FIFTH™: A Stack Based GP Language for Vector Processing

Kenneth Holladay[1], Kay Robbins[2], and Jeffery von Ronne[2]

[1] Southwest Research Institute, San Antonio, Texas
[2] University of Texas at San Antonio, San Antonio, Texas

Abstract. FIFTH™, a new stack-based genetic programming language, efficiently expresses solutions to a large class of feature recognition problems. This problem class includes mining time-series data, classification of multivariate data, image segmentation, and digital signal processing (DSP). FIFTH is based on FORTH principles. Key features of FIFTH are a single data stack for all data types and support for vectors and matrices as single stack elements. We demonstrate that the language characteristics allow simple and elegant representation of signal processing algorithms while maintaining the rules necessary to automatically evolve stack correct and control flow correct programs. FIFTH supports all essential program architecture constructs such as automatically defined functions, loops, branches, and variable storage. An XML configuration file provides easy selection from a rich set of operators, including domain specific functions such as the Fourier transform (FFT). The fully-distributed FIFTH environment (GPE5) uses CORBA for its underlying process communication.

Keywords: Genetic Programming, vectors, linear GP, GP environment.

1 Introduction

Genetic programming (GP) is a supervised machine learning technique that searches a program space instead of a data space [1]. A crucial factor in the success of a particular GP application is the ability of the underlying GP language (GPL) to efficiently represent solutions in the intended problem domain. For example, Spector [2] observed that PUSH3 was unable to evolve a list sort until the list was stored as an external data structure and list access instructions were added to the GPL. Lack of language expressiveness is particularly acute for large classes of feature extraction problems that arise in many important applications such as time-series data mining and digital signal processing (DSP). Feature extraction and related algorithms are often most compactly expressed using vectors.

A fundamental difficulty in applying traditional GP languages to feature extraction problems is the mismatch between the data handling capacity of the language and the requirements of the applications. When examined from an abstract viewpoint, feature recognition algorithms often contain a series of transformations on vector spaces. Thus, the GP search should explore functions on vector spaces. Unfortunately, common GP languages [2-9], whether stack or tree based, do not treat vectors as single data elements. Even simple vector operations, such as multiplication of a vector by a scalar, require a loop construct with branch and flow control.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 102–113, 2007.
© Springer-Verlag Berlin Heidelberg 2007

While there have been a few published applications of GP to signal processing problems, the recurring theme has been to circumvent native vector handling rather than to increase the expressiveness of the language. For example, to automatically classify sound and image databases, Teller [10] designed a GP system that used special data access primitives with specific bound constraints rather than using generalized vectors. In applying GP to signal processing algorithms, Sharman [11] also avoided vector functions by introducing delay nodes and processing time series data one point at a time. For pattern recognition, Rizki et al. [12] pre-processed the data to reduce dimensionality (feature extraction) prior to applying GP.

In this paper, we introduce a new Turing complete GP programming language called FIFTH with a native vector handling capability that allows functions on vector spaces to be expressed in a compact form. FIFTH is a stack-based language (patterned after FORTH syntax and early FORTH [13] design considerations) whose linear representation supports robust mutation and crossover operations.

The remainder of this paper is organized as follows. Section 2 presents the details of the FIFTH language and discusses its implementation. Section 3 describes the GPE5™ distributed execution environment. Section 4 shows two example problems, and Section 5 discusses the FIFTH design as well as future work.

2 The FIFTH Language

The key features of FIFTH include a syntax that can be easily manipulated, a single data stack that allows vectors to be retained intact and treated as single data elements, and an internal structure that supports rich program control and new word generation as well as the ability to wrap external functions as language primitives. These ingredients are needed for effective search of the program space for vector problems.

