The Coverage of Error Diagnosis in Logic Programming Using Weighted Constraints - The Case of an Ill-defined Domain

Nguyen-Thinh Le and Wolfgang Menzel
University of Hamburg
Department of Informatics
{le, menzel}@informatik.uni-hamburg.de

Abstract

Programming should be considered as an ill-defined domain because for a given programming problem there might be many alternative solution strategies and for each of them a wide spectrum of solutions can be created. Most programming problems, which are used to tutor beginners, are simple and might have well-defined start and goal states. The task statement can be well specified and a solution can easily be checked for being correct or not. However, the activity of solving a programming task is a design problem. One can not only apply different programming techniques to create a solution, but also modularize a program to make the code clear, easy maintainable and reusable. In this paper, we demonstrate how we applied weighted constraints to build a diagnosis component which serves the student model of an intelligent tutoring system (ITS) for logic programming. This technology has been integrated into a web-based ITS. We evaluated the coverage of error diagnosis with 221 student solutions of past examinations and the system was able to produce a correct analysis for 87.9% of collected solutions.

Introduction

In the literature, there is no formal definition what constitutes a "well-defined problem". Instead, we must be content with requirements which have been proposed as criteria a problem must satisfy in order to be regarded as well-defined: 1) a start state is available; 2) there exists a limited number of transformation steps which can be relatively easily formalized; 3) evaluation functions are specified and 4) the goal state is unambiguous (Simon 1973). Most programming problems, which are used to tutor beginners, are simple and might have well-defined start and goal states. The task statement is precisely specified and a solution can easily be checked for being correct or not. However, the activity of solving a programming task is a design problem. One can not only apply different solution strategies or different programming techniques to create a solution, but also modularize a program to make the code clear, easy maintainable and reusable. Thus, the second criterion of well-definedness is not satisfied. Hence, programming can be used as a case of ill-defined domains. We study the treatment of such problems in the domain of logic programming.

Currently, the two approaches, which have the potential to deal with ill-defined problems: model tracing (Ogan, Wylie, & Walker 2006) and constraint-based modelling (CBM) (Ohlsson 1994). Model tracing is used by cognitive tutors which are among the most successful ITS today (Koedinger *et al.* 1997) and CBM is applied to a variety of domains among which SQL is the most succesful one (Mitrovic *et al.* 2004). However, these approaches have been applied to problems to a lesser degree of ill-definedness. Typically, they provide a template-like interface which restricts the users' creativity when designing an individual solution. Another approach is to specify exercise texts so clearly that "students can make correct decisions" (Baghaei, Mitrovic, & Irwin 2006), and it remains a task to map between the exercise requirements and the domain's primitives. In fact, such techniques move a task towards the more well-defined problems, and thus, omit important learning objectives, i.e. the proper arrangement of constructs within an SQL statement. In this paper, we introduce an approach for dealing with ill-defined problems by extending CBM with constraint weights (W-CBM). This technology has been integrated in a web-based ITS (INCOM) and evaluated partly with student solutions of past examinations.

In the next section, we depict the training scenario and general conditions under which logic programming problems provided by INCOM should be considered ill-defined. Then, we introduce knowledge representation techniques which are required for diagnosis and describe how the diagnosis procedure is carried out. After that, we evaluate the diagnosis with respect to its coverage, i.e. its ability to identify the solution strategy and the solution elements correctly. Thereafter, the advantages of our W-CBM approach are discussed and we conclude in the last section that W-CBM is very useful in analysing student solutions and identifying the solution strategy.

Training Scenario

INCOM prompts the student with a programming problem and provides feedback to coach her composing a correct solution. Given a task, the student is guided to go through two phases: 1) task analysis and 2) design and implementation. In the first phase, the student is requested to input an adequate signature for the predicate to be implemented. If the signature is not appropriate, INCOM helps her to understand the task. This way, the student's ability to analyse a problem is determined. In the second phase, the student is invited to compose a solution for the given exercise in an unrestricted form. The user interface neither requires the student to adhere to an anticipated solution strategy nor does it specify the arrangement of solution elements (i.e. no input templates are used).

The student is allowed to create Prolog programs (Prolog is a language of logic programming) excluding cuts, disjunctions or if-then-else operators. No assert, retract, abolish or similar database-altering predicates can be used. The set of built-in predicates which can be employed by the students are: =, =.=, =\=, ==, \==, >, >=, <, =<, =.., +, -, *, /, ^ and 'is'. Helper predicates are provided explicitly or must be defined manually.

