PolyAnalyst data analysis technique

Mikhail V. Kiselev
Megaputer Intelligence Ltd., 38 B.Tatarskaya, Moscow 113184, Russia
megaputer@glas.apc.org

Sergei M. Ananyan
IUCF, Indiana University, 2401 Sampson Lane, Bloomington, IN 47405
sananyan@indiana.edu

Sergei B. Arseniev
Megaputer Intelligence Ltd., 38 B.Tatarskaya, Moscow 113184, Russia
megaputer@glas.apc.org

A B S T R A C T

The data analysis techniques of the PolyAnalyst data mining system [Kiselev 94] are based on an automated synthesis of the functional programs treated as multi-dimensional non-linear regression models. This approach provides the system with two valuable properties: 1) it can discover hidden in the data relations that might be of a great variety of forms, 2) it can explore arbitrarily complexly structured data when the corresponding data access primitives are provided. The paper contains a description of the final version of the basic PolyAnalyst mechanisms, which are utilized in the general case, as well as in a particular case of data organized as a set of attribute values (SAV), which is the most common format for the data explored by KDD methods. Numerous practical results obtained by the users of PolyAnalyst in various fields corroborate the high efficiency of the discussed approach to an automated discovery of numerical dependencies in the SAV-structured data.

KEYWORDS: knowledge discovery, automated program synthesis, non-linear regression

1. Introduction.

A great variety of methods proposed for an automated discovery of numerical relations in data can be ordered according to the generality of the functional relations found or alternatively - according to their computational complexity. On the one pole are the fast algorithms capable of discovering relations of very narrow classes. Linear regression and decision trees would be good examples of such algorithms. On the opposite pole are the methods based on an extensive search in broad sets of all possible relations. The latter methods are able to discover and formalize complex non-linear dependencies, but the price to pay for their generality is a very long computational time. As examples of the systems of this kind one can mention FAHRENHEIT [Zytkow, Zhu, 1991], ARE

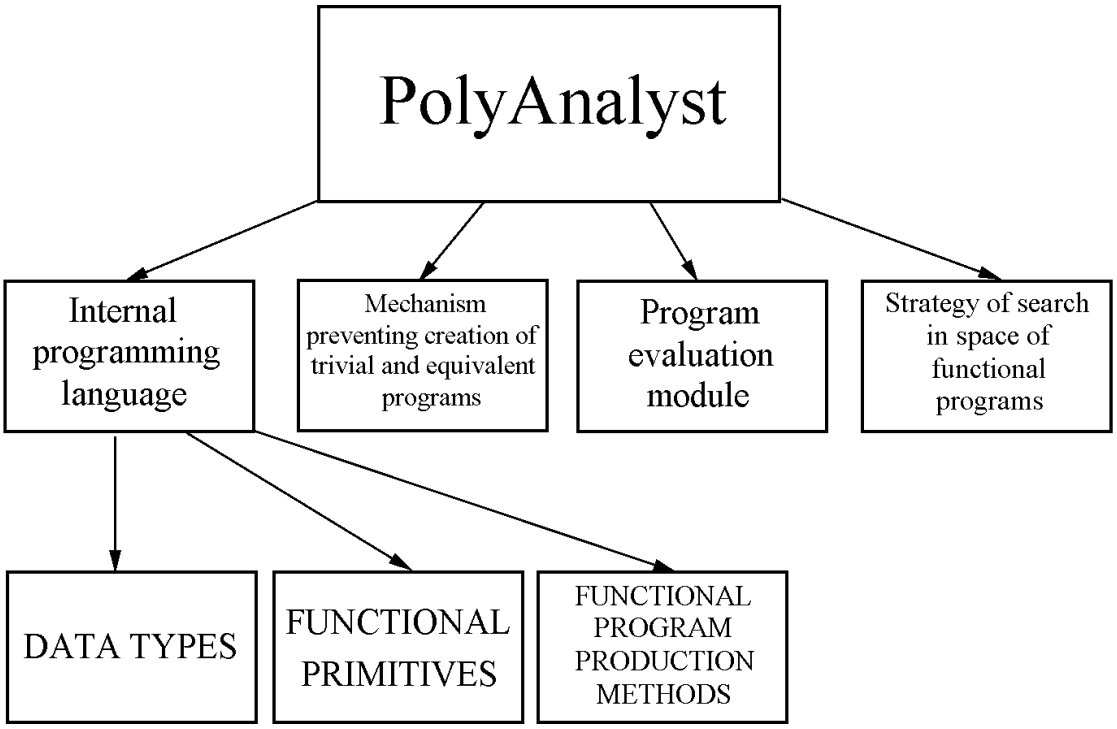

Fig. 1 PolyAnalyst logical structure.