2.1 Syntax, Operators, and the Data Stack

A FIFTH program consists of an input stream of tokens separated by white space. Tokens are either words (operators found in a dictionary) or numbers. All operations take their data from a single data stack and leave their results on this stack in contrast to FORTH and PUSH3, which use separate stacks for each data type. The FIFTH stack holds "containers" with two principal attributes: shape and type. Shape refers to the number of dimensions and the number of elements in each dimension of the data in a container. For data types, FIFTH currently supports `NUMERIC` (integer, real, and complex), `STRING` (UTF-8), and `ADDRESS` (container and word address), as well as arrays in one and two dimensions.

All operators are expected to "do the right thing" with whatever container types they find on the stack. This approach simplifies the syntax for complex vector operations and significantly reduces the number of structural words in the language, effectively reducing the search space and simplifying genetic manipulation.

2.2 Core Vocabulary

Table 1 shows some of the more specialized categories of FIFTH words. The last category in the table, Signal Processing, highlights another strong point of FIFTH.

Using a simple set of wrapper functions, it is easy to incorporate special purpose external libraries such as the GNU Scientific Library. While GP implementations have always tailored their vocabulary to fit the problem, the native vector handling in FIFTH greatly extends the range of available options. This also positively affects the search space by allowing researchers to apply domain specific knowledge.

For example, determining the symbol rate of an encoded signal is an important step in blind radio signal recognition. The phase derivative algorithm (DPDT) [14] can be implemented in several hundred lines of C++ code, but can be expressed compactly in FIFTH as:

```
x ANGLE UNWRAP DIFF MAGNITUDE
FFT MAGNITUDE LENGTH 2.0 / SETLENGTH
LENGTH RAMP 1 + LENGTH Fs SWAP / * 30.0 GT
* LENGTH SWAP MAXINDEX SWAP Fs SWAP / 2.0 / *
```

The token `x` is an input vector representing the signal, and `Fs` is the sampling frequency. The first line develops a feature set from the signal vector (x) by taking the first difference of the unwrapped phase angle. The second line uses a Fourier transform to develop the periodicity of the feature set, while the third and fourth lines find the highest spectral line above 30 Hz and convert its position into a symbol rate using the sampling frequency.

Table 1. Example words in the FIFTH vocabulary

Category	Word Set
Stack	`DROP DUP OVER ROT SWAP`
Vector	`MAX MAXINDEX MIN MININDEX ONES ZEROS LENGTH SETLENGTH`
Matrix	`SHAPE HORZCAT VERTCAT TRANSPOSE FLIP`
Definition and Flow	`BEGIN UNTIL IF ELSE THEN WORD ENDWORD VARIABLE CONSTANT`
Signal Processing	`FFT IFFT FFTSHIFT ANGLE UNWRAP dBMAG HAMMING HILBERT MAGNITUDE`

2.3 Formal Aspects and Validation

FIFTH is a type safe language with formal type rules [15]. The type system can provide information to assist the random program generator and genetic manipulator in constructing syntactically and operationally correct programs. Filtering out incorrect programs reduces the search space and improves run-time performance [16].

A selection of important typing rules is shown in Fig. 1. By convention, an arbitrary FIFTH word is represented $\mathcal{W}$, and a sequence of words by $\mathcal{P}$. The type of an individual slot on the stack is expressed with τ, σ, or ρ (with the top of the stack to the right), and sequences of slots are represented with ϕ. The types of FIFTH words are expressed as functions from initial stack typings to new stack typings. The type rules

$$\Delta \vdash \text{ROT} : \tau\ \sigma\ \rho \rightarrow \sigma\ \rho\ \tau$$

$$\textbf{COMPOSE}\ \frac{\Delta \vdash \mathcal{W}: \phi_1 \rightarrow \phi_2 \quad \Delta \vdash \mathcal{P}: \phi_2 \rightarrow \phi_3}{\Delta \vdash \mathcal{W}\ \mathcal{P}: \phi_1 \rightarrow \phi_3}$$