Under these conditions, the student has a free choice of variable and predicate names, can arrange the parameters within a subgoal or a clause head freely and define helper predicates as needed. Due to this degree of freedom, solving a programming task is a design problem, and thus, logic programming needs to be considered ill-defined.

Knowledge Representation

In order to cover the solution space for a logic programming problem under the conditions above, first, we need a means to represent the semantics required by the problem description. This kind of information is represented in a so-called semantic table. Second, we need a technique to describe the solution space for a programming problem. For this purpose, we apply the CBM approach proposed in (Ohlsson 1994) to model the relationship between structural elements in a solution. Furthermore, we extend the CBM approach with constraint weights to hypothesize the implemented solution strategy and the programming techniques used. Both CBM and constraint weights are required to examine the semantic correctness of the student solution.

Semantic Tables

The semantic table contains two kinds of information: the declaration and implementation semantic requirements. They are used to diagnose the predicate signature and the implementation for a given task, respectively. The second one is specified by means of *generalized* sample solutions which describe the framework of a predicate definition in relational form. That is, clauses, subgoals and argument positions are not restricted to a particular sequential arrangement. In addition, all unification conditions are expressed explicitly and clause heads as well as subgoals are represented in normal form (Table 1). The normal form representation reveals the underlying programming techniques. Thus, the diagnosis becomes more adequate on the conceptual level and the resulting feedback turns more useful. A generalized sample solution is specified for each solution strategy, all of which compose the semantic table for a particular exercise.

Table 1: A semantic table for the sample task

	Head	Subgoal	Description
C1	p(OL,NL)	OL=[]	Basecase list empty
		NL=[]	New list also empty
C2	p(OL,NL)	OL=[N,S|T]	N:Name; S:Salary
		NL=[N,Sn|Tn]	Build a new list
		S=<5000	Salary =< 5000
		Sn is S+S*0.03	Salary is increased
		p(T, Tn)	Recurse old list
C3	p(OL,NL)	OL=[N,S|T]	N:Name; S:Salary
		NL=[N,Sn|Tn]	Build a new list
		S>5000	Salary>5000
		Sn is S+S*0.02	Salary is increased
		p(T,Tn)	Recurse old list

Weighted Constraints

The CBM approach has been proposed in (Ohlsson 1994) to model general principles of a domain as a set of constraints. A constraint is represented as an ordered pair consisting of a relevance part and a satisfaction part: $C =< relevancepart, satisfactionpart >$. The relevance part represents circumstances under which the constraint applies, and the satisfaction part represents a condition that has to be met for the constraint to be satisfied. A constraint is used to describe a fact, a principle or a condition which must hold for every solution contributed by the student. In addition, constraints can also be used to specify the requirements of a task or to handle solution variations. If a constraint is violated, it indicates that the student solution does not conform to the principles of a domain or it does not meet the requirements of the given task.

For a programming problem in an ill-defined domain like logic programming there are many or even uncountably many solutions. We need to create hypotheses about the solution strategy and the programming techniques which have been implemented in the student solution and evaluate them according to their plausibility. These two issues (hypothesis generation and hypothesis evaluation) are carried out at the same time in one and the same decision procedure. For this purpose, we extend the CBM approach with constraint weights which can be conceived of as a measure of importance for a constraint. If the relevance part of the constraint

identifies a solution element, the constraint weight reflects the importance of that solution element. For example, a clause is composed of a clause head and a set of subgoals, each of which is composed of a functor and its arguments. A subgoal contributes more information to the overall correctness of the solution compared to an argument or a functor. Hence, a constraint which examines an argument should be specified as being less important compared to a constraint checking a subgoal. We define a formal representation for weighted constraints as follows: *constraint(Id, Type, Relevance, Satisfaction, Weight, Position, Hint)* where Id is a unique identification of the constraint; $Type$ is a constraint type which guides its invocation; $Relevance$ is the relevance part; $Satisfaction$ is the satisfaction part; $Weight$ indicates the importance of the constraint; $Position$ indicates the error location; and $Hint$ is an instructional message, which explains the error. We use weighted constraints for the following purposes:

- **Declaration constraints**: are used to examine the arity of a predicate as well as type and instantiation mode of each argument position. These constraints are employed exclusively to diagnose the signature of a predicate.
- **Semantics constraints**: are used to examine whether a solution fulfills the requirements of the task statement. Semantic constraints have access to the information from the semantic table. Given a generalized sample solution, the relevance part of a semantic constraint refers to a component of a predicate definition and compares the student solution with respect to that selected component. For example, the following constraint examines whether a clause is a recursive clause according to a requirement.

