GOAL,
A Graph-based Object and Association Language

Jan Hidders
Dept. of Math. and Comp. Science
Eindhoven University of Technology (TUE)
P.O.Box 513, 5600 MB Eindhoven
the Netherlands
e-mail: `hidders@win.tue.nl`

Jan Paredaens
Dept. of Math. and Comp. Science
University of Antwerp (UIA)
Universiteitsplein 1, B-2610 Antwerp
Belgium
e-mail: `pareda@wins.uia.ac.be`

Abstract

A graph-based model for describing schemes and instances of object databases together with a graphical data manipulation language based on pattern matching are introduced. The data model allows the explicit modeling of classes and relations which contain objects and associations, respectively. GOAL consists mainly of two operations, the addition and the deletion. These perform on every part of the instance where a certain pattern is found. We will present the syntax and the semantics of the language, and show its computational completeness.

1 Introduction

In traditional database models like the Entity Relationship Model [1], NIAM [2] and the Functional Data Model [3], labeled graphs are used to represent schemes and sometimes also instances. It is well known that graphically represented schemes are often easier to grasp than textual ones. Moreover, with the introduction of more complex data models like the semantic and, more recently, object-oriented data models [4,5,6,7], this has become even more important. If we want to offer the user a consistent graphical interface to a database then it is desirable that there is also a graphical manipulation language. Unfortunately, the manipulation languages of these models are usually either textual or graph-based but with limited expressive power.

One of the first graph based data models that offered a graph based manipulation language that was computationally complete was GOOD [8] (Graph Oriented Object Database Model). The data representation of GOOD is simply a graph with labeled nodes and labeled edges between them. In PaMaL [9] the data model of GOOD was extended by labeling nodes explicitly as either objects, tuples, sets or basic values. Furthermore, the manipulation language was adapted to fit the meaning of these nodes. This made it possible to conveniently model complex objects in PaMaL.

In the data model of GOAL that we will present here, the data model of GOOD is extended in a slightly different way. In GOAL we distinguish three kinds of nodes being the value nodes, the object nodes and the association nodes. The value nodes are used to represent the so-called printable values such as strings, integers or booleans. The object nodes are used to represent objects and the association nodes are used to represent associations. Both, objects and associations may have properties that are represented by edges. The participants in an association are also considered to belong to the properties of the association. The main difference between associations and objects is that the identity of objects is independent of their properties, whereas associations are considered identical if they have the same properties. Properties may be either functional i.e. have only one value, or multi-valued i.e. consist of a set of values. The type of a property is not constrained in advance such that the property of an object or an association may be either a basic value, an object or an association. This is in contrast with the Entity Relationship Model where, usually, associations are only allowed to hold between entities.

The language of GOAL consists mainly of two operations, the addition and the deletion. Both are based on pattern matching i.e. the finding of all occurrences of a prototypical piece of an instance graph. Wherever such an occurrence is found the addition specifies which nodes and edges to add, and the deletion specifies which nodes and edges to delete. Furthermore, a fixpoint operation is introduced to enable some form of iteration.

The organization of this paper is as follows. In section 2 we introduce the data model of GOAL and how its schemes and instances can be represented as graphs. In section

3 we present the operations of GOAL. In section 4 we will give some extra examples of GOAL programs and discuss the expressiveness of GOAL. Finally, in section 5, we will compare GOAL to other data models and graph manipulation languages such as PaMaL.

2 The Data Model

The basic concept of GOAL is the finite directed labeled graph. It is used to represent schemes and instances as well as the operations on them. Therefore, we will begin with the formal definition of this concept.

Definition 2.1 *A* finite directed labeled graph *with node labels L_n and edge labels L_e is $G = \langle N, E, \lambda \rangle$ with N a finite set of nodes, $E \subseteq N \times L_e \times N$ a finite set of labeled edges, and a labeling function $\lambda : (N \to L_n) \cup (E \to L_e)$ that maps nodes to node labels and edges to edge labels, where for all edges $\langle n_1, l, n_2 \rangle$ in E it holds that $\lambda(\langle n_1, l, n_2 \rangle) = l$.*

Notice that it is not possible in a finite directed labeled graph that there are two edges between two nodes with the same label.

2.1 Object Base Schemes

To introduce the notion of scheme, let us consider the employee administration of a company. Figure 1 shows a possible database scheme of such an administration.

Here, the rectangular nodes represent classes, the rectangular nodes with a diamond shape in it represent relations and the round nodes represent basic value types. The single and double headed arrows that are labeled represent the properties of the objects and associations. If a labeled arrow has a single head it means that the property is functional e.g. the single headed edge *name* indicates that every person has only one name. If an arrow has a double head it means that the property is multi-valued e.g. the double headed edge *sections* indicates that a department may consist of more than one section. The double arrows that are unlabeled indicate the **isa** relation e.g. the arrow between manager and employee indicates that every manager is an employee.

Let us now look at the scheme of Figure 1 in more detail. Here we see the class Person which contains persons that have the properties name, date of birth and address. A subclass of Person is Employee which contains persons that are currently employed by the company. A subclass of Employee is Manager which contains employees that are considered fit to run a section or even an entire department. The second subclass of Employee is Engineer which contains employees that have technical skills in a certain domain. The class Department contains the several departments of the company that have a name, a secretary and a manager assigned to them. A department consists,

