Modularity in Genetic Programming

Martin Dostál

Abstract. This chapter provides a review of methods for automatic modularization of programs evolved using genetic programming. We discuss several techniques used to establishing modularity in program evolution, including highly randomized techniques, techniques with beforehand specified structure of modules, techniques with evolvable structure and techniques with heuristic identification of modules. At first, simple techniques such as Encapsulation and Module Acquisition are discussed. The next two parts reviews Automatically Defined Functions and Automatically Defined Functions with Architecture Altering Operations that enable to evolve the structure of modules at the same time of evolving the modules itself. The following section is focused on Adaptive Representation through Learning, a technique with heuristic-based identification of modules. Next, Hierarchical Genetic Programming is described. Finally, establishing recursion and iteration, a code reuse technique closely related to modularization, is briefly surveyed.

1 Introduction

Genetic programming is a widely known method for automatic program synthesis by evolutionary means [6, 12]. The process of searching a program that solves a given problem can be characterized as breeding candidate programs using evolutionary operations such as crossover, mutation and reproduction. Candidate programs are referred as *individuals* that constitute a *population*. Each individual is evaluated using the so-called *fitness function* that assigns to the individual a number expressing how well the individual solves the problem. Individual evolutionary operations are used to modify existing or creating new individuals. Individuals are reproduced to the new generation on the basis of fitness selection, so better fit

Martin Dostál
Dept. Computer Science, Palacký University Olomouc, 17. Listopadu 12,
77146 OLOMOUC, Czech Republic
e-mail: martin.dostal@upol.cz

I. Zelinka et al. (Eds.): Handbook of Optimization, ISRL 38, pp. 365–393.
springerlink.com

individuals have higher chance to survive to the next generation of evolutionary process than the less fit ones. The process continues repeatedly until a termination criterion is met. Typically, the termination criterion is represented by finding an acceptable solution or exceeding a maximum allowed number of generations. A flowchart of genetic programming is depicted on Fig. 1. In this text we do not discuss the details on evolutionary operations since there are various textbooks such as [6, 12, 14, 16] which provide a thorough description on this topic.

In genetic programming, programs can be represented in different ways such as *tree graphs* or various *textual representations* [7, 18]. In this chapter we will use the tree representation and the corresponding textual representation based on Lisp S-Expressions. Each tree node contains an atomic symbol, which depicts either a function or a terminal symbol. Functions (of non-zero arguments) are contained in non-terminating nodes, whose child nodes represent parameters passed to the function. Terminals, such as variables, numbers and constants are contained in terminal nodes. Functions of zero arguments are also contained in terminal nodes, obviously. Note that a tree can be easily transformed to S-expression, a representation of programs used by Lisp-like programming languages such as Common Lisp or Scheme. S-expressions are *atoms* (for purposes of genetic programming, atoms are terminals and function names) and *lists* composed of S-expressions. Lists represent function calls and use the prefix notation. The first element of a list is treated as function and the other elements as parameters. Fig. 2 depicts a tree representation of an individual corresponding to `(AND (OR (NOT A) B) (AND A (OR A B)))` S-expression.

Available functions constitute the so called *Function set*. The function set should be small, but enough expressive and error-resistant. Error-resistance means that every possible program based on the function and terminal set can be successfully evaluated. These properties are usually referred as *universality*, *sufficiency* and *closure* [12]. For example, consider again the program depicted on Fig. 2. The corresponding function set is equal to $F = \{\texttt{AND}, \texttt{OR}, \texttt{NOT}\}$ and the terminal set $T = \{\texttt{A}, \texttt{B}\}$. This function set is sufficient to express any possible boolean function. To summarize the overview on genetic programming, let us recapitulate the preparatory steps for the use of genetic programming:

1. Define the set of terminals.
2. Define the set of functions.
3. Set the fitness measure.
4. Set the parameters for controlling the run, e.g., the number of generations, population size, crossover and mutation probability.
5. Choose the termination criterion.

In Genetic Programming, the search effort required to find a solution of the problem is described by the $I(M,i,z)$ number which is used as a measure of performance. The $I(M,i,z)$ number expresses the number of individuals that must be processed in order to solve the problem with a certain specified probability (typically put 99%) no later that in specified number of generations. Formula 1 depicts the calculation of the $I(M,i,z)$ number. M is the size of population, i is the number of generations and

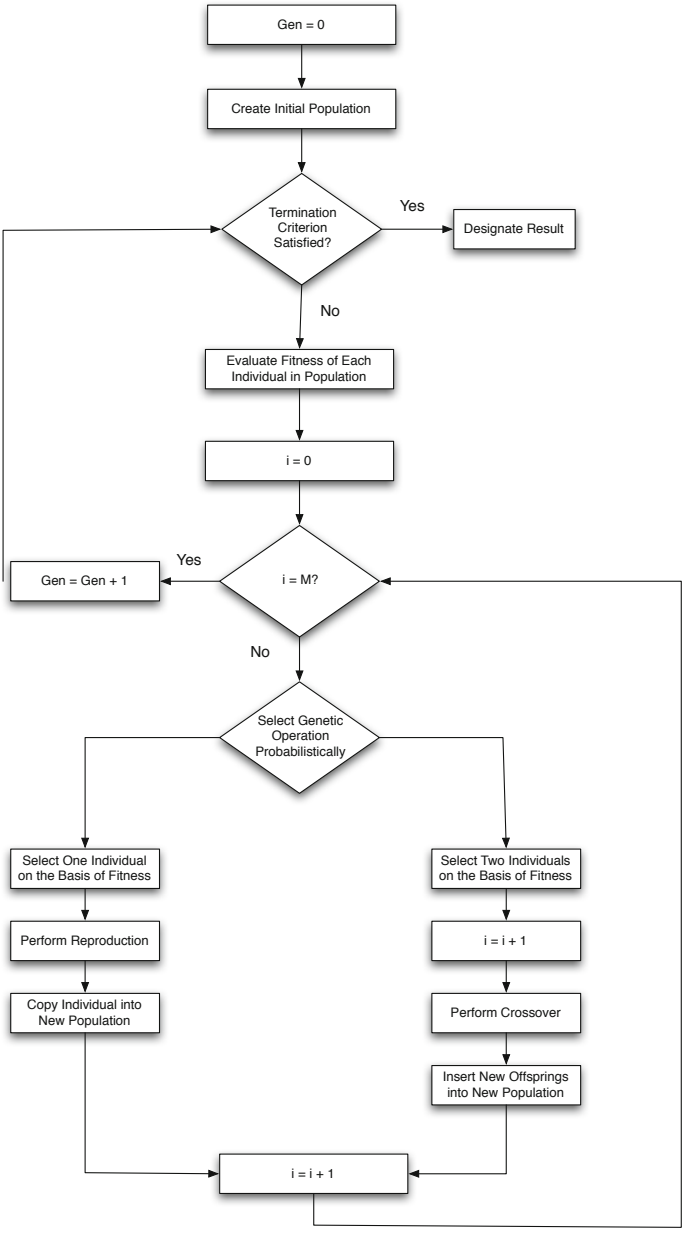

Fig. 1 Genetic programming flowchart

$R(z)$ is the number of independent runs of the experiment required to yield a success with probability at least z. The calculation of $R(z)$ is explained in Formula 2. $P(M,i)$ is experimentally observed cumulative probability of success of the experiment no later than in generation i.

$$I(M,i,z) = M * (i+1) * R(z) \tag{1}$$

$$R(z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil \tag{2}$$

For example, put $z = 0.99$, $M = 1500$ and $i = 100$. Say, that $P(M,i)$ will be equal to 0.44. Then $R(z) = 8$ and thus $I(M,i,z) = 1500 * (100+1) * 8 = 1212000$, that is 1 212 000 individuals must be processed in order to obtain a solution of the problem within 100 generations with probability 99%.

2 Modularity

In general, modularity is a concept of dividing a system into separate components (or constituting a system from such components) that are interchangeable and often also reusable. Modularity is widely utilized principle in nature as well as in artificial systems. In nature, the attributes of modularity can be found in construction of cellular organisms that are, technically speaking, composed of smaller "standardized" units, for instance. Modularity is also a fundamental principle used in artificial systems including computer programming.

