A Survey of Modularity in Genetic Programming

George Gerules
Department of Math and Computer Science
University of Missouri - St. Louis
St. Louis, MO 63121
Email: georgegerules@umsl.edu

Cezary Janikow
Department of Math and Computer Science
University of Missouri - St. Louis
St. Louis, MO 63121
Email: janikowc@umsl.edu

***Abstract*—Here, in this paper, we survey work on modularity in Genetic Programming GP. The motivation for modularity was driven by research efforts, as we shall see, to make GP programs smaller and more efficient. In the literature, modularity has commonly used Koza's term, Automatically Defined Functions ADF. But, we shall see, that the modularity concept has undergone many name and design changes. From the early ideas of Koza and Price's Defined Building Blocks DBB to Binard and Felty's work with System F and GP Briggs and O'Neill's work with Combinators in GP. Our goal in this paper is to survey the literature on this evolution. This will include Automatically Defined Functions ADFs, Automatically Defined Macros ADM, Adaptive Representation Through Learning ARL, Module Acquisition MA, Hierarchically Defined Local Modules HGP, Higher Order Functions using λ calculus LC and Combinators. We also include critiques by researchers on the viability these various efforts.**

I. Introduction

iModularity in GP has been explored in many different ways. It can be roughly categorized into untyped and typed representations. Early work by Koza and Price used untyped ADFs. Some researchers explored modularity in the untyped realm through Automatically Defined Macros ADM [1], Module Acquisition MA [2], Adaptive Representation through Learning ARL and Hierarchical Locally Defined Modules HLDM [3]. While other researchers, inspired by Montana's early work [4], added type systems to their exploration efforts in modularity. These included using Higher Order Functions HOF and methods drawn from the foundations of mathematical logic using λ calculus[5], [6], [7], [8], [9], [10]and combinators[11]. Figure 1 shows the relationship of these various efforts for typed and untyped systems.

There are many similarities between these approaches and many differences. We have four goals with this paper. First, it is our goal to bring together, in one paper, the arc on which untyped and typed ADFs have traveled. Here we use the term ADF in a broad sense when talking about modularity and not Koza's original definition which we will investigate in the next section. Second, our focus has been to include papers that introduce new mechanisms of Automatic Function Definition AFD. Third, we have focused on papers that tend to explain how AFD works. Finally, we have focused on papers where we could obtain implementations.

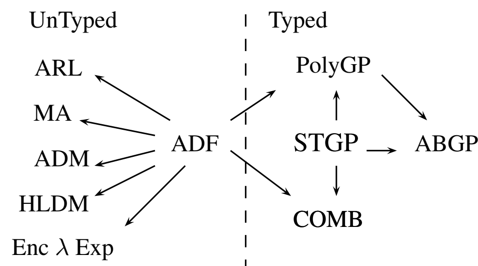


Fig. 1. UnTyped and Typed ADF Systems

II. Untyped AFD Methods

A. Automatically Defined Functions

In this section we take a detailed look at the changing terminology of Koza's work on ADFs. In addition, towards the end of this section we take a look at other early papers on ADFs by Andre and Woodard.

Originally Koza, in a 1990 technical report, calls code reuse Defined Building Blocks DBB. In this report he introduces us to DBB as a way of automatically defining building blocks for a particular GP run. He argues that a programmer will need to solve subproblems in order to solve an overall larger problem. During a GP run a location is chosen in the original tree. That location is pruned on a separate tree and in its location a reference to the newly pruned tree is created. This DBB becomes a zero argument function. The location of the tree removal has a reference to the what is essentially newly created zero argument function. This function label is inserted into the function set. An advantage with this approach is that the newly created function is kept intact and is now immune to potential disruption of the crossover operation.[12]

DBB is discussed again in Koza's first book on GP. Here he makes reference to DBB but now prefers the term Encapsulation Operation EO. What is described as an EO is essentially the same procedure of the DBB in the 1990 report. Later Koza, uses encapsulation to help determine the architecture of a Neural Network via Genetic Programming. Towards the end of the book, he builds a justification for the use of dynamically created functions. It is here that he introduces us to the concept of an ADF. Using the ADF concept he shows how, using the even parity problem, code complexity is reduced through the use of subroutines.[13]

In this part of the first GP book, Koza uses LISP and ADFs on symbolic regression for the even–parity problem. In this portion of his research he uses a LISP `list3` function. The `list3` function for this example problem is comprised of 3 items and its main purpose is to hold together the ADF and the result producing branch RPB. The RPB can call one or more ADF elements. It is the RPB that returns a value that can be compared to a fitness test case. If the returned value from a RPB is close to the fitness test case the overall program is kept around. For this particular problem ADF0 can accept two arguments while ADF1 can accept three arguments. Each ADF and RPB can have their own functions and terminals. None of the ADFs in this section of the book call other ADF entities. Figure II-A shows the overall use of the `list3` function while figure 3 shows how the `list3` function holds together the RPB and the two ADFs, ADF0 and ADF1, which may be used by the RPB to generate a value to be compared against a fitness case value.

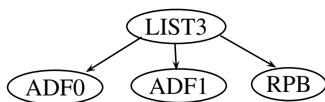


Fig. 2. list3 ADF Overall Structure

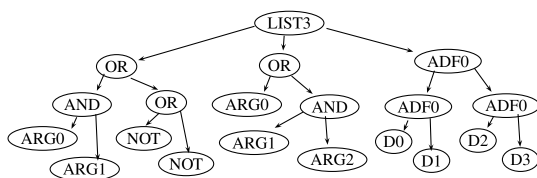


Fig. 3. ADF Use of `list3`

Crossover is constrained so that the root of the tree is not disturbed. Koza terms this a structure preserving crossover.

