer to the conceptual model of domain, and simplify the development and support of software.

The aim of this work is to create a hybrid programming language which will combine object-oriented and/or functional styles with declarative one and will be convenient for conceptual modeling over semi-structured and heterogeneous data. The language should include tools for describing the domain model in the form of a set of concepts and the relations between them as well as for its processing. It should be suitable for creating executable ontologies, the elements of the ontology should be first class citizens of the language. Therefore, it can be attributed to ontology-oriented programming languages [5]. Such features would be interesting both in general-purpose programming languages and in specialized built-in languages for managing data or automating repetitive tasks.

In Section 2 we overview existing hybrid programming languages and the principles of combining various paradigms within a single computational model. The main features that a hybrid programming language oriented to conceptual modeling over semi-structured data should possess are in Section 3. Definitions of main elements of the component of modeling are in Section 4. Main elements of the component of computing are in Section 5. Components interaction issues are discussed in Section 6. Some problems of hybrid language implementation are in Section 7. And Section 8 is devoted to examples of the its practical application.

2 Overview of existing multi-paradigm programming languages

One of the main directions of programming languages development is to increase their expressiveness and clarity of program code. Principles of programming should be natural for human thinking, and the program should be close to the description of a solution to a problem in natural language. Most modern languages support several programming paradigms, each of which offers its own way of describing the solution that convenient in its field of application. This makes it possible to choose appropriate programming style and language constructs depending on a specific of a task. For semi-structured data processing both declarative means of representing knowledge and means of organizing calculations are important. So, let's overview in more detail approaches to combining these paradigms.

2.1 Integration of functional and logic paradigms

Let's start from integration of functional and logic paradigms [10][11]. Both belong to declarative style of programming. The functional style describes the mathematical relationship between data and a goal of computing. The logic one describes logical relationship between facts and rules of the domain. The functions pureness feature simplifies the integration of these paradigms, especially the task of using functions in backtracking search algorithm. It explores alternative branches of a search tree and impure computations in one branch may potentially affect others. Significant results have been achieved in the integration of these paradigms, starting with a simple combination of two paradigms in one computing environment and ending with hybrid programming languages that combine the semantics of both.

The most obvious integration option is to support both paradigms within the same programming platform. Examples of this approach are KnowledgeWorks package for CommonLisp or Racklog for Scheme. They are Lisp-based functional programming languages that include the implementation of the Prolog logic programming system. Due to this, programs in those languages can include both functional and logic expressions. Moreover, logic predicates may include functional expressions. If logic variables are part of functional expressions, then variables must be binded to values before evaluating the expression. Logic predicates are also available within functional expressions. Predicate values can be calculated by calling special functions that allow finding their solutions using built-in inference mechanisms.

The functions allow to specify the order of calculations in an explicit form and simplify the tasks that do not require logic search mechanisms or replace a resolution with a more efficient algorithm suitable for a specific task. It can also greatly simplify the implementation of those tasks that inherently represent a sequence of actions and get rid of controversial control operators such as "`cut`" in Prolog.

A more complex integration option is hybrid functional-logic programming languages that unify functional and logic semantics [11]. When functions in pure functional programming have only one direction of calculation – from input arguments to an output value, the logic approach allows to make it bidirectional. Input and output arguments are equal elements of the relation of equality, the value of the output argument can be used to derive conclusions about the values of the input and vice versa. Because of this, function definitions become more flexible and expressive. Functions obtain inversion property. Inference mechanism make it possible to find all possible values of the input arguments corresponding to the given output value.

As an example, let’s examine the function of combining lists:

append([], L) = L.

append([E | R], L) = [E | append(R, L)].

If it's a part of an equation:

?- append(L,[2,3]) = [1,2,3], (2.1.1)

then obviously, its solution is L=1.

Unfortunately, the interaction between the functional and logic paradigms upon closer examination is not so obvious and is implemented differently in different languages. The core of Ciao-Prolog [12] and Mercury [13] has logic semantics. Functional expressions are translated into logic ones, the solutions of which are found by standard inference mechanism. The opposite approach is to add a logic component to the main functional language, such as in the languages Escher [14], Curry [15] and Toy [16]. More complex cases are also possible. For example, in Oz language [17], the logic and functional paradigms complement the main computation model of distributed and parallel programming.

There are two main approaches to the integration of logic and functional programming: the **residuation** technique emphasizes the determinism of function computation, **narrowing** allows non-deterministic search of its arguments. Some languages such as Oz give priority to the predictability of function computation by limiting the use of non-deterministic logic search. By default, the function calculation process is deterministic, it will only start when all input arguments are binded to values. This technique is called residuation. It is also supported by languages such as HAL [18], Escher [14], Le Fun [19], Life [20], NUE-Prolog [21], Mercury [13]. They also support non-deterministic logic programming when calculating functions. To do this, one must encode the function in a special way and make an explicit call to one of the logic search methods by passing it the function as an input argument.

An option for closer integration of functional and logic paradigms is proposed in languages such as Curry [15] and TOY [16]. Unlike languages using the residuation principle, these languages try to find the values of unknown logic variables used in the definition of the function. This principle is called narrowing. He tries to replace the expressions of the original equation with equal expressions from the function definition in order to harmonize the right and left sides of the equation and unify the logic variables with the values. For the equation (2.1.1), it will look approximately as follows:

append(L,[2, 3]) → [E | append(R, [2, 3])] → [E | append([], [2, 3])] → [E, 2, 3].

The result is the unification of the variable E with a value of 1.

The syntax of these languages is more flexible than that of languages based on residuation. The payback for this is a significant complication of operational semantics. The search for the solution may include enumerating various options of substituting rules from the function definition; various strategies for this enumeration are possible. The process of function calculation becomes non-deterministic, which is not always acceptable in practice. The programmer has to consider these features both when creating functions and when using them, which can complicate the process of software development and puts higher demands on the qualification of developers.

2.2 Integration of imperative and logic paradigms

The integration of the logic and imperative paradigms has received less attention than the functional one. Imperative style involves the use of operators that change state of the program. The state is usually represented by variables and fields of objects that store their values in memory. This is poorly consistent with the backtracking algorithm, which requires returning to the original partial solution if an advanced solution could not be found. Accordingly, it is necessary to restore the initial state of all variables and objects that determine the result of calculations during solution search. This significantly complicates integration of these paradigms.

The most important results in this area were achieved in a project of the OZ language [17]. OZ is a multi-paradigm programming language that includes concepts of most of the popular programming paradigms, including logic, functional, object-oriented, constraint programming, distributed and parallel programming. OZ is positioned as a learning language, Mozart [22] is its main implementation. The main feature of the language is orientation to parallelism.

Imperative programming style is supported by a concept called **Cell** [23]. It includes a container for storing the state, as well as a set of atomic operations for reading and changing it. These atomic operations are needed to support concurrency. Cells are the foundation on which the object-oriented extension of the OZ is built.

OZ also supports logic and constraint programming. The problem of harmonization of backtracking algorithm with the parallel nature of the calculations and the presence of objects with state is solved by a structure called **computation spaces** [24]. Computation space is a repository of constraints, procedures, and cells, as well as a computational thread associated with it. When it is necessary to explore different solutions, computational thread generates child computation spaces. And the whole process of computing can be represented as a process of traversing a tree-like structure consisting of the local computation spaces. If the current branch of the search requires a change in the state of the program, then it happens in the current computation space, not in parent one. If this branch fails, the parent computation space can be used to restore the initial state of the computing system, generate a new branch and continue the search.

The integration of the imperative and logical paradigms is also addressed in IDP project [25]. The authors classify IDP system as a knowledge base system, which consists of 3 parts: a knowledge representation language, a set of inference methods, and integration procedures. The name IDP stands for Imperative-Declarative Programming.

The knowledge representation language is intended for constructing models in a declarative form. It is based on first-order logic and supports modularity, which makes it easy to structure and reuse the code. The project supports an extensive set of powerful and generic inference methods for efficiently solving a wide class of tasks. Among them are query inference, model and satisfiability checking, propagation inference, deduction, symmetry detection and model extraction. A detailed information about these methods can be found in the IDP user manual [26].

Procedural interface is designed to describe main workflow of a program in imperative style: to organize data input, prepare logic objects, call one of the inference methods to solve the main task and output results. Typically, solving these types of tasks in a declarative logic style is not very convenient, so the imperative layer on top of the logic knowledge representation language looks very useful. As an integration procedural language, IDP supports C++ and Lua. Unfortunately, the unification of the imperative and declarative paradigms is limited only to this layer; procedures are not available at the level of knowledge representation language.

The mutual influence of programming paradigms is not limited only to the creation of hybrid languages. For example, F-Logic [27], which is the language of knowledge representation and conceptual modeling, adopted a form of knowledge representation from object-oriented languages. F-Logic was originally designed for logical reasoning over object-oriented databases but is also applied in semantic technologies [28].

F-Logic represents domain entities as objects, each object is described by a set of attributes and relations with other entities. It supports complex compound objects, inheritance, polymorphism, encapsulation, type system, object identities and methods. The language is based on first-order logic, its inference procedure allows to perform queries on an object-oriented database.

Object-oriented form of concepts helps to gather all the information about the simulated entity in one place and operate on it as a single whole. This makes the process of modeling easier and more natural in comparison with other forms of knowledge representation, e.g. the relational one. According to the relational model information about an object is split into several relations called tables that describe individual properties of objects. This solves many problems with data integrity, but on the other hand complicates the work with them.

2.3 Integration of imperative paradigm and data management languages

The most popular tools for data management are domain-specific languages such as **SQL**. Integration of the imperative paradigm and data management languages is the mature field of study with many practical solutions. The most famous one is **PL/SQL**, a procedural extension of the SQL language. This language is designed to process data in a relational database, using both imperative (variables, control statements, functions, objects) and declarative programming styles (SQL expressions). The SQL query code can include functions, the results of query execution can be associated with variables and cursors and processed in imperative style, the query code itself can be generated dynamically by a sequence of imperative commands. The combination of procedures and queries with the help of some tricks allows to implement recursive queries.

Language Integrated Query (LINQ) is a popular component of .NET platform, that allows to include data query expressions into a program code in an object-oriented language in a natural way. The level of integration of imperative and declarative styles in LINQ is comparable to PL/SQL. In addition, LINQ queries can be used to retrieve data not only from relational databases, but also from arrays, enumerated classes, and XML documents. LINQ is convenient not only for receiving data but also for transforming it. For example, it's possible to combine several input sequences into one, which type would be a structured hierarchical object, change the format of the sequence, filter and sort it. The architecture of this component is quite flexible, it allows to create custom data providers and ways to fulfill the query.

GraphQL is a framework for building application programming interfaces (APIs), which includes a query and data manipulation languages as well as a runtime for these queries. The language consists of 3 main components: data types, queries and data receiving functions called resolvers.

Data types are descriptions of objects fields. GraphQL supports scalar types, lists, enumerations, and references to nested types. Since type fields can contain references to other types, the entire data scheme can be represented in the form of a graph. Query is a description of the data structure requested from the API. It includes a list of required objects, their fields, and input attributes that specify the required values of the fields. Each type and each field must be associated with a resolver function. A type resolver describes an algorithm for obtaining its objects, a field resolver – the values of an object field. The resolver is a function in one of the functional or object-oriented languages.

GraphQL combines a declarative description of the data schema with imperative or functional algorithms for obtaining them. The data schema is described explicitly and plays a key role in the solution. The best practice is to create a data scheme that does not repeat the data source scheme but matches the domain model. This makes GraphQL a popular solution for integrating diverse data sources.

Thus, the GraphQL language allows expressing the domain model in a rather clear way, isolating it from the rest of the code, and bringing the model and its implementation closer together. Unfortunately, the declarative component of the language is limited only to the description of the composition of data types; all other relations between model elements must be implemented in the resolver layer. The scope of GraphQL is limited to the construction of the API layer; accordingly, the data scheme will describe only that part of the domain model that belongs to this layer.

2.4 Conclusions

The idea of combining different programming paradigms in one language is obvious and attracts a lot of attention. The integration of functional and logic paradigms is a good example showing that it's possible to not only implement both paradigms within the same runtime but also combine the semantics of a function and a statement into one whole. Several experimental and educational programming languages have been created in this area which, however, have not yet gained popularity in practice. The area of integration of logic and object-oriented paradigms has attracted less attention; but the OZ language shows the practical possibility of this. The F-Logic language proves the fact that the object-oriented form of representing knowledge is convenient even for logic reasoning.

The integration of object-oriented paradigm and DSL data management languages is the de facto standard. Most modern relational databases support procedural extensions to SQL language. LINQ expands the capabilities of object-oriented languages, giving them a tool for querying data which has a relational model. LINQ is a very flexible tool that that can be used for creating custom DSL languages. But LINQ is primarily a query language; using it for knowledge representation is not entirely natural. The popularity of the GraphQL framework shows that tools for describing a data model in a declarative way can and should be the part of modern programming languages. They have wide area of application and help to make the code more concise and clearer, accelerate software development.

3 Main features of hybrid ontology-oriented programming language

The main goal of this work is to create a programming language that would be convenient both for organizing calculations and for describing a model over semi-structured data. The language must satisfy the following criteria:

- Language should be convenient for practical use; the basic concepts of language should be natural for human thinking. It should also be easy to learn. Its basic concepts should have a familiar form, consistent with established practices and approaches.
- The language must support descriptions of sequences of computations.
- The language should be convenient for describing a model based on heterogeneous data collected from diverse sources and having a different nature and structure (relational, semi-structured, document-oriented, in the form of graphs, trees, lists, etc.).
- The data modeling language should be flexible and expressive enough to represent knowledge of domain in a form suitable for both programming and human understanding.
- The model should be convenient for both describing concepts that are important for solving specific narrow tasks and for building complete and holistic ontology of the domain.
- The language should be convenient for describing queries to the model, its implementation should include mechanisms of their performing.
- The components of modeling and computing should be integrated as closely as possible. The form of description of the model must be compatible with the form of the description of the calculations.

So, let's try to highlight the main features of the future language that meet the specified criteria.

Obviously, the **component of computing** should be an implementation of object-oriented and / or functional programming language. Integration of functional style is an easier task. But object-oriented style is also important due its popularity in practice. That's why not only the creation of a new language matters, but also the search for the possibilities of adding tools for modeling over semi-structured data in a declarative way into the most popular programming languages, primarily object-oriented.

There are no obvious and ready-made solutions for the **component of modeling**. Let's try to form its shape on our own. It is convenient to present the model in the form of an ontology that describes the domain using a set of abstract concepts, specific instances, attributes and the relations between them. Instances correspond to source data objects. Abstract concepts describe the structure of the domain and are derived from instances and other abstract concepts. They give meaning to the source data within a specific domain. Attributes define the structure of concepts and instances. Relations describe how concepts and instances relate to each other.

Dependencies between concepts can have a complex structure, including recursively defined ones, for example, graph or tree-like. The component of modeling should be sufficiently flexible and expressive to describe such relationships. In addition, it should have clear and simple operational semantics so that one can specify an effective way to derive one concepts from others. A suitable basis for such a modeling language is first-order logic. Many relations between concepts can be described in the form of logic conditions connecting the values of their attributes. And SLD resolution is a simple and understandable rule of inference.

So, the model over semi-structured data should be represented by a set of concepts interconnected by relations that allow to obtain one concepts from others. Often in practice this model has a utilitarian goal, it's just enough to extract a certain abstract level of information from the source data instead of building complex and exhaustive ontology. Therefore, it was decided to make the component of modeling convenient primarily for describing the transformation of one concepts into others. For this, the definition of the concept should unite a description of a structure of the concept and a list of its relations with other concepts, enough to derive its values from the source data. Concept "*C*" will have the form:

$$C = \{A, Cc, Re\},$$

where "*A*" is the set of attributes, "*Cc*" is the set of parent concepts, "*Re*" is the set of relations between the parent and child concepts. Relations are expressions connected by Boolean operators. Thus, the model can be created by "layers". At the lower level are the instances of the source data. The next layers contain concepts that unite, clarify or generalize the concepts from previous layers. And mostly the concepts from upper layers are the targets of a search. The combination of the structure of the concept and the method of obtaining it in one definition makes it possible to simplify the understanding of the connections between these layers of concepts. The structure of the program becomes more linear, since it describes the transformation of concepts from the input layer to the most abstract one. The structure of concepts and their relations are located side by side in the program, which improves its readability.