In programming, modularity represents a technique of composing a system of smaller, independent program units. It enables a *separation of concerns*, in which the whole problem is decomposed into small, logical parts that can be created and verified separately. The separation of concerns is inherently natural for the human thinking. This concept is often also referred as the *divide and conquer* maxim, coming from politics and sociology. Use of modularity enables to reuse the modules which improves the quality of code. The code is reusable, less redundant, better organized, shorter, more readable and thus better manageable. Modules itself are often represented by functions (or another means to separate a portion of code, such as procedures, subroutines or classes) or libraries, depending on particular programming language.

In fact, standard genetic programming has no built-in support for establishing a modular solution of a problem. It significantly limits the efficiency and, more importantly, the scalability of a system for program synthesis. Actually, even simple problems can often be solved with more efficiency when modularity is utilized properly. Without modules, each reusable portion of code must be evolved repeatedly at the same time. It is analogous to writing a program in the all-at-the-same-time style without decomposing a problem into subproblems. For instance, consider the even-parity problem. It is a simple classification problem. A solution of the problem returns True if an even number of inputs are True. Otherwise, it returns False. The truth

table for the even-3-parity problem is depicted in Table 1. The even-parity problem has been widely used as a benchmark problem for program synthesis systems. With standard genetic programming, the problem scales poorly with increasing the number of inputs, see Table 2 whose first row depicts the computational effort required to solve even-3, 4 and 5 problem. For the even parity problem it would be useful to find and reuse a useful portion of code during the evolution in order to improve the scalability and efficiency. For instance, with a function set composed of `OR` and `NOT` functions, finding a module that computes the `XOR` function would be particularly useful since a solution of the problem could be expressed as `(XOR A (NOT (XOR B C)))`. Most likely, solutions based on the original function set would be more complex (e.g., `(NOT (OR (NOT (OR A B)) (NOT (OR (NOT A) (NOT B)))))`) than using the function set with `XOR` function.

Table 1 The truth table for even-3-parity problem

A	B	C	EP-3
FALSE	FALSE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	TRUE
TRUE	TRUE	FALSE	TRUE
TRUE	TRUE	TRUE	FALSE

Establishing a useful modularity lies in two closely related aspects: finding a proper structure of modules (a number of modules, a number of arguments of each module) and evolving a useful and reusable functionality of each module.

In most methods for modularization of genetic programming, modules are represented as functions that can be automatically discovered during the evolution process. It means that the function set is also subject to evolution. In other words, we search for a good representation language for the problem at the same time with searching a solution of the problem since a small, yet expressive, language can express the solution as a less complex program than in a representation not such specific to the problem whose solution is being searched.

3 Encapsulation

In 1992, Koza [12] suggested a very basic technique to modularize the evolved code. The idea behind the Encapsulation operator is to freeze a part of an individual's code into a new function which code can be then reused in other individuals in a population.

The encapsulation begins with the selection of a promising individual using a selection operation, e.g., roulette wheel selection. Next, a non-terminating node in

the individual is selected at random. A subtree located at the selected node (the root-node of the subtree is the selected node) is then replaced (encapsulated) by the new function call. The function code corresponds to the encapsulated subtree. The new encapsulated function has zero arguments since the code contains functions and terminals from the general function and terminal set. Encapsulated functions are named automatically using the letter "E" and a number in a successive order (e.g., "E0", "E1", "E2", ...) and added immediately to the function set in order to be reused in newly created individuals.

Since the encapsulated code (subtree) is moved from the individual to a new function, it is not subject to evolution any further. In other words, once the module is created, its code is frozen. To demonstrate Encapsulation on an example, consider the following code and corresponding tree on Fig. 2:

```
(AND (OR (NOT A) B)
     (AND A (OR A B)))
```

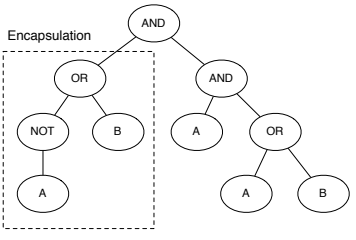

Fig. 2 A tree for `(AND (OR (NOT A) B) (AND A (OR A B)))` individual

The initial function set is equal to $F = \{\texttt{AND}, \texttt{OR}, \texttt{NOT}\}$ and terminal set is equal to $T = \{\texttt{A}, \texttt{B}\}$. The selection point has been chosen at random to the `OR` node on the left branch of the tree. New function `E0` will be defined and added to the function set and the encapsulated subtree will be replaced with the `E0` function call:

```
(DEFUN E0 ()
  (OR (NOT A) B))
```

Now, the function set will be equal to $F = \{\texttt{AND}, \texttt{OR}, \texttt{NOT}, \texttt{E0}\}$ and the code of the encapsulated individual is following:

```
(AND (E0)
     (AND A (OR A B)))
```

Koza in his work [12] applied experimentally Encapsulation to the Boolean 6-multiplexer problem. The probability of encapsulation was set to 20%, that is 200 of 1000 individuals in population of each generation were subject to encapsulation. Koza compared the results obtained with- and without encapsulation operation using the probability of success $P(M, i)$, $M = 1000$. However, no substantial differences in performance were found.

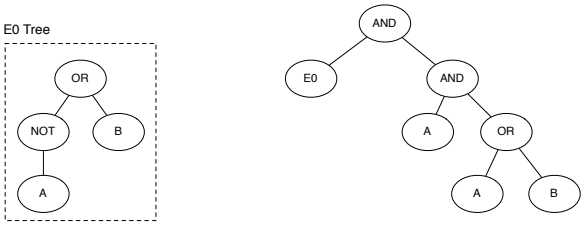

Fig. 3 Encapsulation on an individual

4 Module Acquisition

Angeline and Pollack [1–4] proposed a modularization technique called "Module Acquisiton" (MA). This technique is similar to Koza's Encapsulation operator in several aspects. First, the subtrees to define new modules are selected at random. Second, modules are defined globally. Thirdly, modules are not subject to evolution.

Module acquisition introduces two additional operators to handle the modularity: *Compression* and *Expansion*. Similarly to Encapsulation, Compression operator defines a new function. A randomly selected subtree at given depth is used to define a new function. Note that since a subtree of a given depth is taken, it may not be necessarily the complete subtree as shown on Fig. 4. Newly defined function will take arguments $p_1, p_2, \ldots, p_n$ as required by functions at terminal nodes of the module subtree. For instance, the compression of the subtree at Fig. 4 will result in a function of three arguments since `AND` requires two, and `NOT` requires one argument. New functions are named automatically and added to the function set. Thus, modules are global and static.

New module created by the Compression operator according to the individual on Fig. 4 will have the following definition:

```
(DEFUN NEWMOD (P1 P2 P3)
 (AND (NOT (AND P1 P2))
      (OR B (NOT P3))))
```

The Expansion operator counterparts the Compression operator. A randomly selected function call corresponding to a previously evolved module is reversely replaced by the original subtree. Once a module has been expanded in an individual, it is subject to evolution again. Both operators are applied with a given probability.

5 Automatically Defined Functions

Automatically Defined Functions (ADF) were proposed by Koza and Rice in 1992 [14, 17]. This approach is quite different from the above introduced Module Acquisition and Encapsulation. Basically, Automatically Defined Functions are locally defined modules in contrast to globally defined modules in previously discussed

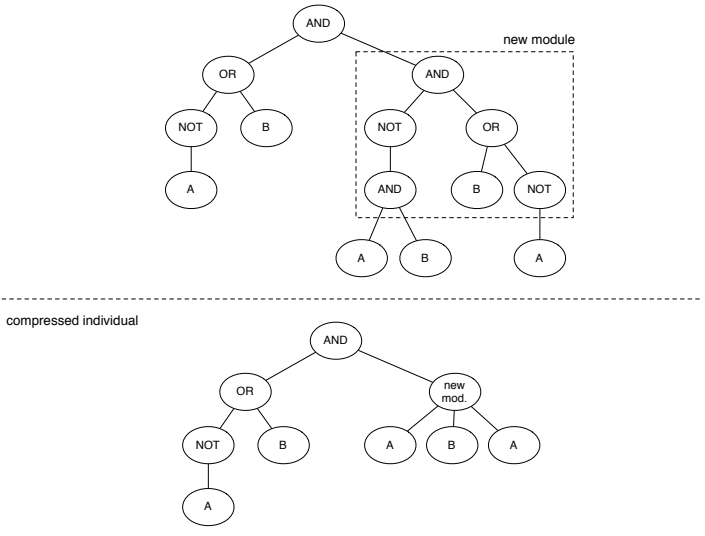

Fig. 4 Module Compression

approaches. The modules are contained within an individual, so that they cannot be called by another individuals.

