ACGP/CGP lil-gp 2.1;1.02
Technical & User's Manual

version July 21, 2004

Cezary Z. Janikow[1]

Department of Mathematics and Computer Science
University of Missouri – St. Louis
janikow@radom.umsl.edu

This document describes extensions to *lil-gp* facilitating dealing with constrains and heuristics, CGP - as outline in [5], and to adapt those, ACGP - as outlined in [6][7]. *ACGP lilgp 2.1;1.02* in built on the top of *CGP lil-gp 2.1;1.02*. This document combines both the tech manuals for *CGP2.1* as well as for *ACGP2.1*.

Both *CGP2.1* and *ACGP2.1* are based on *lil-gp 1.02*, and thus resulting extensions are referred to as *CGP/ ACGP lil-gp 2.1;1.0*2 (the first version# is for the extension, the second for the utilized *lil-gp* version). Unless explicitly needed to avoid confusion, version numbers are omitted.

CGP is a methodology to process both *strong* constraints and *weak* constraints (*heuristics*) in GP utilizing tree representation. Both kinds of constraints are only *first-order*, that is constraints on parent-child relationship (plus constraints on the root node). *CGP2.1* is the resulting implementation. In CGP, trees invalidating the strong constraints are never produced, while the heuristics are used as preference criteria for initialization, crossover and mutation.

ACGP is a methodology to adapt the employed heuristics to improve problem-solving. At present, the improvement is based on both fitness and chromosome size, as determined by observing population statistics.

ACGP v2.1 differs from ACGP v1.1 by collecting different distribution information since the information expressed in v2.1 trees is richer than that in v1.1.1. The extended distribution information is also used to generate differnt updates of the basic heuristics.

1 Overview

lil-gp 1.02 is a public domain tool [11] for developing Genetic Programming (GP) [8][9][10] applications. Its implementation is based on the *closure* property [9], which states that every function can call any other function and that any terminal can provide values for any function argument. Unfortunately, this property leads to many invalid programs being evolved (invalid with respect to program syntax and semantics, and not program size limitation). In GP, this is dealt with by either penalizing such trees (*e.g.*, by setting evaluation to 0), or by providing extended interpretations (*e.g.*, protected division [9], but these choices are arbitrary in general). The objectives of Constrained Genetic Programming (CGP) is to provide mean to specify syntax and semantic constraints, and to provide mechanisms to enforce them [4]. *CGP lil-gp 1.02;1.02*[2] is the result of implementing CGP into *lil-gp 1.02*. *CGP lilgp 1.1/1.02* (not distributed as of now) extends the former. *CGP lil-gp2.1/1.02* extends the latter by dealing with different types. Please refer to *lil-gp* for licensing information.

CGP lilgp 2.1/1.02 implements constraints processing plus it adds a few parameters over *lil-gp 1.02*. It is capable of processing function constraints as outlined in [4][5]. It also has the abilities to process problem heuristics: in *CGP lil-gp 1.02* the constraints would determine what programs were considered valid, ond only those would evolve. However, except for the pressure of the evaluation, there was no other ways to differentiate between different pro-

1. This research was partly supported through NASA/JSC grant NAG 9-847 and FTTP summer programs.
2. http://www.cs.umsl.edu/~janikow

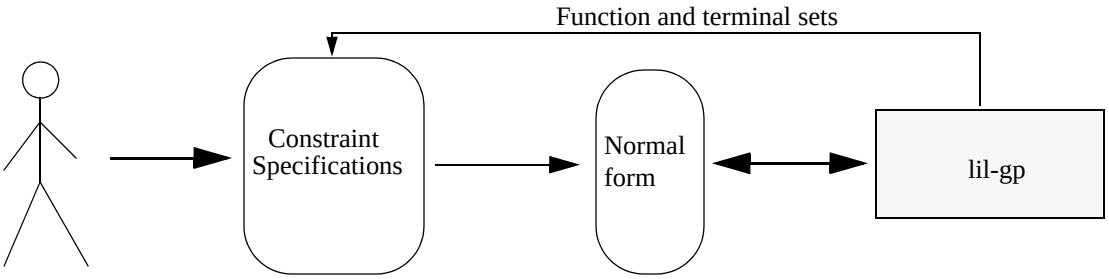
grams.Starting with *v1.1, CGP lil-gp* allows the user to specify different weights to various elements of a tree, based on their context. For example, suppose there are three functions f_1, f_2 , f_3. *CGP lil-gp 1.02* allows the user to restrict arguments of f_1 to call upon f_2 and f_3 only. *CGP lil-gp 2.1* allows the user to additionally specify that out of those two allowed functions, f_2 is twice as good a candidate. Work on automatically evolving those weights is in progress (at the moment they remain constant throught the run).