In later chapters of Koza's first GP book, he discusses Hierarchies of Automatically Defined Functions HADF and their evolution. Here any ADF can call any other ADF to solve a particular problem. In fact an ADF can be composed of other ADFs. So, the RPB may call these hierarchies with its own function and terminal sets made up of ADFs that are composed of other ADFs. When these hierarchies are created, function and terminal sets are carefully constructed as to not cause infinite loops. Symbolic regression and the even parity problem were used as a test bed. As a result, the number of individuals evolved in finding a good solutions is drastically reduced.

Koza's entire second Genetic Programming book GP2 it is devoted to the topic of ADFs.[14] This 26 chapter book explores a wide variety of problem implementations related to ADFs in GP. Early in the book he credits James Rice as the inventor of the idea of ADFs and directs the reader to a patent both of them filed in 1992. In the patent ADFs are called Automatic Function Definitions AFD.[15]

Early in the GP2 book he expands discussion for justification and design (HADFs). He points out the ADFs are particularly useful when there is "regularity" in the problem to be solved. A complex problem may be subdivided in to smaller solvable problems in a divide and conquer manner. It is this identification and grouping of tasks, or regularity as he calls it, is what an ADF can exploit in GP when solving a problem.

Eight claims are provided in GP2 on how ADFs help GP. They can be summarized as follows: 1), ADFs subdivide problems into smaller subproblems. 2), ADFs exploit regularities in the problem space. 3), ADFs solve problems more efficiently with less computation and evaluation. 4), the ending solution generated by a GP system using ADFs are often smaller than a solution generated from a non ADF system. 5), when scaling up problems using ADFs, the intermediate solutions grow in program size at a slower rate than without ADFs. 6), for problems than can be scaled up, computation effort grows slower than without ADFs. 7), for problems that can be scaled up, the solutions generated by a GP system using ADFs is less complex than GP systems without ADFs. 8), a GP using ADFs is capable of evolving the overall architecture of a particular program.

The question of how many ADFs are needed are explored in Chapter 7. Sometimes it is known ahead of time how many ADFs are needed from the overall structure of the problem. But, as explained in this section of the book, many times it is not known, *a priori*, how many ADFs are needed. Three categories of methods are introduced in this chapter. The first category of methods requires manual analysis. There are two kinds of analysis in this category. In the first method we can analyze the problem and guess how many ADFs are needed based on knowledge of the problem domain *prospective analysis*. In the second method we can randomly try different combinations of ADFs on the same GP problem, look at the results and see which one performs better with a particular combination of ADFs. This is *retrospective analysis*. The second category of methods is based on capacity. The issue here is whether there were a sufficiently correct number of ADFs present, *sufficient capacity*, to solve a problem. This also indirectly relates to the third category on whether enough computer resources are available, *affordable capacity*, to handle various configurations of ADFs. This third category involves whether the number of ADFs can be dynamically determined at run time. Chapters (22–25) explore the structure and use of ADFs when they are dynamically generated.

During this early period in the development of ADFs, a few researchers observed a few points of interest are worth noting.

In a paper by Andre, on Map-making, he advocates that ADFs need to be grouped by functionality. ADFs in this paper have constraints placed on which functions and terminals were used. Specifying a function to be only used in a particular ADF forced that ADF to be used. If that ADF changed, through the evolutionary process, it changed in all locations where that ADF was used.[16]

Woodard, in a paper on modularity in genetic programming, introduces us to a number of proofs that state that the minimum

number of primitives needed to define a particular function is independent of the overall number of primitives if we use modularity to the GP problem.[17]

Around this time, because of these results, several research efforts, ADFs started to be explored with other paradigms. These are discussed next.

B. Module Acquisition

In Angeline and Pollack's Module Acquisition MA, has its roots in their work on Evolutionary Algorithms.[18] Here we will focus on their tree representations when discussing MA.

MA works by removing a random branch in the tree. The removed branch is then trimmed to a desired depth, regardless of what the tree contains. Its functionality is frozen and never changes. This operation is called *compression*. See figure 4. There are two types of compression. The first is a *freezing* operation. Once a module is frozen no more compression operations can be performed on it. Furthermore, this type of module does not contain references to other modules. The second is an *atomization* operation. For this type of module, hierarchical references to other modules can exist and. Fitness is measured on the whole program that may use compressed modules.

The authors' note that a lack of diversity can be introduced into the overall population of trees, if only the compression operation is used. So to counteract this they use what they call an *expansion* operation, which is the insertion of a module back into the tree at random locations.

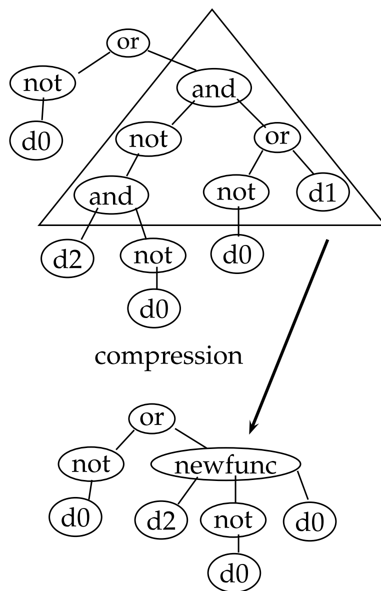


Fig. 4. MA compression operation

The module's parameters are the values that are fed to that portion of the removed tree. The module is kept in a library manager called GLib, based on their earlier research, in Genetic Algorithms, to identify subroutines. [2] GLib, is used to keep track of how many times a particular module is used. But, this number is never used as a fitness measure for the overall fitness function.

The artificial ant and even parity problem where chosen as the benchmark problems to demonstrate this methodology.

In summary, the success of this method is that programs that use these immutable modules, that are created and inserted randomly, will lead to smaller and better programs. It is these programs that contain one or more copies of successful modules will win out over modules with unfit modules. We will see in the next section how several researchers improve on this method.

C. Automatically Defined Macros

Spector introduces us to code reuse with Automatically Defined Macros ADM.[1] In LISP, when a function is called its results are computed in the function body and any results are returned. But, in LISP, there is a macro facility, much like the preprocessor macro facility in the "C" language where the text of the macro body is copied to the location of the macro invocation. So, if a LISP macro was called twice, the entire code that defines the macro body is copied twice and expanded twice at each location of macro invocations.