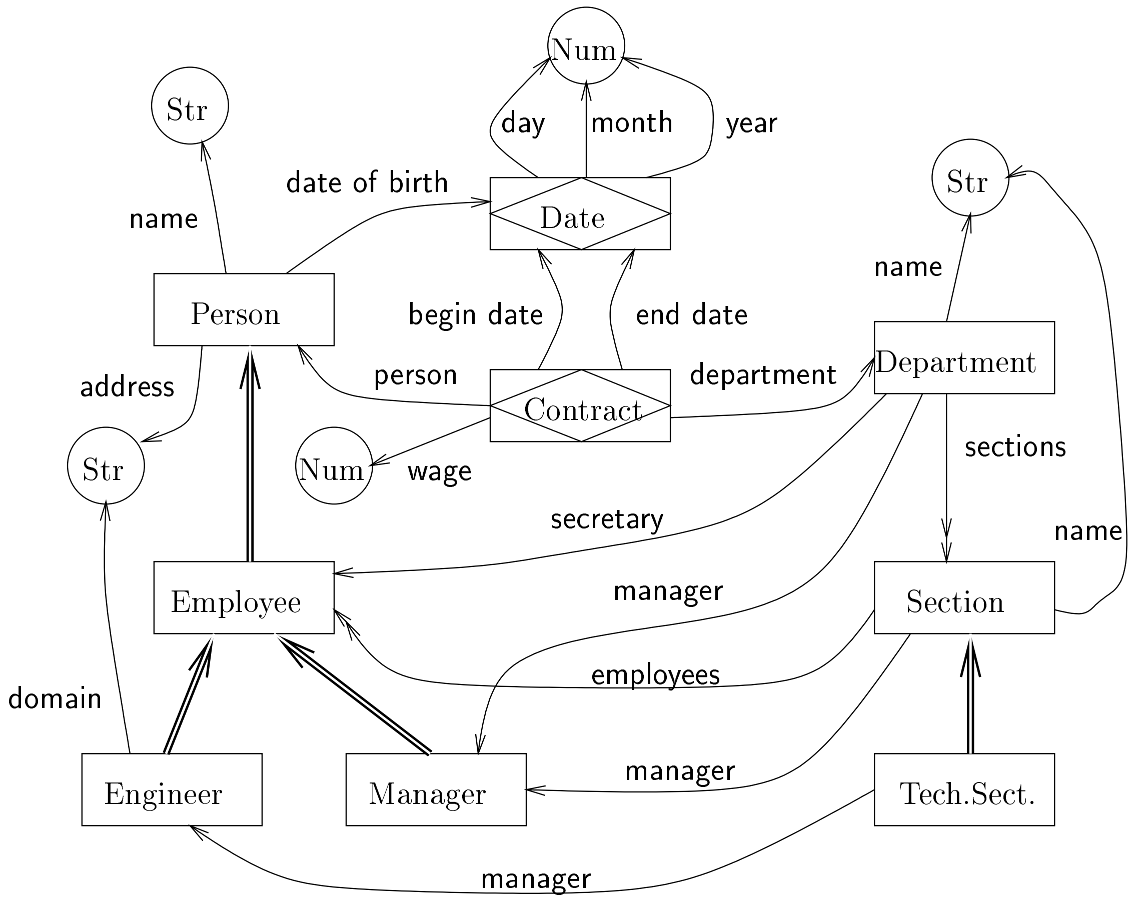

Figure 1: The employee administration database scheme

furthermore, of several sections. The class Section contains these sections that have a name and a manager and a set of employees who work with it. A subclass of Section is the class Technical Section which contains sections where specialized technical work is done, and which have to be run by an engineer. The relation Contract records the contracts between a person and a department. Properties of a contract are the wage, begin date and end date. Finally, there is the relation Date which contains dates represented by a day, a month and a year.

The fundamental difference between objects and associations is that the identity of an object is not dependent upon its properties i.e. two objects with exactly the same properties are not necessarily the same object whereas two associations with the same properties are identical. Thus, the relationships of the Entity Relationship Model and the tuples of object oriented data models can be modelled in GOAL with the same concept i.e. association. Examples of this are the Contract relation and the Date relation in Figure 1.

Before we turn to the formal definition of a scheme we have to define a database context that contains the preliminary concepts which are system-given.

Definition 2.2 *A* database context *is defined as $U = \langle B, V, \delta, C, R, F, M\rangle$ with B a finite set of basic types e.g.* `int`*,* `str`*,* `bool` *etc., V an enumerable set of basic values, $\delta : B \to 2^V$ maps every basic type to its domain i.e. an enumerable subset of the basic values, C an enumerable set of class names, R a countable set of relation names, F an enumerable set of functional edge labels and M an enumerable set of multi-valued edge labels. Furthermore, it must hold that B, C and R are pairwise disjoint and F, M and $\{$***isa***$\}$ are pairwise disjoint.*

The database context is considered to be fixed for all the following definitions. We are now ready to define what exactly constitutes a scheme.

Definition 2.3 *A* scheme *is a finite directed labeled graph $S = \langle N_S, E_S, \lambda_S\rangle$ with node labels $B \cup C \cup R$ and edge labels $F \cup M \cup \{$***isa***$\}$. Furthermore, it should hold that:*

- *all different nodes have different labels.*
- *edges may not leave from nodes labeled with a basic type.*
- *from a node there may not leave two edges with the same label except* **isa** *edges.*
- *the* **isa** *edges are only allowed between two nodes labeled with class names or two nodes labeled with relation names.*

The nodes labeled with a basic type are called *basic type nodes.* and are represented by round nodes. The nodes labeled with a class name are called *class nodes* and are represented by rectangles. The nodes labeled with a relation name are called *relation nodes* and are represented by rectangles filled with a diamond. Edges labeled with a functional edge label or multi-valued edge label are called *functional edges* or *multi-valued edges* and are represented by single headed arrows or double headed arrows, respectively. Edges labeled with **isa** are called **isa** edges and are represented by double unlabeled bold arrows. Notice that **isa** edges are not only allowed between classes but also between relations.

If we look again at Figure 1 it can be verified that it presents a valid scheme except that it has several basic type nodes labeled with the same basic type. To be formally a scheme, these nodes would have to be merged but, informally, we allow the duplication of basic type nodes to increase readability.

The **isa** edges in Figure 1 indicate that every employee is a person, and that every manager is an employee. Evidently, it follows that every manager is also a person. Thus, the subtype relation can be easily derived from the **isa** edges.

Definition 2.4 *The* subtype relation *$\preceq_S$ for a scheme S is a subset of $(B \times B) \cup (C \times C) \cup (R \times R)$ such that $l_1 \preceq_S l_2$ iff there are two nodes n_1 and n_2 in S with the labels l_1 and l_2, respectively, and a, possibly empty, directed path of* **isa** *edges from n_1 to n_2.*

Notice that the subtype relation holds between class names, relation names and basic types, and not between the nodes of the scheme. Also notice that since **isa** edges are not allowed between basic type nodes, the only subtype of a basic type is the basic type itself.