Finally, *CGP lilgp 2.1/1.02* can also process constraints based on data typing. For example, suppose that function f_1, requires two argumemnts. Suppose that it can deal with two integers, in which case the resulting type is an integer. It can also deal with two floats, in which case the returned type is float. Upon mutation or crossover, the second child of this node can only be a function returning integers if the first argument currently resturns integers. However, if the first argument currentlyu returns floats, then the second child can be any function returning either integer or real (given that integers are compatible with reals [4]).

Despite some redundancies among such type constraints and constraints processed in *v1.02/1.1*, no constraints have been removed. This way the user has the option to use either constraint specification means.

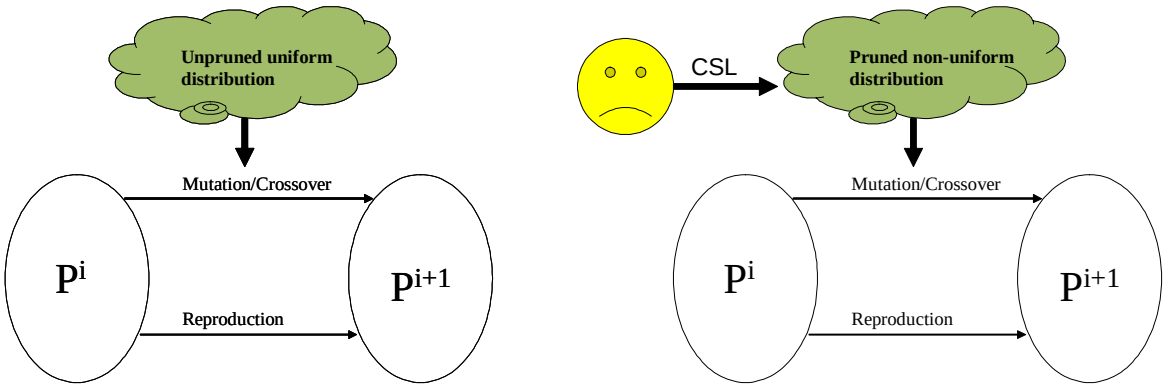
2 CGP Application Environment

Figure 1 The *CGP lil-gp* environment.



The environment for *CGP lil-gp*, and its CGP and *lil-gp* elements are presented in Figure 1. As indicated, the user interacts with *CGP lil-gp* (all interfaces of *lil-gp* are preserved intact and are not shown) by specifying problem constraints in the constraint language (see Section 4.1 for explanations and examples). CSL constraints are transformed into the normal form [5] and stored in so called *mutation sets*. Finally, *lil-gp* interacts with the mutation sets. The result of this interaction is a restriction on *lil-gp*'s space of evolved programs into only those satisfying the constraints.

Figure 2 Another look at the application environment of GP (left) and CGP(right).

