Abstract Expression Grammar Symbolic Regression

Michael F. Korns

Korns Associates, 1 Plum Hollow, Henderson, Nevada 89052 USA
mkorns@korns.com.

Abstract. This chapter examines the use of Abstract Expression Grammars to perform the entire Symbolic Regression process without the use of Genetic Programming per se. The techniques explored produce a symbolic regression engine which has absolutely no bloat, which allows total user control of the search space and output formulas, which is faster, and more accurate than the engines produced in our previous papers using Genetic Programming.

The genome is an all vector structure with four chromosomes plus additional epigenetic and constraint vectors, allowing total user control of the search space and the final output formulas. A combination of specialized compiler techniques, genetic algorithms, partical swarm, aged layered populations, plus discrete and continuous differential evolution are used to produce an improved symbolic regression sytem.

Nine base test cases, from the literature, are used to test the improvement in speed and accuracy. The improved results indicate that these techniques move us a big step closer toward future industrial strength symbolic regression systems.

Key words: Abstract Expression Grammars, Differential Evolution, Grammar Template Genetic Programming, Genetic Algorithms, Particle Swarm, Symbolic Regression.

1 Introduction

This chapter examines techniques for improving symbolic regression systems with the aim of achieving entry-level industrial strength. In four previous papers (Korns, 2006), (Korns, 2007), (Korns, 2008), and (Korns, 2009), our pursuit of industrial scale performance with large-scale, symbolic regression problems, required us to reexamine many commonly held beliefs and, of necessity, to borrow a number of techniques from disparate schools of genetic programming and recombine them in ways not normally seen in the published literature. The techniques of abstract expression grammars were developed, but expored only tangentially.

While the techniques, described in detail in (Korns 2009), produce a symbolic regression system of breadth and strength, lack of user control of the search space, bloated unreadable output formulas, accuracy, and slow convergence speed are all issues keeping an industrial strength symbolic regression system tantalizingly out of reach. In this chapter abstract expression grammars become the main focus and are promoted as the sole means of performing symbolic regression. Using the nine base test cases from (Korns, 2007) as a training set, to test for improvements in accuracy, we constructed our symbolic regression system using these important techniques:

- Abstract expression grammars
- Universal abstract goal expression
- Standard single point vector-based mutation
- Standard two point vector-based cross over
- Continuous vector differential evolution
- Discrete vector differential evolution
- Continuous particle swarm evolution
- Pessimal vertical slicing and out-of-sample scoring during training
- Age-layered populations
- User defined epigenetic factors
- User defined constraints

For purposes of comparison, all results in this paper were achieved on two workstation computers, specifically an Intel Core 2 Duo Processor T7200 (2.00GHz/667MHz/4MB) and a Dual-Core AMD Opteron Processor 8214 (2.21GHz), running our Analytic Information Server software generating Lisp agents that compile to use the on-board Intel registers and on-chip vector processing capabilities so as to maximize execution speed, whose details can be found at *www.korns.com/Document_Lisp Language_Guide.html*. Furthermore, our Analytic Information Server is available in an open source software project at *aiserver.sourceforge.net*.

1.1 Testing Regimen and Fitness Measure

Our testing regimen uses only statistical best practices out-of-sample testing techniques. We test each of the nine test cases on matrices of 10000 rows by 5 columns with no noise, and on matrices of 10000 rows by 20 columns with 40% noise, before drawing any performance conclusions. Taking all these combinations together, this creates a total of 18 separate test cases. For each test a training matrix is filled with random numbers between -50 and +50. The target expression for the test case is applied to the training matrix to compute the dependent variable and the required noise is added. The symbolic regression system is trained on the training matrix to produce the regression champion. Following training, a testing matrix is filled with random numbers between -50 and +50. The target expression for the test case is applied to the testing matrix to compute the dependent variable and the required noise is added. The regression champion is evaluated on the testing matrix for all scoring (i.e. out of sample testing).

Our two fitness measures are described in detail in (Korns 2009) and consist of a standard least squared error which is normalized by dividing LSE by the standard deviation of Y (dependent variable). This normalization allows us to meaningfully compare the normalized least squared error (NLSE) between different problems. In addition we construct a fitness measure known as tail classification error, TCE, which measures how well the regression champion classifies the top 10% and bottom 10% of the data set. A TCE score of less than 0.20 is excellent. A TCE score of less than 0.30 is good; while, a TCE of 0.30 or greater is poor.

2 Previous Results on Nine Base Problems

The previously published results (Korns 2009) of training on the nine base training models on 10,000 rows and five columns with no random noise and only 20 generations allowed, are shown below [1].

In general, training time is very reasonable given the difficulty of some of the problems and the limited number of training generations allowed. Average percent error performance varies from excellent to poor with the *linear* and *cubic* problems showing the best performance. Minimal differences between training error and testing error in the *mixed* and *ratio* problems suggest no over-fitting.

Surprisingly, long and short classification is fairly robust in most cases including the very difficult *ratio*, and *mixed* test cases. The salient observation is the relative ease of classification compared to regression even in problems with this much noise. In some of the test cases, testing NLSE is either close to or exceeds the standard deviation of Y (not very good); however, in many of the test cases classification is below 0.20. (very good).