On the surface it may see that all an ADM does is delay evaluation of a part of code and increase program size. But as Spector points out, ADMs can help in modifying the overall program architecture in ways that a purely functional approach cannot. If a GP problem requires the use of functions that have *side effects* ADMs perform worse than ADMs because the semantics of a particular ADM may not be consistent where a macro expansion happens at a particular site in the GP syntax tree. *Side effects* can happen when a function changes the program state outside of the function body.

For the implementation, changing a Koza type ADF to a Spector type ADM is done by modifying the *fast–eval* code in Koza's LISP original implementation, so it treats a function like a macro. Macro bodies are evolved in a similar fashion to ADF function bodies. Note, that if a ADM is evolved it is not added to the function set and no statistics are kept on the usefulness of a particular ADM. As an aside, Spector mentions that in Zongker's and Punch's *lilgp*, changing an ADF to a ADM is done by replacing the function evaluation type from $EVAL_DATA$ to $EVAL_EXPR$. [19]

Spector does point out that there is an increased evaluation time for all of the expanded macro invocations. This might be a drawback for use of ADMs for some implementations.

The ADMs performed better than ADFs for the Dirt–Sensing and Obstacle–Avoiding robot problem. But the use of ADMs in the Lawn Mower problem performed worse when compared to the same problem using ADMs. Spector also investigated using ADMs on Russell and Norvig's Whumpus World problem where it performed favorably when compare to use of ADFs on the same problem.

D. Hierarchy Locally Defined Modules

In Banzhaf et al. they explore modularity in GP by exploiting a Hierarchy Locally Defined Modules, HLDM.[3] Here, locally defined modules may define other local sub–modules.

In their work they cite previous work by Koza, ADFs, Angeline and Pollack's MA, and Rosca's and Ballard's ARL. They also, cite Gruau's work on using graphs as a basis for cellular encoding for developing neural networks as inspiration for their work. But, when referencing that work, they state that they only use the idea of hierarchical evolution to specify locally aware context sensitive modules.[20]

In their version, hierarchical GP modules are organized into levels. Modules at each level are able to exchange and mutate genetic material only within that level. This concept is also called structure preserving crossover in Koza's work.[13] However in Banzhaf et al., the rate of crossover and exchange lessens at lower levels. Thus, as good modules are created and evolved at lower levels, they won't change as much. Identifying good modules is done by changing a module with a neutral module and comparing the localized fitness. This is based on work done by Rosen his book. The basic idea is that there is a neutral module that doesn't add or detract from the overall fitness of the GP program. Now, if overall fitness of the program is increased when a fitter module is added it must be a good module and we keep it around.[21]

They have two different versions of their HGP system. One being *hGPminor* and the other being *hGP*. For both *hGPminor* and *hGP* there is only one level of modules and the individual calling the module can only call that module once. Also, for both mutation is only allowed on the highest level. Evolution is not allowed for *hGPminor* and is allowed for *hGP*. Evolutionary parameters on *hGP* has two capabilities. One is the capability of having a bad sub tree being replaced by a randomly generated sub tree. The other is setting the crossover probability to a predefined 33%. They report that replacing a bad sub tree with a randomly generated sub tree performed poorly.

For their experiments they chose 4 functions for symbolic regression and 2 different even parity problems. The details of these can be found in table 5 *hGP* performed better than *hGPminor*, which in turn performed better than GP without ADFs.

They report that *hGPminor* out performs standard GP and *hGP* out performs *hGPminor* for all six problems.

Problem	Type	
1	continuous	randomly selected values
2	continuous	steps
3	continuous	$x^6 - 4x^5 - 3x^4 +$ $4x^3 - 2x^2 - x + 4$
4	continuous	$\frac{x^3 - x^2 - x + 3}{x + \frac{5}{9}}$
5	discrete	even–5–parity
6	discrete	even–7–parity

Fig. 5. hGP Test Problems

For their work they created some extensions to the GPC++ 0.4 version framework.

III. TYPED AFD METHODS

Data types were introduced into GP with Montana's paper on Strongly Typed Genetic programming.[4] While his work didn't specifically address ADFs, his work was used as a spring board to constrain the search space for GP. In the next several sections we take a look at data typing mechanisms used in the context of AFD.

A. Adaptive Representation Through Learning

In this section we take a look at Rosca and Ballard's original papers introducing Adaptive Representation through Learning ARL. This is followed by a paper from Dessi et al., analyzing the effectiveness of the ARL method.

Rosca and Ballard introduced ARL in a series of papers in the mid 1990's.[22], [23], [24] They based their work on limitations they noticed in Koza's HGP and Angeline's MA.

First, they point out, in Koza's original HGP, there is no determination of the *intrinsic* value of an evolved subroutine. A subroutine's own fitness is never differentiated from other building blocks of code. The evolved subroutine only returns a computed numerical value which contributes to the overall fitness to a particular problem.

Second, for Angeline and Pollack's MA method, they point out that created routines are never removed, leaving a lot of unfit routines. In addition, they cite work by Kinnear where comparisons are made between Koza's ADFs and Angelines MA methods. In that work ADFs outperform MA for the 4 multiplexer problem when compared to the baseline. In Kinnear's research, ADFs find solutions for a particular GP run much better than GP runs without ADFs. MA doesn't have an effect on finding better solutions when compared to GP runs without ADFs.[25]

In Rosca's and Ballard's work, subroutines, i.e. building blocks, are given their own fitness functions. Each subroutine's fitness function is run against a subset of test cases. If there are any unused variables in a subroutine, the variables are turned into random fixed numbers.

The goal is to dynamically identify new useful functions and add them to the function set of the evolving genetic program. Dynamic identification is done by a predefined, block fitness function. If the population has enough diversity good building blocks will evolve and beat out unfit building blocks.