2.2 Database Instances

To introduce the notion of instance we present a small example of a weak instance of the employee database in Figure 2.

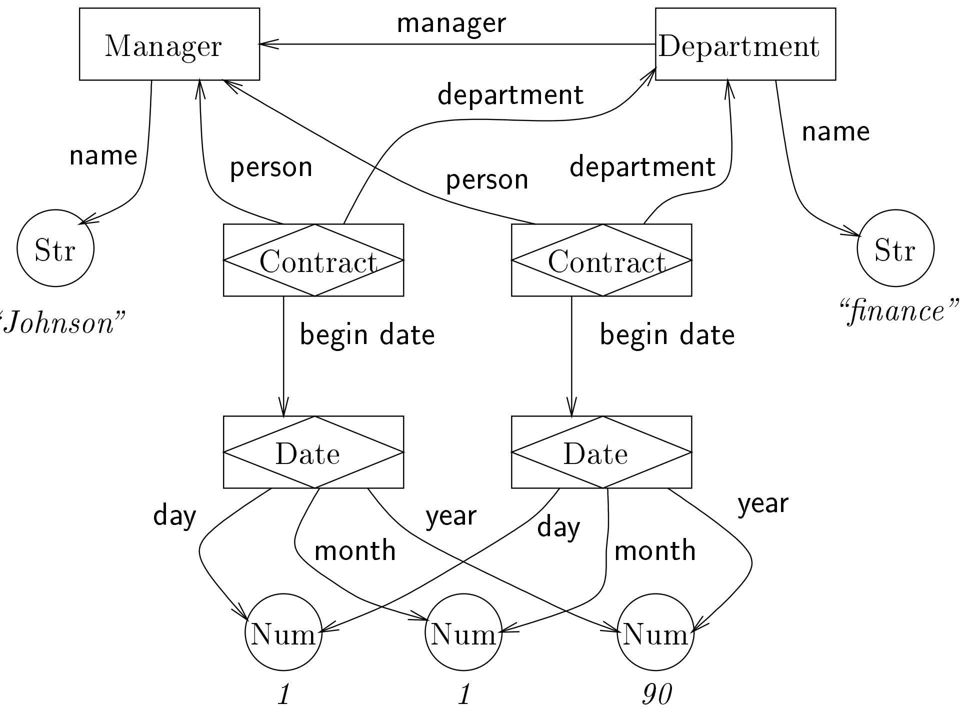

Figure 2: A weak instance of the employee database

The interpretation of the graph is reminiscent of the interpretation of a scheme graph. The rectangle nodes represent objects, the rectangle nodes with a diamond inside represent associations and the round nodes represent basic values.

Here we see a manager with the name "Johnson" who is the manager of the finance department. Apparently, Johnson has two contracts with his department, both with begin dates that have the same day, month and year.

Definition 2.5 *A* weak instance *is a pair $I = \langle\langle N_I, E_I, \lambda_I\rangle, \nu_I\rangle$ where $\langle N_I, E_I, \lambda_I\rangle$ is a finite directed labeled graph with node labels $B \cup C \cup R$ and edge labels $F \cup M$. The partial function $\nu_I : N_I \to V$ maps nodes labeled with a basic type to their basic value. Furthermore, it should hold that:*

- *for every functional edge label there may at most leave one edge with that label from every node.*
- *exactly all the nodes labeled with a basic type are labeled with a basic value.*
- *the basic value of a node labeled with a basic type must belong to the domain of this basic type.*

The nodes labeled with a basic type are called *value nodes.* The nodes labeled with a class name are called *object nodes.* The nodes labeled with a relation name are called *association nodes.* The λ-label and the ν-label of a node are said to be the *type label* and the *value label* of a node, respectively. Edges labeled with a functional edge label or a multi-valued edge label are called *functional edges* or *multi-valued edges*, respectively.

The graphical notation of weak instances is identical to those of schemes. It can be easily verified that the graph presented in Figure 2 is a valid weak instance.

We now turn to the question when a weak instance belongs to a certain scheme. Firstly, it may be clear that the weak instance may not contain objects and associations from classes and relations that were not mentioned in the scheme. Secondly, in GOAL we consider all properties to be optional e.g. the property date of birth of the class Person does not have to be defined for all persons. This enables us to conveniently model incomplete information. On the other hand, an object or an association may only have those properties defined that are prescribed for its type or for a supertype. Finally, the type of a property must be a subtype of all the types that are prescribed in the scheme by the type and the supertypes. For instance, in Figure 1 we see that the manager of a section must be a manager. It can also be seen that the manager of a technical section must be an engineer. Therefore the manager of a technical section must both be a manager and an engineer.

These intuitions about the typing of a weak instance can be formalized in the following way.

Definition 2.6 *A* weak instance I is of scheme S *whenever:*

- *the type labels in the weak instance occur in the scheme.*
- *for every weak instance edge $\langle n_1, l, n_2\rangle$ there is a scheme edge $\langle n'_1, l, n'_2\rangle$ such that $\lambda_I(n_1) \preceq_S \lambda_S(n'_1)$.*
- *for every weak instance edge $\langle n_1, l, n_2\rangle$ and scheme edge $\langle n'_1, l, n'_2\rangle$ it must hold that if $\lambda_I(n_1) \preceq_S \lambda_S(n'_1)$ then $\lambda_I(n_2) \preceq_S \lambda_S(n'_2)$.*

From the intuition of the typing of weak instances it may already be clear that the weak instance presented in Figure 2 is a weak instance of the scheme in Figure 1.

If we look at the weak instance of Figure 2 we see two Date associations with the same properties. Because the identity of an association depends fully upon the properties it has, these two associations should be merged. But then the properties of the two contracts would also become identical. Therefore, they would have to be merged also. Informally speaking, two nodes should be merged if they represent the same value.