  Constraint 1:

  Rel: In the generalized sample solution, there exists a recursive clause.

  Sat: In the student solution, there exists a corresponding recursive clause, (which has a subgoal in the clause body, with the same functor and argument list as in the clause head).
- **Pattern constraints**: Standard solution strategies can be modelled as patterns. A pattern describes the coarse structure of a possible student solution. However, patterns are independent from any task. They are used to create hypotheses about the strategy implemented in the student solution. For that purpose, the student solution is decomposed into components which are then compared with the pattern corresponding to the exercise type. A pattern encapsulates the application of several programming techniques, which construct the desired semantics of the pattern, i.e. the recursive processing of list elements. Since a pattern can be specialised by inheriting the characteristics of a super-pattern and adding new techniques, patterns can be organized in a hierarchy. Internally, a pattern is modelled by so-called pattern constraints which describe the pattern's characteristics. Pattern constraints are partly redundant to semantic constraints. If a pattern constraint is violated, a strategy related remedial hint can be derived and returned to the student. Hence, such constraints can enhance the explanation quality of the diagnostic results but they are not mandatory. Not always a suitable pattern can be found for all possible exercise types and solution strategies. In such a case, no strategy related remedial hints can be given (Le 2006).
- **General constraints**: express the general semantic principles of the programming language. They are not specific to any task and must be satisfied by any correct solution.

Transformation Rules

A programming technique or a construct can be instantiated in many different ways. Especially, arithmetic expressions exhibit a great variety of equivalent formulations. In order to represent a space of alternatives for a programming technique or a construct, we have defined transformation rules. Currently, we do not consider recursively embedded arithmetic expressions. Our system just copes with arithmetic expressions without nesting, for example: A*(B+C). For such arithmetic expressions, the following transformation rules are required:

- Rule 1: transforms the normal form to the simplified form applying the distributive law: $A \circ X \pm B \circ X \rightarrow (A \pm B) \circ X$, where the operator $\circ$ is either * or /. If A and B are numbers, then $(A \pm B) \circ X$ can be transformed to $M \circ X$ where $M = A \pm B$. For example: $(2 + 3) * X$ will be transformed to $5 * X$.
- Rule 2: transforms a product term applying the commutative law: $A * B \rightarrow B * A$

Error Diagnosis

According to the two-phase guide of the training scenario, the diagnosis procedure is separated into two steps: signature and implementation diagnosis. Invoking declaration constraints, the signature diagnosis examines the appropriateness of the predicate declaration. Semantic, pattern, and general constraints are invoked in the subsequent implementation diagnosis which determines the implemented solution strategy, the correct implementation of required programming techniques (i.e. implicit or explicit unification) and the correct application of general semantic principles of Prolog (i.e. the instantiation conditions for arithmetic evaluations).

In principle, the signature and implementation diagnosis are carried out as an interaction of hypothesis generation and hypothesis evaluation. Hypotheses are interpretation variants for the student solution [1]. They are generated by mapping elements of the student solution to the ones of the semantic table. Every hypothesis is evaluated based on the relevant constraints. In addition, the score of the constraints,

[1] The term "student solution" means both student's input for signature declaration and implementation.

which are violated by the selected hypothesis, is computed based on a multiplicative model. That score is used to take a decision for the most plausible interpretation. Diagnostic information about shortcomings of the student solution can be gained from constraint violations. Superfluous and missing elements in the student solution are detected from the hypothesis mapping.

After the declaration diagnosis is finished and the student has provided an appropriate signature declaration, the student is allowed to input her implementation. Once the solution is submitted to evaluation, the implementation diagnosis is carried out as follows: 1) Hypothesis generation: the student solution is mapped iteratively to each of the generalized sample solutions; 2) Hypothesis evaluation: for each mapping, the plausibility is computed based on violated constraints and the mapping which has the highest plausibility score represents the best hypothesis. The interaction of hypothesis generation and hypothesis evaluation takes place on different levels subsequently:

1. Mapping of the clauses;
2. Mapping of the subgoals within a clause;
3. Mapping of arguments and operators;
4. Mapping of summands in a arithmetic expression;
5. Mapping of factors and algebraic signs in a summand.