Statistics are kept on the usefulness of subroutines. These statistics are called complexity measures in their paper. There are three kinds of complexity tracked by their method. *Structural complexity*, is the number of nodes in tree for a subroutine. *Evaluation complexity*, is the number of nodes in tree for a subroutine, plus the number of times calls are made to that routine. *Evaluation complexity*, also keeps track of call hierarchies, in case the global function uses evolved subfunctions. *Description complexity*, using work by Iba et al., is a method using *Minimum Description Length* MDL. To use MDL, a problem can be coded in a minimal way to describe the overall representation of that problem.[26] So, including the first two complexities in the fitness function landscape, their method outperformed HADF, MA when compared a GP

baseline. Here the GP baseline is the 4 even parity problem run without ADFs. They did note that when MDL was used as part of the fitness function an overall GP run was less successful.

Now when a particular GP run is underway, the number of newly created building blocks is tracked. These building blocks are added to the function set. If the number of newly created building blocks is below a threshold, evolution is put on hold while unfit building blocks are removed from individuals and a new population of building blocks is generated. The previous unfit building blocks are replaced with fitter building blocks from the expanded function set. The function set never gets smaller. The replacement can happen upon immediate discovery of new fit building blocks or after some predetermined number of generations have passed where there has been no new building blocks discovered. The number of generations for this to take place is called an *epoch*.

Several test problems were used in their work. A type defined pac–man problem, the 3 and 5 even–parity and odd–parity where chosen. All of which they reported that the ARL approach came with better quality solutions than plain vanilla GP solutions.

The key point for this approach is that typed building blocks have fitness themselves. Next, several researchers propose a number of modifications to the ARL approach.

Dessi et al., analyze the effectiveness of ARL where they propose a number of modifications to the original algorithm. [27] They note that in the original ARL algorithm, the epoch is of fixed length, They propose a variable length to an epoch. The variable length epoch would be adaptively determined through monitoring the diversity of the population. They also propose, using the concept of entropy from Information Theory as a measure of population diversity. This measure and a strategy they call *MaxFit* help guide improvements in the population so that new routines maybe introduced into the population.

For this work, a subroutine is considered to be a subtree of depth between 2 and 4.

A number of heuristics are used to to test the existing ARL algorithms and new algorithms. All of the building block selection methods respect the depth limits of the program trees. These heuristics are briefly described in the next few paragraphs. Their naming scheme for their methods is represented as a change in font in the descriptions that follow.

The RANDOM method selects blocks in the program randomly.

The RANDOMFIT method selects blocks according to either tournament selection or to a fitness proportional selection. This selection is set prior to the run of the program.

The WHOLEPROGFITNESS method selects a block according to the same fitness function used for the entire program.

The FITNESSBLOCK method selects blocks according to the same fitness used for the entire program. For this one on only 60% of the training data was used. The reduced training set was only for the symbolic regression problem and the multiplexer problem. And the selection of the reduced set of input data points was tuned for each problem. The did point out that there was a drawback on the large number of evaluations this added to this method.

The BLOCKACTIVATION method does its selection after crossover. The child is evaluated, if it has a better fitness than at least one of the parents, then that child is a good subroutine and is kept around. Poor performing subroutines will eventually be replaced by better subroutines.

The FREQUENCY method selects a block based on the highest number of times it occurs in the population.

The FREQUENCYPROGRAM method selects a block based on the highest number of times it occurs in a single program.

The SCHEMA method selects a block is according to the average fitness of a block in relation to the average fitness of all programs.

The CORRELATION method selects blocks according to either tournament selection or to a fitness proportional selection. A statistical correlation is performed from the computed output of a program in the tree against subprograms in the tree.

They also, introduce a new selection method called *Saliency*. This method modifies a building block and looks for large scale differences in fitness. If the modification of a subroutine causes a large negative change then the subroutine must have a high semantic relevance to the overall fitness. Saliency is a combination of three equations. These three equations and their relationship can be found below. Equation one measures a subroutine's original output $out(i)$ against the modified output $out'(i)$. N is the number of fitness cases. Previous to invoking this heuristic section of a subroutine is according to either tournament selection or to a fitness proportional selection. For the Saliency algorithm 10% of the population was used. For the Saliency Elitism method 1% of the population was used.

The equation for this approach are below.

$$Sal_{out} = \frac{1}{N} \times \sum_{i=1}^{N} \begin{cases} 0 & \text{if } out(i) = out'(i) \\ 1 & otherwise \end{cases}$$

Equation two, computes the difference in fitness.

$$Sal_{fit} = ABS(f - f')$$

Equation three, combines the two saliency measures into an overall value for saliency.

$$Sal = Sal_{out} \times Sal_{fit}$$

For the non saliency selection methods the authors cite work from Tackett and Kinnear.[28], [29]

Dessi et al. also introduce mutation into their version of ARL. Here they take low fitness subroutines and replace them with randomly created routines. This approach is used in the saliency methods. This, in theory, might generate a subroutine of higher fitness.

For test problems they used the 6 bit multiplexer, symbolic regression, and sorting problems for their experiments. For the symbolic regression routine the function is $f(x, y) = 2 \times x \times x - 3 \times y \times y + 5 \times x \times y - 7 \times x + 11 \times y - 13$. The range of the function $f(x, y)$ from -4 to 4 for both the x and y values. The

equation used for their test problem is a two variable equation where Koza's original equations in his first two books only use one variable.

When discussing the results of their experiments, they compared the previous selection methods against genetic programming without subroutine discovery. Genetic programming without subroutine discovery was called canonical genetic programming *cgp*.

For the symbolic regression none of the selection methods performed better than genetic programming without subroutine discovery. The correlation method was the only method that equaled *cgp*.

For the sort problem, only the RANDOMFIT and FREQUENCY methods performed better than *cgp*.

In the 6 multiplexer problem, the RANDOM selection method and the SALIENCY ELITISM method performed better than *cgp*.

The authors did not investigate whether program size was reduced or whether the number of evaluations was reduced as a result of subroutine discovery.