$$\textbf{STACK}\ \frac{\Delta \vdash \mathcal{P}: \phi_1 \rightarrow \phi_2}{\Delta \vdash \mathcal{P}: \theta \cdot \phi_1 \rightarrow \theta \cdot \phi_2}$$

$$\textbf{COMPILE}\ \frac{\begin{array}{c}\Delta, \mathcal{W} : \phi_1 \rightarrow \phi_2 \vdash \mathcal{P}_1 : \phi_1 \rightarrow \phi_2 \\ \Delta, \mathcal{W} : \phi_1 \rightarrow \phi_2 \vdash \mathcal{P}_2 : \phi_3 \rightarrow \phi_4\end{array}}{\Delta \vdash \text{WORD}\ \mathcal{W}\ \mathcal{P}_1\ \text{ENDWORD}\ \mathcal{P}_2 : \phi_3 \rightarrow \phi_4}$$

$$\textbf{TAUT}\ \Delta, \mathcal{W}: \phi_1 \rightarrow \phi_2 \vdash \mathcal{W}: \phi_1 \rightarrow \phi_2$$

Fig. 1. Example basic type rules for FIFTH

assume an environment of mappings from declared words to their type; these are represented with Δ.

The **COMPOSE** rule allows sequences of FIFTH words to be executed as long as each word's ending stack typing matches the beginning stack typing of the next word. The **STACK** rule says that if a sequence has a valid typing, additional slots can be added to the bottom of the stack (as long as the top of the stack matches the signature) to create new valid typings for that sequence. The **COMPILE** rule allows new FIFTH words to be defined and added to the environment. These can then be used in accordance with the **TAUT** rule.

In addition to formal typing, FIFTH includes support for tracing programs and for comparing the execution of an algorithm in FIFTH with execution in MATLAB through `READMAT` and `WRITEMAT` file operations. These operations read and write MATLAB [17] revision 4 binary files and are used for data I/O as well as debugging. We selected the MATLAB format because it is capable of representing vectors using multiple data types including integer, float, and complex representations. These features allow us to implement all or parts of an algorithm in both languages and exchange data for automatic comparison.

3 The FIFTH Genetic Programming Environment

The Genetic Programming Environment for FIFTH (GPE5) consists of three major components, as illustrated in Fig. 2. GP5 provides random program generation, genetic manipulation, program interpretation and parsing, as well as an interactive FIFTH terminal. DEC5 (Distributed Evaluation Controller) manages the distributed evaluation of programs for each generation. One or more DPI5 (Distributed Program

Interpreter) components are required to run the programs against the problem data sets. Each DPI5 is controlled by the DEC5 through CORBA interfaces.

All components are written in C++ and have been tested on Windows XP, Solaris, and Linux operating systems. Where operating system specific interaction is required, the ACE libraries provide an intermediate layer. CORBA communication between the distributed components is based on the TAO [18] libraries.

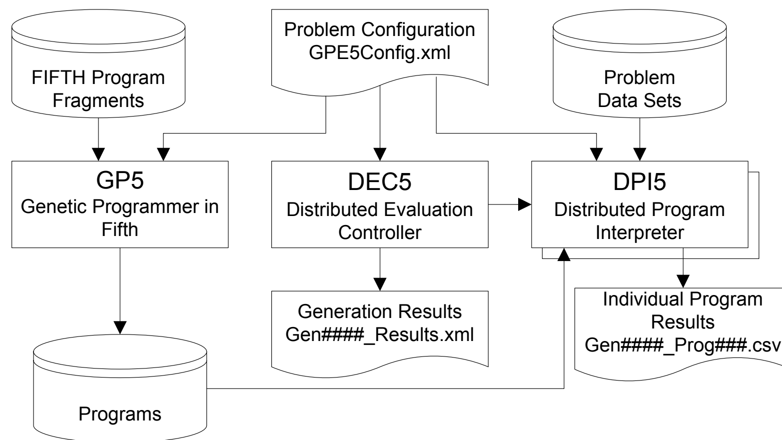

Fig. 2. Block diagram for the FIFTH Genetic Programming Environment (GPE5)

3.1 Random Program Generation

Another way to reduce the search space in GP is to ensure that only syntactically correct programs are allowed into the gene pool. FIFTH was designed so that each word in the dictionary has a stack signature that consists of the number and types of containers expected on the data stack when it is executed and the number and types of containers left on the stack when it finishes.