Definition 2.7 *Two nodes n_1 and n_2 are said to be* value equivalent *or $n_1 \cong_I n_2$ iff for all $i \in I\!N$ it holds that $n_1 \cong_I^i n_2$ where $\cong_I^i \subseteq N_I \times N_I$ is defined as:*

- *$n_1 \cong_I^0 n_2$ iff n_1 and n_2 are association nodes with the same type label or if they are the same class node or if they are value nodes with the same type label and the same value label.*
- *$n_1 \cong_I^{i+1} n_2$ iff $n_1 \cong_I^i n_2$ and for every edge $\langle n_1, l, n_1' \rangle$ there is an edge $\langle n_2, l, n_2' \rangle$ such that $n_1' \cong^i n_2'$ and vice versa.*

Notice that if two nodes n_1 and n_2 are labeled with the same relation name, it holds that $n_1 \cong_I^0 n_2$. Furthermore, it can be observed that two value nodes are value equivalent iff they have the same type and value label, and two object nodes are value equivalent iff they are the same node.

Since we did not forbid associations to directly or indirectly refer to themselves, it is possible to represent certain infinite values in a weak instance. Consider, for example, the weak instance in Figure 3.

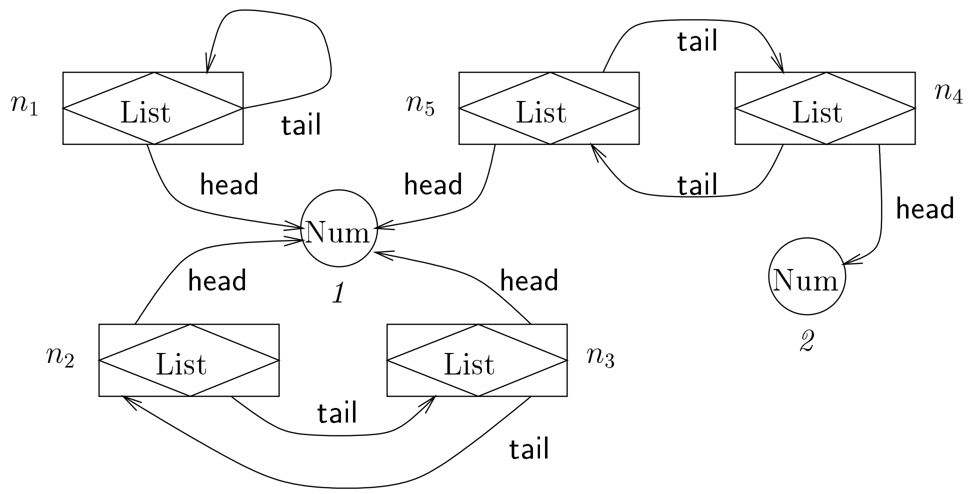

Figure 3: An instance with infinite values

Here, the nodes n_1, n_2 and n_3 are association nodes that represent the same infinite list of 1's. They are therefore value equivalent. The nodes n_4 and n_5 both represent infinite lists with alternating 1's and 2's. They are not value equivalent because n_4 represents the list starting with a 2 and n_5 the one starting with 1.

We are now ready to define instances i.e. weak instances that respect the notion of associations without identity.

Definition 2.8 *An* instance *is a weak instance I for which it holds that all different nodes are not value equivalent.*

Definition 2.9 *For every weak instance there is an instance that can be obtained by merging nodes that are value equivalent. This instance is called the* reduction *of a weak instance.*

For example, if in Figure 2 the two contracts, the two dates and the two value nodes with value label 1 would be merged, then we would obtain an instance.

2.3 Inheritance Conflicts

The scheme in Figure 1 has got a small problem. The manager of a section has to be a manager, but the manager of a technical section also has to be an engineer. In the GOAL data model, objects only belong to the class they are labeled with and its superclasses. This implies that an object can only belong to two classes at once if these classes have a common subclass. Since the classes Manager and Engineer do not have a common subclass there can not exist a manager that is an engineer as well. Formally, this does not present a problem since the manager of a technical section may always be left undefined, thus avoiding the contradiction. Of course, this is not what was intended with the scheme. Therefore, we introduce the notion of consistent scheme that prevents these problems.

Definition 2.10 *A* consistent scheme *is a scheme S where for every two edges $\langle n_1, l, n_2\rangle$ and $\langle n_1', l, n_2'\rangle$ in S for which $\lambda_S(n_1') \preceq_S \lambda_S(n_1)$ it holds that there is a node n_3 in S such that $\lambda_S(n_3) \preceq_S \lambda_S(n_2)$ and $\lambda_S(n_3) \preceq_S \lambda_S(n_2')$.*

Notice that the scheme of Figure 1 can be made consistent by adding a class Technical Manager that inherits from both Engineer and Manager.

3 The Language

In this section we will present the operations of GOAL. These operations will all be defined over a fixed scheme S i.e. all begin, intermediate and final instances must be of this scheme.

3.1 Pattern Matching

The language of GOAL contains two operations: the addition (that adds new nodes and/or edges to the running instance) and the deletion (that deletes some nodes and/or edges from the running instance). Both operations follow the same principle: everywhere some "pattern" is found in the running instance, the operation is applied. A pattern has the form of a weak instance where not necessarily all value nodes have a value label.