The C language was used to conduct their research. The framework they developed has many features in common with the Zongker's and Punch's lilgp system. [19]

So to summarize this section, the results are mixed for ARL. None of the test problems overlapped between the two groups, so it might be difficult to draw conclusions when one group is analyzing the original effectiveness of the ARL approach. As a side note, one reason for Dessi's heuristics not performing better than plain vanilla GP might be due to the non commutative nature of the function chosen for an evolutionary target. For an in depth look at how commutative non–commutative functions impact symbolic regression by Janikow and Aleshunas.[30]

B. λ Calculus and Combinatory Logic Introduction

Several researchers use λ Calculus LC and Combinators found in the foundations of mathematics and computer science for GP. In the next few paragraphs we will first describe LC and then Combinators before discussing their use in GP implementations.

LC has its origins in the work of Alonzo Church in 1932. It was revised a year later in 1933. In its original incarnation, he needed a better platform to explore foundations of mathematical logic than the platform of Russell's type theory or Zermelo's set theory. For LC he chose functions as a base concept rather than sets. The symbol λ came about as a modification to Whitehead's and Russell's symbol $\hat{x}$ for class abstraction. [31]

In LC there are two primitive operations, one of application and the other abstraction. Application has to do with applying a function to a function argument. For example $f(5)$. Abstraction is a symbolic expression of what the function does. For example, $\lambda x.x^3$. We will see later how LC is used in GP.

Schönfinkel's first introduced Combinators, in a 1924 paper, but due to health reasons abandoned the topic. Later, Haskell Curry rediscovered them and continued work on Combinators. Combinators are a notational system on how to combine objects. More on how these evolved can be found in Cardone and Hindley's paper and in Seldin's paper.[31], [32]

In Combinatory Logic CL the only primitive application allowed is application. What a function *does* is built up from combination operations, *Combinators*. These combinators are labeled with bold faced capital letters and they denote term rewriting rules. For example the **B** combinator is the composition operator which is defined in terms of other combinators **S** and **K**. So to build the **B** combinator, which is **S(KS**K. The **K** combinator is **KXY** $\triangleright$ **X**. The $\triangleright$ symbol means reduces to the thing on the right. The **S** combinator is defined as **SXYZ**$\triangleright$**XZ(YZ)**. For a full description of this and many other combinators see Seldin's chapter in the following work.[32]

There is a connection between Functional Programming FP, LC and Combinators. The FP can be simplified to LC expressions which are equivalent to Combinators. These Combinators can be a *template* for building functional genetic programs. One of the nice properties of Combinators is that variables are not bound to values.

1) PolyGP System: Functional programming techniques for GP are explored in Yu's thesis where polymorphism, implicit recursion and higher order Functions HOF are used on several bench mark problems.[33], [8]

In the introduction section, she cites two methods of problem solving in the context of programming. The first is *problem decomposition* and the other is *contextual checking*. In problem decomposition, a complex problem is divided into smaller problems until solutions can be programmed for each smaller problem. The entire system system is built from smaller programs. Contextual checking on the other hand is performed on code being created. Yu uses lexical and semantic checking, done by the compiler, on a program to find errors and verify that a program conforms to the grammar of a given language. MA, ARL and MA are all mentioned as precursors for her work using HOFs.

Three techniques, found in functional programing languages, are used in the `PolyGP` system. They are *polymorphism higher order functions* and *implicit recursion*. In functional languages polymorphism is carried out through a type system. This polymorphism is different than the `C++` parameterized types found in template meta programming. A higher order function in a functional language is a function that can take and return functions as a program argument. Implicit recursion is a type of recursion that will always terminate based on input. For example the LISP functions `map`, `foldr`, `foldl` and `nth` functions are functions that use implicit recursion. These functions apply a recursive operation on a finite list of elements. The recursion ends when the list is traversed. Specifically, the `foldr` function works by applying an operation on a list of elements from the right side. Yu points out that in Koza's first book ([13] p97) the genetic programming process, of *create test modify* is similar to the process to carried out by a programmer. In her work she adds functional programming techniques to Koza's GP process, since these methods have helped out programmers

to solve problems that would be hard to solve with other less powerful methods. The `PolyGP` system uses and evolves HOF is based on LC. Subroutines are the actual Lambda Abstractions LA, and the system can accommodate multiple types. Type checking is done via polymorphic type checking system.

Polymorphic functions, in functional programming, is achieved through a type system, since there exist unnamed λ functions in the functional programming paradigm. In the introductory section she points out that in Koza's original implantation of GP, type information is not used. Yu refers to this as "untyped" GP. The search space for "untyped" GP can be very large. Constraints can be placed on the search space in the form of introducing a type system for generating valid GP programs. In order to introduce polymorphism to GP a type system has to be added to GP. This type system is different than Montana's work involving Strongly Typed Genetic Programming STGP, where types are stored in a look up table. [4]

In `PolyGP` the type system determines dynamically what are correct types for functions using a unification algorithm first introduced by Robinson. Robinson unification algorithm originated in the area of computer theorem proving. The algorithm finds out whether two types are the same when they are instantiated. When the unification algorithm completes, it either returns the most general unifier or it returns nothing. The phrase, "most general unifier", refers to the kind of unifier that is returned if there is more than one unifier found. If more than one unifier is found it returns the unifier that is the least specialized. [34], [35]

There are two kinds of grammars at work in the `PolyGP` system to help with polymorphism and the process of generating type correct programs. The first is an *expression syntax*. See figure 6. The second is a *type syntax*. See figure 7. Both of these, in the paper, are expressed in a form similar to functions definition HASKELL language and Backus–Naur Form BNF. [34]

expression ::	c	constant symbols or keywords
	x	identifier
	p	built in primitive operation
	x	application of one expression to another

Fig. 6. Expression Syntax

σ ::	τ	built in type
	υ	type variable
	$\sigma_1 \rightarrow \sigma_2$	function type
	$[\sigma_1]$	list of elements all of type σ_1
	$(\sigma_1 \rightarrow \sigma_2)$	bracketed function type