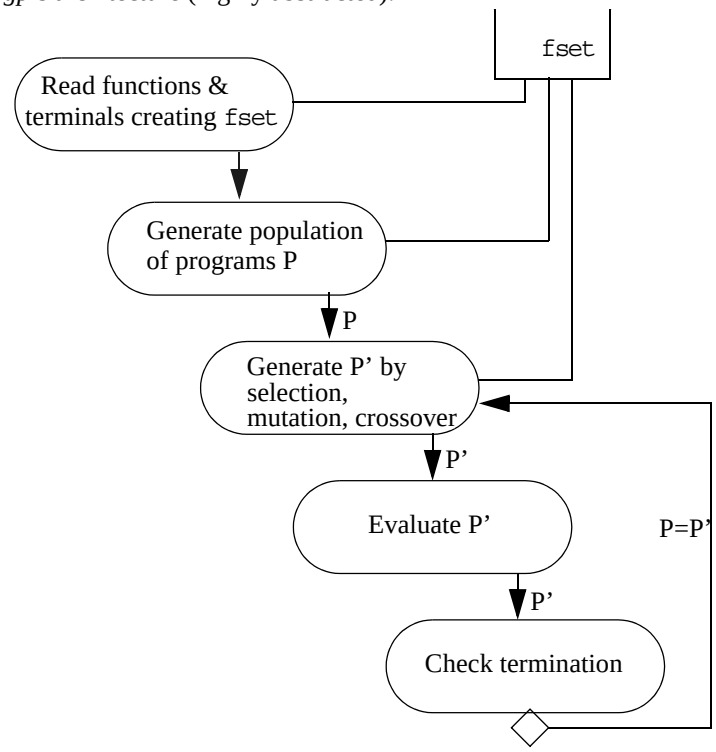


The CSL-expressed constraints must refer to the specific application, identified by the set of functions and terminals (but not the evaluation function) the user defines for *lil-gp*. Thus, these sets must be made available to CGP processing.

3 Overview of *lil-gp* Modifications

Disregarding many elements not affected by (affecting) the merger between CGP and *lil-gp*, the dataflow architecture of *lil-gp 1.02* can be simplified to that of Figure 3. `fset` is the *lil-gp*'s data structure storing the user provided definitions for functions and terminals. These definitions are needed to generate the initial programs, to apply mutation and crossover, and to evaluate evolving programs.

Figure 3 *lil-gp*'s architecture (highly abstracted).



To create *CGP lil-gp*, we have implemented minimal changes into *lil-gp*. This will facilitate other extensions, as well as it will allow relatively easy upgrades to future versions of *lil-gp*. In fact, no lil-gp's user nor internal interfaces has been modified. These modifications/extensions are illustrated in Figure 4 with dotted lines. The changes can be summarized as follows:

1. After `fset` sets are read and stored by *lil-gp* (and after the functions are ordered in *lil-gp 1.02* and later), a new module `create_MS_czj` is called. This module accesses global `fset` to obtain information about functions/terminals, and based on that information it interacts with the user to read the problem constraints. At the moment, a very simple interface is provided, but it may be replaced with any other interface at a later stage.
2. Constraints are transformed into the normal form, and expressed as mutation sets `MS_czj`, which data is global and available to *lil-gp*.
3. Population initialization, mutation, and crossover of *lil-gp* are modified to interact with `MS_czj` in preserving the integrity of the generated and explored programs with respect with the user-specified constraints. The first two have minimal changes. Crossover have been reimplemented.

In addition, *CGP lil-gp 1.1* provides a few minor modifications over *lil-gp 1.02*, which we felt were sometimes desired (default behavior is equivalent to *lil-gp*'s):

1. Several new parameters are added to those of *lil-gp*:
   a. Initializtion parameters `init.depth_abs`, `={true,false}`, default=`false`
      In the default mode, this parameter has no effect over *lil-gp 1.02*
      When set, `init.depth=m-n` will cause rejection of initial trees shallower than `m`
   b. Mutation parameter `depth_abs`, `={true,false}`, default=`false`
      In the default mode, this parameter has no effect over *lil-gp 1.02*
      When set, `depth=m-n` will cause rejection of mutation offspring shallower than `m`

(it is a good idea to have `keep_trying` set to `true` as well)

c. Additional parameters have been added to the Crossover parameters.
The `internal` and `external` parameters are replaced with 2 pair of parameters which seperate their functionality between the source and destination parents. The new parameters are:

- `internal` (internal nodes in source parent), default = 0.9
- `internal_dst` (internal nodes in destination parent), default = `internal`
- `external` (external nodes in source parent), default = 0.1
- `external_dst` (external nodes in destination parent), default = `external`

These act the same way as in *lil-gp 1.02* except that they allow there to be different frequencies for internal/external node selection in the source and destination parents.

d. Crossover parameters `internal` & `external` have been extended. Using them in the normal way results in standard *lil-gp 1.02* operation. Two extra parameters have been added `internal_dst` & `external_dst`. Both default to the matching `internal` & `external` values. These allow different internal/external selection frequencies to be selected for source and destination trees.