[Shen, 1990], or 49er [Zembowicz, Zytkow, 92]. These systems construct more involved formulae that express the relations in data by combining simpler formulae using a functional composition mechanism, and eventually finding a sufficiently accurate form for the relation. If one performs a full search in the space of all possible formulae, then the number of tested algorithms grows super-exponentially with the complexity of the built algorithms. In order to reduce this dramatic growth rate and arrive at an acceptable computational time in such an approach one needs to develop a better mechanism for the navigation in the space of all possible algorithms. A combination of advanced navigation mechanisms had been implemented in the PolyAnalyst data mining system [Kiselev 94; Kiselev, Arseniev 96; Kiselev, Ananyan, Arseniev 97] described in this paper.

At present PolyAnalyst occupies probably the most extreme position in the family of data mining systems performing a full search in the broad space of all possible relations. The main module of PolyAnalyst automatically formulates and tests hypotheses about the sought relation in the form of programs expressed in a simple internal functional language (henceforth this main module of the system will be referred to as simply PolyAnalyst). The internal programming language has a sufficient expressive power for formalizing any relation which can be expressed in an algorithmic form, when a set of necessary functional primitives is provided.

Since the search for the best regression model in a set of functional programs is a very difficult problem, the logical structure of the PolyAnalyst system is quite complex. The system includes several components working cooperatively as a single structure (Fig. 1), which are

described in detail in the present paper. PolyAnalyst implements a very general mechanism for generating and testing the regression models. For example, this mechanism can work with data that have an arbitrary structure because programs generated by PolyAnalyst access the explored data via special data access primitives, which can provide an access to the specific elements of vectors, matrices, lists, and other arbitrarily complex data structures. At the same time, in the case when the database records are represented as sets of scalar values, some of the PolyAnalyst algorithms can be implemented much simpler. An exact definition of this data format and the corresponding modification of the PolyAnalyst techniques specialized for the exploration of data represented as sets of scalar values, are the subject of the last section of the paper.

2. PolyAnalyst internal programming language.

We start the description by considering the PolyAnalyst internal programming language, which is used for formulating hypotheses about the relations in data. This language is a functional programming language in the sense defined in [Backus 1978]. Similar to any other programming language, the PolyAnalyst internal language includes three principal components: data types, functional primitives, and control structures, which can also be considered as the methods for constructing more complex programs from the simpler programs.

The data types of the PolyAnalyst internal language form two classes: universal data types, which are defined for all application domains, and user-defined domain-specific data types. The former class includes only two data types, namely, *boolean*, denoted as **L**, and *numerical* (real number), denoted as **N**. The **N** type is the only data type containing an infinite number of values. All other data types, including **L** and all user-defined types, involve finite sets of values. To clarify the term “user-defined data types” it is necessary to explain our understanding of the concept of structure of the explored data. We consider the records of the analyzed database as sets of mappings from some sets to other sets. For example, the data represented as a two-dimensional matrix of numerical values can be considered as a mapping from the direct product of the sets of the vertical and horizontal positions to **N**: $POSX \times POSY \rightarrow \mathbf{N}$. Generally speaking, any data structure can be thought of as a mapping from some sets of keys determining the access to individual values, to the sets of these values, so that such an approach does not limit our ability to work with arbitrarily complexly organized data.

In our formalism the properties of every data type are determined by two characteristics. The first characteristic describes the ordering properties, namely, whether the relation “greater than” or the operator “next” are defined for the data of this type. The second characteristic is called enumerability. It determines whether the instruction "For each value from the data type X perform the following actions …" makes sense for the considered data type.