$\tau :: string|bool|generic_i$

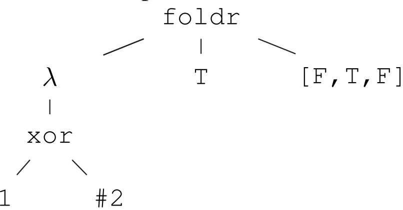
$\upsilon :: dummy_i|temporary_i$

Fig. 7. Type Syntax

The term *structural abstraction* is introduced in an earlier work and used in this research as a name for the patterns that emerge from usage of HOFs.[36] Yu argues that GP simultaneously evolves the *structure* and the *contents*. It is this *meta* level processing by using HOF and the type system the PolyGP system can perform better than systems that use ADFs, MA, or ARL. When solving the even parity problem the HOF `foldr` was chosen to help find good solutions. A boolean `xor` example using the `foldr` function for structural abstraction can be seen next. This example shows how the `foldr` function would evaluate a list of values.[37]

```
foldr (xor) F [T,F,F]
= T or (T or (F or F))
= T or (F or T)
= T or T
= F
```

LA are embedded in the previous example. The program tree for the above example is as follows.



$\#1$ and $\#2$ are program arguments for `foldr`. F is xor'd on each item in the list from right to left. Overall structure of the program is kept intact via the type system. For crossover LA can only crossover with other similar LA.

The `PolyGP` system is made up of a creator, an evolver, an evaluator and a type system. The creator is responsible for generating a population of programs. Each program is a tree which is grown from the top down to a specified depth. The dynamically created type correct programs based on an contextual instantiation method. This instantiation method is done by a unification algorithm.

Constraining the search space is a constant theme in this dissertation. Yu cites the work of Andre, 1994 where they recommend that ADFs be grouped by functionality. This will lead to ADFs that evolve separately if each ADF has its own evolution parameters. There are three potential structures inside a GP program with ADFs, as she points out. First an ADF is statically defined for every program in the GP population. Each program in the GP population has the same structure. Second, during initial program creation for the initial population, each program is created with a random structure. Then during succeeding generations the structure changes bases on overall fitness goals and the third is architecture altering operations placed on the structure of the GP program. Yu cites Koza's six architecture altering operations on how to achieve this here. Yu notes that if an incorrect sequence of architectural altering operations are chosen, the GP program may miss out on other better choices as the GP program evolves. Yu observes that for the `PolyGP` system, that these operations can be computationally expensive. This system was developed using the Haskell language on the 1.4 version Glasgow Haskell compiler.

Results based on this research can be found in a number of Yu's papers. In a paper on recursion and lambda abstraction the PolyGP system successfully evolved programs to learn the Even-N-Parity problem. This system produced smaller programs than the ADF approach and found correct solutions in less generations. Yu cites structure abstraction as a key feature of this success.[36], [33]

In another paper they revisit Montana's STGP where they grow the LISP mapcar and nth programs 6 to 7 times faster. They point out that their search space is smaller due to generation of type correct programs. And, their initial population is smaller due to their improved crossover operator.[34], [33]

2) System F and Abstract Based Genetic Programming: A system called Abstraction Based Genetic Programming ABGP, introduced by Binard, combines ideas from many areas of computer science and mathematics. In this work he uses Girard–Reynold SYSTEM F, a typed LC, within a GP system. [5], [6], [7]

His work presupposes knowledge of Type Theory and a background in LC. A detailed history with examples on usage, can be found Cardone and Hindley's paper.[31] Briefly, SYSTEM F was invented independently first by Girard in 1972 and later by Reynolds in 1974. [31], [38], [39] LC itself was invented in the 1930s by Alonzo Church and seeks to describe function abstraction. [40] When LC was first created it was in an untyped form. Girard introduced a types to LC and called it SYSTEM F. It can handle universally quantified types and has an abstraction feature for its types that makes it attractive to in the area of reducing terms in mathematical proofs. In 1974 Reynolds independently created a very similar system. His main motivation was in creating a system that could handle polymorphism used in programming languages.

A SYSTEM F is essentially defined by two grammars, one for the data types permitted in the terms, and the other for the terms themselves. Binard uses this property to further constrain the search space of allowable programs population. SYSTEM F also has the property of Curry–Howard Isomorphism. This property states that second order logic statements can be expressed with SYSTEM F syntax. So second order logic statements involving $\exists x$ and $\forall x$ can be expressed and transformed using second order proof methods. So an individual in an ABGP system's population is essentially a statement to be proved using second order logic. His system evolves provably correct individuals through the expressivity of its SYSTEM F type system.

In the ABGP system, a GP object is made up of a kingdom. A kingdom is made up of a proof and a species population. The species population is made up of genotypes. The ecosystem has species. Species have a genotype and proofs. The genotype is made up of genes parsed nodes in the SYSTEM F parse tree. The genes are the data types and program fragments that make up a SYSTEM F term. Normalization of SYSTEM F programs are done using term elimination rules that make up the second order logic portion of the SYSTEM F framework. The species is essentially a proof of correctness in the SYSTEM F framework. The proof system is based on intuitionistic type theory and is also known as constructive type theory.

During a GP run an initial population is generated randomly. Individuals in the population have as size limit imposed. He calls this *maxComp*. Only correct individuals can be generated in the population based on the constraints imposed by the predefined SYSTEM F statements.

Crossover in ABGP needs special care because of the expressiveness of SYSTEM F terms. In SYSTEM F an expression can take several forms. SYSTEM F has a *strong normalization* property meaning that depending on the abstraction context the parse tree for a SYSTEM F expression can be represented differently but normalize to the same irreducible expression.[5] So crossover needs to happen on individuals within the normalization context. For purposes of ABGP a species is an individual of the same normalization context. Crossover is handled in three separate ways. First material was swapped within the same species. Second, material was swapped and genes can be mutated within species. And third crossover scheme was explored where material was swapped and mutated across species.