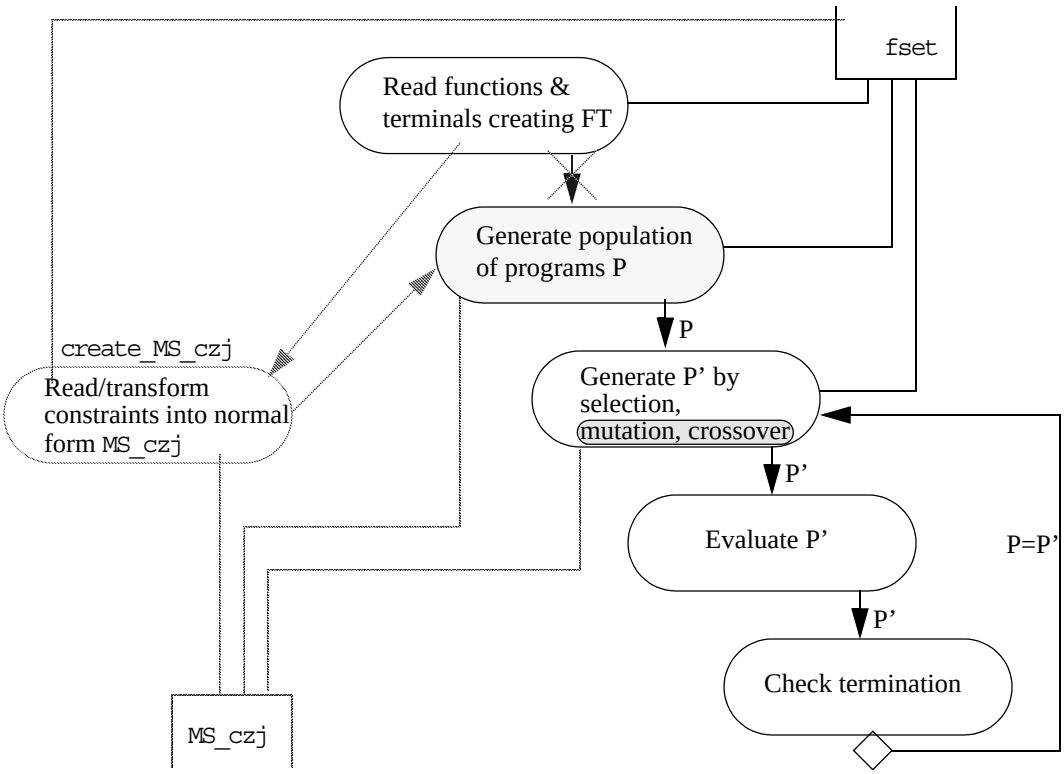
2. A new crossover-like operator has been added; collapse. Collapse acts like crossover except that only one parent is used, and the source subtree is selected from within the destination subtree. More information on this can be found in the Users Manual.
3. `function`'s `index` member is adjusted after functions are sorted by *lil-gp* (if at all). *lil-gp 1.02* is apparently not using this member except for setting it improperly, *CGP lil-gp 1.1* is using it thus it updates it after sorting. In addition, the sorting function used by `qsort` has been modified to sort first by arity and then alphabetically by function name. This was to resolve different sorting results on different platforms.

Except for these, no other changes in *lil-gp 1.02* are implemented. Note that since *lil-gp* does not deal with sets but only with functions (terminals are not specified by sets but by generating functions) the implementation and interface will somehow divert from those outlined in [4] (consult [5] instead).

It is important to note that actual mutation and initialization are not modified directly. Instead, the low level functions generating subtrees have been modified. All modifications can be accessed by searching for `czj` string, except for the crossover operator (`operator_crossover()`) which has been completely rewritten.

Note that *CGP lil-gp 1.1/1.02* does not deal with ADFs [10] at the moment, but an extension is planned for the near future.

Figure 4 *CGP lil-gp*'s architecture (highly abstracted).



4 Technical Manual for CGP 2.1

The methodology for processing problem constraints is presented in [5]. Here we present necessary details for understanding what kinds of constrains, and how, can be used in *CGP lil-gp 1.1/1.02*.

4.1 *CSL* constraint specifications

lil-gp 1.02 does not provide for explicit (neither extensive nor even intensive) set specifications for terminals. Instead, sets are defined by functions generating random elements, which are provided by the user. Accordingly, disregarding ADFs, there are three different kinds of functions – we will call them functions of type I, II, and III, and their sets will be denoted as F_I, F_{II}, and F_{III}. Unless explicitly stated, all references to functions imply all function types (denoted F), and all references to terminals imply functions of type II and III. It is important to mention that *lil-gp 1.02* orders functions of `fset` so that type I functions precede all other functions.