The second substantial difference lies in the structure of modules. In ADF, modules have a predefined structure in contrast to Module Acquisition and Encapsulation where modules are chosen at random. Thus in Module Acquisition and Encapsulation, modules may also have a different structure. When using ADF, the experimenter must specify the number of automatically defined functions and the number of arguments for each ADF during the preparatory steps. In fact, this is a kind of additional knowledge about the problem which can significantly improve the evolution process, when provided properly. To be clear on the structure, in ADF the structure of modules is predefined and it is also common for all individuals. However, the code of each particular ADF in each individual is specific.

When using ADF, the structure of an individual is divided into *branches*. Branches are child nodes of the root node which contains the so called *placeholder*. The placeholder is an operation which defines how many branches are defined in the individual. The rightmost branch contains the individual's code and is called *result-producing branch*. Other branches represent the definitions of ADF's, each branch corresponding to one ADF. These branches are called *function-defining branches*, see Fig. 5. The `PROGN` function acts as a placeholder.

A different structure of individuals makes it necessary to modify the crossover operator. The crossover must be performed per branches, so that it is not possible to combine the code from different branches, e.g., `ADF0` and `ADF1` or `RPB` and `ADF0`. The reason is obvious; individual branches have its own function sets and may also use a different number of arguments.

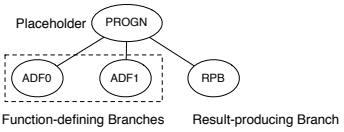

Fig. 5 The structure of an individual

5.1 Preparatory Steps:

1. Choose the number of function-defining branches.
2. Choose the arity of each branch.
3. Choose the function and terminal set for each branch.
4. Determine, if references between function-defining branches are allowed.

Since the experimenter specifies the function set for each ADF it is possible to include also other ADF's into the function set of a particular ADF. However, it must be handled carefully since it may result in infinite loops. For example, consider an `ADF0` with `ADF1` in its function set and `ADF1` with `ADF0` in its function set. It is possible that `ADF0` will call `ADF1` in its body and `ADF1` will call `ADF0`, which would result in an infinite loop.

5.2 Example: Even-4-Parity Problem

For the even-4-parity problem we have chosen two automatically defined functions, `ADF0` a `ADF1`. ADF0 will take two arguments, ADF1 will take three arguments.

- population size $M = 4000$
- maximal number of generations = 51
- fitness-cases: all possible combinations
- $fitness = 2^4 - x$, x is the number of correct outputs
- termination criterion: $fitness = 2^4$
- function set for $F = \{\texttt{AND}, \texttt{OR}, \texttt{NAND}, \texttt{NOR}\}$
- function set for RPB $F_R = F \cup \{\texttt{ADF0}, \texttt{ADF1}\}$
- terminal set $T = \{\texttt{D0}, \texttt{D1}, \texttt{D2}, \texttt{D3}\}$
- function set for `ADF0`: $F_0 = F$
- terminal set for `ADF0`: $T_0 = \{\texttt{ARG0}, \texttt{ARG1}\}$
- function set for `ADF1`: $F_1 = F$
- terminal set for `ADF1`: $T_1 = \{\texttt{ARG0}, \texttt{ARG1}, \texttt{ARG2}\}$

Koza in [12] performed an experiment with the above described settings. For 168-times repeated experiments there was a solution found in 93% of runs in less than 10 generations. It follows that $I(M, i, z) = 4000 * (9 + 1) * 2 = 80\,000$.

Example of a solution:

```
(PROGN (DEFUN ADF0 (ARG0 ARG1 ARG2)
         (VALUES
          (OR (AND ARG0 ARG1)
          (AND (NOT ARG0) (NOT ARG1)))))
        (DEFUN ADF1 (ARG0 ARG1)
         (VALUES
          (AND ARG0 (AND (ARG1 ARG2)))))
        (VALUES
         (ADF0 (ADF0 D0 D1) (ADF0 D2 D3))))
```

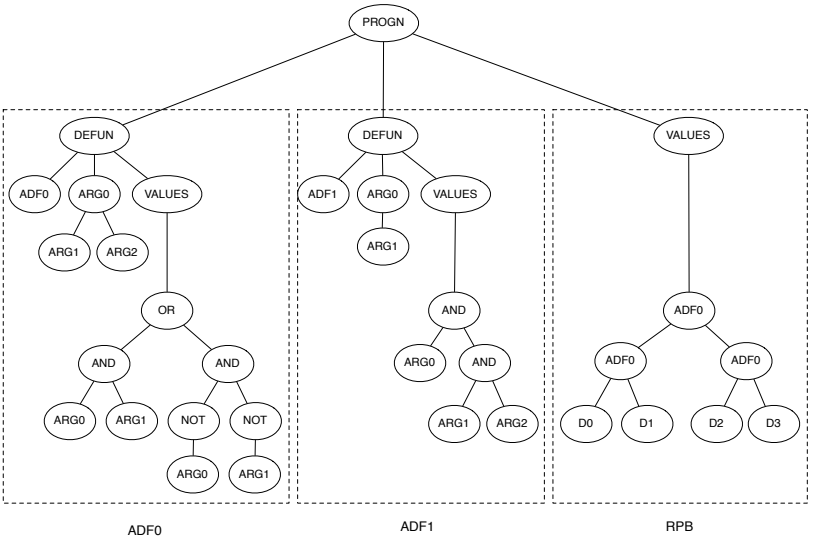

Fig. 6 An even-4-parity problem solution

Automatically defined functions, when the experimenter defines a promising structure of individuals (that is the number of function-defining branches and the number of arguments for each branch), can provide a substantial improvement over the standard genetic programming. For instance, Koza in [12] compared the standard genetic programming to genetic programming with utilization of ADF on even 3-,4-, 5- parity problem and odd 5-parity problem. The obtained $I(M,i,z)$ values are summarized in Table 2. For details on the control parameters that Koza used see [12]. Another comparison with a different setup of parameters has been performed by Koza in [14], we summarize the results in Table 3.

Table 2 Standard GP vs GP with ADF

system	ep-3	ep-4	ep-5	op-5
GP	80 000	1 276 000	7 840 000	912 000
GP(ADF)		80 000	152 000	276 000

Table 3 Standard GP vs GP with ADF

system	ep-3	ep-4	ep-5	ep-6
GP	96 000	384 000	6 528 000	70 176 000
GP(ADF)	64 000	176 000	464 000	1 344 000

6 Automatically Defined Functions with Architecture-Altering Operations

When using Automatically Defined Functions, the experimenter must propose the number of ADF's and the number of arguments for each ADF. In other words, a promising structure of the problem solution must be proposed in advance. Unfortunately, for many problems this preparatory step is difficult since we may not know how a solution of the problem could be structured. Ideally, a system for evolving programs should require from the experimenter as few as possible information on how to solve the problem.

To address this issue, Koza introduced [13, 15, 16] the Architecture-altering operations (ADF-aao) which enable to evolve the structure of individuals at the same time of evolving the solution itself. Architecture-altering is composed of six operations (Branch Duplication, Argument Duplication, Branch Creation, Argument Creation, Branch Deletion, Argument Deletion), each handling a structure of an individual in some sense.

In the following parts we introduce individual architecture altering operations including an example of application of the operation to a sample individual depicted on Fig. 7.

6.1 Branch Duplication

Branch Duplication (also called Subroutine Duplication) operates on a copy of selected individual. The operation inserts a new function-defining branch in the individual. The new function-defining branch is created by duplication of an existing branch. The operation may also modify the code of the result-producing branch, but preserves the program semantics. More precisely, the operation acts as follows:

1. Select an individual from the population probabilistically on the basis of fitness.
2. Make a copy of the individual. The individual will undergo the operation.