The random program generator (RPG) uses a two step approach to creating syntactically correct programs with the desired output signature. After first generating a randomly selected fraction of the maximum program size, RPG parses the program to identify errors and fixes them by inserting additional words and terminals.

3.2 Crossover and Mutation in FIFTH

FIFTH supports crossover, mutation, and cloning as standard genetic manipulations, but the infrastructure supports easy addition of any technique that operates on one or two parents. Crossover is performed as follows. First, select two parent programs. In the first parent, select a random number of sequential words in a random location in the program (within configurable constraints) then characterize the input-output effect for the sequence. In the second parent, find all sequences (within configurable constraints) that match the input-output effect then randomly select one. Swapping the two compatible sequences completes the operation. Mutation uses the same initial

sequence selection and characterization, but the sequence is deleted then replaced by a randomly generated sequence that matches the characterization.

A typical problem with mutation and crossover operations in genetic programming is the high probability that the result will destroy the viability of the offspring because the program no longer executes correctly after the operation. Tchernev [19] states that the destructive effects of crossover can be mitigated in FORTH-like GP languages by requiring that the exchanged segments have matching stack signatures. Our implementation of FIFTH uses a stack signature that tracks not only stack size but also branch and loop depth. FIFTH mutation allows a single word or block of words to be replaced by a word or block with a matching stack signature. Crossover operations are similarly constrained. While this choice limits the search space, it increases the probability that the result of evolution will not be destructive.

4 Using GPE5 to Solve a Problem

As with any GP environment, solving a problem using GPE5 requires a few preparatory steps, most of which involve editing the XML configuration file from which all of the components read their control information. The following descriptions are reasonably analogous to the preparatory steps outlined by Koza [20].

4.1 Identify the Terminal Set

In GPE5, there are two classes of terminals: problem data inputs and ephemeral numbers. Problem data inputs may be scalars, vectors, or arrays. For each data set, the values are stored as variables in MATLAB V4 .mat file format. The variable names are in the configuration file for selection during the initial random program generation. An inherent part of this preparatory step is the identification and organization of the data set files. Each data set must be in a separate .mat file. When read into the FIFTH program, each variable in the .mat file becomes a FIFTH CONSTANT. The second category consists of numbers that may be selected during random program generation. In the configuration file, groups of terminal tokens are assigned separate probabilities of selection.

4.2 Identify the Function Set

The function set for GPE5 is divided into three categories controlled by the configuration file. Groups of tokens within each category can be assigned an independent probability of selection. The first category consists of the subset of FIFTH words that are appropriate for the problem under consideration. These may be math operations, logical comparisons, and stack manipulations, as well as domain specific functions such as filters and Fourier transforms.

The second category of functions, unique to GPE5, consists of user user-defined FIFTH program fragments. Each fragment is a valid FIFTH program stored in a file. If a fragment is selected when randomly generating a program, the fragment code is inserted into the program so that it is indistinguishable from the purely random material. In subsequent generations, the fragments are subject to genetic manipulation just like the random parts of the program.

The third category describes the architectural and structural components of a program. These include automatically defined words (WORD, ENDWORD), intermediate storage (VARIABLE, CONSTANT), as well as branch and loop constructs (IF, ELSE, THEN, BEGIN, UNTIL).