Definition 1 *A function appearing in a node of a tree is said to* ***label*** *that node.*

I. *Ordinary functions*. These are functions of at least one arguments. Therefore, they will label the internal nodes.

II. *Ordinary terminal*. These are functions of no arguments. Therefore, they can label the external nodes. However, they are not instantiated in a tree – they are instantiated at evaluation by external data.

III. *Ephemeral random constant terminal*. These are functions of no arguments. Therefore, they can label the external nodes. Moreover, they are instantiated (possibly differently) in each appearance in each tree.

Example 1 Follow the lawnmower example from *lil-gp 1.02 User's Manual*, section 6.3.1. Disregarding ADFs, there are three type I functions (F_I={*frog*, *vma*, *prog2*}), with $a_1 = 1, a_2 = 2, a_3 = 2$, two type II functions (F_{II}={*left*, *mow*}), and one function of type III (F_{III}={*Rvm*}).

Definition 2 *Let us define* ***set compatibility*** *denoted* $\Rightarrow$. *That is,* $X \Rightarrow Y$ *means that the set X can replace the set Y. When speaking of functions, a set indicates the range of the function. When speaking of function arguments, a set indicates the domain of the argument. When speaking of a program tree, a set indicates the range of the function labeling the Root node. When speaking of problem specifications, a set indicates the*

range of values to be returned by a program.

Proposition 1 $X \Rightarrow Y \leftrightarrow X \subseteq Y$.

$\therefore X \Rightarrow Y$ *means that in places where values from Y are valid one may place any value from X, or any function returning a value from X. To guarantee that no out-of-domain values are used for the original Y, X may not contain values not found in Y. Therefore, it must be a subset of Y, or it must equal Y.*

Proposition 1 will help the user determine and specify the constraints. Unfortunately, it cannot be automated since *lil-gp* does not operate on explicit (neither intensive nor extensive) set definitions.

Definition 3 *Define the following* ***Tspecs*** *(syntactic constraints):*
i) T^{Root} *– the set of functions which return data type compatible with the problem specification.*
ii) $T^{*}_{*} - T^{j}_{i}$ *is the set of functions compatible with the jth argument of* f_i .

In terms of a labeled tree, T^{Root} is the set of functions which, according to data types, can possibly label the *Root* node. T^{j}_{i} is the set of functions that can possibly label the *j*th child node of a node labeled f_i .

Tspecs reduce both the space of program templates (and thus structures) and the space of instances of those structures. Therefore, they are very powerful constraints. *lil-gp* allows any function of type I to label any internal node, and any function of type II and III to label any external node. Obviously, this is not true in actual programs – different functions take different arguments and return different ranges. *Tspecs* allow expressing such differences, thus allowing reduction in the space of program instances per program structure. Moreover, some *Tspecs* also implicitly restrict what function can call other functions, effectively reducing the space of possible program templates. Therefore, some *Tspecs* can be seen analogous to *function prototypes* available in high level languages.

Example 2 Assume $F_I = \{f_1, f_2, f_3\}$ with arities 3, 2, and 1, respectively. Assume $F_{II} = \{f_4\}$ and $F_{III} = \{f_5, f_6, f_7\}$. Assume that the three type III functions generate random boolean, integers, and reals, respectively. Assume f_4 reads an integer. Assume f_1 takes boolean and two integers, respectively, and returns a real value. Assume f_2 takes two reals and returns a real. Assume f_3 takes a real and returns an integer. Also assume that the problem specifications state that a solution program should compute a real number. These assumptions are expressed with the following *Tspecs*:

$$T^{Root} = \{f_1, f_2, f_3, f_4, f_6, f_7\}$$
$$T^{1}_{1} = \{f_5\},\ T^{2}_{1} = \{f_3, f_4, f_6\},\ T^{3}_{1} = \{f_3, f_4, f_6\}$$
$$T^{1}_{2} = \{f_1, f_2, f_3, f_4, f_6, f_7\},\ T^{2}_{2} = \{f_1, f_2, f_3, f_4, f_6, f_7\}$$
$$T^{1}_{3} = \{f_1, f_2, f_3, f_4, f_6, f_7\}$$

Note that in Example 2 Proposition 1 is used by making integers compatible with reals (but not the other way around). The example also assumes that booleans are not compatible with integers (nor with reals), which assumption can be reversed by assuming additional interpretations (such as those in the C programming language).