Suppose, X is a set of expressions on the level to be considered, which are extracted from the semantic table, i.e. on the clause head/subgoal level X consists of a clause head and a set of subgoals and Y is a set of expressions of the same level extracted from the student solution. From a horizontal perspective, on each level the diagnosis works as a generate-and-test procedure as follows:

1. **Create a set Z of hypothesis mappings between X and Y:** A mapping is a combination of expressions in X and Y, where the following cases are considered:
   - If X is empty and Y is not empty, then take a $y \in Y$ and add $map(NIL, y)$ to the current mapping. If Y is empty and X is not empty, then take a $x \in X$ and add $map(x, NIL)$ to the current mapping. Otherwise, for all $x \in X$ and $y \in Y$ add $map(x, y)$ to the current mapping.
   - If $x \in X$ contains transformable expressions, then apply mathematical transformations to create variants of x which extend the size of Z.
2. **Compute the plausibility of hypothesis mappings:**
   - Select constraints of the current diagnosis level.
   - The plausibility for the mapping $z \in Z$ is computed by the formulae: $P(z) = \prod_{i=1}^{N} S_i$ where S_i is the weight score of the constraint which is violated and $0 < S_i < 1$. Compute the plausibility for each mapping $z \in Z$ iteratively.

Table 2: A student solution for the sample task

SC1	gehalt([], []).
SC2	gehalt([N,B|R],E):-B>5000, B is B+B*0.02, gehalt(R,E), E is [N,B|E].
SC3	gehalt([N,B|R],E):-B is B+B*0.03, gehalt(R,E), E is [N,B|E].

   - The mapping z, which has the highest plausibility $P(z)$, is the best hypothesis.

For example, there is one generalized sample solution for the sample task (Appendix A) in the semantic table (Table 1) and a sample student solution is given in Table 2. According to the procedure above, the diagnosis will begin at the clause level. First, it maps clauses of the semantic table with clauses of the student solution. The following hypothesis mappings are created: *Mapping1=*{*map(C1;SC1), map(C2;SC2), map(C3;SC3)*} and *Mapping2=*{*map(C1,SC1), map(C2;SC3), map(C3;SC2)*}. As *C1* and *SC1* are of the same clause type (base case), there is just one possible mapping whereas *C2, C3, SC2* and *SC3* are recursive cases, therefore 2*1=2 possible mappings need to be considered on the clause level. Second, the hypothesis generation iterates through the subgoal, argument/operator, summand, factor levels and the amount of mappings rises with the diagnosis depth. Finally, the generated mappings are evaluated by invoking the corresponding constraints and the one with the highest score is selected.

Evaluation of the diagnostic coverage

Since the student might follow a range of many solution strategies, the ability of the system to interpret and to analyse a student solution correctly is an indispensable prerequisite for deriving a precise diagnosis. To measure this ability, we conducted a preliminary evaluation. We selected appropriate exercises and solutions from past written examinations. The participants were students who had chosen their major in different branches of Informatics. The examination candidates had attended a course in logic programming which was offered as a part of the first semester curriculum in Informatics. The evaluation was meant to determine the number of student solutions whose strategy was identified correctly by INCOM.

We selected seven exercise tasks (a sample is described in Appendix) from the examinations of the years 1999 and 2000, for which there were 221 solutions available. Each of the exercise tasks requires different skills to master, among them: arithmetic calculation, arithmetic test, database relationships, list (de)composition, recursion, and unification. While collecting student solutions from past examinations,

Table 3: Evaluation of the diagnostic coverage

Task	Sol	Not understandable	Correctly analysed	Incorrectly analysed
1	10	0	10	0
2	11	0	10	1
3	6	2	3	1
4	17	1	16	0
5	58	2	54	2
6	81	0	79	2
7	38	2	34	2
Sum	221	7	206	8

we filtered out solutions which are not sufficiently elaborated for applying a diagnosis, i.e. highly fragmentary clauses. Furthermore, we added appropriate predicate declarations, because during the examination, students were not asked to provide that information about meaning, types and modes of each argument position.