4.3 Determine the Fitness Evaluation Strategy

Fitness is a measure of how well a program has learned to predict the output(s) from the input(s). Generally, continuous fitness functions are necessary to achieve good results with GP. While fitness can be any continuous measure, GPE5 uses standardized forms where zero represents a perfect fit. These include error, error squared, and relative error. Although relative error is not often encountered in the GP literature, it works well for problems where the acceptable deviation from the actual output is proportional to the output value.

To avoid overfitting during program evolution, GPE5 uses a different subset of the available data files for each generation. The number of files included in the subset is controlled by a configuration parameter.

4.4 Select the Control Parameters

The GPE5Config.xml file provides a mechanism for experimenting with a number of control parameters including minimum and maximum number of generations, fitness function, fitness target, maximum execution stack depth, and maximum number of words to execute for a single program (to avoid endless loops).

5 Example Problems

5.1 Polynomial Regression

We selected polynomial regression for initial testing of the GPE5 to demonstrate its ability to solve standard GP test problems. Fig. 3 shows an example run for the quadratic equation $y = 0.5x^2 + x/3 - 2.2$. Pertinent configuration parameters were: population size = 1000, minimum program size = 15, maximum program size = 150, parent pool size = 900, and exponential fitness ranking bias = 0.9. The genetic operations included cloning, mutation, and crossover with respective probabilities of 0.05, 0.50 and 0.45. Both mutation and crossover used a uniform length selection of between 1 and 4 program tokens. Table 2 shows the terminal values and functions configured for use by the random program generator.

A typical human generated solution would require about 17 program tokens:

```
1.0 2.0 / x * x * x 3.0 / + 1.0 5.0 / 2.0 + -
```

A longer (43 tokens) but correct general solution was evolved in 33 generations.

```
x x * 1.0  + 1.0  x -2.0  1.0  / - 5.0  - -3.0  -3.0  / -3.0
+ -4.0 - -5.0  / -5.0  + -3.0  / x + -3.0  / OVER - -4.0  / -
- - 2.0  / -4.0  - -5.0  +
```

Table 2. Tokens used by the random program generator for polynomial regression problems

P(selection)	Terminals and Functions in the set
0.20	x (data input)
0.30	1 2 3 4 5 -1 -2 -3 -4 -5
0.40	* + - /
0.10	DUP SWAP OVER ROT

Fig. 3 shows the evolutionary progress. The solid line shows the best fitness at each generation. The dotted line shows the fitness of the best ancestor in each generation for the final program (shown above) that met the fitness termination criteria.