However, syntactic fit does not necessarily mean that a function *should* call another function. One needs additional specifications based on program semantics. These are provided by means of *Fspecs*, which further restrict the space of program templates.

Definition 4 *Define the following* ***Fspecs*** *(semantic constraints):*
i) F^{Root} *– the set of functions disallowed at the Root.*
ii) $F_{*} - F_{i}$ *is the set of type I functions disallowed as direct callers to* f_i *(generally, a function is unaware of the caller; however, GP constructs a program tree, which represents the dynamic structure of the program).*
iii) $F^{*}_{*} - F^{j}_{i}$ *is the set of functions disallowed as* arg_j *to* f_i .

Example 3 Continue Example 2. Assume that we know that the sensor reading function f_4 does not provide the solution to our problem. We also know that a boolean value (generated by f_5) cannot be the answer (this information is redundant as it is provided in *Tspecs*). Also assume that for some semantic reasons we wish to exclude f_3 from calling itself (*e.g.*, this is the integer-part function, which yields identity when applied to itself). These constraints are expressed with the following *Fspecs* (the other sets are empty):

$F^{Root} = \{f_4, f_5\}$

$F_3 = \{f_3\}$

4.2 Transformation of the CSL Constraints

4.2.1 Normal form

Given the above *Tspecs* and *Fspecs*, which can be used to express some problem constraints (those expressible with this language), an obvious issue is that of possible redundancies, or that of existence of sufficiently minimal specifications. Surprisingly, after certain transformations, only a subset of *Tspecs* and *Fspecs* will turn out to be sufficient to express all such (though not all in general) constraints. This observation is extremely important, as it will allow efficient constraint enforcement mechanisms after some initial preprocessing.

The sufficient minimal set is thus important for efficient constraint processing, but not for constraint specifications, which are more easily expressed with the original *Tspecs* and *Fspecs*. This is why we need both, along with the necessary transformations (see Figure 1).

The *normal form* is a set of F^{Root} and all F^*_* constraints (properly transformed, see [5]).

Proposition 2 *The normal form is sufficient to express all constraints of the Tspec/Fspec language.*

$\therefore$ [5]

Example 4 Constraints of Example 2 and Example 3 have the following normal form:

$F^{Root} = \{f_4, f_5, f_6\}$

$F_1^1 = \{f_1, f_2, f_3, f_4, f_6, f_7\}, F_1^2 = \{f_1, f_2, f_5, f_7\}, F_1^3 = \{f_1, f_2, f_5, f_7\}$

$F_2^1 = \{f_5\}, F_1^2 = \{f_5\}$

$F_3^1 = \{f_3, f_5\}$

The normal form expresses the constraints. According to Figure 1, these transformed constraints are consulted by GP to restrict the search space. According to Figure 4, initialization, mutation, and crossover consult the constraints in *CGP lil-gp*. An obvious question remains: how those operators can use the information. We propose to express the normal form differently – in mutation sets – to facilitate efficient consultations.

4.2.2 Mutation sets

lil-gp allows parameters determining how deep to grow a subtree while in mutation. That is, *lil-gp* allows differentiation between functions of type I and terminal nodes (labeled with type II or III). We need to provide for the same capabilities.

Definition 5 *Define F_N to be the set of functions of type I that can label (thus, excluding useless functions) node N without invalidating an otherwise valid program tree containing the node. Define T_N to be the set of terminal functions that can label node N (making it a leaf) the same way.*

Definition 6 *Let us denote T_{Root} and F_{Root} the pair of mutation sets associated with the Root. Let us denote T_i^j and F_i^j the pair of mutation sets for the jth child of a node labeled with f_i .*

The information expressed in the normal form can be expressed with $2 \cdot (1 + \sum_{i=1}^{|F_I|} a_i)$ different function sets [5], while only two sets (one pair, expressed in `fset`) are needed in *lil-gp* itself. These sets alone are sufficient to initialize *CGP lil-gp* programs with only valid trees, to mutate valid trees into valid trees, and to crossover valid trees into valid trees [5].

4.2.3 Constraint feasibility

Unfortunately constraints may be so severe that only empty or only infinite trees are valid. This would be detected in the initialization when no trees could be generated. However, this could be detected earlier, and the troublesome functions can be identified and possibly removed from the function set.

This feature is not fully implemented as of now – the only check is to ensure that no function has both mutation

sets empty, which in fact detects *useless* functions [5].