The functional primitives of the PolyAnalyst internal language are at the same time the simplest programs of this language, so that prior to describing them it is necessary to explain how the programs synthesized by the system are represented. A program P is considered as an object having a certain set (possibly empty) of inputs **in**(P) and one output. Every input $\alpha \in$ **in**(P) is marked by its data type DT(α) and also by some other attributes discussed below. The data type of a program output is denoted as DT(P). Every input α of the program can be assigned some value $p(\alpha)$ in accordance with its data type. A set of all possible mappings from the set of inputs

in(P) to the set of their values will be denoted as *EVIN*(P). For every $p \in EVIN(P)$ the value P(*p*) returned by a program can be computed as a result of the sequence of operations determined by an internal structure of the program. These operations depend on a set of functional primitives entering the program and the control structures utilized. If a program contains the so called data access primitives (see below) then a database record for which the program is evaluated should be specified. In that case the output value of the program depends on the database record number *i* explicitly: P(*p*, *i*).

The functional primitives which can be included in the programs created by PolyAnalyst, are also broken into two classes: universal and user-defined primitives. The first class includes various operations defined on the universal data types **L** and **N**. These are the boolean operations AND, OR, and NOT, which are represented as primitives with two (AND, OR) or one (NOT) inputs of the type **L** and an output of the type **L**. As a generalization of the numerical relational operators a ternary primitive *inr* with the prototype *inr*:**L**(*x*:**N**, *y*:**N**, *z*:**N**) is used. The value of this primitive is the value of the proposition $y \le x < y + z$. In addition to these primitives the universal primitives include a so called TF-commutator *if*:**N**(*b*:**L**, *x*:**N**, *y*:**N**). If $b = 1$ then the value of this primitive is *x* otherwise it is *y*. The mission of this primitive is similar to the mission of the **if** construction of the C programming language. The user-defined primitives are divided to primitives generated automatically for defined data types, data access primitives, and special primitives. For example, for each data type ***T*** a primitive expressing the equality relation with the prototype **L**(***T***, ***T***) and a TF-commutator with the prototype ***T***(**L**, ***T***, ***T***) are created automatically. The primitives implementing the "greater than" relation and the "next" operation are produced if the respective data types are described as ordered. The prototypes and the bodies of the data access primitives are determined by the structure of the analyzed database records. For example, if the records are organized as two-dimensional matrices containing the values of the type *D*, then PolyAnalyst creates a primitive with the prototype *D(PosX, PosY)* where *PosX* and *PosY* are the data types representing the horizontal and vertical positions in the matrix. Thus, PolyAnalyst has no internal limitations on the structure of the explored database records if the necessary data access primitives are supplied. The data can be organized as sets of scalar values, vectors, matrices, lists, or any other structures. And finally, a subgroup of special primitives includes the primitives corresponding to operations specific for an explored application domain. For example, a calculation of the sine function might be required only in a narrow class of application domains. Therefore a primitive implementing the calculation of sine should be defined explicitly by its body and prototype when necessary.

As it has been mentioned before, functional primitives are considered as the simplest programs. In order to create more complex programs from the simpler programs, several production methods (or control structures) are used. The PolyAnalyst internal language has two basic types of the production methods: functional composition and iteration/recursion.

1. **Functional composition**. This is a simple and clear way of creating new programs from existing ones. Connecting the outputs of some programs P_1, ..., P_n to the inputs of a program Q we build a new program Q(..., P_1, ..., P_n,...). It is required that data types of the corresponding inputs and output match. A formal definition of this production method can be found in Appendix.