Combining concepts and their relationships in one definition is not always possible. In some cases, all the concepts connected by a relation may be the goal of the search, and it is impossible to place the relation in only one of the definitions of concepts. In other cases, the creation of an "universal" relations that can be applied to concepts of a different nature may be useful. For example, spatial relationships can be applied to all concepts that include spatial coordinates. Therefore, the language must also support the definition of relation. Such definition of relation will have the following form:

$$R = \{Cc, Re\},$$

where “Cc” is the set of concepts connected by the set of relations “Re”. It corresponds to a predicate of logic programming. Therefore, such relations can be used in the definitions of concepts or be the target of queries to the model.

The traditional orientation of logic programming languages to rules is not very convenient for data processing. More suitable is approach when the data structure takes central stage, such as the object-oriented approach used in F-Logic language [27]. It is more consistent with the component of computing and provides more opportunities for the integration of the paradigms. In addition, it allows to naturally expand or refine the definitions of child concepts through inheritance. Unfortunately, the syntax of the F-Logic language at first glance looks rather unusual and may take some time to learn, so it’s better to choose a more traditional and familiar SQL-like style. Thus, the definition of a concept will resemble SQL query and consist of the concept name, lists of its attributes, parent concepts and relations between the concepts.

The minimum level of **components integration** should be like that of the PL/SQL language. Expressions of relations between concepts can include functions declared in the component of computing. And with the help of the latter one it should be possible to prepare instances of the source data, dynamically assemble the definitions of concepts and execute queries to obtain their specific values. But the proximity of the meaning and form of the classes of the object-oriented model and the concepts of ontology allows deeper integration. So, concepts correspond to classes with a predefined constructor that creates their instances using inference. From the other side classes correspond to concepts with an arbitrarily defined method of instantiating their attributes. This will make it possible to enrich the model with elements with arbitrary structure and inference algorithms implemented in general-purpose functional or object-oriented programming languages. Relations have analogies in a functional world. They can be represented as functions whose input arguments are objects, the result is a Boolean value.

It is also necessary to determine a **type system** of the hybrid language. It was decided to focus on weak dynamic typing. Many data formats, such as JSON, do not contain data type information. Also, the data types system of the data source may not coincide with those of an application. Weak dynamic typing will make it easier to connect data types of the input sources and the application. And implicit type binding will make concepts more flexible. In addition, dynamic typing is easier to implement. Static typing and parametric polymorphism also have important advantages and may be the goal of future research.

Let's try bringing the syntax style of the future language closer to JavaScript, primarily because of its popularity, simplicity, and ease of learning. It is also a widespread built-in scripting and data access language on many platforms including web browsers, SAP-HANA, MongoDB, CouchDB, Clusterpoint, Google Spreadsheets and Sites, IBM xPages, etc. These platforms would also benefit from executable ontologies language.

4 Description of the component of modeling

Let's start the description of the experimental hybrid language from the component of modeling. According to the requirements described in the previous section, the main elements of the component are data instances (facts), abstract concepts and relations between them. Facts and concepts have a name and are described by a set of attributes. Attributes of facts are always associated with specific values, the values of the attributes of concepts are determined as a result of logic inference (or other method of generating them). Concept is based on other concepts or facts; let's call the initial concepts parents, and derivatives – child. To do this, let's add relations to the definition of the concept. Their role is to bind values of the attributes of different concepts and make inference of the attributes of child concept possible. We also need a mechanism that allows concepts to inherit the attributes and relations of other concepts, expanding or narrowing them. As a separate type of concept, we introduce an abstract relation that will allow to specify only dependencies between parent concepts without introducing the attributes of the child concept. Since the component of modeling is based on first-order logic, it is worth determining the meaning of such concepts as logic variables, negation, and elements of higher-order logic. It is also useful to borrow nested queries, outer joins, and aggregation from the world of relational algebra and SQL. Let's consider all these elements of the component of modeling in details.

4.1 Facts

Fact is a description of specific knowledge of domain in the form of a named set of key-value pairs:

```
<fact definition> ::= "fact" <fact name> "{"
  <attribute name> ":" < attribute value>
  {"," <attribute name> ":" < attribute value>}
"}"
```

For example:

```
fact product {
  name: "Cabernet Sauvignon",
  type: "red wine",
  country: "Chile"
}
```

The name of the fact may not be unique. For example, there may be many products with a different name, type and country of origin which can be described as fact "product".

The meaning of the facts of the component of modeling is consistent with the facts of Prolog language. Except that in Prolog, the arguments of the terms are identified by their position, and the attributes of the facts of the component of modeling are identified by name.

Facts are considered identical if their names coincide, as well as the names and values of their attributes.

4.2 Concepts

Concept is a data structure that describes an abstract entity and is based on other concepts and facts. The definition of concept includes a name, lists of attributes, and child concepts. As well as a logic expression describing the dependencies between its (child concept's) attributes and attributes of parent concepts and allowing to derive the values of the attributes of the child's concept:

```
<concept definition> ::=
"concept" < concept name> [< concept alias>] "("
  <attribute name> ["=" <expression>]
  {"," <attribute name> ["=" <expression>]}
")"
"from"
  <parent concept name> [<parent concept alias>] ["("
    <attribute name> "=" <expression>
    {"," <attribute name> "=" <expression>}
  ")"]
  {"," <parent concept name> [<parent concept alias>]
  ["("
    <attribute name> "=" <expression>
    {"," <attribute name> "=" <expression>}
  ")"]}
["where" <expression of relations>]
```

The definition of the concept is like a SQL query, but instead of the name of the tables are the names of the parent concepts, and instead of the returned columns are the attributes of the child concept. In addition, the concept has a name by which it can be accessed in the definitions of other concepts or in queries to the model. The parent concept can be a concept itself or a fact. Expression of relations in the "`where`" section is a Boolean expression that can include logic operators, equality conditions, arithmetic operators, function calls, etc. Their arguments can be variables, constants and links to attributes of both parent and child concepts. Link to attribute has following format:

```
<attribute expression> ::=
<concept alias>"."<attribute name>
```

Expression of relations has a conjunctive form and should include equality conditions for all attributes of the child concept, enough to derive their values. Also, it may include conditions that limit the values of parent concepts or link them to each other. If some of the parent concepts are not binded to each other in the "`where`" section, then the inference mechanism returns all possible combinations of their values as a result.

For convenience, some part of the equality conditions of the attributes can be in the attribute section of the child and parent concepts. Such syntactic sugar makes the dependencies between attributes more explicit and distinguish them from other conditions.

Here is an example of the definition of "`profit`" concept based on the concepts of "`revenue`" and "`cost`":

```
concept profit p (
  value = r.value - c.value,
  date
) from revenue r, cost c
where p.date = r.date = c.date
```

Concepts correspond to rules in Prolog but have slightly different semantics. Prolog focuses on the construction of logically related statements and queries to them. Concepts are primarily intended for structuring the source data and extracting information from it. Concept attributes correspond to Prolog variables.

Since the list of parent concepts and conditions of relations are divided into separate sections, inference algorithm has its own characteristics. Let's briefly describe it there, more detailed information can be found in Section 7. Inference of the parent concepts are executed in the order in which they are specified in the "`from`" section. The search for a solution for the next concept is performed for each partial solution of the previous concepts in the same way as in SLD resolution. For each partial solution, a validation of the expression of relations from the "`where`" section is performed. Since this expression takes conjunctional form, each subexpression is checked separately. If the subexpression is false, then this partial solution is rejected and the search proceeds to the next one. If some of the arguments of the subexpression are not yet defined (not binded to the values), then its check is postponed. If the subexpression is the equality operator and only one of its arguments is defined, then the inference system will calculate its value and try to associate it with the free argument. This is possible if this free argument is concept attribute or logic variable. For example, if in the subexpression "`concept1.attr = concept2.attr`" the only one defined argument is "`concept1.attr`", then the inference mechanism will calculate its value and bind it to "`concept2.attr`". When the inference process will reach the "`concept2`" concept, the value of its "`attr`" attribute will be already known and will be treated as the input argument for this branch of the search tree. As a result of the inference, all attributes of the child concept must be binded to the values. And, the expression of the relations should be true and not contain free subexpressions. It is worth noting that failed inference of parent concepts may be acceptable result, for example in operations of negation. The order of parent concepts in the "`from`" section determines the order of traversal of the search tree. This fact makes it possible to manually optimize the search for a solution, starting with those concepts that have less possible values and make the search space narrower.

The task of the inference is to find all possible substitutions of the attributes of the child concept and to represent each of them as concept instances. The instance has the form of an object. These instances, in turn, can be used to infer the values of other concepts, or they can be treated as objects of the component of computing. The instances are considered identical if the names of their concepts and the names and values of attributes coincide.

It is considered acceptable to create several concepts with the same name but with a different implementation, including a different set of attributes. Such concepts can describe alternative ways of obtaining different versions of the same entity, for example, from different sources. Or combine related concepts under one name. The inference algorithm processes all existing definitions of the concept and combines their results.

4.3 Concepts inheritance

One of the most common relations between concepts is hierarchical relation. Its feature is that the structures of the child and parent concepts are very close. Therefore, maintaining the inheritance mechanism at the syntax level is very important; programs will be filled with repetitive code without it. When building a network of concepts, it would be convenient to reuse both their attributes, parent concepts and relations. The list of attributes is easy to expand, shorten or redefine some of them. The situation with the modification of relations is more complicated. Since they are logic expressions in conjunctive form it is easy to add additional subexpressions to it. But deleting or modifying them may require significant syntax complexity. The benefits of this are not so obvious, so it was decided to postpone this task for the future.

One can declare a concept based on inheritance using the following construct:

```
<concept definition> ::=
"concept" <concept name> [<concept alias>] "is" (
  <parent concept name> [<parent concept alias>]
  {"," <parent concept name> [<parent concept alias>]}
["with"
  <attribute name> ["=" <expression>]
  {"," <attribute name> ["=" <expression>]}]
["without"
  <parent attribute name>
  {"," <parent attribute name> }]
|
  <parent concept alias> {"," <parent concept alias>}
["with"
  <attribute name> ["=" <expression>]
  {"," <attribute name> ["=" <expression>]}]
["without"
  <parent attribute name>
  {“,” <parent attribute name> }]
“from”
  <parent concept name> [< parent concept alias>]
  [“(”
    <attribute name> “=” <expression>
    {“,” <attribute name> “=” <expression>}
  “)”]
  {“,” <parent concept name> [<parent concept alias>]
  [“(”
    <attribute name> “=” <expression>
    {“,” <attribute name> “=” <expression>}
  “)”]}
)
[“where” <expression of relations>]
```

The “`is`” section contains a list of inherited concepts. Their names can be specified directly in this section. Or, a complete list of parent concepts can be specified in the “`from`” section, and in “`is`” aliases of only those ones that will be inherited. In “`with`” section one can expand the list of attributes of inherited concepts or redefine some of them; in “`without`” section – remove them.

Let's look at some examples of inheritance mechanism application. Inheritance allows to create a concept based on an existing one, getting rid of those attributes that make sense only for the parent concept and not for the child one. For example, if the source data is presented in the form of a table, then it is possible to give names to cells of certain columns and get rid of the column’s numbers:

```
concept revenue is tableCell c without c.column
where c.column = 2
```

Inheritance is also useful when it is needed to convert several related concepts into one generalized form. Using “`with`” section it is possible to convert some of the attributes to a unified format and add the missing ones:

```
concept cv is cvV1 with skills = 'N/A'
concept cv is cvV2 c with skills = c.coreSkills
```

The extension of the attribute list is a common task. It may be changing the format of attributes, adding attributes functionally dependent on existing attributes or external data, etc. For example:

```
concept price is basicPrice
with valueUSD = valueEUR * getCurrentRate('USD', 'EUR')
```

It is also possible simply combine several concepts under one name without changing their structure. For example, to indicate that they belong to the same genus:

```
concept webPageElement is webPageLink
concept webPageElement is webPageInput
```

Or create a subset of the concept by filtering out part of its values:

```
concept exceptionalPerformer is employee e
where e.performanceEvaluationScore > 0.95
```

Multiple inheritance is also possible. According to it the child concept inherits the attributes of all parent concepts. If the parent concepts have the same attribute names, priority will be given to the most left one in the "`is`" section. One can also resolve this conflict manually by explicitly overriding the desired attribute in the "`with`" section. For example, this type of inheritance would be convenient if one needs to collect several related concepts in one "flat" structure:

```
concept employeeInfo is employee e, department d
where e.departmentId = d.id
```

Inheritance without changing the structure of concepts complicates the verification of the identity of objects. As an example, consider the definition of "`exceptionalPerformer`" concept. Requests to the parent ("`employee`") and child ("`exceptionalPerformer`") concepts returns the same employee instance. The objects representing it are identical in meaning. They have a common data source, the same names and values of the attributes, but different concept name, depending on which concept the query was made to. Therefore, the operation of equality of objects must take this feature into account. Names of concepts are considered equal if they coincide or are connected by a transitive relation of inheritance without changing the structure.

So, inheritance is a useful mechanism that allows to explicitly express such relations between concepts as "genus-species", "class-subclass", "specific-general", "set-subset", etc. And get rid of code duplication in definitions of concepts and make the code more understandable.

4.4 Concept of relation

In some cases, the relations between concepts should be declared explicitly instead of being included into the definition of one of the concepts. It may be some universal relation applicable to different concepts. Or the concepts associated by such relation are equal. And the relation can be used to derive any of the concepts participating in it.

Such a relation can be represented in the form of a concept whose attributes are nested concepts participating in it. For example, a relation describing rectangles nested in each other can be defined as follows:

```
concept nestedSquare(
  inner = i,
  outer = o
) from square i, square o
where i.xLeft > o.xLeft and i.xRight < o.xRight
  and i.yBottom > o.yBottom and i.yUp < o.yUp
```

But besides logic semantics, this relation also has a functional one. The relation is a function of Boolean type that checks whether the indicated concepts correspond to it:

```
function nestedSquare(inner, outer) {
  return exist(nestedSquare{inner: inner, outer: outer});
}
```

Therefore, it makes sense to combine these two definitions of concept and function in one definition of relation:

```
<relation definition> ::=
"relation" <relation name> ["("
  <outer alias of nested concept> ["=" <expression>]
  {"," <outer alias of nested concept>
      ["=" <expression>]}
")"]
"between"
  <nested concept name> [<nested concept alias>]
  ["("
    <attribute name> "=" <expression>
    {"," <attribute name> "=" <expression>}
  ")"]
  {"," <nested concept name> [<nested concept alias>]
  ["("
    <attribute name> "=" <expression>
    {"," <attribute name> "=" <expression>}
  ")"]}
["where" <expression of relations>]
```

This definition creates a relation between the concepts indicated in "`between`" section that correspond to the logic expression from "`where`" section. The outer aliases section allows to specify the aliases of concepts for external access; they correspond to the attribute names of the child concept in concept definition example and the input arguments of the function definition example. This section is optional. By default, the list of nested concepts corresponds to those specified in the `'between`" section, outer aliases correspond to their inner aliases.

So, the concept can be defined as relation as follows:

```
relation nestedSquare
between square inner, square outer
where inner.xLeft > outer.xLeft
  and inner.xRight < outer.xRight
  and inner.yBottom > outer.yBottom
  and inner.yUp < outer.yUp
```

And it can be used in the definition of other concepts both as concept:

```
concept htmlFormElement is e
from htmlForm f,
     nestedSquare(inner = e, outer = f),
     htmlElement e
```

and as function:

```
concept htmlFormElement is e
from htmlElement e, htmlForm f
where nestedSquare(e, f)
```

These two use cases have different behavior during inference. In the first case the relation is treated as usual concept in "`from`" section. Inference algorithm starts from finding instances of "`htmlForm`" concepts, then binds them to "`outer`" nested concept of "`nestedSquare`" relation and finds the instances of the "`inner`" one. In the second case it is treated as function and its calculation will be delayed until all its arguments are binded to values. Inference finds all combinations of the concepts "`htmlElement`" and "`htmlForm`" and then filters out those that do not match the "`nestedSquare`" relation.

4.5 Concept as function

A declarative logic form of concepts definitions is not always convenient. In some cases, it is more natural to specify a sequence of calculations, which result is a list of instances of the concept. Functional style may be appropriate to describe an infinite sequence that generates the instances of a concept, for example, a sequence of integers. It is common task to download facts from an external source, for example, from a database or a web service. In this case the concept can be represented by a function that translates a query to the concept into a query to the database and returns the result of it. Also, sometimes it makes sense to replace the inference algorithm with a custom implementation that takes into account the specifics of a task.