[1] The nine base test cases are described in detail in (Korns, 2007).

Table 1. Result For 10K rows by 5 columns no Random Noise

Test	*Minutes*	*Train-NLSE*	*Train-TCE*	*Test-NLSE*	*Test-TCE*
linear	1	0.00	0.00	0.00	0.00
cubic	1	0.00	0.00	0.00	0.00
cross	145	0.00	0.00	0.00	0.00
elipse	1	0.00	0.00	0.00	0.00
hidden	3	0.00	0.00	0.00	0.00
cyclic	1	0.02	0.00	0.00	0.00
hyper	65	0.17	0.00	0.17	0.00
mixed	233	0.94	0.32	0.95	0.32
ratio	229	0.94	0.33	0.94	0.32

The previously published results (Korns 2009) of training on the nine base training models on 10,000 rows and twenty columns with 40% random noise and only 20 generations allowed, are shown below.

Table 2. Result for 10K rows by 20 columns with 40% Random Noise

Test	*Minutes*	*Train-NLSE*	*Train-TCE*	*Test-NLSE*	*Test-TCE*
linear	82	0.11	0.00	0.11	0.00
cubic	59	0.11	0.00	0.11	0.00
cross	127	0.87	0.25	0.93	0.32
elipse	162	0.42	0.04	0.43	0.04
hidden	210	0.11	0.02	0.11	0.02
cyclic	233	0.39	0.11	0.35	0.12
hyper	163	0.48	0.06	0.50	0.07
mixed	206	0.90	0.27	0.94	0.32
ratio	224	0.90	0.26	0.95	0.33

Clearly the previous symbolic regression system performs most poorly on the test cases *mixed* and *ratio* with conditional target expressions. There is no evidence of over-fitting shown by the minimal differences between training error and testing error. Plus, the testing TCE is relatively good in both *mixed* and *ratio* test cases. Taken together, these scores portray a symbolic regression system which is ready to handle some industrial strength problems except for a few serious issues.

The output formulas are often so bloated, with intron expressions, that they are practically unreadable by humans. This seriously limits the acceptance of the symbolic regression system for many industrial applications. There is no user control of the search space, thus making the system impractical for most specialty applications. And of course we would love to see

additional speed and accuracy improvements because industry is insatiable on those features.

A new architecture which will completely eliminate bloat, allow total user control over the search space and the final output formulas, improve our regression scores on the two conditional base test cases, and deliver an increase in learning speed, is the subject of the remainder of this chapter.

3 New System Architecture

Our new symbolic regression system architecture is based entirely upon an Abstract Expression Grammar foundation. A single abstract expression, called the **goal expression**, defines the search space during each symbolic regression run. The objective of a symbolic regression run is to optimize the goal expression.

An example of a goal expression is: y = **f0**(**c0***x5)+(**f1**(**c1**)/(**v0**+3.14)). As described in detail in (Korns 2009), the expression elements **f0**, **f1**, *, +, and / are abstract and concrete functions(*operators*). The elements **v0**, and x5 are abstract and concrete features. The elements **c0**, **c1**, and 3.14 are abstract and concrete real constants. Since the goal expression is abstract, there are many possible concrete solutions.

- y = **f0**(**c0***x5)+(**f1**(**c1**)/(**v0**+3.14)) (...to be solved...)
- y = **sin**(**-1.45***x5)+(**log**(**22.56**)/(**x4**+3.14)) (...possible solution...)
- y = **exp**(**38.16***x5)+(**tan**(**-8.41**)/(**x0**+3.14)) (...possible solution...)
- y = **square**(**-0.16***x5)+(**cos**(**317.1**)/(**x9**+3.14)) (...possible solution...)

The objective of symbolic regression is to find an *optimal* concrete solution to the abstract goal expression. In our system architecture, each individual solution to the goal expression is implemented as a set of vectors containing the solution values for each abstract function, feature, and constant present in the goal expression. This allows the symbolic regression system to be based upon an all vector genome which is convenient for genetic algorithm, particle swarm, and differential evolution styled population operators. In addition to the regular vector chromosomes providing solutions to the goal expression, epigenetic wrappers and constraint vectors provide an important degree of control over the search process and will be discussed in detail later in this chapter. Taken all together our new symbolic regression system is based upon the following genome.

- Genome with four chromosome vectors
- Each chromosome has an epigenetic wrapper
- There are two user contraint vectors

The new system is constructed using these important techniques.

- Universal abstract goal expression
- Standard single point vector-based mutation
- Standard two point vector-based cross over
- Continuous vector differential evolution
- Discrete vector differential evolution
- Continuous particle swarm evolution
- Pessimal vertical slicing and out-of-sample scoring during training
- Age-layered populations
- User defined epigenetic factors
- User defined constraints

The *universal* abstract goal expression allows the system to be used for general symbolic regression and will be discussed in detail later in this chapter. Both single point vector-based mutation and two point vector-based cross over are discussed in (Man 1999). Continuous and discrete vector differential evolution are discussed in (Price 2005). Continuous particle swarm evolution is discussed in (Eberhardt 2001). Pessimal vertical slicing is discussed in (Korns 2009). Age-layered populations are discussed in (Hornby 2006) and (Korns 2009). User defined epigenetic factors and user defined constraints will be discussed in detail later in this chapter.