2. **Iteration/recursion**. In contrast with a simple and clear functional composition, the production method dedicated to creating iterative and recursive constructions is very complex. It realizes computational structures suitable for expressing a great variety of unconditional and

conditional loops, as well as more specific concepts such as ∀ and ∃ quantors. Let us enumerate the structural components which may enter this constructions and list their purposes.

a. Loop variables Ω. These are internal objects of an iteration/recursion construction which are connected to the inputs of sub-programs which are parts of the construction. They have definite data types (which should be discrete) and get different values from their data types during the process of evaluation of a considered construction. They are used for the implementation of the calculations that should be performed for each element of some set.

b. The procedure **PRED** realizing a predicate used for the definition of the loop variable value

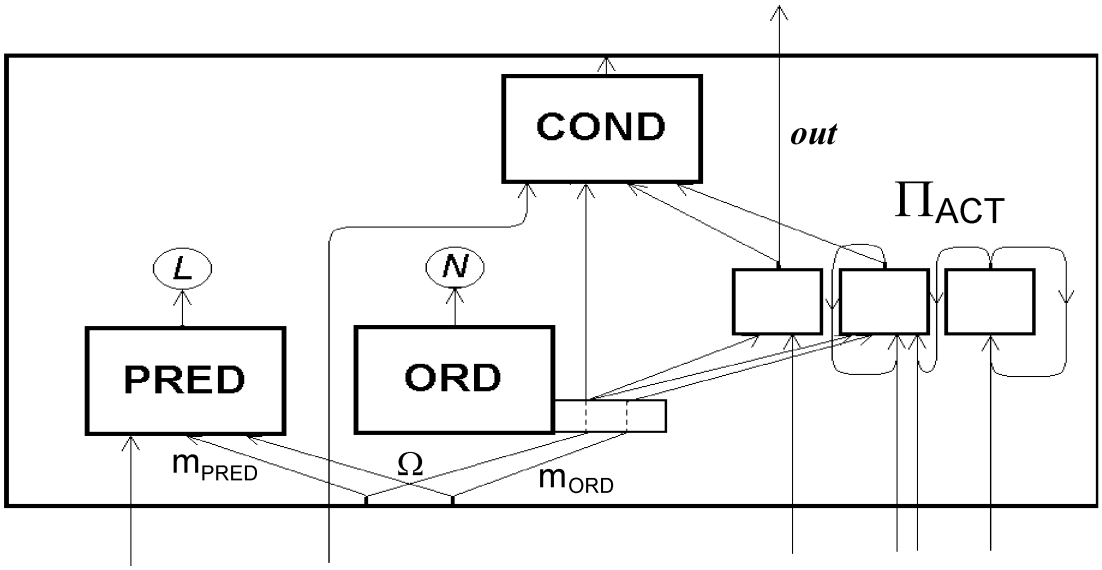

Fig. 2 Iteration/recursion production method.

combination subset for which iterative computations are performed.

c. The procedure **ORD** defining an order on the set of values of loop variables. A sequence in which values are assigned to loop variables in the process of computation is determined by this order.

d. The procedure **COND** realizing a termination condition for iterative computations (for conditional loops only).

e. Procedures Π_{act} representing the body of an iterative construction. During one iteration these procedures are executed. If the termination condition is not satisfied, the values returned by these procedures are passed to some of their inputs and a next iteration starts. Beside that a value ***out*** returned by some of these procedures becomes an output value of the whole iterative construction after the computation termination.

Schematically this construction is depicted in Fig. 2. Some comments to this figure along with a formal definition of this production method also can be found in Appendix.

Although the two discussed production methods are sufficient for providing the PolyAnalyst internal language with an expressive power of a universal programming language, an important special case of the functional composition was selected as a third production method. This mechanism is used for representing numerical dependencies. It is based on the

inclusion of the programs returning numerical values in the form of rational expressions (i.e. polynomial divided by polynomial). The reasons for singling out this special form of numerical relations, as well as the details of the implementation of this production method are described in [Kiselev, Arseniev 96] to which we refer an interested reader.

3. Prevention of building trivial and equivalent programs.

Applying the production methods to existing programs we obtain new programs. However, one cannot guarantee that the new programs will be "good" programs from the semantic point of view. The synthesized programs should satisfy the following three requirements to be considered as semantically correct:

1. *Dependence on all inputs*. We say that procedure P depends on its *i*-th input if $P(x_1,..., x_{i1},..., x_n) \neq P(x_1,..., x_{i2},..., x_n)$ for some combination of its input values. A semantically correct procedure should depend on all its inputs. This requirement can be expressed rigorously as: $\forall \alpha \in \mathbf{in}(P) \exists p,q \in EVIN(P): P(p) \neq P(q) \wedge \forall \beta \in \mathbf{in}(P) \setminus \{\alpha\}: p(\beta) = q(\beta)$. Naturally this is applicable if $\mathbf{in}(P) \neq \varnothing$.