A removal operation has the option to delete alleles, or whole individuals from the population. The population is rebuilt to its maximum size to get ready for the next round of evolution.

Individuals are evaluated either by test cases or by the results of a fitness function. If an individual evaluates correctly for all test cases or by good results via a fitness function, it is considered a solution.

3) Encoding λ Expressions: In a very brief two page paper Tominaga, Suzuki and Oka use an encoding scheme to evolve λ expressions for the objective function $f(x) = 2x$ using Church numerals. [10]

Church numerals are a way of describing the natural numbers using untyped λ expressions.[31].

The number 2, as they state in their work, would be the following untyped LC expression.

$$(\lambda x.(\lambda y.(x(xy))))$$

Another example is the function D which is defined as:

$$(\lambda x.(\lambda y.(\lambda z.((xy)(xy)z))))$$

So if the function D is applied to 2 using the above equations we have:

$$(D2) \equiv ((\lambda x.(\lambda y.(\lambda z.((xy)(xy)z))))(\lambda x.(\lambda y.(x(xy)))))$$

Now, if β reductions are applied we would get:

$$(\lambda x.(\lambda y.(x(x(x(xy))))))$$

And the number 4 would be produced.

Notice that the number 4 is defined as recursive function.

The authors' developed a β conversion library for Zongker's lilgp system. λ expressions are evolved and later evaluated by this system. A property of β reduction always produces valid

expressions. So no invalid expressions will be generated by this system as a result of the GP process.

They successfully generated the λ expressions, LE, of the described above in 13 out of 25 runs, with an initial population of 1000, 10% reproduction with elitism selection of 1%, 89.9% crossover and 0.1% mutation.

C. Combinators

Briggs and O'Neill in their paper of using Functional Programming FP in GP use the concept of Combinators. A listing of some of the combinators can be found in their work. It is interesting to note that some of the combinators can be defined in terms of compositions of other combinators.[41]

They point out, that using meta level combinators to evolve functions causes problems for local variables. Does a local variable's meaning change as elements of a function are combined or taken apart. If a local variable is in one sub routine and a crossover operation is performed, that locally defined variable might not be valid in another subroutine.

IV. SUMMARY

In this work we have traced, in GP, how ADFs are used. Starting as a way in Koza's original book on GP, ADFs were used as a way to reduce complexity and size of genetic programs. Others, after this, started researching different uses for ADFs in GP, with MA, ARL, ADM and various hierarchal representations. And still others approached ADFs from a meta level, with methods found in the foundation of mathematics, `PolyGP`, ABGP and combinators.

A variety of computer languages were used to explore ADFs. These ranged from the procedural languages such as C language, to C++, to functional languages like LISP, HASKELL and OCAML.

Success for these various frameworks are varied and problem dependent. There was no unified test bed of problems although there were themes through out many researchers' work including, symbolic regression, even parity problems, and multiplexer problems. Others used problems suited to the domain they were operating in, like theorem proving and the higher order functions of `nth`, `foldr` and `map`.

REFERENCES

[1] L. Spector, "Evolving control structures with automatically defined macros," in *Working Notes for the AAAI Symposium on Genetic Programming*, E. V. Siegel and J. R. Koza, Eds. MIT, Cambridge, MA, USA: AAAI, 10–12 Nov. 1995, pp. 99–105.

[2] P. J. Angeline and J. B. Pollack, "The evolutionary induction of subroutines," in *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*. Bloomington, Indiana, USA: Lawrence Erlbaum, 1992, pp. 236–241.

[3] W. Banzhaf, D. Banscherus, and P. Dittrich, "Hierarchical genetic programming using local modules," *InterJournal Complex Systems*, vol. 228, 2000.

[4] D. J. Montana, "Strongly typed genetic programming," Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, BBN Technical Report #7866, 7 May 1993.

[5] F. Binard and A. Felty, "An abstraction-based genetic programming system," in *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2007)*, P. A. N. Bosman, Ed. London, United Kingdom: ACM Press, 7-11 Jul. 2007, pp. 2415–2422.

[6] ——, "Genetic programming with polymorphic types and higher-order functions," in *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, M. Keijzer, G. Antoniol, C. B. Congdon, K. Deb, B. Doerr, N. Hansen, J. H. Holmes, G. S. Hornby, D. Howard, J. Kennedy, S. Kumar, F. G. Lobo, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, J. Pollack, K. Sastry, K. Stanley, A. Stoica, E.-G. Talbi, and I. Wegener, Eds. Atlanta, GA, USA: ACM, 12-16 Jul. 2008, pp. 1187–1194.

[7] F. J. L. Binard, "Abstraction-based genetic programming," Ph.D. dissertation, Ottawa-Carleton Institute for Computer Science, School of Information Technology and Engineering, Faculty of Engineering, University of Ottawa, Ottawa, Canada, 2009.

[8] T. Yu and C. Clack, "PolyGP: A polymorphic genetic programming system in haskell," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds. University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann, 22-25 Jul. 1998, pp. 416–421.

[9] J. Rosca, "Towards automatic discovery of building blocks in genetic programming," in *Working Notes for the AAAI Symposium on Genetic Programming*, E. V. Siegel and J. R. Koza, Eds. MIT, Cambridge, MA, USA: AAAI, 10–12 Nov. 1995, pp. 78–85.

[10] K. Tominaga, T. Suzuki, and K. Oka, "An encoding scheme for generating lambda-expressions in genetic programming," in *Genetic and Evolutionary Computation – GECCO-2003*, ser. LNCS, E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, Eds., vol. 2724. Chicago: Springer-Verlag, 12-16 Jul. 2003, pp. 1814–1815.

[11] F. Briggs and M. O'Neill, "Functional genetic programming and exhaustive program search with combinator expressions," *International Journal of Knowledge-Based and Intelligent Engineering Systems*, vol. 12, no. 1, pp. 47–68, 2008.

[12] J. Koza, "Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems," Dept. of Computer Science, Stanford University, Technical Report STAN-CS-90-1314, Jun. 1990.