The principles for working with such concepts should be the same as with the other concepts described above. The search for solutions should be called by the same methods. They can be used as parent concepts in definitions of concepts in the same way. Only the internal implementation of the procedure generating instances of the concept should be different. Therefore, let's introduce another one way to define concept and call it "concept as function":

```
<concept definition> ::=
"concept" <concept name> "("
  <attribute name>
  {"," <attribute name> }
")"
"by" <instances generation function>
```

Concept by function definition includes a list of attributes and an instances generation function. It was decided to make the list of attributes a mandatory element of the definition, as this simplifies the use of this type of concept. There is no need to study the function in order to understand the structure of the concept.

The instances generation function should return an iterator over a collection of inference results. This makes the definition of the concept more flexible. For example, it's possible to implement lazy evaluation of entities or their infinite sequence. Developers of the generation function can use the iterator of any standard collection or create its custom implementation. An element of the collection should be an associative array of attributes of the concept. The instances of the concept are automatically created from them. The input arguments of the function are a query object (which specifies the constraints on the values of the result) and an inference mode (all possible results, only the first result or verification of the existence of the result). The query and the inference mode will be discussed in more detail below.

Let's consider some examples of the definition of the concept by function. A concept of a time interval may look as follows:

```
concept timeSlot15min (hour, minute)
by function(query, mode) {
  var timeSlots = [];
  for(var curHour = 8; curHour < 19; curHour += 1) {
    for(var curMinute = 0; curMinute < 60;
        curMinute += 15) {
      timeSlots.push({
        hour: curHour,
        minute: curMinute
      });
    }
  }
  return timeSlots.iterator();
}
```

The function returns an iterator for all possible values of the 15-minute time interval. It doesn't check the output for compliance with the query. This will be done automatically when converting the array of attributes into the instances of the concept. But if necessary, query fields can be used to optimize the function. For example, to generate a query to a database based on the fields of the query to the concept.

As an another example, let's determine the values of a certain exponential scale in the form of a concept:

```
concept expScale2 (value, position)
as function(query, mode) {
  return {
    _curPos = 0,
    _curValue = 1,
    _limit = 100,
    next: function() {
      if(!this.hasNext()) {
        throw OutOfBoundException;
      }
      var curItem = {
        value: this._curValue,
        position: this._curPosition
      };
      this._curPos += 1;
      this._curValue = this._curValue * 2;
      return curItem;
    },
    hasNext: function() {
      return this._curPos < this._limit;
    }
  };
}
```

The function returns an iterator that generates concept instances using lazy calculations. The size of the sequence is limited in the "`hasNext`" method, but if necessary, it can be turned into infinite. Concepts based on infinite sequences must be used very carefully, as they do not guarantee completion of the inference.

This section proposes some of the options for how the concept as function may look like. But the safety and usability issues of such concepts require more detailed research in the future.

4.6 Logic variables

In logic programming variables are placeholder for arbitrary terms that connects elements of rules. In the modeling component, concept attributes primarily play this role. But in some cases, it would be convenient to declare a logic variable that is not the part of the attributes of any of the concepts but is used in the expression of relations. Logic variables can be used as references to subexpressions when it is needed to split complex expression into smaller ones. Or when a subexpression is used several times.

In order to distinguish logic variables from the variables of the component of computing, the names of the former must begin with the symbol "`$`". Example of variable usage:

```
concept pointInAnnulus (point = p, annulus = a)
from point p, annulus a
where $dist <= a.R
  and $dist >= a.r
  and $dist = Math.sqrt((p.x - a.x) * (p.x - a.x) +
                        (p.y   - a.y) * (p.y - a.y))
```

4.7 Negation

In this subsection, we consider the meaning an implementation of a negation operator of the component of modeling.

Logic programming systems usually include the rule of negation as failure [29] in addition to the operator of Boolean negation. It infers "*not p*" if the inference of "*p*" fails. In different systems of knowledge representation, both the meaning and the algorithm of the inference of negation rule may differ.

In systems that adhere to the "open-world assumption", the knowledge base is considered incomplete, therefore, the statements missing in it are considered unknown. The statement "*not p*" can only be deduced if it is definitely possible to derive from the knowledge base statements that "*p*" is false. Such negation is called strong. Systems adhering to the "closed-world assumption" [30] infer "*not p*" also when there is no information about "*p*" in the knowledge base. It is believed that in such systems the knowledge base is complete, and the missing statements in it are considered false. Such a negation operator is called negation as failure.

In different logic programming systems, the meaning of negation as failure also has some features. For example, in the case of cyclic definitions:

p <— not q

q <— not p

classic SLDNF resolution is not able to complete. Such definitions are not valid in Prolog language. But logic programming systems with stable model semantics [31] and well-founded semantics [32] can work with such definitions. Stable model semantics allows to find many possible solutions: "*{p, not q}*" and "*{q, not p}*". And well-founded semantics allows to find a partial solution, leaving some of the conclusions uncertain.

It was decided not to add the operator of negation as failure to the component of modeling. First, the source data can correspond to both the complete and the incomplete state of knowledge, it depends on the nature of the data source. So, the component of modeling is closer to the "open-world assumption". Also, concepts in the component of modeling and statements in logic programming have different meaning. Terms in logic statements can be either false or not inferred from the knowledge base. Concepts of the component of modeling have more complex structure and the mixture of Boolean negation and non-deductibility doesn't make sense. Boolean negation can be applied only to concept attributes; they can be both false and true. It is impossible to directly apply it to concepts; they can consist of different attributes and it is not clear which one should be responsible for the falsity or truth of the concept as a whole. Also, attribute may be undefined if it is not included in the specific instance of the concept. The entire child concept may be non-deductible from the source data.

Therefore, it makes sense to introduce separate operators for each type of negation listed above. Attribute falsity can be checked by the traditional Boolean operator "`not`", whether concept contains an attribute by built-in function "`defined`", and the result of inferring the concept from the source data by function "`exist`". Three separate operators are more predictable, understandable, and easy to use than the complex operator of negation as failure. An attempt to add the negation as failure operator to the component of modeling seems redundant now, especially of its complex versions in the form of semantics of stable models or well-founded semantics.

Let's look at some examples. A concept can be considered false if certain of its attributes have a value indicating this. Negation of attributes allows to find such entities:

```
concept unfinishedTask is task t where not t.completed
```

Or if it doesn't have certain attributes. Functions checking whether attribute is defined can help with that:

```
concept unassignedTask is task t
where not defined(t.assignedTo) or empty(t.assignedTo)
```

The function of checking the existence of the concept is indispensable when working with recursive definitions and hierarchical structures:

```
concept minimalElement is element greater
where not exist(
  element lesser where greater.value > lesser.value
)
```

In this example, a check for the existence of a smaller element is performed as a subquery. The creation of subqueries will be discussed in Section 4.9.

4.8 Elements of higher-order logic

In higher-order logic, not only statements about specific facts are allowed but also about statements, including statements about statements, etc. In Prolog, elements of such logic are implemented using several built-in predicates, the arguments of which are other predicates. For example, predicate "`call`" allows to dynamically add its arguments to a list of goals of a current rule, "`findall`", "`bagof`" and "`setoff`" to find all solutions of the specified predicate. Also, in Prolog there are built-in tools for finding predicates in the knowledge base and manipulating their attributes. HiLog [33] supports higher-order logic at the syntax level. It allows to use arbitrary terms (for example, variables) at the positions of predicates.

In the component of modeling higher-order logic can take the form of using arbitrary expressions in positions of concept names and their attributes. To do this, let's introduce a special operator of dynamic substitution of concept names and their attributes:

```
<name operator> ::= "<" <expression> ">"
```

It converts the value of the expression to a concept name, its alias, or attribute name, depending on the context. Let's discuss some of these features.

The most obvious example is the unification under one name of all concepts that satisfy certain conditions, e.g. having certain attributes. A concept of a point can be defined as unification of all concepts that include x and y coordinates:

```
concept point is <$anyConcept> a
where defined(a.x) and defined(a.y)
```

Inference binds variable "`$anyConcept`" with all possible names of concepts (of course, except for itself) and filter out those that do not have coordinate attributes.

A more complex example is to declare a generic relation that can be applied to many concepts. E.g. a transitive parent-child relation between concepts with the same name:

```
relation ancestorOf
between <$parent> parent, <$child> child
where $parent = $child and defined(parent.id)
  and defined(child.parent)
  and (
    parent.id = child.parent
    or exist(
      <$parent> intermediate
      where intermediate.parent = parent.id
        and ancestorOf(intermediate, child)
    )
)
```

Also, dynamic name substitution is useful if attributes of one concept are links to the names of other concepts or their attributes. For example, when the source data contains not only facts, but also their structure, e.g. XML or database schema definitions. The source data may also include additional information about facts, such as data types, formats, default values, validation rules etc. Let's consider an example when the source data includes certain facts as well as the rules for validating the attributes of these facts as a separate entity:

```
fact validationRule {
  attributeName: "...",
  rule: function(value) {…}
}
```

Validation results can be described by the following concept:

```
concept validationRuleCheck (
  attributeName = r.attrName,
  result = r.rule(o.<r.attrName>)
) from validationRule r, someObject o
where defined(o.<r.attrName>)
```

Also, the source data can describe a model of something, and the component of modeling may be responsible for building the metamodel. Working with texts in natural language also assumes that the source data includes not only facts, but also statements on them. The expressibility of first-order logic is not enough in all these cases, and the more expressive higher-order logic is needed.

The topic of higher-order logic for metamodeling is quite complex and requires more detailed and thorough research. Both in terms of choosing a convenient language design and performance of the inference algorithm. It is out of the scope of this paper but is interesting area of future work.

4.9 Nested concept definitions

In the SQL world, subquery is a tool often used when it is needed to obtain intermediate data for further processing in a main query. In the component of modeling there is no such urgent need for intermediate data, since the way of obtaining them can be formalized as a separate concept. But there are cases when nested definitions of concepts would be convenient. For example, when it is needed to get the instances of a concept using the "`exist`" or "`find`" functions, without including it in the list of parent concepts. (These functions will be discussed in more detail below.) Often there is a need to slightly modify the concept, select some of its attributes, filter their values. It is easier to do this in the body of the definition of the external concept instead of creating a new one. Sometimes it makes sense to use the nested definition of the concept in the "`from`" section. For example, one can create the nested concept definition in the "`from`" section and transfer part of the conditions out of the "`where`" section into it. Then they will be checked only at the stage of searching for a solution to the nested concept. This makes it possible to check the existence of its solution in the "`where`" section of the external concept. This is the analog of the "`OUTER JOIN`" section of the SQL language, "`where`" clause of the nested concept corresponds to the "`ON`" section.

Let's discuss the syntax of the nested concept definition. In general, it follows the syntax of common concept definitions, except that in some cases one can omit attribute lists and even the name of the child concept. If the nested concept is used as an argument to the "`exist`" function, then its name and list of attributes are not important and can be omitted. In the "`find`" function, one can omit the concept name if the objects obtained as search results are converted to lists or associative arrays of attribute values. If the concept name or the attributes are not specified, then they will be inherited from the parent concepts.

Let's try to clarify the above with a few examples. Using the "`exist`" function, one can check the derivability of the nested concept:

```
concept freeExecutor is executor e
where not exist (
  task t
  where t.executor = e.id
    and t.status in ('assigned', 'in process')
)
```

The "`find`" function returns a list of all values of the concept. The list can be binded with an attribute or variable or be an argument to an expression:

```
concept customerOrdersThisYear is customer c with orders
where c.orders = find(
  order o
  where o.customerId = c.id
    and o.completedDate > '2019-01-01'
)
```

Declaring the nested concept in the "`from`" section allows to separate part of the relations conditions from the main expression in the "`where`" section. And thus, make the failure of the inference of the parent concept acceptable:

```
concept taskAssignedTo is t
from task t, (user where id = t.assignedTo) u
with assigneeName = if(
  exist(u),
  u.firstName + ' ' + u.lastName,
  'Unassigned'
)
```

4.10 Aggregation

Aggregation is the integral part of both relational algebra and logic programming. In SQL, "`GROUP BY`" section allows to group rows that have the same key values into summary rows. It helps to remove duplicate values and is usually used with aggregate functions such as "`sum`", "`count`", "`min`", "`max`", "`avg`". For each group of rows, aggregate functions return an ordinary value calculated for all rows of this group. In logic programming, aggregation has more complex semantics [34]. This is because in some cases of rule recursive definitions, SLD resolution falls into an endless loop and is not able to complete. As in the case of negation as failure, the recursion problem in the aggregation operation is usually solved using the semantics of stable model or well-founded semantics. But since the semantics of the component of modeling should be as simple as possible, a standard SLD resolution is preferable. And the problem of avoiding infinite recursion is better to left to developers.

Aggregation could naturally be implemented using the component of computing of the hybrid language. For this, a function that collapses the results of inference by unique groups and calculates aggregate functions for each of them would be enough. But dividing the definition of concept into logic and functional parts is not the most convenient solution for such an important tool as aggregation. It is better to expand the syntax of the concept definition by adding a grouping section and aggregation functions to it:

```
<concept definition> ::=
"concept" <concept name> [<concept alias>] "("
  <attribute name> ["=" <expression>]
  {"," <attribute name> ["=" <expression>]}
")"
["grouped by" <attribute name> {"," <attribute name>}]
"from"
  <parent concept name> [< parent concept alias>]
  ["("
    <attribute name> "=" <expression>
    {"," <attribute name> "=" <expression>}
  ")"]
  {"," <parent concept name> [< parent concept alias>]
  ["("
    <attribute name> "=" <expression>
    {"," <attribute name> "=" <expression>}
  ")"]}
["where" <expression of relations>]
```

The "`grouped by`" section, like the "`GROUP BY`" section in SQL, contains a list of attributes by which grouping is performed. The expression of relations may also include aggregation functions. Terms containing such functions are considered undefined until the moment when values of all parent concepts are found, and grouping is performed. After that, their values can be calculated for each group, associated with attributes and / or used in filtering. With this approach, there is no need for "`HAVING`" section that separates filtering conditions before and after grouping.

The main functions of aggregation are "`aggregate`", "`count`", "`sum`", "`avg`", "`min`", "`max`". The purpose of most of the functions follows from their names. Separately, it is worth highlighting the "`aggregate`" function. Its input argument is an expression; the function returns a list of the results of calculating this expression for each element of the group. By combining it with other functions, one can implement any arbitrary aggregation function.

Example of aggregation:

```
concept totalOrders (
  customer = c,
  orders = aggregate(o),
  ordersCount = count(o),
  ordersSumTotal = sum(o.sum)
) grouped by customer
from customer c, order o
where c.id = o.customerId
```

5 Description of the component of computing

Now let's move on to the component of computing. The choice of features of this component is not as critical as for the component of modeling. The global goal of the work is to find ways to integrate component of modeling with all the widely used programming languages. It was decided to take the JavaScript language as the basis for the component of computing of an experimental version of the hybrid language. The goal is not to implement the entire ECMAScript specification, but rather to borrow a set of elements from it that is minimally necessary to create the experimental version of the language. The implementation of the component of computing is slightly different from the ECMAScript standard, mainly due to the personal preferences of the author and in order to speed up and simplify the development. Let's briefly consider its main elements.

A dynamic type system is chosen. Variables and fields of objects, initially associated with values of one type, can be reassigned with values of another type. The language supports automatic type casting in implicit situations. For example, the string '1' can be automatically converted to the number 1 and vice versa, the expression 1 + '2' returns 3, and '2' + 1 returns '21'. But if necessary, the value can be explicitly converted to the desired type. Such a type system is very simple to implement and flexible enough to combine type systems of an application and diverse data sources. The primitive data types are "`null`", "`number`", "`string`" and "`boolean`". It was decided to abandon "`undefined`" data type.

Lists (similar to the JavaScript arrays), objects, and dates are supported. Lists are enclosed in square brackets:

```
var emptyList = [];
//list contains variable "a" and number, string and
// boolean values
var list = [a, 1, "string", false];
```

List item can be accessed by specifying its index in square brackets:

```
var someVariable = list[0];
list[1] = 2;
```

The language supports built-in operations over lists, such as getting a head, tail, list size, getting part of the list, adding and removing items, etc.

An object can be represented as an associative array that connects the fields (or slots) of the object with values. Objects are created using curly braces:

```
var emptyObject = {};
var obj = {
  field1: 1,
  field2: 2
};
```

Such built-in methods of associative arrays as retrieving and deleting an element by key, check of key existence, obtaining list of keys, values and key-value pairs, etc. are available. For lists and objects, methods from the world of functional programming such as "`map`", "`filter`", "`find`", "`forEach`", and "`fold`" are also useful.