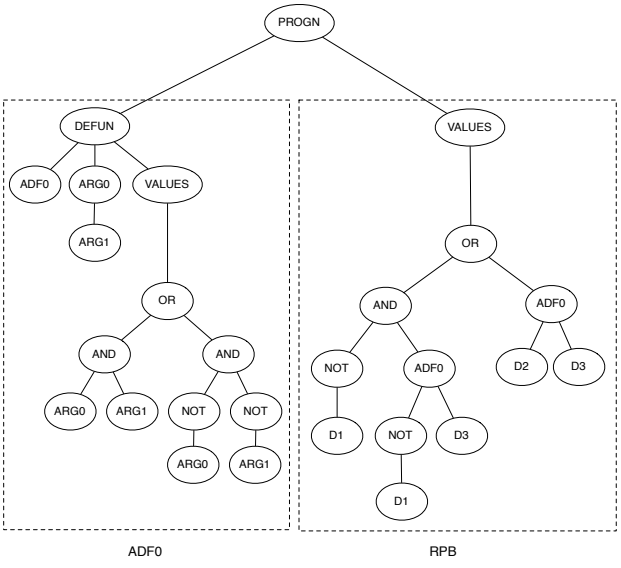

Fig. 7 An individual with one function-defining branch

3. The individual should have at least one function-defining branch and less than a maximal number of function-defining branches. Otherwise, abort the operation and continue with the reproduction operation.
4. Select a function-defining branch in the individual.
5. Duplicate the selected branch in the individual and assign an unique name automatically using the "ADF" sequence and a number in successive order. The new branch has the same argument list and body as the duplicated one.
6. Update the function set with the newly defined branch.
7. Select randomly (with given probability) some nodes within the result-producing branch that represent a call to the duplicated function-defining branch and replace them with a call to the newly defined branch. Child nodes (arguments) remain unchanged.

The Branch duplication operation has two control parameters: a probability of the operation execution and a preestablished maximum number of function-defining branches in the individual.

Although the operation itself preserves the individual's program semantics, it may be altered by subsequent application of recombination operators.

To demonstrate the operation consider the original program depicted in Fig. 12 and altered program in Fig 8 using the branch duplication. The `ADF0` function-defining branch has been selected to create a new branch using the duplication operation since it is the only function-defining branch available in the individual. New branch named `ADF1` is created from `ADF0`. The next step is to replace some nodes

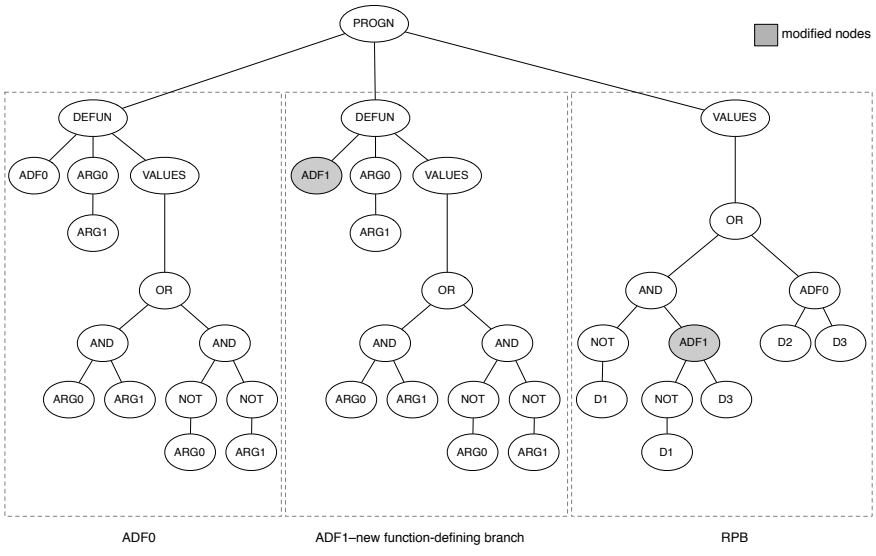

Fig. 8 Branch duplication example

representing a call to `ADF0` with `ADF1`. In this example the changed node is depicted with gray background.

6.2 Argument Duplication

The Argument Duplication operation duplicates one of the arguments present in a function-defining branch of selected individual. The operation modifies some parts of the individual. First, it modifies the argument list of the selected function-defining branch. Second, the body of the selected function-defining branch is modified on a probabilistic basis. Afterwards, the result-producing branch is modified in order to update the calls to modified function-defining branch. However, the operation preserves individual's program semantics. The Argument Duplication operates in the following way:

1. Select an individual from the population probabilistically on the basis of fitness.
2. The individual should have at least one function-defining branch with one argument at least. The branch must have a less than maximum number of arguments established for the individual itself. Otherwise, abort the operation.
3. Randomly choose a function-defining branch of the selected individual.
4. Randomly choose an argument to be duplicated.
5. Add a new argument with unique name to the argument list of the picked branch and update the terminal set for the individual.

6. For each occurrence of the argument-being-duplicated in the function-defining branch, replace randomly with a given probability the original argument with the newly added one.
7. Update each occurrence of the picked function-defining branch in the individual. That is, duplicate the subtree corresponding to the argument-to-be-duplicated in order to increase the number of arguments applied to call the picked function-defining branch by one.

The operator is controlled using the following parameters: a probability of executing the operation, a maximum number of arguments for each function-defining branch.

Fig. 9 depicts an application of the operation on individual from Fig. 7. Argument `ARG1` in the function-defining branch has been selected to duplicate. That is, new argument `ARG2` is inserted into the argument list and some occurrences of `ARG1` in the function-defining branch are replaced with `ARG2`. The next step is to modify the result-producing branch. Each node representing a call to `ADF0` must be updated with a third parameter to be passed to `ADF0`. Since the second parameter i.e., `ARG1` of the function-defining branch has been duplicated, subtrees corresponding to second parameter in each call to `ADF0` are duplicated and passed to `ADF0` as third argument. Coincidentally, in both calls to `ADF0` in the result-producing branch, the subtree to be duplicated as third argument is equal to `D3`.

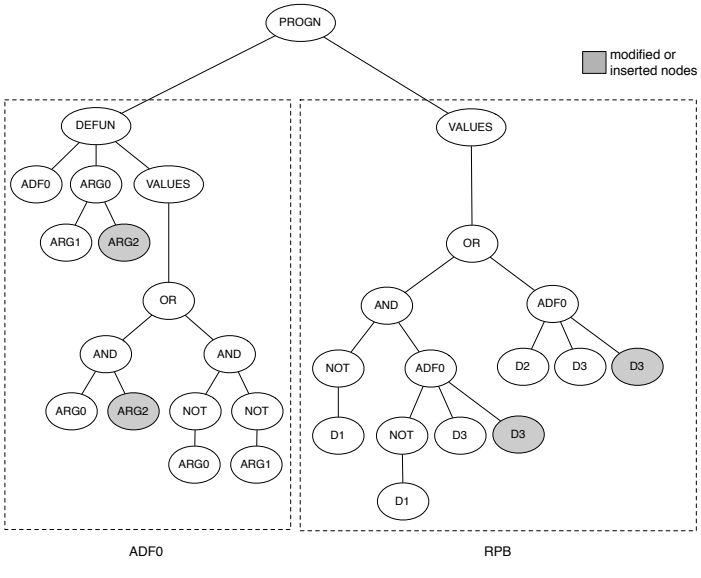

Fig. 9 Argument duplication example

6.3 Branch Creation

The Branch Creation operation (also called Subroutine Creation) operates on one selected individual. The operation creates a new function-defining branch by choosing a subtree in one of the branches available in the selected individual. Although the operation may significantly alter the individual's code structure, it does not modify the semantics. Branch creation operates in the following way:

1. Select an individual from the population probabilistically on the basis of fitness.
2. The individual should have a less than maximum allowed number of function-defining branches. Otherwise, abort the operation.
3. Randomly select one branch within the selected individual. It may be either one of function-defining branches or the result-producing branch.
4. Randomly pick a node in the selected branch and begin traversing the subtree below the picked node in a depth-first manner.
5. For the currently visited node, choose one of the following steps on a random basis:

   a. Designate the current node as the root node for an argument subtree. No further traversal will be performed on the subtree.
   b. Continue traversal. That is, repeatedly apply (5).