To evaluate whether a solution can be analysed correctly by INCOM, an expert investigates every student solution manually. Student solutions which can not even be understood by a human expert, are sorted out to the group "*not understandable*" (an example in Appendix). All "*understandable*" solutions are input into the system which resulted in a list of violated constraints. The expert examines the list of violated constraints and decides whether the system has analysed the student solution correctly. Accordingly, it is assigned to the category "*correctly analysed*" or "*incorrectly analysed*".

Table 3 summarizes the statistics of the evaluation. The amount of available student solutions is indicated in the second column. The third column represents the number of solutions which are sorted to the category "*not understandable*". The fourth and the last column show the amount of solutions which belong to the category "*correctly analysed*" and "*incorrectly analysed*", respectively.

Overall, 87.9% (s=17.1%) of the student solutions have been analysed correctly by INCOM. Those solutions, which have been excluded by the human expert, were implemented with several arbitrary helper predicates. This indicates that the student is disoriented and does not follow a specific solution strategy. The reasons for an analysis failure were that students defined helper predicates which were not modelled in the semantic table. In addition, one student solution was implemented using an operator ";" to concatenate two clauses which is not supported by our system. In other solutions, a comma was used instead of a decimal point, which is the standard for numbers in German, but forced Prolog to break the arithmetic expression into two subgoals. The ratio between "*not understandable*" and "*incorrectly analysed*" solutions is 7:8 which indicates that almost half of the solutions, for which INCOM is not able to produce a correct analysis, can not be understood by the human expert either.

Discussion

In comparison with other application domains, for which constraint-based ITS have been developed, the problem tasks in logic programming poses a challenge for system development. From the perspective of a domain model, we are confronted with novel requirements, because simplified assumptions, which have been made for other application contexts, are no longer valid. Those assumptions are: 1) The absence of a unique ideal solution; 2) The variance, which arises from the free choice of identifier names; 3) The variance, which results from different solution strategies and implementation techniques; 4) The ability to determine the solution strategy implemented in the student solution. According to these four criteria, other constraint-based ITS published so far provide problem tasks, which are usually simplified to a degree that they become more well-defined. Typically, an ideal solution is assumed and constraints serve primarily to enumerate the variations of the solution (Baghaei, Mitrovic, & Irwin 2006). This practice fails, if there are many or even uncountably many solutions for a problem task like in the domain of programming. Hence, programming problems should rather be considered to belong to the class of ill-defined domains, even though any software solution has a formal semantics. For this situation, we have developed an approach which avoids ideal solutions as far as possible. For the purpose of diagnosis, concrete sample implementations are generalized to abstract schemata, which are then restricted by additional co-reference and general linearisation conditions. Therefore, the advantages of using weighted constraints are most convincing in correspondence with the treatment of possible serialisation variants.

In addition, our approach of using generalized sample solutions in the semantic table results in a preference for simple, i.e. elegant solutions. For example, in the arithmetic expression X+1-1+1-1, the system will consider the elements +1, -1 as superfluous, although the term is semantically correct. This way, the principle of parsimony, provides the diagnosis with a means to deal with yet another aspect of ill-definedness.

In general, CBM tutoring systems are considered to be unable to identify the implementation strategy of a student solution. If a remedial hint makes a wrong assumption about the student strategy, then the feedback can be misleading (Martin 2001). This deficiency is compensated in our prototype by the concept of patterns. From a didactic point of view, patterns fulfil two purposes: 1) they can be used to make assumptions about the strategy implemented in the student solution; 2) from those assumptions, strategy related feedback can be derived (Le 2006). From the perspective of knowledge representation, the hierarchical organisation of patterns provides advantages with regard to system development and system extension. This is a feature which has not been proposed for alternative approaches so far and might even be difficult to realise.

Our system provides a two-tiered interaction design which offers advantages with regard to the diagnosis quality. It makes available the predicate schema, thus providing the diagnosis with essential information about the exercise specific allocation of arguments, and therefore considerably reduces the number of mapping hypotheses that have to be analysed. Furthermore, the two phases of solution development correspond to important stages of the programming process: signature declaration and implementation of the required semantics.