Functions are implemented as a special kind of built-in objects. One can create an instance of the function, assign it to a variable, pass it as an argument to another function, or return it as a result of calculation. This makes functions the first-class citizens of the language. They can be declared in the traditional way by defining a function:

```
function add(x, y) {
  return x + y;
};
```

or assigned to a variable using a functional expression:

```
var add = function(x, y) {
  return x + y;
};
```

Creation of anonymous functions are allowed:

```
var filteredList = list.filter(function(item) {
  return item.value > 0
});
```

The scope of variables in component of computing is slightly different from that in JavaScript and is closer to traditional options, such as in Go language. The scope of the variable is limited to the block in which it is declared; when the block is exited, its value is lost. For example, a function declared as:

```
function f() {
  var x = 1;
  println(x); // 1
  for(var i = 10; i < 30; i = i + 10) {
    var x = i;
    println(x); //10 and 20
  }
  println(x); // 1
}
```

will output following lines: 1, 10, 20, 1. In the "`for`" block, the local version of the variable x is declared, and after the end of this block its value will be lost. There is no block visibility in JavaScript and modifying the variable "`x`" in the "`for`" block changes its global value. It was also decided to abandon the principle of hoisting, in which variable declarations are moved to the start of the scope. The scope of a variable in the component of computing starts from the position where it is declared. Function

```
var scope = 'global';
function f() {
  println(scope); // global
  var scope = 'local';
  println(scope); // local
}
```

first displays the value of the global variable and then of the local one. In JavaScript, the local variable will be declared at the beginning of the function, but initialized in the middle, so the first "`println`" statement outputs the value "`undefined`".

The scope of functions is the same as in JavaScript. Functions declared using a definition are available throughout the block and can be used above their declarations. If the function is created using an expression and assigned to a variable, then it is available only after assignment but not higher.

The component of computing supports a traditional set of arithmetic, Boolean and comparison operators. The comparison operator “==” checks the equality of the values of the operands after type casting, for example `1 == '1'` returns true. The identity operator “===” checks the equality of both values and types of operands. When comparing objects, equality and identity operators compare the contents of object fields. In this, the component of computing is different from JavaScript, which checks whether the operands reference the same object.

The main control structures are branching operators:

```
if(condition) {
  consequent operators
} else {
  alternative operators
}
result = condition ? consequent expression : alternative
expression
```

and loop operators:

```
for(initial; condition; loop statement) {
  loop statements
}
for(iterator in list) {
  loop statements
}
while(condition) {
  loop statements
}
```

It was decided to postpone for the future design and implementation of such elements of object-oriented model as inheritance and constructors. This is certainly important features, but their implementation requires considerable effort and time, and it is not the main priority of this work.

6 Integration of the components of modeling and computing

Integration of the components of computing and modeling allows to achieve synergy; both components benefit from it. To begin with, let's consider this interaction from the side of the component of computing, then from the side of the component of modeling and separately the issue of the scope of facts and concepts.

6.1 Facts and concepts in the component of computing

In the component of computing, facts and concepts have the form of special built-in objects. Facts are represented by objects having such fields as a unique identifier "`id`", name "`name`" and an associative array of attributes and their values "`attributes`". Built-in methods for adding and removing a fact from a namespace are available:

```
function register()
function forget()
```

Mechanism of namespaces will be discussed below in Section 6.3. The operators of comparison "==" and identity "===" of facts are available, they compare both the attributes and the name of the fact. One can create a new fact instance using "`new`" operator:

```
new Fact(name, attributes)
```

Objects representing concepts have a more complex structure. Fields of concept include a unique identifier "`id`", name "`name`", a list of attribute names "`attributes`", a list of parent concepts "`parentConcepts`", a list of grouping attributes "`grouped`", and a list of expressions of relations "`relations`". The elements of the list of parent concept names are pairs containing their names and aliases. Items of the "`relations`" list are conjunct expressions of expression of relations. The "`grouped`" list may be empty.

Fields of inheritance-based concepts also contain a unique identifier "`id`", a name "`name`", a list of parent concepts "`parentConcepts`", and a list of expressions of relations "`relations`". The "`relations`" list may be empty. In addition, this concept may include lists of overridden and excluded attributes "`additionalAttributes`" and "`removedAttributes`". The elements of the "`removedAttributes`" list are pairs consisting of the alias of the parent concept and the attribute name. The "`attribute`" list may be empty.

Fields of relation concept include a unique identifier "`id`", name "`name`", a list of expressions of relations "`relations`", as well as a list of concept names participating in the relation "`nestedConcepts`" and their external aliases "`externalNames`". The lists of "`relations`" and "`externalNames`" may be empty.

The fields of a concept as function are a unique identifier "`id`", a name "`name`", a list of attribute names "`attributes`", and an instances generation function "`resolver`".

One can create concept objects using constructor functions or auxiliary builder objects. Given the large number of fields, the second method is preferred:

```
var profitDef = Concepts.buildConcept(
   ["profit", "p"]
).attributes(["value", "row"])
 .from([["revenue", "r"], ["cost", "c"]])
 .where([
    () => p.row == r.row,
    () => r.row == c.row,
    () => p.value == r.value - c.value
  ]).build();

var revenueDef = Concepts.buildConceptInheritance(
  "revenue"
).from([["cell", "c"]])
 .without(["c", "column"])
 .where([() => c.column == 2])
 .build();

var authorOfRelDef = Concepts.buildRelation(
  "authorOf"
).from([["author", "author"], ["book", "work"]])
 .where([() => work.authorId == author.id])
 .build();

var databaseOperationDef =
    Concepts.buildConceptAsFunction(
  "databaseOperation"
).as(function() {
  return ["SELECT", "INSERT", "UPDATE", "DELETE", "CALL"]
  .map(
    function(item) => {
      return {name: item};
    }
  ).iterator();
}).build();
```

Concepts created in this way can be passed as arguments to functions, returned from a function as a result, added to collections, etc. "`new`" operator allows to dynamically create concepts and facts. For example, it is possible to transform data from external source to facts or create a definition of a concept as a result of interaction with a user, etc.

To make the concepts created as objects to be visible to inference mechanism, they must be added to the current namespace using "`register`" method. The concepts and facts created in the component of modeling using the "`concept`" and "`fact`" keywords are added to the namespace automatically.

Inference of instances of a concept can be performed using the "`find`", "`findOne`", and "`exist`" methods. The first one finds all possible values, the second one only the first of them, the third checks the existence of at least one solution. The input argument to these methods is a query object. The query object is an associative array containing the attribute values to which the results of the inference should correspond. For example, a call:

```
var profitRow1 = profitDef.findOne({row: 1});
```

will return an object representing an instance of the concept of profit, whose row attribute is 1. Query object may be empty. Also, there are methods “`find`”, “`findOne`”, and “`exist`” of a built-in object “`Concepts`” that allow to perform inference by the name of the concept:

```
var allProfits = Concepts.find(“profit”, {});
```

Functions to extract concepts and facts registered in a current namespace might also be useful:

```
var conceptsList = Concepts.getConcepts();
var profitDefinitions = Concepts.getConcepts(, “profit”);
var factsList = Concepts.getFacts();
var cells = Concepts.getFacts(“cell”);
```

It is possible to get all definitions or filter them by name. Functions that return definitions of concepts that have specified attributes would also be useful. But its effective implementation has been postponed for the future. This is because concepts based on inheritance do not have a fixed list of attributes and inference may be necessary to obtain them.

The concept of relations, in addition to logic semantics, also has functional semantics. Therefore, relations definition objects have a built-in method “`apply`”, the arguments of which are objects, the result is a Boolean value. This method performs inference and returns “`true`” if its arguments can be derived in the current namespace and match the expression of relations.

The main purpose of concepts and facts is to describe the domain model. But they can also be used as an auxiliary tool, that makes the application architecture more dynamic. For example, the component of modeling could replace design patterns such as service locator or publisher/subscriber.

6.2 Functions and variables in the component of modeling

The component of modeling also benefits significantly from the integration. The most obvious benefit is the use of custom functions in the expression of relations. In many cases, this is more convenient than implementing them as logic predicates:

```
concept userProfile is user u
with birthDate = getBirthDateFromID(u.personalId);

function getBirthDateFromID(personalId) {
  var birthDateStr = personalId.split(“-”)[0];
  if(birthDateStr.length() != 6) {
    return null;
  }
  var day = Number(birthDateStr.substring(0,2));
  var month = Number(birthDateStr.substring(2,2));
  var yearDigits = Number(birthDateStr.substring(4,2));
  var year = yearDigits <= Date.getDate().getYear() ?
    2000 + yearDigits : 1900 + yearDigits;
  return Date(day, month, year);
};
```

In definitions of concepts, not only functions can be used, but also other elements of the component of computing, such as constants, variables, and objects. Variables can store the results of previous calculations or data input. For example:

```
var currentDate = Date.getDate();
concept currentTask is task t
where t.performAt = currentDate
  and t.status = 'scheduled';
```

If a variable is used, the concept will use its value calculated at the time the concept was declared. It will remain unchanged even if the variable changes or disappears after going beyond its scope.

Integrating objects into the component of modeling is a more complex task, because a reference to the object is transferred to the definition of the concept. So, when calling the inference function, the current state of the object will be used, not the one that was at the time the concept was defined. In addition, this state may change during the inference process. This can be done by the methods of the object, called, for example, when checking conditions of the expression of relations. This contradicts backtracking search algorithm, according to which the alternative branches of the search for solutions should not influence each other. So, changing the state of the object when processing one branch of the search tree can change the behavior of the object when processing the following branches. In most cases, it is necessary to restore the state of the object when returning to a narrower solution, but to save them when moving to an advanced one. Moreover, it is desirable to do this automatically, this would greatly simplify the work with objects in the component of modeling. Although it should be noted that in some cases it would be convenient to have the opposite behavior at which objects keep their state changed. For example, to collect statistics or debug information about the search process. Or to cache the results of complex calculations that can be reused in alternative branches of the search. Therefore, it makes sense to allow developers to choose the behavior of objects in the process of backtracking search. This mechanism will be described in more detail in Section 7.1.

6.3 Scope of facts and concepts

A set of defined concepts and facts require tight control. The fact is that creating concepts with the same name and different definitions is acceptable. Inference procedure is applied to each of these concepts and the results are combined. Accidental coincidence of concept names can seriously affect search results. In logic programming systems, facts and rules are usually combined in a common space. In most cases, this space is static, so it is easy to control it. But the component of modeling must deal with the set of concepts and facts that can be formed dynamically by the component of computing. They can be created inside functions that call one another. These functions can be called repeatedly. As a result, considering a function is almost impossible to control how it was called and what concepts were created earlier. Therefore, clear, transparent and easy-to-use rules are needed to define the scope of definitions of concepts and facts and eliminate their unintentional mutual influence.

To begin with, the lexical scope, which is defined statically by the code structure, is more convenient for concepts. According to it, concepts and facts that are declared inside a current function should be visible. Also, the scope of concepts includes a function in which the current function is declared. And a function that this function belongs to, and so on to the root of the file. In other words, the concept declared in a certain block of code should be available in all functions declared in this block, including functions embedded in these functions, etc.
Example 6.3.1:

```
concept c1 …;
function rootFunc(...) {
  concept c11 …;
  function nestedFunc() {
    concept c111 …;
  }
}
function anotherRootFunc(...) {
  concept c21 …;
}
```

So, in Example 6.3.1. in the "`nestedFunc`" function, concepts "`c111`", "`c11`" from the "`rootFunc`" function in which it is declared, and "`c1`" from the root of the file are available.

Some of the concepts that are common to the entire file can be statically declared in its root. And in nested functions – concepts whose declarations need to be formed dynamically, for example, depending on the input parameters of the function. Also, if there are some groups of concepts that should not intersect with each other, then they can be placed in different functions, and shared concepts for all groups can be placed in the root of the file. So, the concept "`c1`" is global to the functions "`rootFunc`" and "`anotherRootFunc`", and each of them introduces its own additional local concepts "`c11`" and "`c21`", available only to them.

So, the lexical version of the scope of concepts arranges concepts and facts into a hierarchical structure that corresponds to the structure of functions declarations. This approach simplifies the control of the scope of concepts and facts. Obviously, having a file before eyes there should not be a problem with understanding what concepts will be available at any time of the program execution.

Another important task is to ensure the reuse of concepts. A mechanism, that allows the use of concepts declared elsewhere in the program, is needed. In functional and object-oriented languages, the problem of logical grouping of unique identifiers (variables, functions, classes, etc.) is solved using namespaces. In languages such as C++, C#, and PHP, the namespace is specified using the "`namespace`" keyword, which defines the boundaries of one or more pieces of code. The rules for structuring program code are more stringent in Java and Python. It should be divided into files called packages, each of which has its own namespace. One can access the identifier either by its full name, including the file name and path to it. Or use the "`import`" directive to access identifiers from the specified package or module with just a simple name. In JavaScript, namespaces can be defined using objects; functions and variables are simply included into the object as its fields.

The approach of splitting the program into files and linking them with import directives is quite convenient and encourages the structuring of the program, so let's choose it. To do this, import directives that can open access to concepts declared from outside the current namespace are needed. Since the concepts can be declared both in the root of a file and in functions, two directives are needed. The first one allows adding concepts and facts defined in the root of the external file to the current namespace:

```
concepts from file <file name>;
```

the second is for concepts defined inside the function:

```
concepts from function <function call>;
```

Both directives add both static and dynamic definitions of concepts and facts. To do this, first directive executes the code located in the root of the file. Second directive also executes code of the function so input arguments should be passed to it. Since the concepts defined in the root of the file are available in the function, the second directive makes them visible too. If the imported file or function in turn contains import directives, then the imported concepts will also be exported to the outside.

Even if the same file is imported twice, its concepts should be added to the current namespace only once. Import from a function can be performed multiple times, but only for different combinations of values of its input arguments. For example, a function of reading concepts and facts from a file may add the contents of different files to the current namespace, but the contents of each file cannot be added twice. The implementation of this principle may be different. For example, ignore the recall of the import directive, or replace old results with new ones. For now, let's choose the first option.

Let's consider the following example, in which concepts are distributed across several files:
Example 6.3.2.
file1.oop:

```
concept c1 …;
function createConcept(choice) {
  if(choice) {
    concept c11 …;
  } else {
    concept c12 …;
  }
}
```

file2.oop:

```
use 'file1.oop';
concepts from function createConcept(true);
concept c2 …;
```

file3.oop:

```
use 'file1.oop';
concepts from function createConcept(false);
concept c3 …;
```

file4.oop:

```
concepts from file 'file2.oop';
concepts from file 'file3.oop';
```

file5.oop:

```
use 'file1.oop';
concepts from function createConcept(true);
concept c5 …;
function doSomething() {
  concepts from function createConcept(false);
}
```

File “file2.oop” imports the concepts “c1” and “c11” from “file1.oop”, “file3.oop” imports “c1” and “c12”. The concept “c2”, directly imported, and the concept “c1”, imported in the files “file2.oop” and “file2.oop”, will be available in “file4.oop”. As well as the concepts “c11” and “c12” from the “createConcept” function, called with different arguments in the files “file2.oop” and “file3.oop”. The concept “c11” will be available at the root of “file5.oop”, and the concepts “c11” and “c12” will be available in the “doSomething” function. The lists of available concepts in each block of code of Example 6.3.2 are shown in Figure 1.

In some cases, the dynamic creation of a set of concepts can lead to conflicts and inconsistencies in the knowledge base. The resolution of these conflicts is left to developers.

The proposed method for determining the scope of concepts is intended for the simplest case when the program is a set of procedures and functions distributed over different files. But the same principles can be applied to more complex cases, for example, object-oriented programming. In this case, the basic concepts can be placed in the root of the class, and alternative sets of additional concepts in its methods. This is only one of several possible ways to determine the scope of concepts. The best option depends on how the hybrid language is used, the balance between the components of computing and modeling, and between statically and dynamically defined concepts. At this stage, the search for this balance seems premature, it is better to postpone this task for the future.