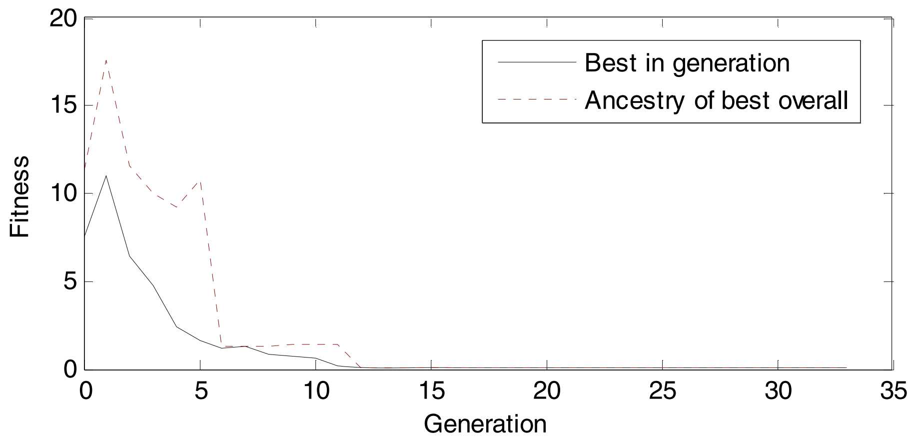

Fig. 3. Best fitness progression for an example run of polynomial regression

5.2 Symbol Rate for Phase Shift Key Modulated Signals

A problem that better illustrates the vector manipulation capabilities of FIFTH and GPE5 is the determination of the symbol rate of phase shift key (PSK) digitally modulated signals. Our previous work [21, 22] compared several common symbol rate algorithms for PSK signals in the High Frequency (HF) radio bands. To prepare a data set for this problem we selected signal property values that are known to work well with the DPDT algorithm presented in section 2.2. Typical values for these properties as well as the specific values used are shown in Table 3. For each unique combination of these properties we generated 10 different signals containing random sequences of symbols. The resulting data set contained 2550 signals. The sample rate was fixed at 8000 complex (In Phase and Quadrature) samples per second, and the signal length was fixed at 16384 samples. Using 1% relative error as the cutoff for a correct symbol rate calculation, the DPDT algorithm achieves 100% correct responses against the entire test data set.

Table 3. Properties of the signals used for the PSK symbol rate example

Property	**Typical Values in HF**	**Values Used**
Modulation	FSK, MSK, PSK, DPSK, OQPSK, QAM, ASK	Phase Shift Key
Pulse shape	None, raised cosine (RC), root RC (RRC), Gaussian	Raised Cosine
Excess bandwidth (rolloff)	Limit: 0.00 to <1.00. Typical: 0.10 to 0.35	0.1
Symbol rate	Typical: 10 to 2400 symbols per second	300 to 2400 in increments of 25
Symbol states	2, 4, 8, 16	2, 4, 8
Signal to noise ratio (SNR)	Practical range: 0 to 60 dB	Infinite (no noise)

A simplified form of this problem evolves only the feature extraction stage as described in [21], followed by standard FFT and peak search stages to derive the symbol rate. With the available vector functions in FIFTH, GPE5 quickly produced modified forms of two common algorithms (phase derivative and magnitude squared) that achieved 100% correct responses.

Evolving the entire symbol rate algorithm has been more challenging. We are in the early phases of experimenting with the wide range of configuration options built into GPE5, but initial results are promising. For example, Fig. 4 shows the best fitness progress for a run using a configuration consisting of population size = 4000, minimum program size = 40, maximum program size = 400, parent pool size = 3500, exponential fitness ranking bias = 0.9, with terminals and functions shown in Table 4. The genetic operations included cloning probability = 0.05, mutation probability = 0.55 with uniform length selection between 1 and 10 tokens, and crossover probability = 0.40 with uniform length selections between 1 and 50 tokens.

The best result from this run achieved a performance of almost 70% correct identification against the entire training set.

Table 4. Tokens used by the random program generator for the symbol rate problem

P(selection)	**Terminals and Functions in the set**
0.10	`x (signal vector)`
0.02	`Fs (sample rate)`
0.13	`1.0 2.0 3.0 4.0 5.0 10.0 -1.0 -2.0`
0.10	`DUP SWAP ROT OVER`
0.05	`LENGTH GT MAX MAXINDEX`
0.25	`ANGLE UNWRAP DIFF MAGNITUDE FFT SQRT REAL IMAGINARY RAMP SETLENGTH MAGSQRD`