Our transformation approach starts with a normalised expression from the semantic table and produces semantically equivalent alternatives, which are compared with expressions from the student solution. Thus, the direction of our transformation is different from the one adopted in comparable works, i.e. (Gegg-Harrison 1993), where expressions to be examined in the student solution are transformed to a normal form, for which the semantic equivalence with a correct solution has to be proven. Such an approach, however, comes with a serious drawback. Since the transformation takes place before the proper diagnosis, the localization of the error becomes problematic. This results also in a negative effect on the precision and comprehensibility of resulting remedial hints. After all, our transformation approach considers a wide spectrum of alternative variants of arithmetic expressions. Modelling a comparable diversity of variants will probably be difficult in other frameworks.

Conclusion and Future Works

We have proposed an extended CBM approach using weighted constraints to model the solution space for logic programming problems. Error diagnosis is based on a knowledge representation for which we applied two features: a semantic table which contains mandatory semantic information and weighted constraints which describe the solution space for a programming problem. The diagnosis procedure is carried out by an interaction between hypothesis generation and hypothesis evaluation. We conducted an evaluation of the model's coverage with 221 student solutions. INCOM was able to produce a correct analysis for 87.9% of student solutions. Next, we are going to extend our semantic table with helper predicates in order to improve the diagnosis coverage. In addition, we evaluate the diagnostic accuracy and conduct an online evaluation to assess the educational effectiveness of INCOM.

References

Baghaei, N.; Mitrovic, A.; and Irwin, W. 2006. Problem-solving support in a constraint-based intelligent tutoring system for uml. *Technology, Instruction, Cognition and Learning* 4(1-2).

Gegg-Harrison, T. S. 1993. *Exploiting Program Schemata in a Prolog Tutoring System*. Number 27708-0129. Durham, North Carolina: Department of Computer Science, Duke University.

Koedinger, K.; Anderson, J.; Hadley, W.; and Mark, M. 1997. Intelligent tutoring goes to school in the big city. *International Journal of AI in Education* 8:30–43.

Le, N.-T. 2006. Using Prolog design patterns to support constraint-based error diagnosis in logic programming. In Ashley, K.; Aleven, V.; Pinkwart, N.; and Lynch, C., eds., *Proceedings of the Workshop on ITSs for Ill-Defined Domains, the 8th Conference on ITS*, 38 – 46.

Martin, B. 2001. *Intelligent Tutoring Systems: The Practical Implementation Of Constraint-based Modelling*. Ph.D. Dissertation, University of Canterbury.

Mitrovic, A.; Suraweera, P.; Martin, B.; and Weerasinghe, A. 2004. Db-suite: Experiences with three intelligent, web-based database tutors. *Journal of Interactive Learning Research (JILR)* 15(4):409–432.

Ogan, A.; Wylie, R.; and Walker, E. 2006. The challenges in adapting traditional techniques for modeling student behavior in ill-defined domains. In Ashley, K.; Aleven, V.; Pinkwart, N.; and Lynch, C., eds., *Proceedings of the Workshop on ITSs for Ill-Defined Domains, the 8th Conference on ITS*, 29 – 37.

Ohlsson, S. 1994. Constraint-based student modelling. In Greer, J. E., and McCalla, G. I., eds., *Student Modelling: The Key to Individualized Knowledge-based Instruction*. Berlin: Springer-Verlag. 167–189.

Simon, H. A. 1973. The structure of ill structured problems. *Artificial Intelligence* 4(3):181–201.

Appendix

A sample task: A salary database is implemented as a list whose odd elements represent names and even elements represent salaries measured in Euro. For example: [meier,3600,schulze,5400,mueller,6300,...,bauer, 4200]. Define a predicate which computes a new salary list based on the given one according to following rules: 1) a salary below or equal 5000 Euro will be raised by 3%; 2) a salary above 5000 Euro will be raised by 2%.

A "*not understandable*" sample solution:

```
gehalttarif(Gehaltvorher,Gehaltnachher):-
  gtacc(Gehaltvorher,[],Gehaltnachher).
gtacc(GLvor, Acc, GLnach).
gtacc([GLvorName, GLvorDM|GLvorTail], Acc, GLnach):-
  gtacc(GLvorTail, [GLvorName,GLneuDM|Acc], GLnach),
  (GLneuDM is GLvorDM*103/100, Gehalt<=5000);
  (GLneuDM is GLvorDM*105/100, Gehalt>5000).
```