Definition 3.1 *A* pattern *is a weak instance but with the value function ν allowed to be undefined for some value nodes.*

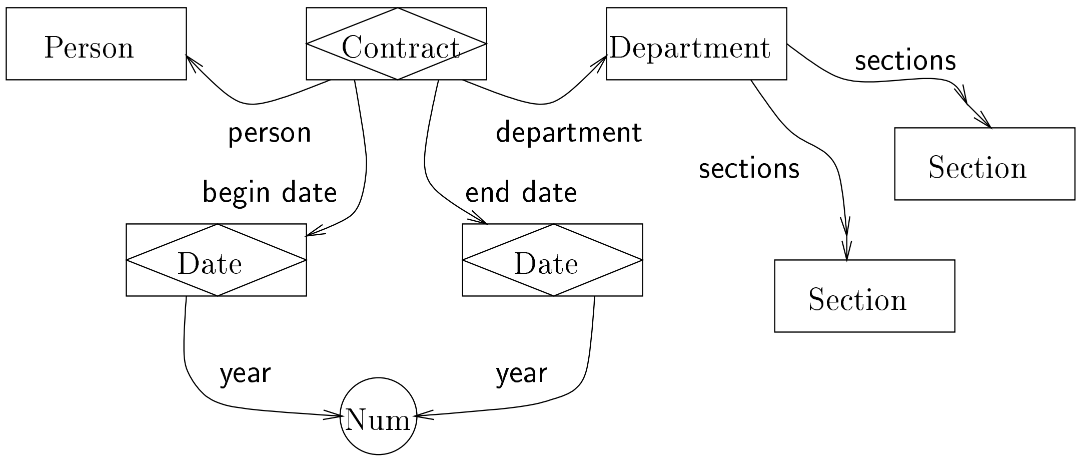

Figure 4: An example of a pattern

Figure 4 shows a pattern over the scheme of Figure 1, that represents all the persons that have a contract that is signed with a department with at least two sections and that has different begin and end dates with identical year.

Clearly a pattern can match with several parts of an instance. Each such a matching is called an embedding of the pattern into the instance.

Definition 3.2 *An* embedding of a pattern J into a weak instance I *is a total injective function $\phi : N_J \rightarrow N_I$ such that:*

- *pattern nodes are mapped to weak instance nodes the type labels of which are subtypes.*
- *pattern nodes with value labels are mapped to weak instance nodes with the same value label.*
- *if the edge $\langle n_1, l, n_2 \rangle$ is in J then $\langle \phi(n_1), l, \phi(n_2) \rangle$ must be in I.*

The set of all embeddings of J into I is denoted as $Emb(J, I)$.

Note that distinct nodes in the pattern are mapped to distinct nodes in the instance. Furthermore a node in the pattern is mapped to a node in the instance with the same type or with a subtype. Recalling Figure 1, an Employee node in the pattern can be mapped, for example, to a Manager node in the instance, indicating that the Manager node is considered here as of type Employee.

3.2 Additions

An addition is used to add new nodes and/or edges, for every embedding of a given pattern that is found in the running instance. The addition is represented by that pattern, J_m augmented with the bold nodes and/or edges that have to be added. These bold nodes and edges together with J_m have also to form a pattern, that is called J_a.

Definition 3.3 *An* addition *is a pair $\mathcal{A}\langle J_m, J_a\rangle$ with patterns J_m and J_a where J_m is a subpattern of J_a and both patterns are of the scheme that the language is defined over.*

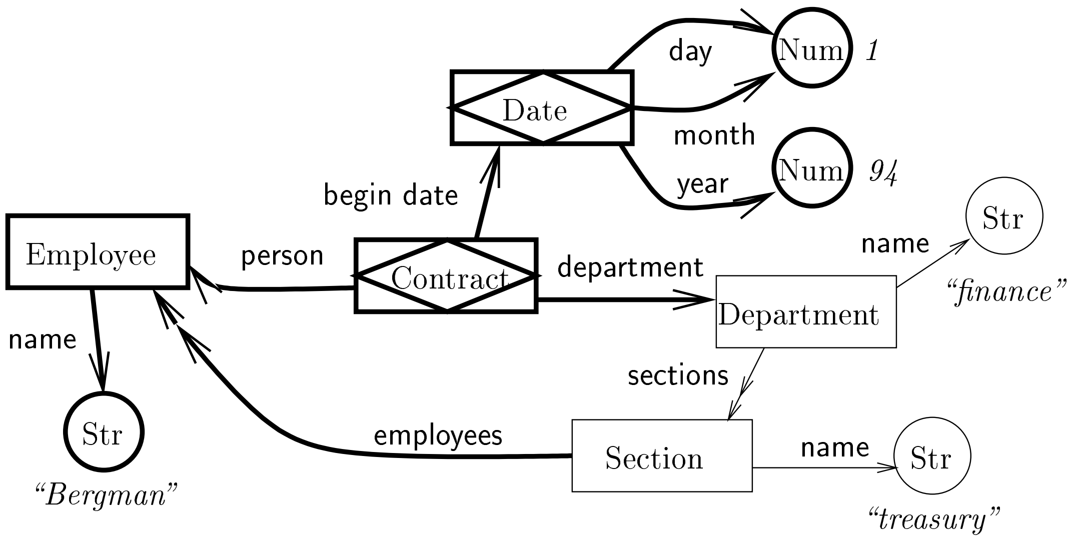

Figure 5: An example of an addition

Figure 5 represents an addition where a new employee Bergman and a contract that starts on 1-1-94 between Bergman and the finance department is added. Bergman also becomes an employee of the treasury section.

Actually, for every bold node (edge) a new node (edge) is created. But after the creation of the new nodes and/or edges, we obtain an instance that can be weak. Therefore the semantics of the addition ends with a reduction.

Definition 3.4 *The* semantics *of an addition $\mathcal{A}\langle J_m, J_a\rangle$ applied to an instance I is defined as the reduction of I' the minimal weak superinstance of I such that every embedding $\phi \in Emb(J_m, I)$ has an extension $\phi' \in Emb(J_a, I')$ with:*

- *ϕ' is equal to ϕ on the nodes of J_m.*
- *each pattern node of $J_a - J_m$ is mapped by each extension to a different node, which does not belong to I.*

Remark, that the addition of Figure 5 creates a new association node for the new contract. A new association node for the date 1-1-94 is also created, but it merges with the existing date 1-1-94, if the latter exists in the running instance. A new object node for Bergman is created of type Employee. If such a node would already exist this node will not merge with it. The basic values such as "Bergman", 1 and 94 are also drawn in bold because otherwise the pattern would not match if these values did not already exist in the object base and therefore nothing would be added.

Clearly the semantics is uniquely defined up to an isomorphism on the new object and association nodes. Notice that the semantics may sometimes be undefined e.g. when a person receives a new name without removing the old one. The result would then be a person with two names but since the property name is functional this is not possible in any instance.