However, before proceeding to discuss the details of the system implemenation, we will review abstract expression grammars as discussed in detail in (Korns 2009).

3.1 Review of Abstract Expression Grammars

The simple concrete expression grammar we use in our symbolic regression system is a C-like functional grammar with the following basic elements.

- **Real Numbers**: 3.45, -.0982, 100.389, and *all other real constants.*
- **Row Features**: x1, x2, x9, and *all other features.*
- **Binary Operators**: +, *, /, %, max(), min(), mod()
- **Unary Operators**[1]: sqrt(), square(), cube(), abs(), sign(), sigmoid()
- **Unary Operators**[2]: cos(), sin(), tan(), tanh(), log(), exp()
- **Relational Operators**: <, <=, ==, ! =, >=, >
- **Conditional Operator**: (*expr* < *expr*) **?** *expr* : *expr*)
- **Colon Operator**: *expr* : *expr*
- **noop Operator**: noop()

Our numeric expressions are C-like containing the elements shown above and surrounded by regression commands such as, **regress()**, **svm()**, etc. Currently we support univariate regression, multivariate regression, and support

vector regression. Our conditional expression operator (...) **?** (...) **:** (...) is the C-like conditional operator where the ? and : operators always come in tandem. Our **noop** operator is an idempotent which simply returns its first argument regardless of the number of arguments: noop(x7,x6/2.1) = x7. Our basic expression grammar is functional in nature, therefore all operators are viewed grammatically as function calls. Our symbolic regression system creates its regression champion using evolution; but, the final regression champion will be a compilation of a basic concrete expression such as:

- (*E1*): f = (log(x3)/sin(x2*45.3))>x4 ? tan(x6) : cos(x3)

Computing an NLSE score for f requires only a single pass over every row of X and results in an attribute being added to f by executing the score method compiled into f as follows.

- f.NLSE = f.score(X,Y).

Suppose that we are satisfied with the form of the expression in (E1); but, we are not sure that the real constant 45.3 is optimal. We can enhance our symbolic regression system with the ability to optimize individual real constants by adding abstract constant rules to our built-in algebraic expression grammar.

- **Abstract Constants**: c1, c2, and c10

Abstract constants represent placeholders for real numbers which are to be optimized by the symbolic regression system. To further optimize f we would alter the expression in (E1) as follows.

- (*E2*): f = (log(x3)/sin(x2*__c1__))>x4 ? tan(x6) : cos(x3)

The compiler adds a new real number vector, C, attribute to f such that f.C has as many elements as there are abstract constants in (E2). Optimizing this version of f requires that the built-in score method compiled into f be changed from a single pass to a multiple pass algorithm in which the real number values in the abstract constant vector, f.C, are iterated until the expression in (E2) produces an optimized NLSE. This new score method has the side effect that executing f.score(X,Y) also alters the abstract constant vector, f.C, to optimal real number choices. Clearly the particle swarm (Eberhardt 2001) and differential evolution algorithms provide excellent candidate algorithms for optimizing f.C and they can easily be compiled into f.score by common compilation techniques currently in the main stream. Summarizing, we have a new grammar term, c1, which is a reference to the 1st element of the real number vector, f.C (in C language syntax c1 == f.C[1]). The f.C vector is

optimized by scoring f, then altering the values in f.C, then repeating the process iteratively until an optimum NLSE is achieved. For instance, if the regression champion agent in (E2) is optimized with:

- f.C == < 45.396 >

Then the optimized regression champion agent in (E2) has a concrete conversion counterpart as follows:

- f = (log(x3)/sin(x2*__45.396__))>x4 ? tan(x6) : cos(x3)

Suppose that we are satisfied with the form of the expression in (E1); but, we are not sure that the features, x2, x3, and x6, are optimal choices. We can enhance our symbolic regression system with the ability to optimize individual features by adding abstract feature rules to our built-in algebraic expression grammar.

- **Abstract Features**: v1, v2, and v10

Abstract features represent placeholders for features which are to be optimized by the nonlinear regression system. To further optimize f we would alter the expression in (E1) as follows.

- (*E3*): f = (log(**v1**)/sin(**v2***45.3))>**v3** ? tan(**v4**) : cos(**v1**)

The compiler adds a new integer vector, V, attribute to f such that f.V has as many elements as there are abstract features in (E3). Each integer element in the f.V vector is constrained between 1 and M, and represents a choice of feature (in x). Optimizing this version of f requires that the built-in score method compiled into f be changed from a single pass to a multiple pass algorithm in which the integer values in the abstract feature vector, f.V, are iterated until the expression in (E3) produces an optimized NLSE. This new score method has the side effect that executing f.score(X,Y) also alters the abstract feature vector, f.V, to integer choices selecting optimal features (in x). Clearly the genetic algorithm (Man 1999), discrete particle swarm (Eberhardt 2001), and discrete differential evolution (Price 2005) algorithms provide excellent candidate algorithms for optimizing f.V and they can easily be compiled into f.score by common compilation techniques currently in the main stream. The f.V vector is optimized by scoring f, then altering the values in f.V, then repeating the process iteratively until an optimum NLSE is achieved. For instance, the regression champion agent in (E3) is optimized with:

- f.V == < 2, 4, 1, 6 >

Then the optimized regression champion agent in (E3) has a concrete conversion counterpart as follows:

- f = (log(**x2**)/sin(**x4***45.396))>**x1** ? tan(**x6**) : cos(**x2**)

Similarly, we can enhance our nonlinear regression system with the ability to optimize individual functions by adding abstract functions rules to our built-in algebraic expression grammar.