2. *Dependence on a database record*. This requirement is applicable to the programs containing data access primitives. It is required that such programs return different values for some pair of database records under the same set of input values: $\exists p \in EVIN(P) \exists i \exists j: P(p,i) \neq P(p,j)$.

3. *Inequivalence to existing programs*. If a program performs exactly the same computation as some existing program it should be discarded. Mathematical representation of this requirement is rather complicated:

For all existing programs Q for which there exists an isomorphism $f: \mathbf{in}(P) \xrightarrow{f} \mathbf{in}(Q)$ such that DT($f(\alpha)$) = DT(α) and for all such isomorphisms: $\exists p \in EVIN(P): P(p) \neq Q(f[p])$, where *f[p]* is defined by the equality $f[p](\alpha) \stackrel{def}{=} p(f^{-1}(\alpha))$.

In order to sift out the programs not obeying these conditions the following mechanisms are utilized:

a. Inputs and outputs of the programs are marked by additional flags called "consistency types" so that inputs and outputs of certain combinations of consistency types cannot be connected in the process of creating new programs. For example, the equivalence NOT(a<b) $\Leftrightarrow$ a=b $\vee$ a>b is eliminated by declaring the input of the NOT primitive and the output of the LESS_THAN primitive as inconsistent.

b. The symmetry properties of the program inputs are considered. Only one representative of symmetrically equivalent productions is selected. For example, OR(P(...), ...) is retained, while OR(...,P(...)) is rejected.

c. Similarly, the requirement that some inputs cannot be connected to the same output is taken into account. This helps in avoiding the equivalencies like OR(P(...), P(...)) $\Leftrightarrow$ P(...).

d. For the commutative productions a certain fixed order of their application is selected. This measure eliminates the equivalencies of the kind: P(NOT*(x), y, z, w*) = IF(Q*(x, y), z, w*), where P(*x, y, z, w*) = IF(OR*(x, y), z, w*), Q(*x, y*) = OR(NOT*(x), y*).

e. Direct tests are applied to check whether the requirements 1 and 2 hold. These tests are based on a comparison of the values returned by the built programs under a variation of the values assigned to their inputs.

f. For each constructed program PolyAnalyst calculates a special value called an input-output signature, which depends on the values of its inputs and the respective returned values. The procedure of calculating this signature guarantees that the programs equivalent in the sense defined in requirement 3 have the same signature values. The procedures with equal signatures are tested for satisfying the requirement 3.

4. Evaluation of the constructed programs.

As had been mentioned before, the created programs are considered as solutions to a certain problem. For example, when solving the problem of the discovery of a numerical relation, the programs constructed by PolyAnalyst are treated as regression functions and are evaluated in terms of standard error of the respective regression models. Alternatively, when solving a classification task, PolyAnalyst evaluates the constructed algorithms to achieve the minimal number of incorrect classifications. Depending on the problem solved, the module performing the evaluation of programs can implement an arbitrary functional ***Ev***[P], which should be minimized. This feature provides the system with a great flexibility: the ability to solve a wide class of KDD and optimization problems.

Strictly speaking, not every program is considered as a potential solution: the programs, which involve no data access primitives, and even some classes of programs including such primitives serve only as components for other programs. It should be noted that each program P represents in general a set of mappings from a set of database records to DT(P), parametrized by the values of its inputs. Therefore for an individual program P the problem of finding the best values for its inputs must be solved. Depending on the structure of the functional ***Ev***[P] and the form of the program P, methods of combinatorics, numerical optimization, or other approaches may be used to solve this problem.

5. GT-search.

The last but not least part of the PolyAnalyst system is a module that selects the production methods, and components for building new programs. The generation of new programs is performed by two processes. The first process, with a lower priority, implements the full search in the space of programs in the order of increasing complexity (which approximately equals to the number of primitives constituting the program). Assigning this process a lower priority ensures that the slow full search process runs in the background of the other quick process efficiently developing a few currently most promising branches of the search tree. This other process, which has a higher priority, creates new programs using the so-called generalizing transformations (GT) [Kiselev, Arseniev 96].