3.3 Deletions

A deletion is used to delete existing nodes and/or edges, for every embedding of a given pattern that is found in the running instance. The deletion is represented by that pattern, J_m, where the nodes and/or edges that have to be deleted are drawn in dashed lines. The nodes and edges that are not dashed have to form a pattern, called J_d.

Definition 3.5 *A* deletion *is a pair $\mathcal{D}\langle J_m, J_d\rangle$ with patterns J_m and J_d where J_d is a subpattern of J_m and both patterns are of the scheme that the language is defined over.*

Figure 6 represents the deletion of the contracts that end in 1993 and removes the contracted employees from the sections that belong to the department the contract was with.

Actually, the deletion starts with the removal of the indicated nodes and/or edges. But after this removal we obtain an instance that can be weak. Therefore the semantics of the deletion ends with a reduction.

Definition 3.6 *The* semantics *of a deletion $\mathcal{D}\langle J_m, J_d\rangle$ applied to an instance I is defined as the reduction of I' the maximal weak subinstance of I for which it holds that for all embeddings $\phi \in Emb(J_m, I)$:*

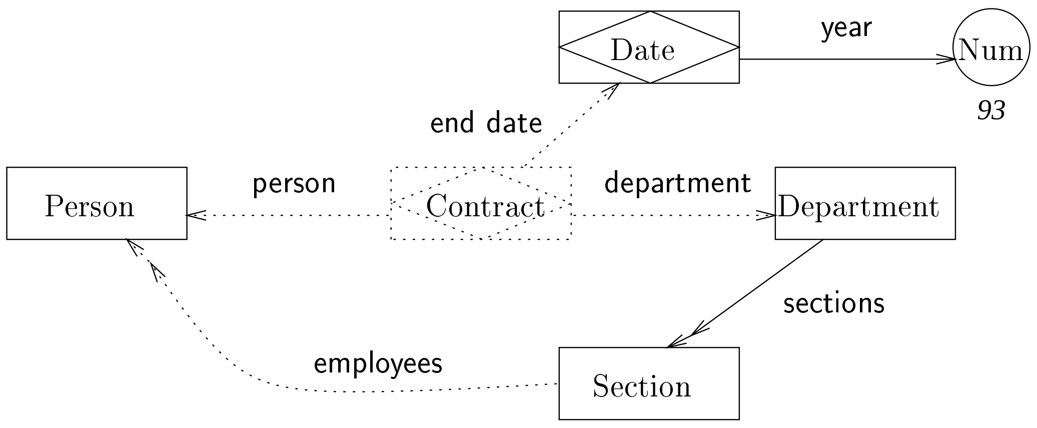

Figure 6: An example of a deletion

- *if a node n in J_m is not in J_d then $\phi(n)$ is not in I'.*
- *if an edge $\langle n_1, l, n_2 \rangle$ in J_m is not in J_d then $\langle \phi(n_1), l, \phi(n_2) \rangle$ is not in I'.*

Clearly the semantics is uniquely defined up to an isomorphism on the association nodes that result from a merging during the reduction.

3.4 Recursion

A transformation is a finite list of additions, deletions (and fixpoints). In order to handle the recursion we define a fixpoint of a transformation.

Definition 3.7 *A* fixpoint *is defined as $\mathcal{F}\langle\mathcal{T}\rangle$, where $\mathcal{T}$ is a transformation, i.e. a finite list of additions, deletions and fixpoints.*

The result of a fixpoint is obtained by first iterating the list of transformations on the running instance until a fixpoint is reached. If no fixpoint is reached the semantics is not defined.

Definition 3.8 *The* semantics *of a fixpoint $\mathcal{F}\langle\mathcal{T}\rangle$ applied to a strong instance I is defined as the first instance I_j in the infinite list $I_0, I_1, I_2, \ldots$ for which it holds that I_j is equivalent to I_{j+1} up to isomorphism on the value and association nodes and where:*

- $I_0 = I$
- *I_{i+1} is the result of consecutively applying the elements of $\mathcal{T}$ to I_i*
- *if a node n in I_i is deleted in I_{i+1} then it may not return in I_k where $k > i$.*

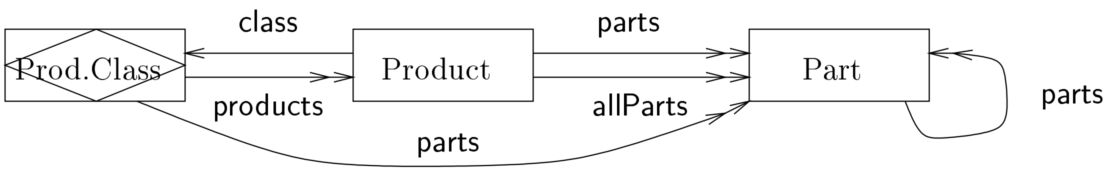

Figure 7: A scheme for products and parts

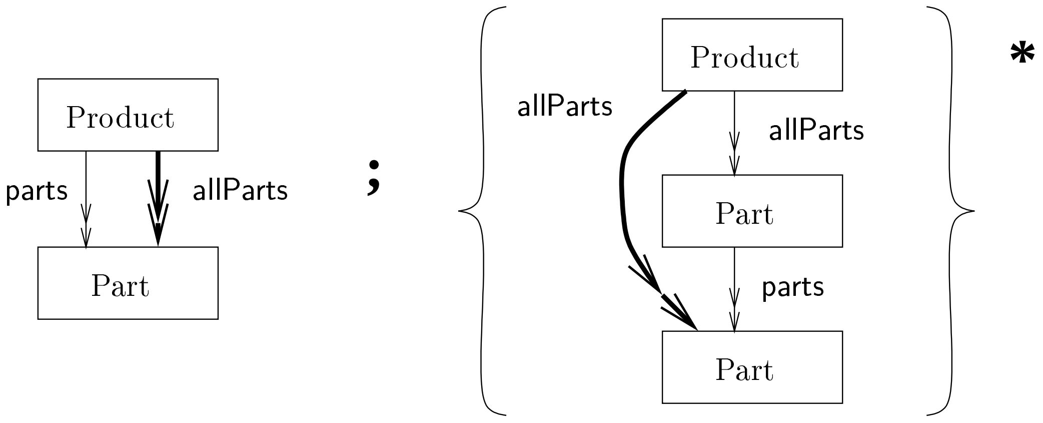

Figure 8: An example of the fixpoint operator

Figure 7 shows the scheme of a database that represents products that are built from parts, that in their turn are built from parts, and so on. In this scheme the edge labeled allParts represents all the parts needed to build a product recursively.