- **Abstract Functions**: f1, f2, and f10

Abstract functions represent placeholders for built-in functions which are to be optimized by the nonlinear regression system. To further optimize f we would alter the expression in (E2) as follows.

- (*E4*): f = (**f1**(x3)/**f2**(x2*45.3))>x4 ? **f3**(x6) : **f4**(x3)

The compiler adds a new integer vector, F, attribute to f such that f.F has as many elements as there are abstract features in (E4). Each integer element in the f.F vector is constrained between 1 and (number of built-in functions available in the expression grammar), and represents a choice of built-in function. Optimizing this version of f requires that the built-in score method compiled into f be changed from a single pass to a multiple pass algorithm in which the integer values in the abstract function vector, f.F, are iterated until the expression in (E4) produces an optimized NLSE. This new score method has the side effect that executing f.score(X,Y) also alters the abstract function vector, f.F, to integer choices selecting optimal built-in functions. Clearly the genetic algorithm (Man 1999), discrete particle swarm (Eberhardt 2001), and discrete differential evolution (Price 2005) algorithms provide excellent candidate algorithms for optimizing f.F and they can easily be compiled into f.score by common compilation techniques currently in the main stream. Summarizing, we have a new grammar term, f1, which is an indirect function reference thru to the 1st element of the integer vector, f.F (in C language syntax f1 == funtionList[f.F[1]]). The f.F vector is optimized by scoring f, then altering the values in f.F, then repeating the process iteratively until an optimum NLSE is achieved. For instance, if the valid function list in the expression grammar is

- f.functionList = < log, sin, cos, tan, max, min, avg, cube, sqrt >

And the regression champion agent in (E4) is optimized with:

- f.F = < 1, 8, 2, 4 >

Then the optimized regression champion agent in (E4) has a concrete conversion counterpart as follows:

- f = (**log**(x3)/**cube**(x2*45.3))>x4 ? **sin**(x6) : **tan**(x3)

The built-in function argument arity issue is easily resolved by having each built-in function ignore any excess arguments and substitute defaults for any missing arguments.

Finally, we can enhance our nonlinear regression system with the ability to optimize either features or constants by adding abstract term rules to our built-in algebraic expression grammar.

- **Abstract Terms**: t1, t2, and t10

Abstract terms represent placeholders for either abstract features or constants which are to be optimized by the nonlinear regression system. To further optimize f we would alter the expression in (E2) as follows.

- (*E5*): f = (log(**t0**)/sin(**t1***t**2**))>**t3** ? tan(**t4**) : cos(**t5**)

The compiler adds a new binary vector, T, attribute to f such that f.T has as many elements as there are abstract terms in (E5). Each binary element in the f.T vector is either 0 or 1, and represents a choice of abstract feature or abstract constant. Adding abstract terms allows the sytem to construct a universal formula containing all possible concrete formulas. Additional details on Abstract Expression Grammars can be found in (Korns 2009).

4 Universal Abstract Expressions

A general nonlinear regression system accepts an input matrix, X, of N rows and M columns and a dependent variable vector, Y, of length N. The dependent vector Y is related to X thru the (quite possibly nonlinear) transformation function, Q, as follows: Y[n] = Q(X[n]). The nonlinear transformation function, Q, can be related to linear regression systems, without loss of generality, as follows. Given an N rows by M columns matrix X (independent variables), an N vector Y (dependent variable), and a K+1 vector of coefficients, the nonlinear transformation, Q, is a system of K transformations, Q_k : $(R_1 x R_2 x ... R_M) -> R$, such that $y = C_0 + (C_1 * Q_1(X)) + ... (C_K * Q_K(X)) + err$ minimizes the normalized least squared error.

Obviously, in this formalization, a nonlinear regression system is a linear regression system which searches for a set of K suitable transformations which minimize the normalized least squared error. If K is equal to M, then Q is

dimensional, and Q is *covering* if, for every m in M, there is at least one instance of X_m in at least one term Q_k.

With reference to our system architecture, what is needed to implement general nonlinear regression, in this formalization, is a method of constructing a universal goal expression which contains all possible nonlinear transformations up to a pre-specified complexity level. Such a method exists and is described as follows.

Given any concrete expression grammar, suitable for nonlinear regression, we can construct a universal abstract goal expression, of an arbitrary grammar node depth level, which contains all possible concrete instance expressions within any one of the K transformations in Q. For instance, the universal abstract expression, U_0, of all Q_k of depth level 0 is **t0**. Remember that t0 is either v0 or c0. The universal abstract expression, U_1, of all Q_k of depth level 1 is f0(t0,t1). In general we have the following.