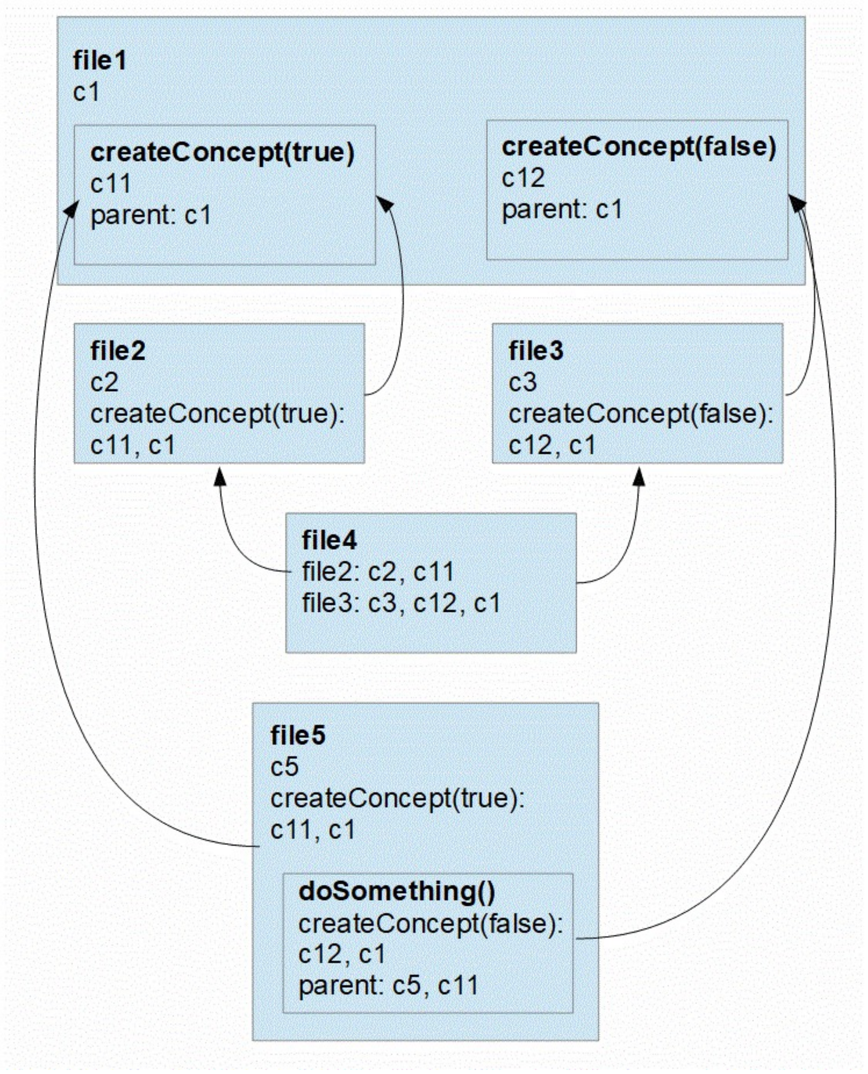


Fig. 1. Concepts available in each code block of Example 6.3.2

7 Basic principles of the implementation of the hybrid ontology-oriented programming language

In this section, we will try to consider the implementation of the basic mechanisms of the hybrid ontology-oriented programming language, which allow to combine the logic and imperative programming paradigms. The time has not yet come for a formal description of its operational semantics. We restrict ourselves to describing in free form only the basic principles of its internal implementation, enough to create a experimental version of the language.

7.1 Structures for storing program state

The most difficult task in the implementation of the language is to resolve the contradictions between the backtracking search engine of the component of modeling and the imperative nature of the component of computing. Variables and object fields of the components of computing can change their state during the calculation. But when performing backtracking search, alternative search branches should not affect each other, therefore, the change in the state of a variable in one branch should not be visible in other branches. And when returning to the original partial solution, the values of all variables and fields of the objects should also be restored.

The most famous solution to this problem are the "computation spaces" of OZ language [24]. Computation space is a container that stores the values of variables, object fields, functions, and constraints. Constraints are variables associated by an equality condition with values or with other variables. Each computation space is associated with its own thread of execution. The basic rules for working with computation spaces are as follows:

— Variables, constraints, and other elements may belong to only one of the computation spaces. Similarly, a thread belongs to only one of them.
— Threads can generate (explicitly or implicitly) new computation spaces that are nested towards the parent one. Thus, the entire set of computation spaces form a tree structure.
— Threads have full access to the elements of their computation space and can change only them. They can also read elements of all parent spaces. Elements of child spaces are completely inaccessible to them.
— The child computation space can either be dropped or merged with the parent. In the latter case, all elements from the child space are copied to the parent and the child is destroyed.
— When trying to add a constraint to the computation space, its compatibility with all other constraints, including parent space constraints, is automatically checked. If the constraint contradicts the existing ones, an exception will occur in the thread. The addition will be aborted and thus the repository of constraints will always be consistent.

Computation spaces play the same role as local data stores in the call stack. They contain data that should be available only in the current branch of the calculations and should be destroyed after it ends. Unlike the call stack, computation spaces have a tree structure and support the parallel execution of their branches, isolating them from each other.

Computation spaces organize the execution of nondeterministic constructions of OZ language, such as "`or`", "`cond`", or "`dis`" – disjunctive constructions of the nondeterministic choice. Disjunctive expressions are usually a set of sentences consisting of a condition "`Gi`" (guard) and a body "`Si`" (code block):

```
<operator>
G1 then S1
[] G2 then S2
...
[] GN then SN
end
```

The operator must execute the body of the sentence whose condition is true. In a simplified form, the process of executing such an operator can be described as follows. OZ suspends the execution of the current thread and creates new nested computation space for each sentence. Then for each nested computation space starts its own thread, which executes and checks the corresponding guard. If the execution of one of the conditions Gi succeeds, the logic variables, cells, and constraints of that guard's space are merged into the parent. And the parent thread will continue its work executing the body of the sentence Si. If the execution of all child spaces fails, then the execution of the parent space is considered unsuccessful. Since child threads can work in parallel and share logic variables and cells, nested computation spaces allow changing their values locally without affecting other threads.

Let's try to adapt the principles of computation spaces described above for backtracking search of the component of modeling. Constraints will be represented by values substitutions for logic variables and attributes. The principles of working with such constraints are very similar to those of the OZ language:

- Computation spaces store the substitutions of logic variables and concept attributes. All these elements may change during the backtracking search and calculations in alternative branches should not influence each other.
- Each step of the backtracking search generates new nested computation space for each alternative search branch.
- Substitutions of variable and attribute values can only be created in the current computation space. Elements of parent spaces are read-only; elements of child spaces are not available.
- After return to the original partial search solution, the child space is discarded, changes to the elements made in it are lost.
- If the child branch of the search succeeds, then its search results are merged into the current space. If there are several such branches, then the results of all of them are

combined. So, after the completion of all branches, parent space will receive the combined search results from all child branches.

Computation spaces are also useful for the component of computing. First, the structure of computation spaces can replace the function call stack for storing local data. Their structure is like that of the parent pointer tree, which is used in some programming languages to implement the call stack. Therefore, it also makes sense to move the local data of function calls to the structure of computation spaces:

— Functions calls and execution of nested blocks of code (in loop statements or conditions) also lead to the creation of the child computation space. And control is given to the function or the block of code. Only one child space is created.
— After return from the function or block of code, the child space is discarded, changes to the elements made in it are lost. After return from the function, the parent space gains access to the result of its execution.
— The behavior of computation spaces created by functions, code blocks, and logic search has its own features. The “`return`” statement stops the execution of a function, even if it is called in the computation space of the nested code block inside the function's body. “`Break`” and “`continue`” statements can be applied only to computation spaces created by loop or branch statements. And the computational spaces of backtracking search react differently to modifications of global variables, which will be described below.

Secondly, computation spaces allow to make changes to variables visible only for the current search branch. For this, the structure of computation spaces should be responsible for all operations on variables - creation, reading, and modification. Since the component of computing allows changing not only local, but also global variables, the logic of working with variables will differ from that of the OZ language:

— Computation spaces store bindings of variable names to values.
— Variable declaration adds the binding to the current computation space.
— Modification of the variable defined in the current space simply changes its value.
— Redefinition of a global variable creates in the current space a new binding of the existing variable name to the new value. Instead of the global variable, a local one with the same name is created.
— When the global variable changes, two options are possible depending on whether logic search is performed in the current calculation process. If it is not, then it is just enough to change the value of the variable in the computation space in which it was defined. If it is, then it is necessary to redefine the variable in the computation space that was created by the search engine. Thus, the changes are visible only to the current search branch and will be deleted after returning from it.
— The modification of the global variables can be implemented as follows. It is necessary to sequentially look through the computation spaces in the order of their nesting starting from the current one and find the first space in which this variable was defined, or the first space created by the search engine. And then in the found space to change the value of the global variable or create a local one.

- To read the value of the variable, it is also necessary to sequentially look through the chain of nested computation spaces in search of variable bindings. The first binding found will contain the current value of the variable.
- The search for variable bindings in the chain of nested computation spaces must be performed considering their lexical scope. Variables with the same name, declared in the code block surrounding the current function, and in the function that called it, are considered different variables:

  ```
  var x = 1;
  function foo() {
    var x = 2;
    bar();
  }
  function bar {
    print(x); // 1
  }
  ```

  The chain of computation spaces will consist of spaces created by the code in the root of the file, function "`foo`", and then "`bar`". But "`bar`" outputs 1, since the variable "`x`" in the function "`foo`" is local.
- After completion of the block of code or function, the current computation space is deleted and all changes to the variables made in it or in nested calls are lost. The same happens after return from the search branch.
- When working with composite data types, changes of the state of their component parts, such as list items, object fields, external variable closures, are also possible. These elements also need to be replaced in the structure of computation spaces. The values of these elements themselves can be of a composite type, stored on the heap and not have or lose the name binding in the structure of computation spaces. In order to redefine the values of these elements in the structure of computation spaces, they need unique identifiers. To do this, let's wrap each instance of a composite type in a special container and assign it a unique identifier. And its component elements can be identified by the identifier of the instance and the name or serial number of the element. When changing the value of the element, it is also needed to create a substitution for it in the structure of computation spaces according to the same rules as for variables. Reading their values also requires search for substitutions.
- In some cases, the locality of variable changes is undesirable. For example, for caching, logging, debugging, synchronization between calculation threads or search branches, etc. Therefore, it should also be a possibility to change the value of the variable in the space where it was declared and ignore the spaces created by search engine. To do this, let's introduce a special "`global`" modifier for variable declarations:

  ```
  global var x = 1;
  ```

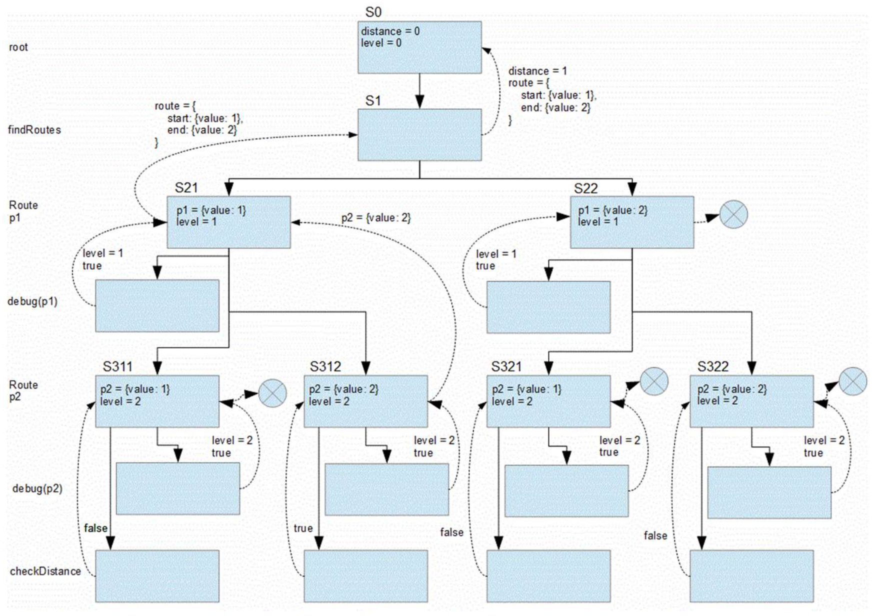


Fig. 2. An example of the structure of computation spaces

Let's discuss an example of a function call sequence and the corresponding structure of computation spaces:

Example 7.1.1.

```
fact startPoint {value: 1};
fact startPoint {value: 2};
fact endPoint {value: 1};
fact endPoint {value: 2};
concept route (start = p1, end = p2)
from startPoint p1, endPoint p2
where debug(p1)
  and debug(p2)
  and checkDistance(p1.value, p2.value);

var distance = 0;
var level = 0;
function findRoutes () {
  distance = 1;
  return find route {};
};

function checkDistance (v1, v2) {
  return v1 - v2 == distance;
};

function debug(parentConcept) {
  level++;
  println("Level: " + level + ", concept found: " +
           parentConcept);
  return true;
}
var routes = findRoutes();
```

In the example above, two instances of the facts "`startPoint`" and "`endPoint`", one concept of "`route`" and three functions are declared. The "`findRoutes`" function defines the relation between parent concepts "`startPoint`" and "`endPoint`" of "`route`" concept, "`debug`" outputs information about current step of inference, "`findRoutes`" starts the process of inference. Let's discuss the execution progress of this program. The states of its computation spaces are shown in Figure 2 and are also described below:

1. The program begins by creating the root computation space "`S0`", in which function declarations and the values of the variable "`distance = 0`" and "`level = 0`" are placed.
2. Call of the function "`findRoutes`" creates one nested space "`S1`". Modification of the variable "`distance`" to 1 will be done in the root space "`S0`" where it was declared, but not in the current space "`S1`".
3. Execution of the "`find`" statement starts backtracking search for the instances of the concept "`route`". It will begin by searching for possible instances of the first parent concept "`startPoint`", for each of them its own nested computation space ("`S21`" and "`S22`" respectively*)* will be created. In the first of them, the parent concept "`p1`" will be associated with the value "`{value: 1}`", in the second with "`{value: 2}`".
4. After binding "`p1`" to the value the first call of the function "`debug`" can be executed. Each call creates new computation space which will be destroyed after completing the call. It increments the variable "`level`" to 1 in the computation space created on previous step. In both spaces "`S21`" and "`S22`" its value is equals to 1. Value of "`level`" will be reverted after completing each branch.
5. The search for the next parent term "`endPoint`" continues in each of the spaces "`S21`" an "`S22`" independently. There are also two possible instances of "`endPoint`" concept. Accordingly, each branch of the calculation is split into two (into "`S311`", "`S312`", "`S321`", and "`S322`") with constraint "`p2`" equals to "`{value: 1}`" and "`{value: 2}`", respectively.
6. Binding "`p2`" to the value allows to call function "`debug`" once again with another input argument. Value of "`level`" will be incremented to 2 in spaces "`S311`", "`S312`", "`S321`", and "`S322`".

7. After the parent concepts “`p1`” and “`p2`” are associated with the values, it will be possible to call the “`checkDistance`” function and check the expression of the relations of the concept. The function will return the true value only for the combination of input parameters “`p1`” = 1 and “`p2`” = 2; the search will end successfully only for the branch “`S312`”. The constraint “`p2 = {value: 2}`” will be merged into the “`S21`” branch and used to generate search results: “`[route {start: {value: 1}, end: {value: 2}}]`”. The search results will be returned first to the space “`S1`” and then to “`S0`”.

7.2 Implementation of the scope of concepts and facts

Let's consider the principles of the implementation of the scope of concepts and facts and the main operations that require access to them. Since their scope is determined by the static structure of the code and import directives, computation spaces are not suitable for this purpose. Therefore, let's introduce a special repository called knowledge base.

The knowledge base has the form of a graph. The vertices of this graph correspond to functions and store a list of concepts and facts declared in them. Since the function can be called with different input arguments and create different sets of concepts and facts, it can be represented by several vertices of the knowledge base for each combination of values of its input arguments. Edges of the graph define the scope of concepts. All vertices of the function are always connected with a vertex that corresponds to its parent function or root of the file, where it was declared. In addition, the vertex is connected with those vertices which are referenced by import directives inside the body of its function. An example of a knowledge base graph can be found in Figure 1.

The main operations on the knowledge base are adding and removing concepts and facts, importing from other functions and search for them.

The **search** operation returns all concepts or facts from the current vertex and from all vertices associated with it by the visibility relations of the concepts. The rules for determining the scope of concepts and facts are described in detail in Section 6.3. According to them, the concepts of the current vertex, parent vertex and vertices specified in import operations are visible. Visibility relationships are transitive. If the vertex A has access to the concepts of the vertex B, and that in turn has access to the contents of the vertex C, then the contents of the latter are accessible to the vertex A. To search for concepts and facts, it is necessary to crawl the knowledge base graph in depth. The search operation algorithm looks as follows:

1. Declare a current search result as an empty array. Mark all vertices of the graph as not visited.
2. Take a list of concepts or facts of the current vertice, filter them according to the search query, merge the result with the current search result.
3. Mark the current vertice as visited.
4. Find the first vertice that is not yet visited and is connected with the current vertice by import operations in the opposite order to that in which import operations follow in the program code.