The application of a GT to an existing program yields a new program called a GT-derivative. A GT-derivative of a program P has the same inputs as the original program, plus some additional inputs so that for some values assigned to these additional inputs the GT-

derivative becomes identically equal to P. One can say that GT-derivatives have additional degrees of freedom in comparison with the program P and thus they can be called its generalizations. In terms of the formalism introduced in section 2 the program P′ is called a GT-derivative of the program P (that is denoted as $P \overset{GT}{\phi} P'$) iff:

1. There exists an isomorphism $f: \mathbf{in}(P) \xrightarrow{f} I \subset \mathbf{in}(P')$ such that DT($f(\alpha)$) = DT(α);

2. There exists an isomorphism $F: EVIN(P) \xrightarrow{F} E \subset EVIN(P')$ such that $q = F(p) \Rightarrow q(f(\alpha)) = p(\alpha)$ and $\forall \alpha \in \mathbf{in}(P') \setminus I \forall p,q \in E: p(\alpha) = q(\alpha) \wedge \forall p \in E: P'(p) = P(F^{-1}(p))$.

For any program there exists a large number of classes of transformations of the program structure, which lead to the creation of a GT-derivative of this program. One should note that the GT are applied only to the most promising branches of the seat tree, thus facilitating a dramatic reduction of the computational time. The success of the utilization of the GT-search for navigating in the space of programs is based on an obvious fact that being a generalization of the program P, the program P′ cannot be worse than the program P in terms of the criterion mentioned in the previous section: $P \overset{GT}{\phi} P' \Rightarrow \boldsymbol{Ev}[P'] \leq \boldsymbol{Ev}[P]$. Indeed, since there always exists such a combination of the values of inputs from $\mathbf{in}(P') \setminus I$, which makes P′ identical to P, thus the value of the ***Ev*** functional for the former program is at least not greater than the value of ***Ev*** for the latter program. Furthermore, if $\mathbf{in}(P') \setminus I$ includes an input of the type **N** and a partial derivative of $\boldsymbol{Ev}[P']$ with respect to that input is different from zero, then we can always decrease the value of $\boldsymbol{Ev}[P']$ by slightly changing the value of this input in the respective direction. This property makes it possible to organize a GT-based search in the space of programs in the following way. When the process of the full search finds a program for which ***Ev***[P] is sufficiently small, this program becomes a parent for a generation of programs created from this program with the help of GT. If one of these programs demonstrates a significant decrease in ***Ev***, this program in turn is taken as a starting point for building new programs with the help of GT and so on. By utilizing this approach the system can build rather complex programs over a reasonable period of time.

This description of the GT-search concludes the discussion of the internal PolyAnalyst mechanisms employed in a general case. The next section is devoted to discussing a special case when the data has a “set of attribute values” (SAV) structure.

6. Specialization of PolyAnalyst mechanisms for the case of data represented as a set of attribute values (SAV).

First of all, let us furnish a precise definition of the SAV data format. From the point of view of PolyAnalyst a data format is defined completely by specifying the data access primitives, a set of possible data types, and a set of the user-defined primitives. The term “set of attribute values” implies that each considered database record constitutes a set of scalar values of different types. Therefore for every position of the data record a data access primitive with no inputs is generated. The data type of the output of this primitive matches the data type of the attribute value in the respective position. Also an additional data type is introduced for every position of the record, which contains unordered non-numerical values. Beside the access primitives, the

only class of the user-defined primitives which can be introduced for a domain of that kind is the class of equality primitives for additional data types. It can be easily shown that this set of data types and functional primitives leaves very few possibilities for employing the iteration/recursion production method for building new programs. For this reason, and also because the functional composition method is much easier to implement, it is reasonable to use in the SAV application domains only the functional composition method. Furthermore, in this case the majority of programs are generated utilizing an important subclass of the functional composition method, namely the production of rational expressions.