- $\mathbf{U}_0$: t0
- $\mathbf{U}_1$: f0(t0,t1)
- $\mathbf{U}_2$: f0(f1(t0,t1),f2(t2,t3))
- $\mathbf{U}_3$: f0(f1(f2(t0,t1),f3(t2,t3)),f4(f5(t4,t5),f6(t6,t7)))
- $\mathbf{U}_k$: f0(U_{k-1},U_{k-1})

Given any suitable functional grammar with features, constants, and operators, we add a special operator, **noop**, which simply returns its first argument. This allows any universal expression to contain all smaller concrete expressions. For instance, if f0 = noop, then f0(t0,t1) = t0. We solve the arity problem for unary operators by altering them to ignore the rightmost arguments, for binary operators by altering them to substitute default arguments for missing rightmost arguments, and for N-ary operators by wrapping the additional arguments onto lower levels of the binary expression using appropriate context sensitive grammar rules. For example, let's see how we can wrap the 4-ary conditional function(operator) **?** onto multiple grammar node levels using context sensitive constraints.

- y = **f0**(**f1**(*expr*,*expr*),**f2**(*expr*,*expr*))

Clearly if, during evolution in any concrete solution, the abstract function **f0** were to represent the **?** conditional function, then the abstract function **f1** would be restricted to one of the relational functions(operators), and the abstract function **f2** would be restricted to only the colon function(operator). Therefore one would have any number of possible solutions to the goal expression, but some of the possible solutions would violate these context sensitive constraints and would be unreasonable. The assertion that certain possible solutions are *unreasonable* depends upon the violation of context sensitive constraints implicit with each operator as follows.

- y = **f0**(**f1**(*expr*,*expr*),**f2**(*expr*,*expr*)) (goal expression)
- y = **?**(<(*expr*,*expr*),**:**(*expr*,*expr*)) (reasonable solution)
- y = **?**(**max**(*expr*,*expr*),**mod**(*expr*,*expr*)) (unreasonable solution)
- y = **?**(+(*expr*,*expr*),**:**(*expr*,*expr*)) (unreasonable solution)
- y = +(**mod**(*expr*,*expr*),/(*expr*,*expr*)) (reasonable solution)
- y = +(**mod**(*expr*,*expr*),**:**(*expr*,*expr*)) (unreasonable solution)

Applying our system architecture to solve the problem of general nonlinear regression absolutely requires the implementation of context sensitive grammar rules to keep the various concrete solutions *reasonable* during the evolution process. This unavoidable mathematical property of unrestricted nonlinear regression transformations requires us to extend the genome to include context sensitive contraints. Since the genome must be extended to include context sensitive constraints, we use this opportunity to extend the genome to give much greater implicit and explicit user control of the search process.

In our new system architecture, the genome is extended such that each genome has both epigenetic and constraint wrapper vectors which, in addition to enforcing appropriate context sensitive grammar rules, can be promoted to give the user much greater implicit and explicit control of the search space. Control of the search space will become a very important aspect of future nonlinear regression systems and will be discussed in detail later in this chapter.

5 Constraints

In order to perform general symbolic regression with a universal abstract goal expression, the genome must be context sensitive. This implies that for some solutions of the abstract goal expression, certain choices of concrete features, concrete real numbers, or concrete functions are unreasonable. Consider the following goal expression: y = **f0**(**f1**(*expr*,*expr*),**f2**(*expr*,*expr*)). If we have no additional information about any particular solution to this goal expression, then we must assume that the constraints for abstract functions f0, f1, and f2 are as follows (i.e. unconstrained).

- *constraints*: **f0**(+ * / % max min mod sqrt square cube abs sign sigmoid cos sin tan tanh log exp < <= == != >= > ? : noop)
- *constraints*: **f1**(+ * / % max min mod sqrt square cube abs sign sigmoid cos sin tan tanh log exp < <= == != >= > ? : noop)
- *constraints*: **f2**(+ * / % max min mod sqrt square cube abs sign sigmoid cos sin tan tanh log exp < <= == != >= > ? : noop)

However if we know that a particular solution has selected **f0** to be the operator **?**, then we must implicitly assume that the constraints for abstract functions f0, f1, and f2, *with respect to that solution* are as follows.

- *constraints*: **f0**(?)
- *constraints*: **f1**(< <= == != >= >)
- *constraints*: **f2**(:)

In the goal expression genome, **f0** is a single gene located in position zero in the chromosome for abstract functions. The constraints are wrapped around each chomosome and are a vector of *reasonable* choices for each gene. In a context insensitive genome, chosing any specific value for gene f0 or gene v6, etc. has no effect on the contraint wrappers in the genome. However, in a context sensitive genome, chosing any specific value for gene f0 or gene v6, etc. may have an effect on the contraint wrappers in the genome. Furthermore, we are not limited to implicit control of the genome's contraint wrappers. We can extend control of the genome's contraints to the user in an effort to allow greater control of the search space. For instance, if the user wanted to perform a univariate regression on a problem with ten features but desired only logrithmic transforms in the output, the following abstract goal expression would be appropriate.

- y = **f0**(**v0**) where f0(cos sin tan tanh)

Publishing the genome's contraints for explicit user guidance is an attempt to explore greater user control of the search space during the evolutionary process.

6 Epigenome

In order to perform symbolic regression with a single abstract goal expression, all of the individual solutions must have the same shape genome. In a context insensitive architecture with only one population island performing only a general search strategy, this is not an issue. However, if we wish to perform symbolic regression, with a single abstract goal expression, on multiple population islands each searching a different part of the problem space, then we have to be more sophisticated in our approach.