[13] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[14] ——, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge Massachusetts: MIT Press, May 1994.

[15] J. R. Koza and J. P. Rice, "A non-linear genetic process for data encoding and for solving problems using automatically defined functions," U.S. Patent 5343554, May 1992, application filed May 11, 1992, issued August 30, 1994, 5,343,554.

[16] D. Andre, "Evolution of mapmaking ability: Strategies for the evolution of learning, planning, and memory using genetic programming," in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, vol. 1. Orlando, Florida, USA: IEEE Press, 27-29 Jun. 1994, pp. 250–255.

[17] J. R. Woodward, "Modularity in genetic programming," in *Genetic Programming, Proceedings of EuroGP'2003*, ser. LNCS, C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, Eds., vol. 2610. Essex: Springer-Verlag, 14-16 Apr. 2003, pp. 254–263.

[18] P. J. Angeline and J. Pollack, "Evolutionary module acquisition," in *Proceedings of the Second Annual Conference on Evolutionary Programming*, D. Fogel and W. Atmar, Eds., La Jolla, CA, USA, 25-26 Feb. 1993, pp. 154–163.

[19] D. Zongker and B. Punch, "lilgp 1.01 user's manual," Michigan State University, USA, Tech. Rep., 26 Mar. 1996.

[20] F. Gruau, "Cellular encoding of genetic neural networks," Laboratoire de l'Informatique du Parallilisme. Ecole Normale Supirieure de Lyon, France, Technical report 92-21, 1992.

[21] R. Rosen, *Life Itself: A Comprehensive Inquiry into the Nature, Origin, and Fabrication of Life (Complexity in Ecological Systems)*. Columbia University Press, 2005.

[22] J. P. Rosca and D. H. Ballard, "Discovery of subroutines in genetic programming," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr., Eds. Cambridge, MA, USA: MIT Press, 1996, ch. 9, pp. 177–201.

[23] ——, "Genetic programming with adaptive representations," University

of Rochester, Computer Science Department, Rochester, NY, USA, Tech. Rep. TR 489, Feb. 1994.

[24] ——, "Hierarchical self-organization in genetic programming," in *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann, 1994.

[25] K. E. Kinnear, Jr., "Alternatives in automatic function definition: A comparison of performance," in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. MIT Press, 1994, ch. 6, pp. 119–141.

[26] H. Iba, H. de Garis, and T. Sato, "Genetic programming using a minimum description length principle," in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. MIT Press, 1994, ch. 12, pp. 265–284.

[27] A. Dessi, A. Giani, and A. Starita, "An analysis of automatic subroutine discovery in genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., vol. 2. Orlando, Florida, USA: Morgan Kaufmann, 13-17 Jul. 1999, pp. 996–1001.

[28] L. Altenberg, "The evolution of evolvability in genetic programming," in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. MIT Press, 1994, ch. 3, pp. 47–74.

[29] W. A. Tackett, "Recombination, selection, and the genetic construction of computer programs," Ph.D. dissertation, University of Southern California, Department of Electrical Engineering Systems, USA, 1994.

[30] C. Janikow and J. Aleshunas, "Impact of commutative and non-commutative functions on symbolic regression with ACGP," in *2013 IEEE Conference on Evolutionary Computation*, L. G. de la Fraga, Ed., vol. 1, Cancun, Mexico, Jun. 20-23 2013, pp. 2290–2297.

[31] F. Cardone and J. R. Hindley, "History of lambda-calculus and combinatory logic," *Handbook of the History of Logic*, vol. 5, 2006.

[32] D. Gabbay and J. Woods, *Logic from Russell to Church, Volume 5 (Handbook of the History of Logic)*. North Holland, 2009.

[33] G. T. Yu, "An analysis of the impact of functional programming techniques on genetic programming," Ph.D. dissertation, University College, London, Gower Street, London, WC1E 6BT, 1999.

[34] C. Clack and T. Yu, "Performance enhanced genetic programming," in *Proceedings of the Sixth Conference on Evolutionary Programming*, ser. Lecture Notes in Computer Science, P. J. Angeline, R. G. Reynolds, J. R. McDonnell, and R. Eberhart, Eds., vol. 1213. Indianapolis, Indiana, USA: Springer-Verlag, Apr. 13-16 1997, pp. 87–100.

[35] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *J. ACM*, vol. 12, no. 1, pp. 23–41, Jan. 1965. [Online]. Available: http://doi.acm.org/10.1145/321250.321253

[36] T. Yu and C. Clack, "Recursion, lambda-abstractions and genetic programming," in *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, R. Poli, W. B. Langdon, M. Schoenauer, T. Fogarty, and W. Banzhaf, Eds. Paris, France: CSRP-98-10, The University of Birmingham, UK, 14-15 Apr. 1998, pp. 26–30.

[37] T. Yu, "Structure abstraction and genetic programming," in *Proceedings of the Congress on Evolutionary Computation*, P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, Eds., vol. 1. Mayflower Hotel, Washington D.C., USA: IEEE Press, 6-9 Jul. 1999, pp. 652–659.

[38] J.-Y. Girard, P. Taylor, and Y. LaFont, *Proofs and Types*, ser. Cambridge Tracts in Theoretical Computer Science. Great Britain: Cambridge University Press, 1989.

[39] J. C. Reynolds, "Towards a theory of type structure," in *Colloque sur la Programmation*, ser. LNCS, vol. 19. Springer-Verlag, 1974, pp. 408–425.

[40] A. Church, "A set of postulates for the foundation of logic," *Annals of Mathematics (2nd Series)*, vol. 33, no. 2, pp. 346–366, 1932.

[41] F. Briggs and M. O'Neill, "Functional genetic programming with combinators," in *Proceedings of the Third Asian-Pacific workshop on Genetic Programming*, T. L. Pham, H. K. Le, and X. H. Nguyen, Eds., Military Technical Academy, Hanoi, VietNam, 2006, pp. 110–127.