5. If the vertice was found create a child computation space for it, execute the code of its function and recursively execute the given algorithm for it. Then remove the computation space. Filter out concepts deleted in current vertice. Merge the result with the current search result. Return to Step 4.
6. If the parent vertex is still not visited, then perform the actions specified in Step 5 for it.
7. Return the current search result.

The input argument of the search operation is the search query. For a fact, the query includes the names and values of the attributes that it should possess. For a concept, it's only the attribute names. Also, query may include the name of the concept or the fact. The search operation is one of the steps of inference algorithm performed by the "`find`" command. It will be described in more detail in the next section. But one can also search for concepts directly with the "`find concepts {<query>}`" and "`find facts {<query>}`" commands.

Adding a concept or a fact changes the contents of the current vertex. This operation is performed by the "`concept`" and "`fact`" commands.

The **import** operation adds an edge between the current vertice and the vertice corresponding to the imported function. It is executed by the "`concepts from file`" and "`concepts from function`" commands. Also, each function call should change the structure of the knowledge base by creating a new vertice corresponding to the current combination of input arguments and connecting it with the parent function. Since the link to parent function is the same for all function calls it can be allocated in an additional special vertex that stores all shared information for all function calls. In this case, when calling the function, it is needed to perform the following actions:

1. Find a vertice that stores shared information about this function.
2. If there is no such one, create it and associate it with the parent function, if any.
3. Find the vertice of the function for the current combination of input argument values.
4. If there is no such one, then create it and connect it with the shared vertice from Step 1.
5. If there is one, clear the lists of its concepts and facts.

Removing concepts and facts from the knowledge base is also needed. It includes operations for removing individual concepts and facts, deleting import results, and completely clearing the knowledge base. Removing individual concepts and facts looks like this:

```
forget concepts <query>;
forget facts <query>;
```

One can also find concepts and facts in the knowledge base using the "`find concepts`" and "`find facts`" commands and then call the "`forget`" method of their definition objects. If the deleted entity is stored in the current vertice of the knowledge base, then it will be deleted directly. But if it was imported, then in the current vertice it will be marked as deleted and excluded from the search.

Removing import results can be done in the following way:

```
forget concepts from file <file name>;
forget concepts from function <function call>;
forget concepts from root;
forget facts from file <file name>;
forget facts from function <function call>;
forget facts from root;
```

As in the previous case, they either delete the edges for the current vertice or mark them as deleted in the case of nested imports.

One can completely clear the contents of the knowledge base using the commands:

```
forget concepts;
forget facts;
```

and clear the contents of only the current vertice:

```
forget local concepts;
forget local facts;
```

The main purpose of the knowledge base is to provide access to the hierarchy of declarations of concepts and facts. But it can also be responsible for caching search results and indexing concepts and facts to speed up the search. Consideration of strategies for efficient caching and indexing is beyond the scope of this work.

7.3 Inference algorithms

The "`find`" command starts inference process and derives instances of concepts from the knowledge base. Its input arguments are the name of the concept and the query object including the names of the arguments and their values. The result of its execution is a list of all instances of the concept, the attribute values of which correspond to the query. It is based on the inference procedure, which can be described in a recursive manner. Let's discuss it in more detail.

The algorithm of this procedure can be divided into two parts – external and internal. External one is responsible for the complete solution for the whole concept, internal is for the solution of its remaining goals. The argument to the external part is the search query. The input arguments to the internal part are:

- parent concepts for which it remains to find a solution,
- substitutions already found for logic variables and concept attributes.

The result of the execution of internal part is an expanded list of the substitutions.

The algorithm of the external part of the procedure includes steps to convert the request into a list of substitutions, call the internal part, check the sufficiency of the obtained substitutions for creating the concept instances and creation of them:

Algorithm 7.3.1. Inference of instances of a concept.

1. Convert the query object to the attribute substitutions.

2. Perform the internal part of the procedure by passing it the substitutions and the list of parent concepts.
3. Filter the results:
4. 3.1. Discard those sets of substitutions that are not enough to bind elements of the expression of relations to values.
5. 3.2. Discard those sets of substitutions that are not enough to bind the attributes of the child concept to values.
6. Create the instance of the child concept for each remaining set of substitutions.
7. Return the list of instances of the child concept.

The algorithm of the internal part is more complex. It includes an attempt to find solutions for the first parent concept from the list and a recursive search for solutions for the remaining parent concepts:

Algorithm 7.3.2. Inference of substitutions for concept attributes.

1. Create a new nested computation space and make it current.
2. Add input substitutions to the computation space.
3. Deriving of new substitutions:
   a. Check the added substitutions for compatibility with the expression of relations.
   b. If they are not compatible, delete the current computation space and return an empty list of search results. The end.
   c. Try deriving new substitutions from existing constraints. This issue will be discussed in more detail below.
   d. If the derivation of the new substitutions was successful, then add them to the computation space and repeat Step 3.
4. Take the first concept from the list of parent concepts. This will be the current goal of inference.
5. Prepare a query for the current goal. To do this, pick substitutions only for its attributes and convert it to the query format.
6. Create an empty list of results of the current goal.
7. Find all facts in the knowledge base whose name and attributes correspond to the query for the current goal. Add the result to the list of results of the current goal.
8. Find all concepts in the knowledge base whose name and attributes correspond to the query to the current goal.
9. For each concept found, perform inference:
   a. Recursively perform inference of the concept using the external part of the procedure (Algorithm 7.3.1) for the definition of the found concept. The input argument is the query to the current goal.
   b. Transform inference results into substitutions for attributes of the current goal.
   c. Add the found substitutions to the list of results of the current goal.
10. If the list of results of the current goal is empty, put a result indicating failed inference of the current goal to the list of results of the current goal.
11. Create an empty list of general results.
12. For each element of the list of results of the current goal, perform inference of the remaining goals:

   a. Create a new nested computation space and make it current.
   b. Add substitutions of the current result to the computation space.
   c. Perform the derivation of new substitutions as in Step 3.
   d. If the new substitutions do not comply with the existing constraints, discard the current result and delete the computation space.
   e. Otherwise, check the tail of the list of parent concepts for emptiness.
   f. If the tail is empty, add the current result to the list of general results.
   g. If not empty, perform inference of the remaining goals:
      (1) Recursively perform inference for the tail of the list of parent concepts using current algorithm. Use the current result as its input argument.
      (2) If the result of the inference is not empty, then for each of its values create a computation space and check it for compliance with the constraints as in Step 3
      (3) If it complies, add it to the list of general results; otherwise, discard it.
      (4) Delete the current computation space.

13. Return the list of general results.

It is also worth considering the issue of deriving new substitutions in Step 3 of the algorithm 7.3.2 of the internal part of the inference procedure. The substitutions can be represented in the form of equality constraints of attributes and variables to the corresponding values. Together with the expression of relations of the definition of the concept, they form a set of constraints of the current inference task. Let's borrow some ideas from the area of constraint programming that would be useful in the component of modeling, such as the technique of constraints propagation. In the algorithm described above, inference of the instances of the parent concepts is alternated with checking the expression of relations. Inference at each step connects the attributes of the corresponding parent concept with values. Constraints propagation uses new information about the value of a variable to analyze the constraints that are stated over it and try reducing a domain of possible values of other variables. These changes, in turn, can also be used to analyze the corresponding constraints and further reduce the domain of the solution and so on. Attributes substitutions found in this way allows to cut off inappropriate search branches and significantly optimize inference for the remaining parent concepts.

In the simplest case, it is possible to derive values of operands of equality relations, arithmetic, Boolean, and string operations. If the value of one of the arguments of the relation of equality is known, then the second argument can be associated with it too. If the result of the arithmetic operation and the value of one of the operands are known, then in many cases the value of the second operand can also be derived. For example, for the following expression of relations:

```
c1.row = c2.row AND c1.value = c2.value * 2
```

the known substitutions for the concept “`c1`” allow to immediately find substitutions for the “`row`” and “`value`” attributes of the concept “`c2`”:

```
c2.row ← c1.row
```

```
c2.value ← c1.value / 2
```

It is not a difficult task to implement such a simple derivation for domains of Boolean values, integer and rational numbers and strings. It will be very useful since such type of constraints is widely used.

It is also worth noting that if the attributes or variables binded to values are not enough to check some clauses of the expression of relations, then this check is postponed until the moment of binding. If at the end after inferring of all parent concepts there are still free attributes and pending checks of clauses, then the inference of the child concept is considered unsuccessful. Since the expression of relations is immutable, it is enough to store only substitutions in the computation spaces, not the entire system of current constraints. A more detailed description of the design of the constraints propagation system is beyond the scope of this work.

The recursive version of the algorithm is clear and easy to implement but not always applicable in practice. A large search depth, for example, in the case of recursive definitions of concepts, can lead to overflow of the call stack. The traditional solution is to transfer recursion from the hardware stack to the stack formed by traversing a decision search tree. Nodes of this tree are structure that store information about the solution search process for the corresponding search goal: a list of nested goals, the current nested goal, found substitutions of the attributes of the concepts. The nested goals describe tasks of the inference of parent concepts, each nested goal represents a branch of the search tree.

A non-recursive inference algorithm starts by creating a search tree node for a given concept and searching for the first nested goal. This search enumerates parent concepts and looks for facts and definitions of concepts for them. It retrieves the first parent concept, which has its own nested goals (i.e., the knowledge base contains the definitions of concepts for the given concept name and not just the facts). Then, control will be transferred to this nested goal; new tree node will be created for it, and the current one will be saved on the stack. If the current node has no more nested goals, then the previous node will be extracted from the stack. It will receive control and a list of found solutions of the current node. The algorithm stops when the current node has run out of nested goals and there is no parent node in the stack. A non-recursive version of inference of concept instances has the following form:

Algorithm 7.3.3. Non-recursive inference of instances of a concept.

```
1. Take the input concept as the current goal of the search, create new node of the
   search tree for it.
2. Create an empty stack of nodes.
3. Find the next nested goal of the current node according to algorithm 7.3.4.
4. While there is the next goal or the stack of nodes is not empty, continue:
   a. If the next goal exists:
      (1) Create a node of search tree for the next goal.
      (2) Push the current node onto the stack of nodes.
      (3) Make the next node current.
   b. If the next goal doesn't exist:
```

(1) Transform substitutions of attributes found by algorithm 7.3.4. into instances of the current concept.
(2) If the node stack is empty, return the found instances. The end.
(3) Pop a previous node from the stack.
(4) Submit the results of the current node to the previous one.
(5) Make the previous node current.

The most complex part of this algorithm is to find the next nested goal in Step 3. The nested goal is the inference task for that parental concept, which has its own parent concepts. In other words, for that parent concept, which is represented in the knowledge base by concept definitions and not by facts. If only facts are available for the parent concept, then it is just enough to transform their instances into attribute substitutions and move on to the next parent concept. The search for nested goals takes into account the fact that the parent concept can have several alternative definitions and iterates over all of them. In addition, if several alternative solutions were found for the current parent concept "Ci", then it is necessary to run inference of the next parent concept "Ci+1" for every of them. Therefore, before moving on to the next nested goal, algorithm must wait until the search for all the nested goals of the previous concept is completed and the inference results are submitted in Step 4.b.(4). Only after this it will be possible to take one of the solutions and derive constraints from it for the inference of the next parent concept. Thus, the algorithm has a two-level form: for each partial solution represented by valid options for attributes substitutions, it sequentially enumerates all the parent concepts, and for each concept it combines the results of the inference of all its alternative definitions. The algorithm of the search for the next nested goal looks as follows:

Algorithm 7.3.3. Search for a next nested goal.

1. Take the first parent concept as the current nested goal. Put the input query to the list of results of the first nested goal "R0". Take the first element of the list "R0" as the current result.
2. If the search for the next nested goal is started for the first time for the current parent concept:
   a. Create new computation space, add substitutions of the current result to it, derive new substitutions, and add them too.
   b. If the derivation fails, delete the current computation space, go to Step 6.b.
   c. Transform substitutions of the current result into a query for the current nested goal.
   d. Find all facts for the current nested goal according to the current query, add them to the list of results of the current nested goal "Ri".
   e. Find all concept definitions for the current nested goal.
   f. Put the result to a list of definitions of the current nested goal. If it is not empty, return the first element of it and the current query. The end.
3. If the search for the next nested goal has been already run for the current parent concept:
   a. Check if the end of the list of definitions of the current goal is reached.

   b. If not reached, then return the next element of it and the current query. The end.
4. Check if the end of the list of parent concepts is reached.
5. If not reached, then take the next parent concept as the current nested goal. Return to Step 2.
6. If reached, then:
   a. Add the results of the current nested goal to the list of inference results.
   b. Delete current computation space.
   c. Check if the end of the list of results of the current nested goal “Ri” is reached.
   d. If not reached, then take next value from the list of results of the current nested goal “Ri” as current result, return to step 2.a.
   e. If reached, then check if the current parent concept is the first.
   f. If not the first, make the previous parent concept and the list of results of the nested goal “Ri-1” current ones, return to step 6.c.
   g. If the first, then return an empty value. The end.

Completion of the algorithm in Steps 2.f and 3.b means that it has found the first or next definition of the parent concept and algorithm 7.3.3 needs to proceed to processing of the nested node. End of the algorithm in Step 6.g means that for all the remaining parent concepts no definitions left that require the creation of nested goals.

Algorithm 7.3.4 finds all the definitions of the first parent concept that require inference to obtain its instances. Its subsequent calls, starting from Step 2, find alternative definitions of the current parent concept or go to the next one. Also, after processing of all parent concepts it returns to alternative search branches for other results found for previous nested goal. The transition to the next parent concept is possible only after inference of the previous one is completed, since this requires its results. That's why the algorithm 7.3.3 in Step 4.b.(4) transfers the results of the inference of the previous concept to the current goal and starts the search for the next nested goal.

The non-recursive inference algorithm also processes each nested goal in its computation space to prevent their mutual influence. A new computation space is created when moving to the next substitution option in Step 2.a.

The algorithms described in this section are applicable to all types of concepts (except for the concept as function that explicitly implements the custom inference algorithm). The difference between the types of concepts lies in the way the constraints are generated and the concept instances are created from sets of substitutions. The proposed algorithms are designed to be executed in one thread, when the next nested goal is sequentially selected, processed and the result is returned to the previous goal. But it can also be modified to support parallel execution. Inference for all found solutions of the goal can be performed in parallel. It is one of the important future tasks of this work.

7.4 Details of the component of computing implementation

This section is devoted to the implementation of the main commands of the component of computing. It has specific characteristics, since the commands and expressions of the component of computing must interact with the elements of the component of modeling. First, they must deal with the structure of computation spaces instead of a stack of function calls. And second, they are part of the expression of relations of the concept definition. Therefore, expressions must support the residuation principle, which delays the evaluation of an expression until all its arguments are binded to values.

The purpose and logic of the main statements are shown in Table 1, the expressions – in Table 2.

Table 1. Main statements of the component of computing

Name	Purpose	Execution logic
Block of code `{commands}`	Combines several commands into a code block. It can be used as an element of the statements of branching, looping, function declarations, etc.	Creates computation space. Sequentially executes nested statements until the end of the list of nested statements is reached or execution is stopped by one of special commands. Deletes computation space.
Variable declaration `var variableName`	Creates new variable	Adds a new variable with the specified name to the current computation space. Binds it to *null*.
Assignment of a value to a variable `varName = expr`	Assigns a value to a variable. Can be combined with a variable declaration statement.	Calculates the value of the expression in the right side. Redeclares the variable in the computation spaces if it was changed in the process of inference. The rules for finding the computation space for the variable redefinition are described in Section 7.1. Binds the variable to the calculated value in the computation space where it was (re)declared.

Branching
`if (cond)`
`  thenCommand`
`else`
`  elseCommand`
	Executes one of the blocks of nested statements depending on a value of a condition.	Creates computation space. Calculates the value of a condition `cond`. If the value of the condition is true, then executes the `thenCommand` statement, otherwise the `elseCommand` one. Deletes computation space.

Loop with precondition
`while (cond)`
`body`
	Executes a nested statement until the condition value is true.	Creates computation space. Calculates the value of a condition `cond`. Until it is true repeats execution of the nested `body` statement and condition recalculation. Deletes computation space.