We have already seen how constraints can be used to control, both implicitly and explicitly, evolutionary choices within a single gene. But what if we wish to influence which genes are chosen for exploration during the evolutionary process? Then we must provide some mechanism for choosing which genes are to be chosen and which genes are not to be chosen for exploration.

Purely arbitrarily and in the sole interest of keeping faith with the original biological motivation of genetic algorithms, we choose to call genes which are chosen for exploration during evolution as *expressed* and genes which are chosen NOT to be explored during evolution as *unexpressed*. Furthermore, the wrapper around each chomosome, which determines which genes are and are not expressed, we call the **epigenome**.

Once again, consider the following goal expression.

- **regress**(**f0**(**f1**(*expr*,*expr*),**f2**(*expr*,*expr*))) where f0(?)

Since we know that the user has requsted only solutions where **f0** has selected to be the operator **?**, then we must implicitly assume that the constraints and epigenome for abstract functions f0, f1, and f2, *with respect to any solution* are as follows.

- *constraints*: **f0**(?)
- *constraints*: **f1**(< <= == ! = >= >)
- *constraints*: **f2**(:)
- *epigenome*: **ef**(f1)

We can assume the epigenome is limited to function f1 because, with both gene f0 and gene f2 constrained to a single choice each, f0 and f2 are implicitly no longer allowed to vary during evolution, *with respect to any solution*. Effectively both f0 and f2 are *unexpressed*.

In the goal expression genome, **ef** is the epigenome associated with the chromosome for abstract functions. The epigenomes are wrapped around each chomosome and are a vector of *expressed* genes. In a context insensitive genome, chosing any specific value for gene f0 or gene v6, etc. has no effect on the contraint wrappers or the epigenome. However, in a context sensitive genome, chosing any specific value for gene f0 or gene v6, etc. may have an effect on the contraint wrappers and the epigenome. Of course, we are not limited to implicit control of the epigenome. We can extend control of the epigenome to the user in an effort to allow greater control of the search space. For instance, the following goal expression is an example of a user specified epigenome.

- (*E6*): **regress**(f0(f1(f2(v0,v1),f3(v2,v3)),f4(f5(v4,v5),f6(v6,v7))))
- (*E6.1*): **where** {}
- (*E6.2*): **where** {ff(noop) f2(cos sin tan tanh) ef(f2) ev(v0)}

Obviously expression (E6) has only one genome; however, the two **where** clauses request two distinct simultaneous search strategies. The first where clause (E6.1) tells the system to perform an unconstrained general search of all possible solutions. The second where clause (E6.2) tells the system to simultaneously perform a more complex search among a limited set of possible solutions as follows. The *ff(noop)* condition tells the system to initialize all functions to noop unless otherwise specified. The *f2(cos sin tan tanh)* condition tells the system to restrict abstract function f2 to only the trigonometric functions starting with cos. The *ef(f2)* epigenome tells the system that only f2 will participate in the evolutionary process. The *ev(v0)* epigenome tells the

system that only v0 will participate in the evolutionary process. Therefore, (E6.2) causes the system to evolve only solutions of a single trignonometric function on a single feature i.e. tan(x4), cos(x0), etc. These two distinct search strategies are explored simultaneously. The resulting champion will be the winning (*optimal*) solution across all simultaneous search strategies.

7 Control

The user community is increasingly demanding better control of the search space and better control of the output from symbolic regression systems. In search of a control paradigm for symbolic regression, we have chosen to notice the relationship of SQL to database searches. Originally database searches where highly constrained and heavily dictated by the choice of storage mechanism. With the advent of relational databases, searches became increasingly under user control to the point that modern SQL is amazingly flexible.

An unanswered research question is how much user control of the symbolic regression process can be reasonably achieved? Our system architecture allows us to use abstract goal expressions to better explore the possibilities for user control. Given the immense value of search space reduction and search specialization, the symbolic regression system can benefit greatly if the epigenome and the constraints are made available to the user. This allows the user to specify goal formulas and candidate individuals which are tailored to specific applications. For instance, the following univariate abstract goal expression is a case in point.

- (*E7*): **regress**(f0(f1(f2(v0,v1),f3(v2,v3)),f4(f5(v4,v5),f6(v6,v7))))
- (*E7.1*): **where** {}
- (*E7.2*): **where** {ff(noop) f2(cos sin tan tanh) ef(f2) ev(v0)}
- (*E7.3*): **where** {ff(noop) f1(noop,*) f2(*) ef(f1) ev(v0,v1,v2)}
- (*E7.4*): **where** {ff(noop) f0(cos sin tan tanh) f1(noop,*) f2(*) ef(f0,f1) ev(v0,v1,v2)}
- (*E7.5*): **where** {f0(?) f4(:)}

Expression (E7) has only one genome and can be entered as a single goal expression requesting five distinct simultaneous search strategies. Borrowing a term from chess playing programs, we can create an *opening book* by adding where clauses like (E7.2), (E7.3), (E7.4), and (E7.5).

The first where clause (E7.1) tells the system to perform an unconstrained general search of all possible solutions.