6. Insert new function-defining branch into the individual. Assign a unique name to the new branch. The branch will take the same number of arguments as the number of designated nodes during the traversal. The body of new branch is constituted using a modified copy of the subtree starting at the node picked in step (4) of this algorithm. The modification of the copied subtree is performed on nodes designated during the traversal. Designated nodes (and thus also the corresponding subtrees, if present) are replaced by the corresponding local arguments of the newly defined function-defining branch. It finishes the constitution of a new body of the newly defined function-defining branch.
7. Replace the subtree used to create the new branch with the node containing the name of the new function-defining branch. The new node will have the same parent as the root node of the subtree. In other words, the subtree used to constitute the new module is replaced with the call to such module. In the next step, the corresponding parameters are assigned to the new branch call.
8. For each node in the subtree below the node designated during the traversal, use the designated node and the subtree below as a parameter used to the new function-defining branch call. This step finishes the modification of a branch used to create a new module.
9. The terminal set of the new branch is equal to the terminal set of the branch selected in step (3) minus arguments replaced in the newly defined branch by local arguments, plus those local arguments. Function set of the selected branch is updated with the newly defined branch name. The function set of branches that include the selected branch is updated with the newly defined branch name.

The operator is controlled using the following parameters: a probability of executing the operation and the maximum number of function-defining branches. Koza in his work [16] on page 97 introduces some other control parameters, such as minimum and maximum number of arguments of function-defining branches and maximum size of a branch. However, no details on how the Subroutine creation operation handles these parameters are provided.

To illustrate the operation, we perform branch creation on an individual depicted on Fig. 7. We deliberately selected the `AND` node in the result-producing branch as a starting node for new branch. Then we performed the subtree traversal staring at the selected node. The subtree and designated nodes are depicted on Fig 10; the subtree is enclosed in a dashed cloud area and the designated nodes are depicted with gray background. Designated nodes in the subtree are replaced by new local arguments for the new branch, namely `ARG0`, `ARG1` and `ARG2`. Now, the subtree can be used to define the body of a new function-defining branch as shown on Fig. 11. The result-producing branch must be updated now. The subtree used to create new branch is replaced by a call to the newly defined branch, that is `ADF1`. Parent node to `ADF1` is equal to the parent node of the subtree's root node. Corresponding parameters are passed to `ADF1`, that is `(NOT D1)` for first argument, `(NOT D1)` for second argument and `D3` as third argument.

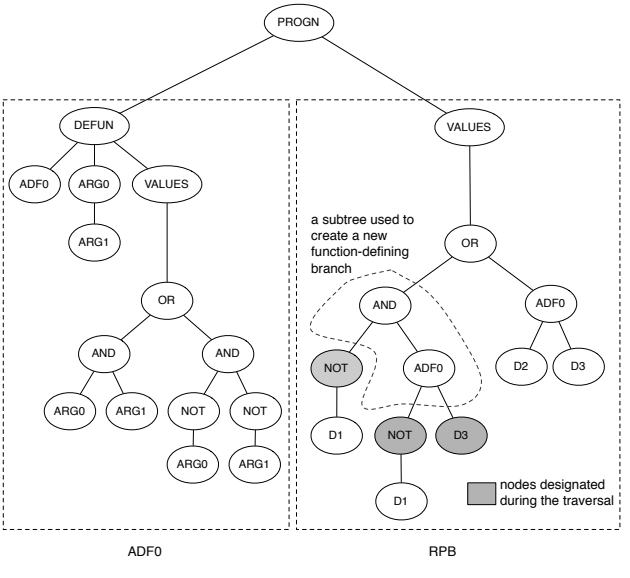

Fig. 10 Branch creation example: first phase

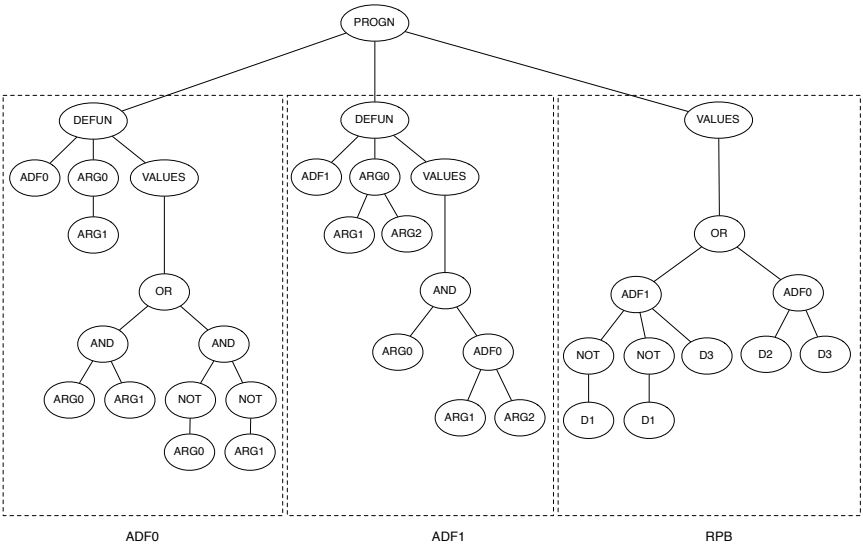

Fig. 11 Branch creation example: second phase

6.4 Argument Creation

The Argument creation adds a new argument into a selected function-defining branch. The new argument is used to replace a subtree within the selected function-defining branch. Other branches are modified in order to call the modified function-defining branch appropriately. Thus, the operator preserves the individual's program semantics as the most other architecture altering operators do. The operator acts as follows:

1. Select an individual from the population probabilistically on the basis of fitness.
2. Select a function-defining branch within the selected individual.
3. Insert a uniquely named new argument to the argument list of the selected function-defining branch.
4. Select a node from the function-defining branch. The node (if represents a function) must have a less than maximum allowed number of arguments established for the problem.
5. Replace the subtree starting at the selected node with node representing the new argument. The replaced subtree will be used to modify branches that call the to-be-modified-function-defining branch.
6. Add an additional parameter to each call of the function-defining branch selected in step 2. A modified subtree obtained in the previous step will be passed as new parameter. The modification of the subtree is done on local variables used in the function-defining branch since these may not be available in other branches. Such local variables are replaced with corresponding parameters (that is nodes

or subtrees) passed to the call of the to-be-modified-function-defining branch. By corresponding parameters we mean parameters of the same order, e.g., first argument, second argument etc.

7. The terminal set of the selected function-defining branch is enlarged with the newly created argument. The arity of the branch is incremented.

The operator is controlled using the following parameters: a probability of executing the operation and a maximum number of arguments for each function-defining branch.

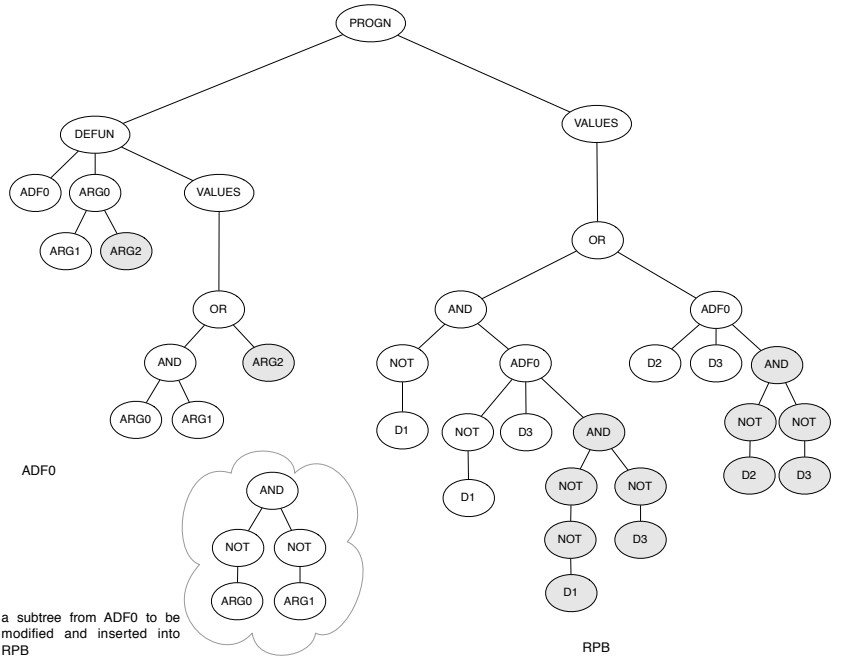

Fig. 12 Argument creation example

To demonstrate the operation we provide an example on Fig. 12 that depicts the application of the operator on a program shown on Fig. 7. The program contains only one function-defining brach, so that the operation will be performed on that branch. The original branch contains two arguments, namely `ARG0` and `ARG1`. New argument `ARG2` will be inserted into the function-defining branch. Now we select a node for replacing the corresponding subtree with `ARG2`. We deliberately selected the `AND` node, the second argument of `OR`. The corresponding subtree is depicted on Fig. 12 in a dashed cloud area. The subtree in the function-defining branch is replaced with a new node for the new argument `ARG2`.