Loop with counter `for( init; cond; iter ) body`	A loop with an initialization statement and a statement executed at each iteration. Commonly used as a loop with an iteration counter.	Creates computation space. Executes an initialization statement `init`. Calculates a condition `cond`. Until it is true repeats the execution of a `body` statement, an iteration statement `iter`, and re-check of the condition. Deletes computation space.
Loop over list `for( iterator in list ) body`	Executes a loop body for each list item.	Creates computation space. Computes the value of a `list` expression, converts it to list type. Declares an `iterator` variable. For each list item, binds it to the `iterator` variable and executes the loop `body`. Deletes computation space.
Function declaration `function funcName( args, … ) body`	Creates a declaration of a function.	Creates a function definition object, associates it with the function name and adds them to the computation space.
Calculation of expression `expression`	Computes the value of an expression. Commonly used to call a procedure.	Computes the value of an `expression`. The result of the calculation is discarded.
Return of value `return expression`	Terminates a function and returns a specified value.	Computes the value of an `expression`. In the structure of computation spaces, finds the closest space that was created by function call. Puts the value of the expression into it. Stops the execution of the found computation space and all its nested spaces.

Table 2. Main expressions of the component of computing

Name	Purpose	Execution logic
Arithmetic operations `+, -, *, /, %`	Perform arithmetic operations of addition, subtraction, multiplication, division, remainder and unary negation	Calculate the values of arguments. Convert the type of the second argument to the type of the first one. Apply the specified operation to the arguments and return the result.

	over integers and real numbers.	
Boolean operations `&&, ||, !`	Perform Boolean operations AND, OR, NOT.	Calculate the values of arguments. Convert the type of the arguments to Boolean. Apply the specified operation to the arguments and return the result.
Strings concatenation `+`	Perform operation of concatenation of strings.	Calculates the values of arguments. Converts the type of the arguments to String. Concatenates the values of the arguments and returns the result.
Comparison `==, !=, <, >, >=, <=`	Perform comparison operations depending on the type of arguments.	Calculate the values of arguments. Convert the type of the second argument to the type of the first one. Apply the selected operation to the arguments and return the result of the Boolean type.
Variable `varName`	Returns the value of a variable.	Finds the value of the specified variable in the computation spaces and returns it.
Literals `1, 0.9, "a", true`	Returns the explicitly specified value of one of the basic types.	Return the specified value.
List `[1, 2, 3 ]`	Creates a list from the specified items.	Creates a new list, adds specified elements to it, and returns the list.
Object `{prop: "value"}`	Creates an object in the form of an associative array of specified key-value pairs.	Creates a new object, adds key-value pairs to it. Returns the object.
Fact `factName {attr: "value"}`	Creates a fact from the specified names and values of attributes.	Creates a new fact with the specified names and attributes values. Returns the fact.
Function calculation `funcName(args)`	Calculates the value of a function.	Finds a variable `funcName` in the computation spaces and converts it into a function declaration object or finds an explicit function declaration with the same name. Computes the values of the arguments. Creates

		new computation space and adds the function arguments to it as local variables. Executes statements of the function body, retrieves the result of the function execution from the computation space. Deletes computation space. Returns the result.
Function definition `function (args) body`	Creates an object of function definition. Commonly used to assign a function definition to a variable, an attribute or object field.	Creates an object of function definition and returns it.
Concept attribute `concept.attribute`	Returns current substitution of an attribute of a concept.	Finds a substitution of specified attribute of a concept in computation spaces and returns it.
Concept instance `conceptAlias`	Returns current substitution of a concept.	Finds a substitution of specified concept in computation spaces and returns it.
Logic variable `$varName`	Returns current substitution of a logic variable.	Finds a substitution of specified logic variable in computation spaces and returns it.

Expressions includes elements of the both components of computing (variables, functions) and modeling, (logic variables, attributes of concepts and their instances). A feature of the latter is that binding of its elements to values occurs gradually, after solutions for parent concepts are inferred. Checking conditions that include elements that are not yet bind to values should be postponed. Therefore, the inference engine needs information from expressions about whether they can calculate values for the current substitutions.

In the case of logic variables and attributes of concepts, it is enough to check whether the computation spaces have substitutions for them. And for the instances of concepts to check whether the inference has already been completed for the corresponding parent concept. Literals and variables of the component of computing are always associated with values. For all composite expressions, it is necessary to recursively check whether all their arguments are binded to values. The exception is the Boolean operations. For an OR operation, it's enough to have only one operand binded to true value, even if the other ones are free. And for the AND operation, with false value.

Nested concept definitions can be arguments to “`find`”, “`findOne`”, “`exist`” functions and they must also support the residuation principle. For this, it is necessary to divide the attributes and instances of concepts in the expressions of the nested concept into two groups depending on their belonging to an external or internal concept. Check whether attributes and concept instances are binded to values is applied only to the components of the external concept. For this, expressions must be able to return a list of attributes and instances of concepts that are included in their composition. The implementation is like checking for bindings. Literals and variables of the component of computing return an empty result. Attributes, concept entities, and logical variables return themselves. The rest of the expressions redirect the request to their arguments.

Implementation of the mechanism of constraints propagation require derivation of the arguments of the expressions if the result of their calculation is known. Therefore, support for inverse operations can be added to the expression implementation. In the simplest case, for most arithmetic, logical, equality and concatenation operations, it looks as follows:

— find free arguments of the expression,
— calculate the values of the rest,
— if possible, using the inverse operation, calculate the value of the free argument,
— recursively call the derivation of the arguments for a subexpression corresponding to this argument.

The recursive derivation ends with expressions of attributes, instances of concepts and logic variables. They simply return the value of the result of the expression as their new substitution.

8 Examples of application of the hybrid ontology-oriented programming language

As examples of an application of the hybrid ontology-oriented programming language, let's consider the creation of a financial model from data stored in a file and a model of a WEB page.

The source data of the first example are earnings and expenses by month. They are stored in a file in CSV format and look like follows:

```
"month", "earned", "spent"
"Jan", 12, 10
"Feb", 24, 26
"March", 21, 14
```

They can be used to calculate profit by months and total indicators:

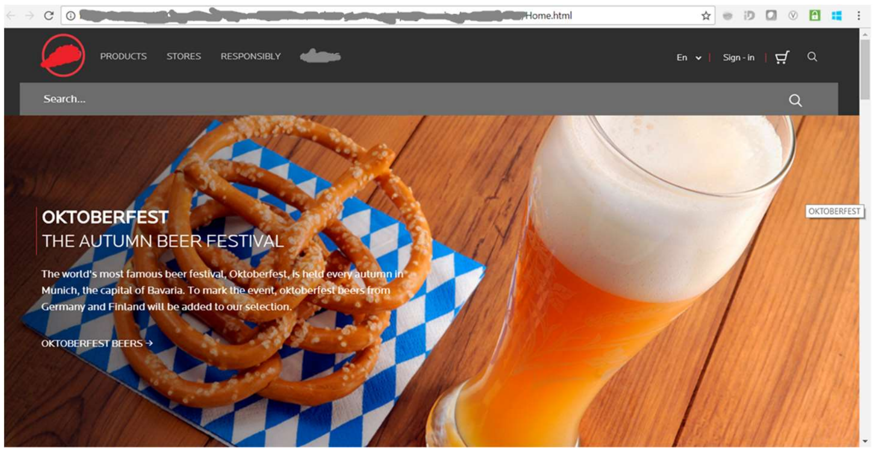
```
use tables;
concepts from function Table.readFromCSVFile(
 "examples/profitExample.csv",
 ","
);

concept month is tableCell where columnNum = 0;
concept earned is tableCell where columnNum = 1;
concept spent is tableCell where columnNum = 2;

concept profit is earned e, spent s
with value = e.value - s.value
where e.rowNum = s.rowNum;

concept results (
  month = m.value,
  earned = e.value,
  spent = s.value,
profit = p.value)
from month m, earned e, spent s, profit p
where m.rowNum = e.rowNum = s.rowNum = p.rowNum;

concept total (
  earned = sum(r.earned),
  spent = sum(r.spent),
  profit = sum(r.profit)
) from results r;

var total = findOne total {};
var output = "Total earned: " + total["earned"] +
", spent: " + total["spent"] + ", profit: " +
total["profit"];
print output;
```

The program starts by importing the results of the execution of the "`Table.read-FromCSVFile`" function, which reads the contents of the file and converts the table cells into facts. The concepts "`month`", "`earned`" and "`spent`" give names to cells depending on the column number, concept "`profit`" calculates the profit for each row of the table. The concept "`results`" combines in one record the values of all the concepts introduced above grouped by month, concept "`total`" calculates the total indicators. The component of computing allows to find the instances of the concepts, format the results and output them. This example demonstrates the benefits of each of the components of the language. The component of modeling allows to describe the data structure in a form of the concepts and use it to extract information from the source data, the component of computing is used to process the search results.

The second example demonstrates the ability to work with the contents of WEB pages. The source data are the contents of the home page and the product page of the e-commerce website. Views of these pages are shown in Figures 3 and 4. A program of this example performs search for products on the web-site's home page, collects information about the products found on the search results page and then prepares a report.

The first part of the program describes a search form by a set of concepts, finds it on the page, enters a text into it, sends a request to a server and loads the contents of a search results page into a knowledge base:

```
var url = "http://<web-site domain>/Home.html";
concepts from function HTML.openWebPage(url);

concept productLink is pageLink where text = "PRODUCTS";

concept storesLink is pageLink where text = "STORES";

concept otherMainMenuItems is PageLink e
where exist (productLink p where inTheSameRow (e, p))
  and exist (storesLink s where inTheSameRow (e, s));

concept searchLink is PageLink e
where e.title = "Search Magnifier Icon‟
  and exist (otherMainMenuItems m where e = m);

searchLink = findOne searchLink {};
HTML.click(searchLink);

concept searchForm (form = f, input = i, submit = s)
from pageForm f,
     pageInput i (form = f.id),
     pageInput s (type = "submit", form = f.id),
     withLabel l (labelText = "Search...", element = i);

var searchForm = findOne searchForm {};
HTML.enterText(searchForm["input"], "Sake");

concepts from function
HTML.followLink(searchForm["submit"]);
forget facts from function HTML.openWebPage(url);
```



The world's most famous beer festival, Oktoberfest, is held every autumn in Munich, the capital of Bavaria. To mark the event, oktoberfest beers from Germany and Finland will be added to our selection.

Fig. 3. Home page of an e-commerce website

In the second line, the program imports the result of executing the "`HTML.openWebPage`" function. It reads the contents of the webpage and converts its HTML elements into facts. The function also imports the definitions of a number of auxiliary concepts that define the spatial relationships between concepts, colors, captions, etc. The concepts "`productLink`" and "`storesLink`" give names to links to product and store pages in the main menu; their purpose is to and help to locate the main menu. The concept "`otherMainMenuItems`" states that all links on the same line as "`productLink`" and "`storesLink`" belong to the main menu. And the concept "`searchLink`" states that the desired link to the search form belongs to the main menu and has the specified caption. Linking "`searchLink`" to the main menu is necessary, since there are other search forms on the page that are currently hidden. After that, the program searches for the "`searchLink`" instance, and then clicks on it using "`HTML.click`" function to open the search form. The concept "`searchForm`" describes the contents of a search form, consisting of an input field and a button to send its contents to the server. The program finds its instance and enters the name of a product into it using "`HTML.enterText`" function. Next, the program using the "`HTML.followLink`" function sends the contents of the form to the server, reads the contents of the received page and loads it into the knowledge base. Then it removes the facts of the home page from the knowledge base. They are no longer needed.

The next stage is to introduce concepts for elements containing information about products, such as name, price, country of origin, etc.

```
concept productTile is pageDivision
where backgroundBasicColorName = "White‶
  and class = "mini-card";

concept productAvailability is PageSpan
with value = colorToAvailability(backgroundColorName)
```

```
where backgroundColorName in
  ["DarkOliveGreen", "Brown", "Crimson"];

function colorToAvailability(color) {
  if (color == "DarkOliveGreen") {
    return "green";
  };
  if (color == "Brown") {
    return "yellow";
  };
  if (color == "Crimson") {
    return "red";
  };
  return "unknown";
};

concept productPrice (
  value = (left.text + "." + right.text).toNumber(),
  leftPart = left,
  rightPart = right
) from pageSpan left, pageSpan right
where left.parent = right.parent
  and left.pos < right.pos
  and !left.text.empty()
  and !right.text.empty();

concept productName is pageDivision
where !text.empty()
  and basicColorName = "Black";

concept productCountry is pageDivision
where !text.empty()
  and basicColorName == "Gray";

concept productVolume is pageDivision
with value = substring(
  text,
  0,
  size(text) - 2
).toNumber()
where basicColorName = "Gray"
  and text.endsWith(" l");
```

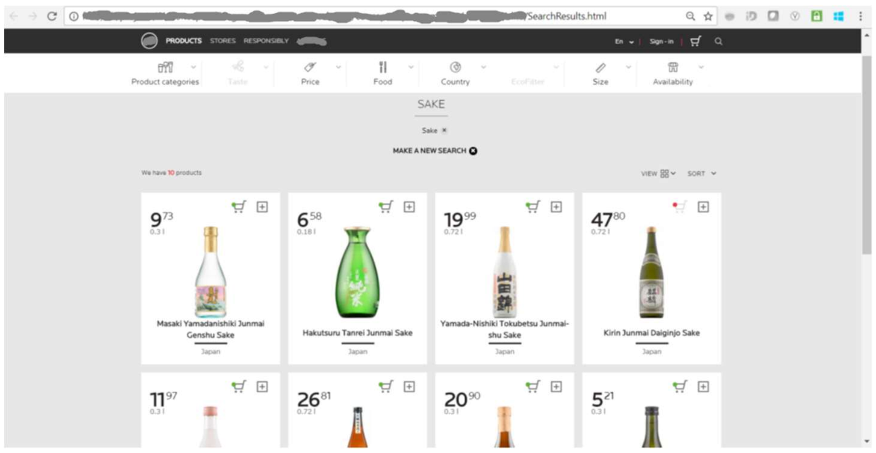


Fig. 4. Products page of an e-commerce website.

The concept “`productTile`” defines a container that contains all other elements of the product description. This is a white area with a fixed value of the class attribute. This attribute corresponds to the class attribute of HTML tags; its use improves the search efficiency of “`productTile`” instances. The concept “`productAvailability`” describes the availability of goods in a warehouse and is defined by the area of one of the three specified colors. The concept “`productPrice`” describes the price of a product and consists of two non-empty text elements having a common parent element. First of them contains the integer part of the price, the second one fractional. The product name “`productName`” is a black text element, and the country of origin “`productCountry`” is gray. Product volume “`productVolume`” is an element of gray color, the text of which ends with the designation of the measure of volume.

A feature of these concepts is that their descriptions are not complete and exhaustive. These descriptions include not only the necessary elements, but also many others. They can unambiguously distinguish product data only if all of them are united:

```
concept product (
  name = nameEl.text,
  price = priceEl.value,
  volume = volumeEl.value,
  country = countryEl.text,
  availability = availabilityEl.value
)
from
  productTile tileEl,
  productName nameEl,
  atTheBottomOf nameRel (
    inner = nameEl,
    outer = tileEl
  ),
  productPrice priceEl,
  atTheTopOf priceRel (
    inner = priceEl.leftPart,
    outer = tileEl
  ),
  productVolume volumeEl,
  atTheTopOf volumeRel (
    inner = volumeEl,
    outer = tileEl
  ),
  productCountry countryEl,
  atTheBottomOf countryRel (
    inner = countryEl,
    outer = tileEl
  ),
  productAvailability availabilityEl,
  atTheTopOf availabilityRel (
    inner = availabilityEl,
    outer = tileEl
  );
```

They can be combined by the concepts of spatial relations “`atTheBottomOf`” and “`atTheTopOf`”, which specify the location of one element inside another. The final concept “`product`” brings together all concepts with product data and limits their position within the product area “`productTile`”. These restrictions allow to discard irrelevant instances of the concepts and uniquely describe the products.

At the last stage, it remains only to find the instances of “`product`” concept, format the result and return it:

```
products = find product {};
output = "Products found:\n";
for(product in products) {
  output = output + product["name"] + ", " +
           product["volume"] + " l, " +
           product["price"] + " eur, " +
           product["country"] + ", " +
           product["availability"] + ".\n";
};
HTML.closeWebPage(url);
return output;
```

Traditional approaches to extract information from WEB pages usually rely on the analysis of their HTML code. This can be done using regular expressions. Or by building a document object model (DOM) and finding its elements using XPATH, CSS, or jQuery selectors. Such a solution is tightly bound to the implementation of the page; changes in the structure of HTML tags, class names, and element identifiers require changing the program that extracts the information.