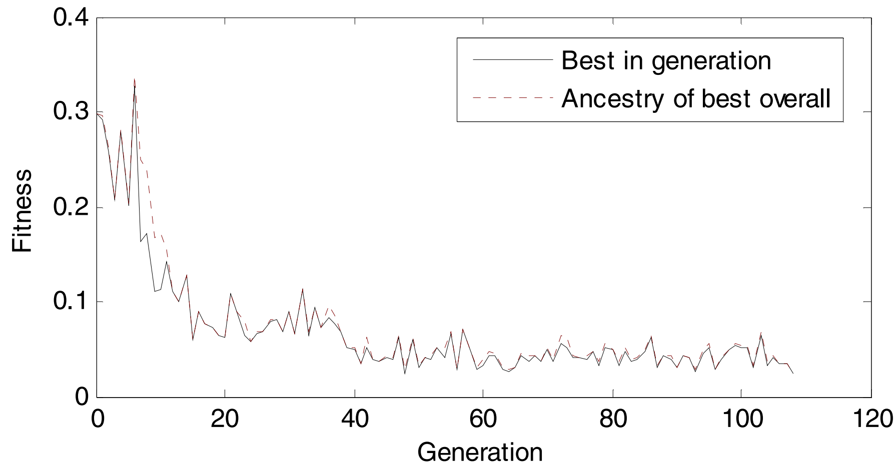

Fig. 4. Best fitness progression for an example run of PSK symbol rate

6 Discussion

The FIFTH programming language was motivated by the need to lift genetic program search to functions on vector spaces by supporting natural, control-free behavior for vector and matrix operations. FIFTH tends to produce mappings from $\mathbb{R}^n \to \mathbb{R}^m$ or in the case of complex values, $\mathbb{C}^n \to \mathbb{C}^m$. These manipulations are natural for a large class of problems including signal processing applications.

Allowing vectors and matrices to be primitive data elements dramatically alters the search space because it eliminates the control operations needed in scalar languages. The probability that such control operations will be damaged during mutation and crossover is high. Even if the control operations are maintained, it is unlikely that intrinsic sizes of vectors will be correctly preserved as successive transformations are performed.

The design philosophy underpinning FIFTH does not mirror the choices that have been made for many linear GPLs and environments. Many authors advocate using function sets and data representation that closely resemble the structure of current CPU hardware, reasoning that this represents a simple structure with significant execution speed advantages. We perceive as a disadvantage to this approach that the resulting programs are almost incomprehensible to humans without significant effort. In addition, for many applications in the feature recognition domain, the performance bottleneck occurs in the execution of matrix manipulation operations such as finding eigenvalues, performing convolutions and filtering, or calculating FFTs. Highly optimized libraries are available for these types of operations. It is highly unlikely that a genetic programming system would ever evolve such complicated operations (optimized or unoptimized) during program evolution. By starting with primitives that reflect the best practices of the domain and using the genetic programming environment as the glue, we are more likely to end up with a working program. In other words, instead of trying to evolve a horse from an amoeba, we try to breed a horse

from a good bloodline. The former process is an example of general evolution, while the latter is an example of special evolution [23].

The details of data handling and support are also critical. Recent publications using stack-based representations have attempted to expand the basic data types available in a GPL by adding separate stacks for each data type. This appears to be an extension of the direction taken by early stack based languages such as FORTH. However, a separate stack design requires adding not only a full set of data manipulation functions for each stack but also functions for data transfer between stacks. As previously indicated, this choice adversely expands the dimensionality of the program search space.

The FIFTH design simplifies the language structure by using a single data stack capable of handling multiple data types. This is accomplished by using a stack that holds containers. Containers provide strong type safety without requiring explicit type annotations or overloaded operations based on explicit data types. Rather, data types are either statically inferred or dynamically tagged. This is similar to the data model found in MATLAB. One reason that MATLAB is popular with signal processing analysts who are not programmers is that variables do not have to be statically declared, and functions automatically determine argument data types at run-time. MATLAB's rich function set together with run-time typing makes the expression of many DSP algorithms both shorter and easier to understand than the same algorithm written in C or C++. Our vision is that similar programs written in FIFTH will be even more compact.

The main disadvantage to this approach is that data handling is more complex since all data, even simple types, must be handled in a structure. Also, since this model is not closely aligned with hardware, there will be more execution overhead.

Our work so far indicates that the FIFTH architecture has the potential of making a large class of vector based feature recognition and signal processing algorithms amenable to a genetic programming approach.