Beside the internal language, the other component of the PolyAnalyst system, which is influenced greatly by the assumption of the SAV data organization, is the GT-search mechanism. Since in this case the **rational** production method plays a very important role, the following two kinds of generalizing transformations applicable to rationals are used most often:

1. Let us denote the rational expression subjected to GT as P = *A/B* where *A* and *B* are polynomials including some programs with outputs of numerical type. If Q is a program returning a numerical value, while *C* and *D* are polynomials, then as it can be readily seen, the program $R = \frac{Q * A + C}{Q * B + D}$ is a GT-derivative of P. Indeed, if a program can be represented as a polynomial depending on programs of type **N**, then the coefficients of this polynomial are treated as inputs of the program. Making all the coefficients of C and D equal to zero we obtain an obvious identity R = P.

2. If we multiply any term in *A* or *B* by a construction IF(Q, *a, b*), where Q is a program returning a boolean value, while *a* and *b* are inputs of the type **N**, then the new rational expression will be a GT-derivative of P. This is true because if $a = b = 1$ then IF(Q, *a, b*) ≡ 1.

It should be noted that the basic commercially available version of PolyAnalyst utilizes only these two classes of GT.

7. Conclusion

A solid history of successful utilization of PolyAnalyst in various fields including banking, marketing, manufacturing, and many other fields corroborates the efficiency of applying the automated program synthesis techniques to KDD problems. The universality of the described approach is achieved due to the absence of any inherent limitations on the structure of the analyzed data, as well as on the procedure used for evaluating the built programs in accordance with arbitrary criteria implemented in the PolyAnalyst program evaluation module. The GT search and the mechanisms suppressing the generation of trivial and equivalent programs solve, or at least soften the problem of combinatorial growth of the number of the generated programs. The assumption of the SAV data structure allows one to make important simplifications, which increase the performance of the system even further.

References

Backus, J. (1978) Can programming be liberated from the von Neumann style? *Commun. ACM*, v.21, pp 613-541.

Kiselev, M.V. (1994) PolyAnalyst - a machine discovery system inferring functional programs, In: *Proceedings of AAAI Workshop on Knowledge Discovery in Databases'94*, Seattle, pp. 237-249.

Kiselev, M.V., Arseniev, S.B. (1996) Discovery of numerical dependencies in form of rational expressions, in; *Proceedings of ISMIS'96 (Ninth International Symposium on Methodologies for Intelligent Systems) poster session*, Zakopane, Poland, pp. 134-145.

Kiselev, M.V., Ananyan, S. M., and Arseniev, S. B. (1997) Regression-Based Classification Methods and Their Comparison with Decision Tree Algorithms, In: *Proceedings of 1st European Symposium on Principles of Data Mining and Knowledge Discovery*, Trondheim, Norway, Springer, pp 134-144.

Shen Wei-Min (1990) Functional Transformations in AI Discovery Systems, *Artif.Intell.*, v.41, pp 257-272.

Zembowicz, R., Zytkow, J. M. (1992) Discovery of Equations: Experimental Evaluation of Convergence, In: *Proceedings of AAAI-92*, AAAI Press, Menlo Park, CA, pp 70-75.

Zytkow J.M., Zhu J. (1991) Application of Empirical Discovery in Knowledge Acquisition, In: *Proceedings of Machine Learning - EWSL-91*, pp 101-117.

Appendix . Formal definition of production methods of PolyAnalyst internal language

1. Functional composition. A program created using the functional composition is defined by the quadruplet $P_{FC} = <P_{up}, \Pi_{down}, A \subset \mathbf{in}(P_{up}), \mu: A \xrightarrow{\mu} \Pi_{down}>$, where Π_{down} is a set of programs (it must be non-empty) and $DT(\mu(\alpha)) = DT(\alpha)$. A new program P_{FC} has the following syntactic characteristics: $DT(P_{FC}) = DT(P_{up})$, $\mathbf{in}(P_{FC}) = \bigcup_{P \in \Pi_{down}} \mathbf{in}(P) \cup \mathbf{in}(P_{up}) \backslash A$. The semantics of this construction is the following. To determine the value of P_{FC} for given input values the values returned by the programs comprising Π_{down} are calculated. Then every input α of P_{up}, which belongs to A, is assigned the value of $m(\alpha)$ and P_{up} is evaluated. Its output value becomes the output value of P_{FC}.