The second where clause (E7.2) tells the system to evolve only solutions of a single trignonometric function on a single feature i.e. tan(x4), cos(x0), etc.

In the third where clause (E7.3), the *f1(noop,*)* condition tells the system to restrict abstract function f1 to only the noop and * starting with noop. The *f2(*)* condition tells the system to restrict abstract function f2 to only

the * function. The *ef(f1)* epigenome tells the system that only f1 will participate in the evolutionary process. The *ev(v0,v1,v2)* epigenome tells the system that only v0, v1, and v2 will participate in the evolutionary process. Therefore, (E7.3) causes the system to evolve champions of a pair or a triple cross correlations only i.e. (x3*x1) or (x1*x4*x2).

In the fourth where clause (E7.4), the *ff(noop)* condition tells the system to initialize all functions to noop unless otherwise specified. The *f0(cos sin tan tanh)* condition tells the system to restrict abstract function f0 to only the trigonometric functions starting with cos. The *f1(noop,*)* condition tells the system to restrict abstract function f1 to only the noop and * starting with noop. The *f2(*)* condition tells the system to restrict abstract function f2 to only the * function. The *ef(f0,f1)* epigenome tells the system that only f0 and f1 will participate in the evolutionary process. The *ev(v0,v1,v2)* epigenome tells the system that only v0, v1, and v2 will participate in the evolutionary process. Therefore, (E7.4) causes the system to evolve champions of a single trignonometric function operating on a pair or triple cross correlation only i.e. cos(x3*x1) or tan(x1*x4*x2).

In the fifth where clause (E7.5), causes the system to evolve only conditional champions i.e. ((x3*x1)<cos(x5)) ? tan(x1*x4) : log(x0).

These five distinct search strategies are explored simultaneously. The resulting champion will be the winning (*optimal*) solution across all simultaneous search strategies.

Of course (E7) alone, with no where clauses, can guide a thorough symbolic regression run; however, assuming there are five features in the problem (x0,x1,x2,x3, and x4) and the twenty-eight operators of our basic grammar, then (E7.1) is searching a space of $(5^8 * 28^7) = 5.27\text{E+}15$ discrete points. Whereas (E7.2) is searching through only (5*4) = 20 discrete points. Expression (E7.3) is searching through only $(5^3 * 2) = 250$ discrete points. Expression (E7.4) is searching through only $(5^3 * 2)^4 = 3.90625\text{E}9$ discrete points. Allowing user specified constraints and epigenomes can greatly reduce the search space in cases where the application warrants.

Our current abstract regression system supports user specified constraints and epigenomes. For general regression problems, with no user specified where clauses, the system supports an implicit opening book looking for linear, square, cube, and trignonmetric unary functions on single features plus all possible pair or triple cross correlations.

We believe that it should be possible to develop libraries of where clauses useful in specific application areas. Such libaries could be developed, published, and shared between user communities. We believe that we have just scratched the surface on understanding the benefits possible with context sensitive constraints, epigenomes, and opening books.

8 Enhanced Results on Nine Base Problems

We used a feature-terminated universal expression, of depth level three, for both problem sets as shown below. In all cases the system was told to halt when an NLSE of less than .15 was achieved at the end of an epoch. The feature-terminated universal expression, $\mathbf{V}_3$, is specified as shown below. Note that $\mathbf{V}_3$ and $\mathbf{U}_3$ are identical except for their terminators.

- $\mathbf{V}_3$: f0(f1(f2(v0,v1),f3(v2,v3)),f4(f5(v4,v5),f6(v6,v7)))
- $\mathbf{U}_3$: f0(f1(f2(t0,t1),f3(t2,t3)),f4(f5(t4,t5),f6(t6,t7)))

For the five column no noise problems, with twenty-eight operators, five features, and five universal transforms of type $\mathbf{V}_3$ to choose from, there are $(5^8 * 28^7)^5 = 4.06753\text{E+}78$ discrete points in the search space. With twenty-eight operators and twenty features to choose from, the magnitude of the search space for the twenty column problems cannot be expressed in our 64bit computer. It is essentially $(20^8 * 28^7)^{20}$. Of course this still does not take into account the difficulties arising from the 40% added noise.

The enhanced results of training on the nine base training models on 10,000 rows and five columns with no random noise and only 20 generations allowed, are shown below in order of difficulty.

Table 3. Result For 10K rows by 5 columns no Random Noise

Test	*Minutes*	*Train-NLSE*	*Train-TCE*	*Test-NLSE*	*Test-TCE*
linear	1	0.00	0.00	0.00	0.00
cubic	1	0.00	0.00	0.00	0.00
cross	9	0.00	0.00	0.00	0.00
elipse	1	0.00	0.00	0.00	0.00
hidden	1	0.00	0.00	0.00	0.00
cyclic	1	0.00	0.00	0.00	0.00
hyper	1	0.03	0.00	0.03	0.00
mixed	35	0.87	0.26	0.88	0.27
ratio	34	0.87	0.26	0.88	0.27

The enhanced results of training on the nine base training models on 10,000 rows and twenty columns with 40% random noise and only 20 generations allowed, are shown below in order as shown above.