References

1. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming–An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, dpunkt.verlag, San Francisco (1998)
2. Spector, L., Klein, J., Keijzer, M.: The Push3 Execution Stack and the Evolution of Control. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, Vol. 2. ACM SIGEVO (formerly ISGEC) ACM Press New York, NY, 10286-1405, USA, Washington DC, USA (2005) 1689–1696
3. Spector, L., Perry, C., Klein, J., Keijzer, M., Hampshire College School of Cognitive Science, Push 3.0 Programming Language Description. Accessed September 2005, http://hampshire.edu/lspector/push3-description.html (2003)
4. Gagn, C., Parizeau, M.: Open BEAGLE: A New C++ Evolutionary Computation Framework. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. Morgan Kaufmann Publishers San Francisco, CA 94104, USA, New York (2002)
5. Perkis, T.: Stack-Based Genetic Programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence. IEEE Press, Orlando, Florida, USA (1994)
6. Spector, L., Robinson, A.: Genetic Programming and Autoconstructive Evolution with the Push Programming Language. In: Genetic Programming and Evolvable Machines (2002)

7. Tchernev, E.: Stack-Correct Crossover Methods in Genetic Programming. In: Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002). AAAI 445 Burgess Drive, Menlo Park, CA 94025, New York, NY (2002)
8. Silva, S.: GPLAB - A Genetic Programming Toolbox for MATLAB (2005)
9. Discipulus™ Genetic-Programming Software. RML Technologies, Inc. (2004)
10. Teller, A., Veloso, M.: Program Evolution for Data Mining. The International Journal of Expert Systems **8** (1995) 216–236
11. Sharman, K.C., Esparcia-Alcazar, A.I., Li, Y.: Evolving Digital Signal Processing Algorithms by Genetic Programming. In. Faculty of Engineering, Glasgow G12 8QQ, Scotland (1995)
12. Rizki, M.M., Tamburino, L.A.: Evolutionary Computing Applied To Pattern Recognition. In: Koza, J.R., Banzhaf, n.W., Chellapilla, n.K., Deb, n.K., Dorigo, n.M., Fogel, n.D.B., Garzon, n.M.H., Goldberg, n.D.E., Iba, n.H., Riolo, n.R. (eds.): Genetic Programming 1998: Proceedings of the Third Annual Conference. Morgan Kaufmann San Francisco, CA, USA, University of Wisconsin, Madison, Wisconsin, USA (1998) 777–785
13. Rather, E., Bradley, M.: Programming Languages - FORTH (X3J14 dpANS-6). American National Standards Institute, Inc. (1993)
14. Mammone, R.J., Rothaker, R.J., Podilchuk, C.I.: Estimation of carrier frequency, modulation type, and bit rate of an unknown modulated signal. In: IEEE International Conference on Communications, Vol. 2 (1987) 1006–1012
15. Cardelli, L.: Type Systems. In: Tucker, A.B. (ed.): CRC Handbook of Computer Science and Engineering. CRC Press, Boca Raton, FL (2004)
16. Haynes, T.D., Schoenefeld, D.A., Wainwright, R.L.: Type Inheritance in Strongly Typed Genetic Programming. In: Angeline, P.J., nd K. E. Kinnear, Jr. (eds.): Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA (1996) 359–376
17. MathWorks, The Mathworks, Inc., MAT-File Format. www.mathworks.com (2005)
18. Schmidt, D.C., Real-time CORBA with TAO. http://www.cs.wustl.edu/~schmidt/TAO.html (2006)
19. Tchernev, E.B., Phatak, D.S.: Control structures in linear and stack-based Genetic Programming. In: Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference, Seattle, Washington, USA (2004)
20. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
21. Holladay, K., Robbins, K.: A Framework for Automatic Large-Scale Testing and Characterization of Signal Processing Algorithms. In: Military Communications (MILCOM), Monterey, CA (2004)
22. Holladay, K., Robbins, K.: Experimental Analysis of Wavelet Transforms for Estimating PSK Symbol Rate. In: IASTED International Conference on Communication Systems and Applications, Banff, Canada (2004)
23. Kerkut, G.A.: Implications of Evolution. Pergamon Press Inc., New York (1960)