The next step is to modify the branches that call the function-defining branch being modified. Function `ADF0` is called from the result-producing branch only. Each

call of the ADF0 in the result-producing branch must be updated with the new argument since one argument (i.e., ARG2) has been added. The selected subtree from the function-defining branch will be modified and passed as third argument to calls of ADF0. The modification of the subtree is performed as follows; the original arguments used in the subtree, that is ARGO and ARG1 will be replaced by corresponding arguments used to call ADF0 in the result-producing branch. That is, the first parameter of the function-defining branch will be replaced by the first parameter of the ADF0 in the result-producing branch and so on. Precisely, for the subtree used as third argument to ADF0 that is present on the left part of the result-producing branch: ARG0 will be replaced by (NOT D1), the first argument passed to the ADF0, and ARG1 will be replaced by D3 since it is the second parameter passed to ADF0. For the second occurrence of ADF0 in the result-producing branch: ARG0 will be replaced with D2 and ARG1 will be replaced with D3.

6.5 Branch Deletion

Branch deletion removes one of the function-defining branches in the selected individual. Each subtree starting with the node corresponding to a call of the removed function-defining branch must be replaced with another subtree. Basically, one of the following methods is applied:

1. Consolidation: the subtree is replaced with a call to another function-defining branch. Note that this operation often does not preserve the semantics of the individual.
2. Regeneration: the subtree is replaced with a new, randomly generated subtree. Regeneration obviously almost never preserves the semantics of the individual.
3. Substitution (Koza calls this operation "Macro expansion"): the subtree is replaced with the body of function-defining branch being deleted. Local arguments of the function-defining branch are replaced with the arguments passed to the call of the function-defining branch to be deleted. Thus, the operation preserves the program semantics.

The operation is controlled with the probability of subroutine deletion. There is also a minimum number of branches parameter that affects the applicability of subroutine deletion.

6.6 Argument Deletion

Argument deletion removes an argument from the selected function-defining branch. Each subtree corresponding to the parameter being removed in branches that call the function-defining branch under argument deletion is removed. The node representing the argument to be removed from the function-defining branch is resolved using one of the following methods:

1. Consolidation: the node representing the argument under deletion is replaced by a node representing another argument. This operation thus almost never preserves the semantics of the individual.
2. Regeneration: the node is replaced with a new, randomly generated subtree. Obviously, this operation almost never preserves the semantics of the individual.
3. Substitution (Koza calls this operation "Macro expansion"). Before we explain how the substitution operates, note that the operation preserves the program semantics.
   The first step is to delete the selected argument from the argument list of the picked branch. In the second step, we create as many copies of the branch under argument deletion as there are invocations in that branch in the individual (although Koza does not state that explicitly, the original branch could be removed obviously).
   For instance, if the result-producing branch calls the branch under argument deletion three times and another function-defining branch (i.e., different than the one under the argument deletion) calls the branch two times, five copies will be made in total.
   Each copy of the branch has assigned a unique name. The third step is to replace each invocation of the branch under deletion with an invocation of one of the copies created in the previous step. The fourth step is to remove the subtree (the subtree represents a parameter passed to a function invocation) corresponding to the argument under deletion that appears in the invocation of the corresponding copy of the branch under argument deletion. In other words, we are modifying a branch that calls the branch under the argument deletion. The removed subtree is used to replace the nodes representing the deleted argument in the corresponding copy of the branch under the argument deletion. This step is repeated for every invocation of a particular copy of the branch under deletion. The operation has one potential disadvantage; it rapidly increases the redundancy of the individual's code since we must create a copy of the branch under argument deletion for each invocation of the branch in the individual.
   The operation is controlled with the probability of argument deletion. There is also a minimum number of arguments parameter that affects the applicability of argument deletion.

Fig. 13 demonstrates an application of the argument deletion operation by substitution on an individual depicted on Fig. 7. Original function-defining branch is removed and two copies, `ADF1` and `ADF2` are created. Argument `ARG0` has been chosen as argument to be deleted, so that `ADF1` and also `ADF2` takes only one argument, namely `ARG1`. In the next, we have to replace the calls to `ADF0` by a call to `ADF1` and `ADF2`, respectively. For the call to `ADF1` we have to remove the first parameter, that is `(NOT D1)` from the result-producing branch and replace all occurrences of `ARG0` in `ADF1` with `(NOT D1)`. Analogously, for the call to `ADF2` we have to remove `D2`, the first parameter passed to the function call, from the result-producing branch and replace all occurrences of `ARG0` in `ADF2` with `D2`.

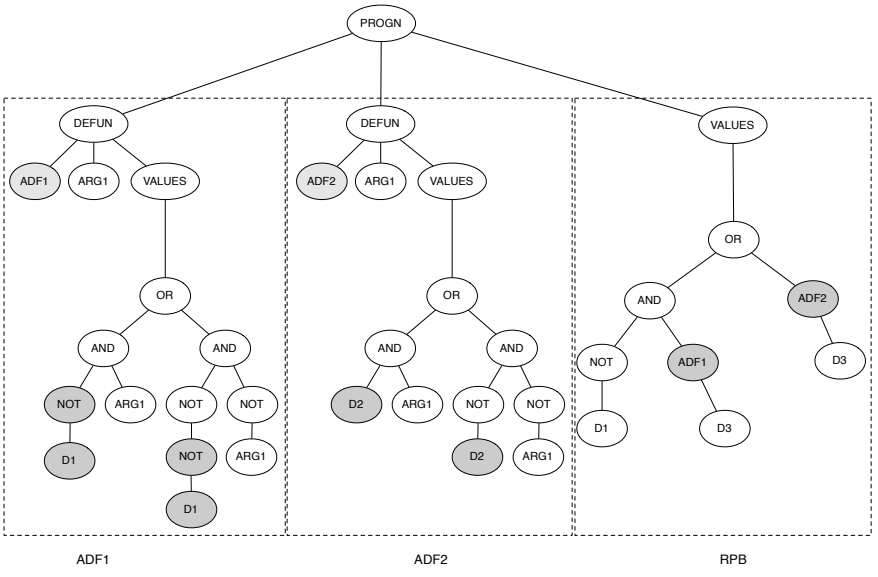

Fig. 13 Argument deletion example

7 Adaptive Representation through Learning

Adaptive Representation Through Learning (ARL) [21, 23, 24] has been proposed by Rosca and Ballard as an extension and modification of their earlier introduced modularization method called Adaptive Representation (AR) [22]. Both AR and ARL heuristically discover promising parts in individuals and define those as new modules. Modules are represented as globally defined functions in contrast to ADF and ADF with Architecture-altering operations where modules are defined locally within individuals. In ARL, the structure of modules is not predefined and modules are not subject to evolution. However, some modules may be deleted or some new modules defined during the evolution.

The most important part of ARL is the identification of promising code in the individuals. A part (i.e., a subtree in tree-based representation) is considered promising if it has a small height (e.g., usually between 3 and 5), high *differential fitness* and high *block activation*.

Differential fitness is the difference in fitness between the individual and its least fit parent. In other words, individuals with highest improvement over its parents are preferred since those individuals presumably contain useful pieces of code. Note that authors of ARL suppose that only a small fraction of population has the differential fitness greater than zero.