In Figure 8 a transformation is given that adds all the necessary allParts edges in a given instance. Remark that we separate the operations by semicolons and that we use $\{\ \}*$ as a notation for the fixpoint.

4 More Examples

In this section we will give some more examples of GOAL transformations. The first example in Figure 9 presents the calculation of what is known in GOOD as an abstraction. The transformation is defined in the context of the scheme of Figure 7. Its intention is to create one Product Class node for every class of products where a class is defined as a set of products that contain the same parts. In the first operation, a product class is created for every product. Notice that these product classes all have a different products property. This prevents them from being merged into one node. In the second operation, the product classes obtain the same parts as the product they belong to. In the third statement, the product class is stripped of its products property. The only property now left is the parts property. Therefore, the product classes with the same parts will be merged. All products consisting of the same parts will now

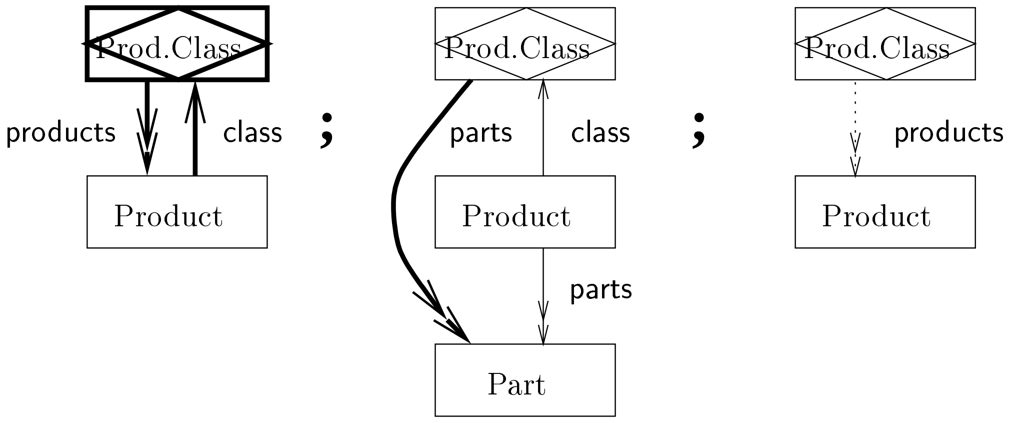

Figure 9: The computation of the Product Classes

point to one and the same product class.

The next example shows how we can simulate computations with numbers. For this purpose we use the scheme as presented in Figure 10.

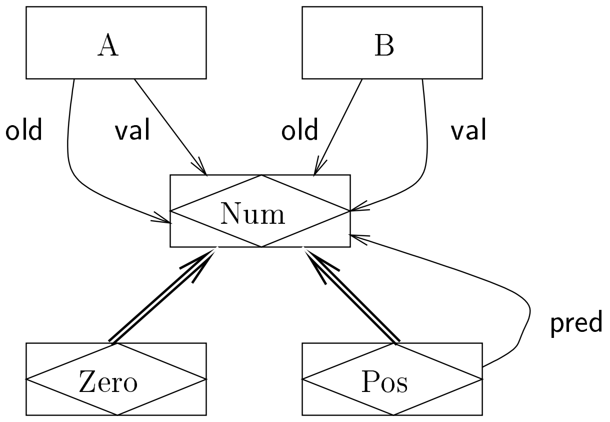

Figure 10: A scheme for numerical computations

Here the classes A and B are meant to each contain one object being the variables a and b, respectively. These variables have two properties, the value and the old value. The value gives the current value of the variable and the old value is used to hold intermediate results. The relation Number holds numbers that are either Zero or Positive. A positive number points to its predecessor, thus numbers are represented as a chained list of positive numbers ending with a zero. In Figures 11, 12 and 13 we present the simulation of some simple computations. In Figure 11 the value of a is incremented by one. In Figure 12 the value of a is decremented by one. Notice that the old number is not removed because it may be shared with other variables. This is very likely because nodes that represent the same number are always merged. In Figure 13