2. Iteration/recursion. The most general form of this construction is expressed by the following twelve components: $P_{iter} = <P_{pred}, P_{ord}, P_{cond}, \Pi_{act}, \Omega, A_{pred} \subset \mathbf{in}(P_{pred}), A_{ord} \subset \mathbf{in}(P_{ord}), A_{iter} \subset \mathbf{in}(P_{cond}) \cup \bigcup_{P \in \Pi_{act}} \mathbf{in}(P)$, $\mu_{pred}: A_{pred} \xrightarrow{\mu_{pred}} \Omega$, $\mu_{ord}: A_{ord} \xrightarrow{\mu_{ord}} \Omega$, $\mu_{iter}: A_{iter} \xrightarrow{\mu_{iter}} \Omega \cup \Pi_{act}$, ***out*** $\in \Omega \cup \Pi_{act}>$, where P_{xxx} are programs, Π_{act} is a set of programs, and Ω is a set of loop variables. From the syntactic point of view, the loop variables are the objects which have a single attribute - their data type (this type should be enumerable).

The iterative/recursive construction is syntactically correct if the following additional conditions hold: $DT(P_{pred}) = DT(P_{cond}) = \mathbf{L}$, $DT(P_{ord}) = \mathbf{N}$, $DT(m_{xxx}(\alpha)) = DT(\alpha)$ for all m_{xxx}. A special pseudo-program without inputs denoted as $\Im$ may be substituted in place of some components of the considered construction. The output value of this pseudo-program always equals to 1. For example, if $\Omega = \varnothing$, then P_{pred} and P_{ord} should be $\Im$. The prototype of P_{iter} is defined as $DT(P_{iter}) = DT(\boldsymbol{out})$, $\mathbf{in}(P_{iter}) = \bigcup_{P \in \Pi_{act}} \mathbf{in}(P) \cup (\mathbf{in}(P_{pred}) \backslash A_{pred}) \cup (\mathbf{in}(P_{ord}) \backslash A_{ord}) \cup (\mathbf{in}(P_{cond}) \backslash A_{iter})$. The semantics of this construction is determined by the following algorithm for its evaluation. (Note that the mappings m_{xxx} describe the method of passing the values to the inputs of programs included in the construction.)

1. If $\Omega \neq \varnothing$, create a list **LOOPVAR** of all combinations of the possible loop variable values for which the value of P_{pred} equals to 1. The method of passing the values of the loop variables to the inputs of P_{pred} is determined by the mapping m_{pred}.
2. If $\Omega \neq \varnothing$, sort the list **LOOPVAR** in the order of ascending values returned by P_{ord} for the combinations of the loop variable values from **LOOPVAR**. The **LOOPVAR** list can be considered as a matrix $LV[i,\omega]$, where i is the variable value combination number, and ω is the loop variable.
3. $i \leftarrow 1$.
4. Calculate the values of all the programs from the set Π_{act}. The values of their inputs are determined by the following rule. If $m_{iter}(\alpha) \in \Omega$, then the value of the input α equals to $LV[1,m_{iter}(\alpha)]$, otherwise it is equal to the value of the respective input of P_{iter}.
5. Evaluate P_{cond}.
6. If $\Omega \neq \varnothing$ and $i =$ <number of rows of LV>, or the value of P_{cond} equals to 0, stop the computation. Take the value of ***out*** as the value of the whole construction P_{iter}.
7. $i \leftarrow i + 1$.
8. Calculate the values of all programs from the set Π_{act}. The values of their inputs are taken from the loop variables, the outputs of the programs that belong to Π_{act}, or from the inputs of P_{iter} in accordance with the mapping m_{iter}. For example, if $m_{iter}(\alpha) = \omega \in \Omega$, then $LV[i,\omega]$ should be taken as the value of α.
9. Go to step 5.

Passing the values between the components of this production is depicted schematically in Fig. 2. In addition to the described general form, the iteration/recursion production method has several special forms which are not considered here.