Block activation is the number of evaluations (executions) of the root node of a subtree (called *a block* in ARL) within the individual. Highly active blocks (that is blocks with high number of activations of the root node and non-zero, or greater

than a minimum percentage, activations of every other node in the block) of a small height (usually between 3 and 5) are considered salient and further transformed to new modules. During the constitution of a new module from a salient block, a random subset of terminals is replaced by new, local terminals (that is local variables) within the block. The process of a module discovery and subsequent constitution of a module can be summarized as follows:

1. Select individuals with highest positive differential fitness.
2. For each selected individual assign to each node the number of activations in the evaluation of fitness.
3. Create a set of candidate blocks (subtrees) by selecting blocks of small height and high activation.
4. Remove all blocks that contain inactive nodes.
5. For each candidate block do:

    a. Determine the terminal subset.
    b. Create new module with parameters corresponding to a random subset of terminals present in the candidate block.
    c. Duplicate the individual with the highest differential fitness.
    d. Replace the candidate block within the individual with a call to newly defined module.

Rosca in [20] states that the use of modularity increases the diversity in a population in comparison to standard genetic programming. In ARL, modules can be dynamically created as well as removed during the evolution process. New modules can be defined either:

1. In each generation. However, the subroutine discovery process can be too computationally intensive to be effectively performed in each generation. Also, there may be a slow progress in early generations or between a small number of consecutive generations.
2. On the basis of epochs. An epoch is a period of consecutive generations throughout which the system operated with a fixed representation.
3. On the basis of decreasing a population diversity. The diversity is measured using a population entropy. Individuals in a population are categorized into classes according to certain properties or behavior, the number of individuals in each class is determined. The diversity is then computed using Shannon's information entropy, see [23, 25] for details.

Modules can also be deleted during the evolution process as we stated above. This is done using the evaluation of module usefulness, where modules of low usefulness are deleted. The usefulness of a module is determined by the average fitness of all individuals that have called the module over a fixed "time window" of past generations.

Rosca and Ballard compared ARL to standard genetic programming and genetic programming with ADF on the PacMan problem [23] and showed the advantages of ARL over the standard GP and GP with ADFs. Dessi et al. analyzed ARL and proposed several improvements, see [8] for more detail.

8 Hierarchical Genetic Programming

Hierarchical Genetic Programming (hGP) [5] is a method for local, context-sensitive modularization of individuals in population. The basic idea behind the hGP is to recursively identify valuable subtrees in individuals. New modules are constituted from promising subtrees of existing modules. Thus, modules are organized hierarchically, see Fig. 14 for an instance of individual with two hierarchical modules. The aim behind this approach is to enable evolution at different speed on each level of hierarchy. The lower in the hierarchy a module is located, the slower it is enabled to evolve. Authors of hGP consider lower level modules as more fundamental for the problem solution thus their evolution should be slower that in higher level modules.

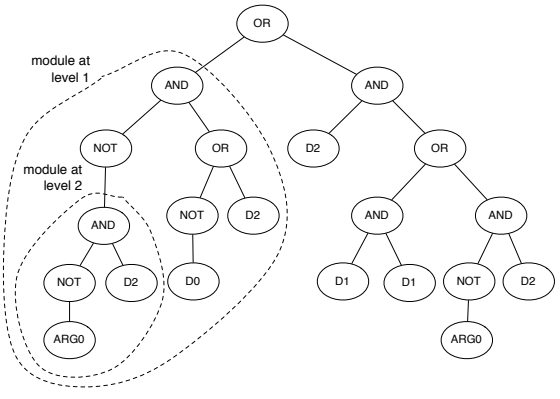

Fig. 14 Hierarchical modules within an individual

In fact, the hGP operates with modularity on a structural level only; it provides a hierarchy within an individual with different speed of evolution on particular levels of hierarchy. But this also means that hGP does not provide the main objective of modularization: a code reuse. In hGP, a module can not be used repeatedly as it is possible in another approaches. Modules in hGP are local to the individual. Thus, the recombination operators perform on the same levels in a hierarchy within individuals similarly to structure preserving crossover in genetic programming with ADF [14].

Modules are discovered in an individual using a searching of valuable subtrees inside. The analyzed subtree is replaced either with a constant value or a randomly generated subtree. The original and modified individual (i.e., a copy of individual with modified subtree under the analysis) are compared on the basis of fitness to discover the usefulness of the subtree within the individual.

Banzhaf et al. [5] reported promising results on even parity problem in comparison to standard genetic programming.

9 Recursion and Iteration

Recursion and iteration represent another important kind of code reuse. In this section we provide a brief outline of the present approaches including references to relevant literature. Obviously, repetitive code execution is essential for establishing a general solution of many problems. For instance, the even parity problem can be solved with standard genetic programming for a given number of input arguments (at least theoretically, owing to exhaustive search space for higher number of input arguments); however, a general solution for any number of input arguments is not feasible in GP without extending the system with a support to repetitive code execution.

Koza proposed proposed language elements for establishing iteration in evolved programs, such as Automatically Defined Iterations (ADI) and Automatically Defined Loops (ADL) [16]. These elements provide some predefined types of iteration similarly as programming languages provide iteration statements, such as FOR or DO WHILE loop.

More attention has been paid to establishing a repetitive code execution using recursion. With recursion it is not required to have special language elements for repetitive code reuse which could be an advantage for designing a system for evolution of programs. To establish recursive definitions it is required no more than enabling the appearance of call(s) to function itself within its body. Note also that recursion is a general concept that enables definition of any possible repetitive code execution. However, a recursive function must have correctly defined both the terminating condition(s) and recursive call(s) in order to not result in an infinite loop. The possibility of creating non-terminating programs introduces several challenges for automatic program synthesis. First, determining whether the program is non-terminating can be handled in some special cases only since it is an undecidable problem. Second issue is handling non-terminating programs. Even a non-terminating program may contain useful parts, so that the problem is how to reflect the program quality in fitness. Third challenge is measuring recursion semantics of evolved programs.

A promising approach to establish recursion in evolved programs is the utilization of the so called *implicit recursion*, a technique known from functional programming languages. In general, implicit recursion is a function that contains some type of recursion and that can be parametrized by another functions to provide a particular computation. In other words, implicitly recursive functions provide an abstracted "skeleton" of certain kind of recursion. The main advantage is that implicit recursion always terminates. Those implicit recursions thus can be a part of a function set. Yu in [26, 27] experimented with implicit recursions from functional programming with promising results. Dostál experimented with the so called ARF functions [9, 10] that represent a different, more general, types of implicit recursion than used by Yu. ARF functions showed promising results in evolving recursive solutions to some problems such as even-n-parity problem or a simple arithmetic based on Church's numerals [11].

To demonstrate the idea behind implicit recursion, consider two following examples of a recursive function. First, a program on Fig. 15 appends two lists. Second,

a program on Fig. 16 computes the even-n-parity problem (the examples require some basic understandings of Lisp-like languages, see [9] for details). Although both these programs compute different problems, their code is very similar in the structure including the termination condition and recursive call.

```
(LABEL APPEND
 (LAMBDA (A B)
   (IF A
       (CONS (CAR A) (APPEND (CDR A) B))
       B)))

(APPEND '(NIL NIL) '(NIL NIL NIL)) => (NIL NIL NIL NIL NIL)
```

Fig. 15 A solution of the append problem using recursion

```
(LABEL EP-N
 (LAMBDA (A B)
   (IF A
       (XOR (CAR A) (EP-N (CDR A) B))
       B)))

(EP-N '(NIL NIL NIL T T NIL) '()) => T
```

Fig. 16 A solution of the even-n-parity problem using recursion

Both these problems could be defined effectively using a function that implicitly abstracts the type of recursion used in both programs. This can be done using the well know `FOLDR` function, see Fig. 17. The `FOLDR` function takes three arguments: a function, a list and a value called *terminator*. The function is initially called with the rightmost item of the list and terminator. In the subsequent applications of `FOLDR`, the function is applied to another elements of list traversed from right to left and the result obtained from previous application of the function. For example: `(FOLDR + '(1 2 3) 10)` yields the following computation: `(+ 1 (+ 2 (+ 3 10))) => 16`. Using `FOLDR`, both `APPEND` and `EP-N` could be defined effectively as shown on Fig. 18. Thanks to the same type of recursion, both definitions are different only in the first argument passed to `FOLDR`.

```
(LABEL FOLDR
 (LAMBDA (F A B)
   (IF A (F (CAR A) (FOLDR F (CDR A) B)) B)))
```

Fig. 17 Implicit recursion using FOLDR

```
(LABEL EP-N
 (LAMBDA (A B)
  (FOLDR XOR A B))))

(LABEL APPEND
 (LAMBDA (A B)
  (FOLDR CONS A B))))

(EP-N '(NIL NIL NIL T T NIL) '()) => T
(APPEND '(NIL NIL) '(NIL NIL NIL)) => (NIL NIL NIL NIL NIL)
```

Fig. 18 `APPEND` and `EP-N` defined using implicit recursion

10 Conclusion

The above discussed methods are more or less different from each other in a variety of viewpoints.