The component of modeling is different in that it allows to describe the structure of a WEB page at an abstract semantic level, using concepts and relations that are natural to humans. For example, spatial relations between elements, hierarchy relations, membership in sets, positions in lists, colors, labels, etc. This form of the model of WEB pages has several advantages over traditional approaches to information extraction. Firstly, it is more resistant to changes to the source code of pages. It will remain valid if the changes do not affect the design of the page, the main properties of its elements and the relations between them. Secondly, the conceptual model is clearer and more understandable. It explicitly describes the structure of the page from the user point of view and is close to specifications of a software product. Thirdly, programs in a hybrid programming language are concise and simple. It is just enough to describe the concepts, call the search function and the result will be ready for further processing.

Because of these features, the hybrid programming language can become a good platform for software products in the field of information extraction from WEB pages, Robotic Process Automation, testing and monitoring of WEB applications. It combines tools for creating a flexible and intuitive page model and powerful and general-purpose tools for processing the search results of the model elements. Also, the hybrid programming language can be useful in other cases when it is necessary to build a complex model from semi-structured data:

- Rule-based information extraction from texts on natural language, such as reports, protocols, court decisions, and social media posts. Analysis of tonality of a text.
- Analysis of logs and sequences of actions. For example, analysis of the actions of visitors of a WEB site, highlighting patterns of their behavior, classifying users according to their behavior, fraud detection, finding the causes of errors, etc.
- Image analysis. Combining graphic primitives into composite objects. Binding text to image elements.

These tasks do not limit the scope of the hybrid ontology-oriented programming language. It will be useful in many cases when the domain model can be described in a declarative form:

- Building a unified model based on data from diverse sources: databases, WEB services, Semantic WEB, document-oriented databases without a clear structure, configuration files, etc.
- Creating application program interfaces for accessing data on request. The component of modeling is more flexible and universal in comparison, for example, with the GraphQL data description language.
- Business process modeling.
- Creation of complex multi-level analytical reports from raw source data. Model of domain in the form of a set of concepts is much clearer compared to complex multi-level SQL queries.
- Work with complex recursively defined data structures such as graphs and trees.

The hybrid ontology-oriented programming language can be used for ontologies development. It allows to embed the ontology description of the application, task or domain level directly into the project code, prepare its elements dynamically, mix them with custom objects, expand its capabilities by custom functions, use ontology and its inference results together with other program elements. Such a programming style would greatly simplify the work with ontologies in medical applications, ERP systems, etc.

9 Conclusions

The main result of this work is a **design** of a hybrid ontology-oriented programming language for semi-structured data processing. The language combines components of the declarative description of a model of domain and the imperative processing of its elements.

The model of domain takes the form of an ontology and consists of **facts** describing specific entities and abstract **concepts**. The facts correspond to objects directly obtained from the source data. They are described by a set of attributes associated with values. The concepts correspond to abstract objects constructed from the facts or other concepts. A definition of the concept includes a set of attributes, a list of parent concepts, and a set of relations linking the attributes of the parent and child concepts with each other. An inference engine can derive instances of the child concept from the parent ones using this definition. To do this, it sequentially finds the instances of the parent concepts, filters out those that do not satisfy the conditions of the relations, derives from them the values of the attributes of the child concept and creates its instances. The ontology is based on first-order logic, which allows to create flexible declarations of concepts including recursively defined ones.

The following principles for creating concepts are proposed:

The principle of **inheritance** allows to create a definition of a concept based on existing concepts by adding, deleting or redefining their attributes and expanding the expression of relations.

The concept of **relation** is a statement about several concepts related by given conditions. These concepts are not divided into child and parent ones and inference can be applied to any of them. The concept of relations is built over an ordinary concept, its attributes are binded to the instances of the parent concepts. It can also be used as a function, the input arguments of which are the instances of the parent concepts, and the output is the result of checking whether they satisfy the expression of relations.

The **concept as function** explicitly generates its instances by a custom algorithm without using built-in inference procedure. This allows to implement a custom search algorithm, download facts from external sources, implement endless sequences of facts, etc.

The mechanism of **nested concepts** creates a definition of a concept within the definition of another one. The nested definition is convenient when it makes sense only in the context of the external concept and its creation as a separate concept is redundant. Nested concepts are often used as arguments to "`find`" and "`exist`" inference functions. Also, nested concepts allow to transfer some of the conditions of relations from the external concept to the internal one and implement an analog of the external join operation of SQL language.

The **aggregation** mechanism allows to split a set of concept attributes bindings into unique groups and calculate aggregate functions for each of them. The results of aggregate functions calculation can be binded to the attributes of the child concept or checked in the expression of relations. Standard aggregate functions such as finding the quantity, amount, average, minimum and maximum values as well as arbitrary operations are supported.

The component of modeling adheres to the "open world assumption", therefore, it separates the operators of **Boolean negation** of attribute values "`not`", **derivability** of a concept instances from input data "`exist`", and the **existence** of the attribute of the concept "`defined`". These three separate operators are flexible, predictable, understandable, and easy to use. If necessary, their combination allows to implement the operation of negation as failure.

Elements of **higher-order logic** allow to dynamically specify the names of parent concepts and attributes depending on the found substitutions of attributes or logic variables. For this, the ability of interpreting the result of expression evaluation as the name of a concept or attribute is introduced. Higher order logic makes definitions of concepts more flexible, for example, it allows to create generic concepts over a whole class of concepts without being tied to their names. Also, concepts and facts can describe the structure of other concepts and facts what can be useful in extracting sentences on natural language, data schemes, meta-models from source data.

The proposed principles describe a conceptual model of a domain in a formal language suitable for performing queries to it. The style of concepts definition is object-oriented, what facilitates understanding of the data structure. The definition of the concept also focuses on the method of obtaining a child concept from parent ones. Due to this, the process of extracting information from the source data can be represented as a sequence of transformations of the original parent concepts into derived child ones, which makes the model simpler and more understandable. Such principles of creating definitions of concepts as inheritance, concept of relation, elements of higher order logic allow to create libraries of generic concepts suitable for reuse in different domains. The inference procedure is simple and unambiguous, the list of parent concepts determines the order of traversal of the decision search tree what gives clues for search optimization. This makes the component of modeling a convenient tool for describing the model of domain based on semi-structured data from diverse sources.

The **component of computing** includes such traditional elements as variables, branching and looping operators, functions, lists, and objects. Functions are first class citizens of the language. A type system of the experimental version of the language is dynamic with automatic type conversion in implicit situations. The syntax of the language is close to the syntax of JavaScript.

Principles of **integration** of the components of modeling and computing have been developed; they give advantages to both. The imperative style of programming can significantly facilitate the preparation of source data for the model, process the results of queries to it, implement those elements of the model that are easier to describe as a sequence of calculations than in the form of logic expressions, replace the built-in inference mechanism with a custom implementation if necessary. The declarative style is convenient for describing complex structures, it is more concise and closer to the natural language. Definitions of concepts and facts are first-class citizens of the component of computing. Objects of definitions of concepts and facts can be created dynamically, assigned to variables, passed to functions as arguments, etc. They can include variables, operators, and functions and other elements of the component of computing. The integration of these components allows not only to describe the conceptual model of domain, but also to prepare data for it, extract information from it and process the results within the single programming language. Also, elements of the component of modeling can replace some architectural patterns of the component of computing and make the application structure more dynamic.

The key ideas of **implementation** of the language are also considered. To structure a set of concepts, a **static scope** and **import directives** are proposed. They allow to split the code of concepts definitions into libraries, reuse them, and limit the mutual influence of concepts from different libraries. To ensure the cooperative work of stateful elements of the component of computing with the backtracking search mechanism, the idea of **"computation spaces"** has been adapted. It is a tree-like data structure whose nodes correspond to the stack of function calls and backtracking search steps. The nodes of the tree of computation spaces are designed to store local values of variables, fields of objects, functions, value substitutions for concept attributes and logic variables. Backtracking search procedure changes the values of variables and substitutions in the current space, not in the space where they were declared. When returning to the previous search step, the current computation space is discarded, all changes to the variables made on current step are lost. Computation spaces also isolate the search branches from each other and allow them to be executed in parallel.

A structure in the form of a graph called “knowledge base” is proposed to **store definitions** of concepts and facts. Its vertices correspond to function calls with various combinations of input parameters. Edges correspond to the static structure of nested function declarations and import directives of concepts and facts. The vertices store lists of concepts and facts declared in the corresponding function. The content of other vertices reachable from the current one is also available in it. The implementation of the basic operations on the knowledge base is described: adding, deleting and searching for concepts and facts, as well as their import.

Backtracking search algorithm is adapted for inferring concept instances. It includes finding solutions to the current parent concept, checking constraints and deriving substitution of concept attributes for each solution found, searching for solutions to the remaining parent concepts for each found combination of substitutions and creating instances of a child concept for each solution. The search for the solutions to the nested concepts is performed recursively. A non-recursive version has also been developed. It dynamically creates a decision tree and traverses it. These algorithms not only passively check whether the found substitutions of concept attributes satisfy the given constraints, but also using the constraints propagation technique derive new values of substitutions from them.

The logic of the execution of the **basic statements and expressions** of the component of computing is described in terms of their interaction with the component of modeling. The call stack and its structures for storing local variables are unified with computation spaces. Also, statements and expressions are endowed with some additional properties necessary for the inference algorithm. They include checking if all elements of the expression are binded to values. It allows the inference algorithm to postpone the constraints check until all attributes of concepts and logic variables included in it are binded to the values. Another such property is the derivation of substitutions of the attributes of concepts and logic variables, for the known result of the expression calculation. As well as the return of all the attributes of the concepts included in the expression, which is necessary for the implementation of nested concepts.

The proposed hybrid ontology-oriented programming language is suitable for solving problems requiring the creation of a complex multi-level conceptual model of domain from semi-structured or heterogeneous data. Such tasks may include integrating data from diverse sources, extracting information from WEB pages, documents, natural language texts, images, event logs, etc., creating multi-level analytical reports, automated testing, process automation, business process modelling, working with complex recursively defined data structures and creating applications based on ontologies.

The considered example of search for products on the e-commerce website demonstrates the advantages of integration of a conceptual model of a domain and tools of its processing. It allows to abstract from the low-level details of the implementation of a WEB page and describe it in terms of the visual properties of its elements and their relative positions. Such a description is concise, understandable, close to the specifications of the software product and is also resistant to changes in the page structure that do not affect its design.

The work on proof of concept of the hybrid language is carried out on GitHub platform, the project page can be found here: https://github.com/Voropay/ConceptualProgramming.

References

1. Buneman P.: Semistructured data. Symposium on Principles of Database Systems (1997)
2. Brachman R. J., Levesque H. J.: Knowledge Representation and Reasoning. Morgan Kaufmann (2004).
3. Hodgson J. P. E.: First Order Logic, Saint Joseph's University, Philadelphia (1995).
4. Clocksin W. F., Mellish C. S.: Programming in Prolog. Berlin; New York: Springer-Verlag (2003).
5. Goldman N.M: Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer. In: Fensel D., Sycara K., Mylopoulos J. (eds) The Semantic Web - ISWC 2003. ISWC 2003. Lecture Notes in Computer Science, vol 2870. Springer, Berlin, Heidelberg (2003).

6. Kalyanpur A., Pastor D., Battle S., Padget J., Automatic mapping of OWL ontologies into Java. In: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), pp. 98–103 (2004).
7. Lamy Jean-Baptiste: Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. Artificial Intelligence in Medicine, Volume 80, pp 11-28 (2017).
8. Clark K. L., McCabe F. G.: Ontology oriented programming in Go. Applied Intelligence 24, pp. 3–37 (2006).
9. Brambilla M., Cabot J., Wimmer M.: Model Driven Software Engineering in Practice. Morgan & Claypool, USA, Synthesis Lectures on Software Engineering #1. 182 pages (2012).
10. Antoy S., Hanus M.: Functional logic programming. Commun. ACM 53.4, pp.74-85 (2010).
11. Hanus M.: Multi-paradigm declarative languages. In: Proceedings of the International Conference on Logic Programming (ICLP 2007), pp 45–75, Springer LNCS 4670 (2007).
12. Casas A., Cabeza D., Hermenegildo M.V.: A Syntactic Approach to Combining Functional Notation, Lazy Evaluation, and Higher-Order in LP Systems. In: Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006), pp. 146–162. Springer LNCS 3945 (2006).
13. Somogyi Z., Henderson F., Conway T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. Journal of Logic Programming, Vol. 29, No. 1-3, pp. 17–64 (1996).
14. Lloyd J.: Programming in an Integrated Functional and Logic Language. Journal of Functional and Logic Programming, No. 3, pp. 1–49 (1999).
15. Hanus M.: A Unified Computation Model for Functional and Logic Programming. In: Proc. of the 24th ACM Symposium on Principles of Programming Languages, pp. 80–93, Paris (1997).
16. Lopez-Fraguas F., Sanchez-Hernandez J.: TOY: A Multiparadigm Declarative System. In: Proc. of RTA'99, pp. 244–247. Springer LNCS 1631 (1999).
17. Smolka G.: The Oz Programming Model. In: J. van Leeuwen, editor, Computer Science Today: Recent Trends and Developments, pp. 324–343. Springer LNCS 1000 (1995).
18. Garcıa de la Banda M.J., Demoen B., Marriott K., Stuckey P.J.: To the Gates of HAL: A HAL Tutorial. In: Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002), pp. 47–66. Springer LNCS 2441 (2002).
19. Ait-Kaci H., Lincoln P., and Nasr R.: Le Fun: Logic, equations, and Functions. In: Proc. 4th IEEE Internat. Symposium on Logic Programming, pp. 17–23, San Francisco (1987).
20. Ait-Kaci H.: An Overview of LIFE. In: J.W. Schmidt and A.A. Stogny, editors, Proc. Workshop on Next Generation Information System Technology, pp. 42–58. Springer LNCS 504 (1990).
21. Naish L.: Adding equations to NU-Prolog. In: Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, pp. 15–26. Springer LNCS 528 (1991).
22. Van Roy P.: Multiparadigm Programming in Mozart/Oz. Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers. Lecture Notes in Computer Science 3389, Springer (2005).
23. Henz M., Smolka G., Würtz J.: Object-oriented concurrent constraint programming in Oz. In: V. Saraswat and P. Van Hentenryck, editors, Principles and Practice of Constraint Programming, chapter 2, pp. 29-48. The MIT Press, Cambridge, MA (1995).
24. Van Roy P, Brand P., Duchier D., Haridi S., Henz M., Schulte C.: Logic programming in the context of multiparadigm programming: the Oz experience. In: TPLP 3(6), pp. 715-763 (2003).

25. De Cat B., Bogaerts B., Bruynooghe M., Janssens G., Denecker M.: Predicate Logic as a Modeling Language: The IDP System. In: M.K. Kifer, Y.A. Liu (Eds.), Declarative Logic Programming: Theory, Systems, and Applications, Chapt. 5, pp. 279-329, ACM Books (2018).
26. The IDP framework reference manual, KU Leuven Knowledge Representation and Reasoning research group, 2015, https://dtai.cs.kuleuven.be/krr/files/bib/manuals/idp3-manual.pdf, last accessed 2020/02/27.
27. Kifer M., Lausen G., Wu J.: Logical foundations of object-oriented and framebased languages. Journal of the ACM, 42, pp. 741–843 (1995).
28. Kifer M.: Rules and Ontologies in F-Logic. Reasoning Web, pp. 22-34 (2005).
29. Clark K.L.: Negation as Failure. In: Gallaire H., Minker J. (eds) Logic and Data Bases. Springer, Boston, MA (1978).
30. Reiter R.: On Closed World Data Bases. In: Gallaire, Hervé; Minker, Jack. Logic and Data Bases. Plenum Press. pp. 119–140 (1978).
31. Gelfond M., Lifschitz V.: The stable model semantics for logic programming. In: Proceedings of the Fifth International Conference on Logic Programming (ICLP), pp 1070–1080 (1988).
32. Van Gelder A., Ross K.A., Schlipf J.S.: The Well-Founded Semantics for General Logic Programs. Journal of the ACM 38(3) pp. 620—650 (1991).
33. Chen W., Kifer M., Warren D. S.: HiLog: A foundation for higher-order logic programming. Journal of Logic Programming. 15 (3), pp. 187–230 (1993).
34. Denecker M., Pelov N., Bruynooghe M.: Ultimate Well-founded and Stable Semantics for Logic Programs with Aggregates. In: International Conference Logic Programming, pp 212–226, Springer Verlag (2001).