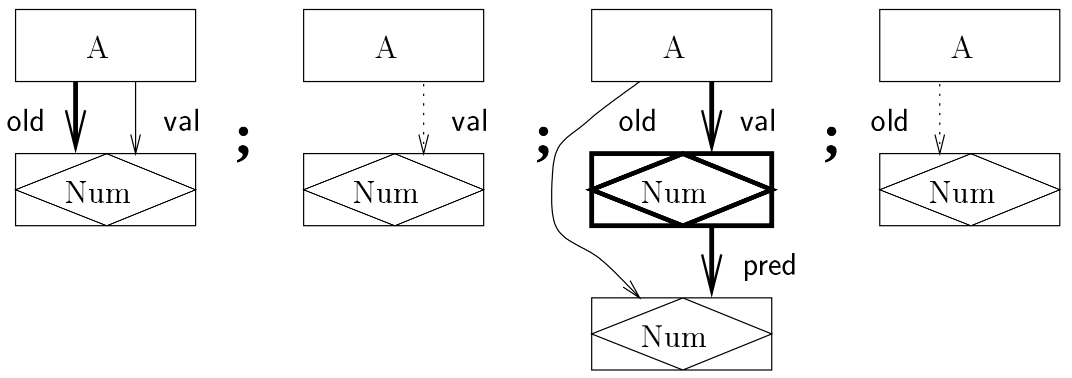

Figure 11: *a* becomes *a* plus 1

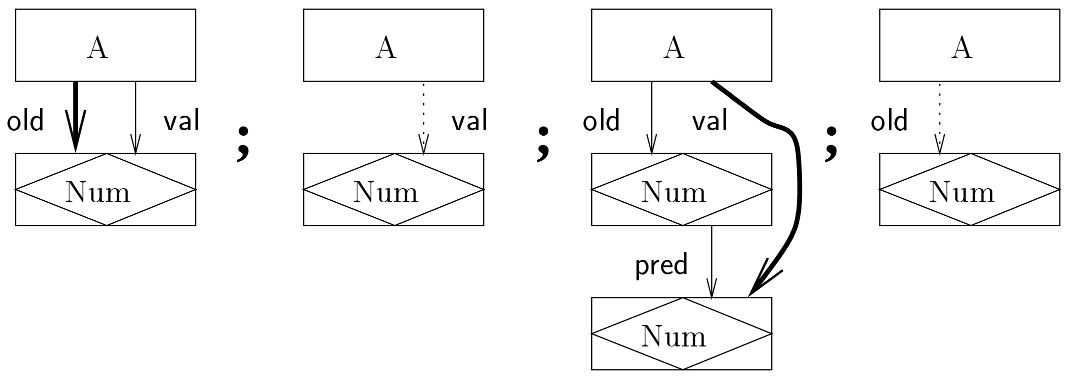

Figure 12: *a* becomes *a* minus 1

the value of b is set to zero if the a is zero. Notice that the first operation of Figure 13 does not match if the value of b is already zero. This is because different nodes in the pattern have to be embedded upon different nodes in the instance. The following addition would not cause any changes either because the edge it tries to add would already exist. The total result would therefore be identical to what was intended.

Finally, some remarks about the completeness of GOAL. From the previous examples we can observe that together with some form of iteration that can be simulated using the fixpoint operation, GOAL can simulate all computable functions on numbers. However, since GOAL is meant as an object database manipulation language, it is more interesting to know whether all "reasonable" manipulations can be computed. Since GOOD is known to be complete in this sense, [10] we can make the following observation.

Theorem 4.1 *GOAL is computationally complete.*

Proof: *All operations of GOOD can be simulated in GOAL. The technical details of this will be omitted here.*

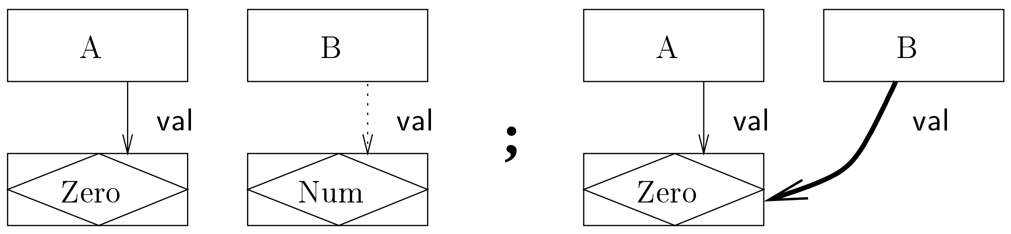

Figure 13: b becomes zero if a is zero

5 Conclusions

In this paper we presented a graphical object-oriented database model and a graph-based manipulation language. This model is the result of the continuing research done on GOOD. The most distinct difference with GOOD is the presence of so-called association nodes. These nodes differ from normal object nodes because they are automatically merged when they represent the same value. This makes them very suitable to model both relationships and tuples. They can, in fact, be seen as a unification of these two concepts from the Entity Relationship approach and the Object-Oriented approach, respectively. Furthermore, the semantics of the association nodes enables us to avoid operations like the abstraction in GOOD, which are not completely pattern based. Another difference with GOOD (and PaMaL) is the injectivity of the pattern matching i.e. two nodes in the pattern can not be embedded upon the same node in the instance. Although not an essential feature we feel that it gives in many cases a more intuitive notion of pattern matching.

The operations of GOAL resemble those of PaMaL and share their simplicity and expressiveness. GOAL, however, does not allow different nodes to represent the same tuple or set in intermediate results. Moreover, GOAL offers relations to conveniently model relationships between more than two objects. Furthermore, GOAL tends to lead to succinct data models because not every tuple and set has to be explicitly represented with a node. A small disadvantage of GOAL is that association nodes that represent nameless tuples need a relation name. Since these tuples usually represent meaningful concepts that deserve a name anyway, this may be considered more of a feature than a bug. Finally, GOAL provides multiple inheritance, even between relations, and a mechanism to detect inheritance conflicts.

References

[1] Chen, P.P.: "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems*, 1 (1976), 9–36.

[2] Nijssen, G.M. and T.A. Halpin: *Conceptual Schema and Relational Database De-*

sign: a fact oriented approach, Prentice Hall, Sydney, Australia, 1989.

[3] Shipman, D.W.: “The Functional Data Model and the Data Language DAPLEX”, *ACM Transactions on Database Systems*, 1 (1981), 140–173.

[4] Abiteboul, S. and R. Hull: “IFO: A formal semantic database model”, *ACM Transactions on Database Systems*, 4 (1987), 525–565.

[5] Abiteboul, S. and P.C. Kanellakis: “Object Identity as a Query Language Primitive”, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, pages 193–204, 1985.

[6] Lécluse, C., P. Richard and F. Velez: “O_2, an object-oriented data model”, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, pages 411–422, 1989.

[7] Beeri, C.: “A Formal Approach to Object-Oriented Databases”, *Data & Knowledge Engineering*, 4 (1990), 353–382.

[8] Gyssens, M., J. Paredaens and D. Van Gucht: “A Graph-Oriented Object Database Model”, *Proceedings of the 1990 ACM Symposium on Principles of Database Systems*, Nashville, pages 417–424, 1990.

[9] Gemis, M. and J. Paredaens: “An Object-Oriented Pattern Matching Language”, *JSSST, International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, pages 339–355, 1993.

[10] Van den Bussche, J., D. Van Gucht, M. Andries and M. Gyssens: “On the Completeness of Object-Creating Query Languages”, *Proceedings 33rd Symposium on Foundation of Computer Science*, pages 372–379, 1992.