Modularization methods approach modularity from different standpoints. Less complex methods, such as Encapsulation or Hierarchical Genetic Programming, are more or less "graph" or "individual" oriented than the other methods which approaches modularity from the programmatic standpoint. The main motivation in "graph" oriented methods is to separate a promising part of an individual in order to preserve it from evolution (this is the case for Encapsulation) or to provide a different conditions for evolution (hGP). Note that in these methods modules lack an important feature of modularity—a module reuse which is obviously a limitation. Methods such as Module Acquisition, Automatically Defined Functions and Adaptive Representation Through Learning are primarily motivated by code reuse. Promising code is transformed to new modules that can be reused (also repeatedly) in individuals' programs. Modules are usually represented as functions that extends the representation language of a problem, thus not only individuals but also the representation is subject to evolution. In other words, we also search an appropriate representation that will enable to express a solution of the problem as a simpler and shorter program.

Modules may either be defined locally or globally. Local modules can be used by corresponding individual only, whereas global modules can be used by any individual in a population. With local modules each individual have its own definition of modules. Local modules are used by ADF, whereas MA and ARL uses globally defined modules.

Evolvability of modules is also an important property of program synthesis systems with modularity. In Encapsulation, MA and ARL are modules, once created, static and thus not subject to evolution, whereas with ADF or hGP modules can be evolved simultaneously with the evolution of individuals. Note that the

conditions for evolving modules are usually substantially different from the conditions for individuals.

Another difference lies in the identification of modules. New modules can either be identified heuristically or at random. Random identification of modules is used by Encapsulation and MA whereas ARL and hGP identifies modules on the basis of fitness. A sophisticated method based on differential fitness and block activation is used by ARL.

An important property is also the structure of modules. The structure may either be predefined (ADF), or it may be inferred from the identified code used to constitute a module (MA, ARL). In ADF with Architecture altering operations, the module structure can be altered during the evolution using architecture altering operations. These operations allow for adding/removing a module or adding/removing an argument of a module.

An evaluation and comparison of the above presented methods on the basis of performance should be interesting. Although the authors of individual methods usually provide some basic evaluations and comparisons, there is a considerable lack of a general and profound comparative analyses that could objectively compare the methods for modularization. However, such a comparison would be quite extensive and laborious since genetic programming itself as well as the particular modularization methods are customized by a plenty of control parameters that could drastically affect the performance. Koza compared ADF to standard GP [14] and ADF to ADF with Architecture altering operations [16] with promising results. Banzhaf et al. compared a constrained version of hGP to standard genetic programming on several symbolic regression problems including even-5 and -7 parity problem. Reportedly [5], hGP outperformed standard genetic programming. Rosca and Ballard compared ARL to standard genetic programming and genetic programming with ADF on the PacMan problem [23] and showed the advantages of ARL over the standard genetic programming and genetic programming with Automatically defined functions.

Modularity is in [19] referred as one of the open problems in genetic programming. Since there is a considerable research interest into evolutionary program synthesis, we believe in remarkable progress in methods and techniques to modularization in future.

References

1. Angeline, P.J.: Genetic programming and emergent intelligence, pp. 75–97. MIT Press, Cambridge (1994)
2. Peter, J.: Angeline and Jordan B. Pollack. The evolutionary induction of subroutines. In: Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society, Lawrence Earlbaum (1992)
3. Angeline, P.J., Pollack, J.B.: Coevolving high-level representations (1993)

4. Peter, J.: Angeline and Jordan B. Pollack. Evolutionary module acquisition. In: Proceedings of the Second Annual Conference on Evolutionary Programming, pp. 154–163. MIT Press (1993)
5. Banzhaf, W., Banscherus, D., Dittrich, P.: Hierarchical genetic programming using local modules. Technical Report 50/98, University of Dortmund, Dortmund, Germany (1998)
6. Banzhaf, W., Francone, F.D., Keller, R.E., Nordin, P.: Genetic programming: an introduction: on the automatic evolution of computer programs and its applications. Morgan Kaufmann Publishers Inc., San Francisco (1998)
7. Brameier, M.F., Banzhaf, W.: Linear Genetic Programming (Genetic and Evolutionary Computation). Springer-Verlag New York, Inc., Secaucus (2006)
8. Dessi, A., Giani, A., Starita, A.: An analysis of automatic subroutine discovery in genetic programming. In: Banzhaf, W., Daida, J.M., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M.J., Smith, R.E. (eds.) GECCO, pp. 996–1001. Morgan Kaufmann (1999)
9. Dostál, M.: On evolving of recursive functions using lambda abstraction and higher-order functions. Logic Journal of IGPL 13(5), 515–524 (2005)
10. Dostál, M.: A functional approach to evolving recursive programs. In: Kitzelmann, E., Schmid, U. (eds.) Second Workshop on Approaches and Applications of Inductive Programming, pp. 27–38 (2007)
11. Hankin, C.: An Introduction to Lambda Calculi for Computer Scientists. College Publications (February 2004)
12. Koza, J.R.: Genetic programming - on the programming of computers by means of natural selection. Complex adaptive systems. MIT Press (1993)
13. Koza, J.R.: Architecture-altering operations for evolving the architecture of a multi-part program in genetic programming. Technical report, Stanford, CA, USA (1994)
14. Koza, J.R.: Genetic programming II: automatic discovery of reusable programs. MIT Press, Cambridge (1994)
15. Koza, J.R., Andre, D.: Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In: Genetic Programming 1996: Proceedings of the First Annual Conference (1996)
16. Koza, J.R., Andre, D., Bennett, F.H., Keane, M.A.: Genetic Programming III: Darwinian Invention & Problem Solving, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (1999)
17. Koza, J.R., Rice, J.P.: Genetic programming: The movie (1992)
18. O'Neill, M.: Automatic programming with grammatical evolution (July 13, 1999)
19. O'Neill, M., Vanneschi, L., Gustafson, S., Banzhaf, W.: Open issues in genetic programming. Genetic Programming and Evolvable Machines 11(3), 339–363 (2010)
20. Rosca, J.P.: Genetic programming exploratory power and the discovery of functions. In: Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming, pp. 719–736. MIT Press (1995)
21. Rosca, J.P., Ballard, D.H.: Genetic programming with adaptive representations. Technical report (1994)
22. Rosca, J.P., Ballard, D.H.: Learning by adapting representations in genetic programming. In: International Conference on Evolutionary Computation 1994, pp. 407–412 (1994)
23. Rosca, J.P., Ballard, D.H.: Discovery of subroutines in genetic programming, pp. 177–201. MIT Press, Cambridge (1996)
24. Rosca, J.P., Ballard, D.H.: Evolution-based discovery of hierarchical behaviors. In: AAAI/IAAI 1996, vol. 1, pp. 888–894 (1996)
25. Shannon, C.E.: A mathematical theory of communication. Bell System Technical Journal 27 (1948)

26. Yu, T.: Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction. Genetic Programming and Evolvable Machines 2, 345–380 (2001)
27. Yu, T.: A higher-order function approach to evolve recursive programs. In: Yu, T., Riolo, R.L., Worzel, B. (eds.) Genetic Programming Theory and Practice III, Genetic Programming, May 12-14, ch. 7, pp. 93–108. Springer, Ann Arbor (2005)