4.2.4 *CGP lil-gp* mutation

lil-gp mutates a tree by selecting a random node (different probabilities for internal and external nodes). The mutated node becomes the root of a subtree, which is grown as determined by some parameters. To stop growing, a terminal function is used as the label. To force growing, a type I function is used as the label. Because we separate the same type I and terminal functions (type II and III), we can use exactly the same mechanisms. The only difference is that to label a node a subset of the original functions are used.

Operator *'mutation'. To mutate a node N, first determine the kind of the node (either the Root, or otherwise what is the label of the parent and which child of that parent N is). If the growth is to continue, label the node with a random element of* F_N *and continue growing the proper number of subtrees, each grown recursively with the same mutation operator. Otherwise, select a random element of* T_N *, instantiate it if from* F_{III} *, and stop expanding N.*
If growing a tree and $F_N = \emptyset$ *, then select a member of* T_N *(guaranteed not to be empty). If stopping the growth and* $T_N = \emptyset$ *, then select a member of* F_N *(this will unfortunately extend the tree, but it is guaranteed to stop if constraints feasiblity are fully checked).*

Proposition 3 *If a valid tree is selected for mutation, mutation will always produce a valid tree. Moreover, this is done with only constant overhead.*

:: [5].

Based on the problem-specific constraints, it may happen that a function will be only allowed to call other type I functions. In the tree, it means that a node being the corresponding child of another node labeled with such a function cannot mutate to become a leaf.

Example 5 Assume the normal form of Example 4. Assume mutating *parent1* as in Figure 5. Assume the node *N* is selected for mutation. It is the 1st child of a node labeled with f_3 . $T_3^1 = \{f_4, f_6, f_7\}$ and $F_3^1 = \{f_1, f_2\}$. If the current mode is to grow the tree, then the mutated node will be randomly labeled with either f_1 or f_2 . If the current node is to generate a leaf, then label *N* with either f_4 , f_6 , or f_7 .

Example 6 Suppose the constraints are so severe that $F_{Root}^t = \emptyset$. In this case, the only possible programs are made of single nodes labeled with functions of type II and type III. While this cannot make a good program, it is our assumption that the user specifies the 'right' constraints for the problem. In some cases it may happen that both $F^{Root} = \emptyset$ and $T^{Root} = \emptyset$. In this case no programs can be constructed at all – this means that there are no valid solutions made of the available functions.

4.2.5 *CGP lil-gp* initialization

Operator *'create a valid tree'. Assume that* $T_{Root} \neq \emptyset \vee F_{Root} \neq \emptyset$ *, and that functions which can only label trees which cannot be instantiated with finite valid trees are removed from the mutation sets. To generate a valid random tree, create the Root node, and mutate it using the mutation operator.*

Proposition 4 *Operator 'create a valid tree' will create a tree with at least one node, the tree will be finite and valid with respect to CSL constraints.*

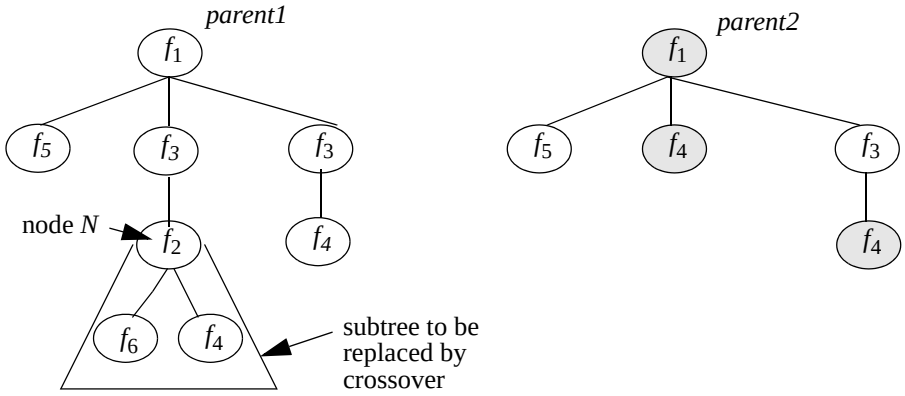
$\therefore$ [5].

4.2.6 *CGP lil-gp* crossover

The idea to be followed is to generate one offspring by replacing a selected subtree from parent 1 with a subtree selected from *parent2*. To generate two offspring, the same is repeated after swapping the parents.