Clearly, in time-constrained training (only 20 generations), the enhanced symbolic regression system is an improvement over the previously published results. While the enhanced system performs poorly on the two test cases *mixed* and *ratio* with conditional target expressions, the performance on all other nine base test cases is acceptable. In addition, the testing TCE scores

Table 4. Result for 10K rows by 20 columns with 40% Random Noise

Test	*Minutes*	*Train-NLSE*	*Train-TCE*	*Test-NLSE*	*Test-TCE*
linear	1	0.11	0.00	0.11	0.00
cubic	1	0.11	0.00	0.11	0.00
cross	49	0.83	0.21	0.81	0.20
elipse	1	0.12	0.00	0.12	0.00
hidden	1	0.11	0.02	0.11	0.02
cyclic	1	0.14	0.00	0.14	0.00
hyper	1	0.12	0.00	0.12	0.00
mixed	56	0.90	0.29	0.90	0.30
ratio	59	0.90	0.29	0.90	0.30

indicate that we can perform some useful classification even in the difficult conditional problems with noise added.

Taken together with the absence of bloat and increased user control of the search space, these results portray a symbolic regression system which is ready to handle many industrial strength problems.

9 Summary

The use of abstract grammars in symbolic regression moves the entire dicipline much closer to *industrial ready* for many applications.

First, we have a formalization which clearly emphasizes our value added as a search algorithm for finding nonlinear transformations. We are no longer cast in a competitive role against univariate regression, multivariate regression, support vector regression, etc. In fact we enhance these regression techniques with our nonlinear search capabilities. For instance, this formalization gives us the opportunity to partner with regression professionals, who have a large body of well thought out statistics for choosing one multivariate model over another. The situation is similar with the support vector community. I believe that these opportunities for cross disciplinary work should be encouraged.

Second, we no longer have a bloat problem of any kind. Further experimentation with context sensitive constraints and epigenomes will improve the symbolic regression process from the user's perspective. Effectively, with bloat, symbolic regression is a black box tool because the resulting expression is practically unreadable by users. However, with user specified goal expressions, constraints, and epigenomes, the symbolic regression process can become effectively white box. From an industrial perspective, a white box tool is far preferable to a black box tool.

Third, we now have a much greater degree of user control over the search space and over the form of the output. We have the potential to specify symbolic regression problems in terms the user can understand which are

specific to the application. Furthermore, using multiple where clauses, the user can be much more sophisticated in specifying search strategy. Opening books of where clauses, useful in specific application areas, can be developed, published, and shared between users.

Financial institutional interest in the field is growing while pure research continues at an aggressive pace. Further applied research in this field is absolutely warranted. We are using our nonlinear regression system in the financial domain. But as new techniques are added and current ones improved, we believe that the system has evolved to be a domain-independent tool that can provide superior regression and classification results for industrial scale nonlinear regression problems.

Clearly we need to experiment with even more techniques which will improve our understanding of constraints and epigenetics. Primary areas for future research should include: experimenting with statistical and other types of analysis to help build conditional WFFs for difficult conditional problems with large amounts of noise; experimenting with opening books for general regression problems, and parallelizing the system on a cloud environment.

References

1. Michael Caplan, Ying Becker (2005). Lessons Learned Using Genetic Programming in a Stock Picking Context, in *Genetic Programming Theory and Practice II*. Springer, New York.
2. Shu-Heng Chen (2002), editor. Genetic Algorithms and Genetic Programming in Computational Finance. Kluwer Academic Publishers, Dordrecht Netherlands.
3. Russell Eberhart, Yuhui Shi, James Kennedy (2001). Swarm Intelligence. Morgan Kaufman, New York.
4. Gregory S Hornby (2006). Age-Layered Population Structure For reducing the Problem of Premature Convergence, in *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM Press, New York.
5. Michael Korns (2006). Large-Scale, Time-Constrained, Symbolic Regression, in *Genetic Programming Theory and Practice IV*. Springer, New York.
6. Michael Korns (2007). Large-Scale, Time-Constrained, Symbolic Regression-classification, in *Genetic Programming Theory and Practice V*. Springer, New York.
7. Michael Korns, Loryfel Nunez (2008). Profiling Symbolic Regression-classification, in *Genetic Programming Theory and Practice VI*. Springer, New York.
8. Michael Korns (2009). Symbolic Regression of conditional target expressions, in *Genetic Programming Theory and Practice VII*. Springer, New York.
9. John R Koza (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge Massachusetts.
10. John R Koza (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. The MIT Press, Cambridge Massachusetts.
11. John R Koza, Forrest H Bennett III, David Andre, Martin A Keane (1999). Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann Publishers, San Francisco California.
12. Riccardo Poli, William Langdon, Nicholas McPhee (2008). A Field Guide to Genetic Programming. LuLu Enterprises.
13. Michael O'Neill, Conor Ryan (2003). Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers, Dordrecht Netherlands.
14. Kenneth Price, Rainer Storn, Jouni Lampinen (2005). Differential Evolution: A Practical Approach to Global Optimization. Springer, New York.
15. Kim-Fung Man, Kit-Sang Tang, Sam Kwong (1999). Genetic Algorithms. Springer, New York.
16. Kenneth Price, Rainer Storn, Jouni Lampinen (2005). Differential Evolution: A Practical Approach to Global Optimization. Springer, New York.