Operator *'crossover'. Suppose that node N from parent1 is selected to receive the material from parent 2. First determine* F_N *and* T_N *. Assume that* F_2 *is the set of labels appearing in parent2. Then,* $(F_N \cup T_N) \cap F_2$ *is the set of labels determining which subtrees from parent2 can replace the subtree of parent1 starting with N. In other words, any subtree of parent 2 whose root is labeled with one of* $(F_N \cup T_N) \cap F_2$ *is a valid candidate for this operator.*

Figure 5 Illustration of mutation and crossover.



Example 7 Assume the normal form of Example 4. Assume *parent1* and *parent2* as in Figure 5. Assume the node *N* is selected for replacing with a subtree of *parent2*. It is the 1st child of a node labeled with f_3. Then, $T_3^1 = \{f_4, f_6, f_7\}$ and $F_3^1 = \{f_1, f_2\}$, and only the subtrees with the shaded roots can be used to replace *N*. Crossover would select a random subtree from so selected set. Note that preferences on replacing with leaves vs. internal nodes can be imposed on the process.

Proposition 5 *If two valid parents are selected for crossover, up to two valid offspring will be generated and no invalid offspring will be generated. Moreover, this is done with the same complexity O(n) as lil-gp's crossover (n is thee size).*

::[5].

4.3 Overview of Modifications

All new functions have been implemented in `cgp_czj.c`. To avoid modifications of existing prototypes, all new data is global. All new prototypes and declarations are provided in `cgp_czj.h`.

Except for `operator_crossover()`, in `crossovr.c`, and `generate_random_full_tree()`/ `generate_random_grow_tree()`, in `tree.c`, which has been rewritten, very minor changes have been introduced to existing code. All other changes can be accessed by searching `czj`. All affected files have `czj` comments in the header.

`cgp2_czj.c/h` provide utilities for handling the constraint interface file, with their own functions and global variables. Reading constraints/heuristics info out of the file is designed as a preprocessor to *cgp2.1*, generating the actual inpout file expected (temporary or retained depending on the `cgp_input` parameter). Details are not described in this manual.

4.4 Detailed Changes for Constraints

`4.4.1 lnode`

`lnode` has been changed to `struct` from `union` to accommodate two new members members: `wheelExt_czj` and `wheelInt_czj`. These accummulate weights for feasible sources for crossover. In other words, these two members create roulette wheel for crossover, distributed over all nodes plausinble for a given crossover, separately for external and internal node crossover.

In addition, there is the `typeVec_czj` member, which is the index of the type vector in `TP_czj`, which itself specifies which overloaded instance of the function labeling the node was in use. For example, if the node is for function `k` (using the fset ordering aftrer sorting), then `TP_czj[k].f.typeVecs[typeVec_czj]` is the array indicating the types of arguments and the returning type as currently used by the function `k`.

4.4.2 Global variables and functions (those accessable outside `cgp_czj.c`)

Variables:

1. `MST_czj`
   Stores mutation sets, along with heuristic weights and roulette wheels constructed for these. The data is represented separately for each function (and Root), each argument of a function, and each type to be produced by

an argument of a function.

2. TP_czj
   For terminals/Root, the returning type.
   For functions, this data structure provides the polymorphic information: types that can be generated and argument types required.
3. Function_czj
   The node to be modified has this parent. If the node to be modified is the Root then this is the number of type I functions (pointing to the index for the Root in the MST_czj array).
4. Argument_czj
   The node to be modified is this argument of its parent, 0 if the node is the Root (this indicates the index in the MST_czj[Function_czj] array.
5. Type_czj
   The return type expected by Function_czj on its Argument_czj. For the Root, it is TP_czj[NumFT].retType. This indicates the index in the array MST_czj[Function_czj][Arg_czj]. (NumF and NumT are the number of functions Type I and the number of terminals (type II and III).
6. const int RepeatsSrc_czj
   Used in crossover to specify how many new destinations can be tried if no feasible sources are found.
7. const int RepeatsBad_czj
   Used in crossover to specify how many new sources can be tried if the resulting tree is too big.

Functions:

1. void create_czj(void)
   Called to read all info and create the global vars.
2. int random_F_czj(void)/random_T_czj(void)/random_FT_czj(void)
   Replace calls to random_int() in generate() functions so that only indexes which label feasible nodes (and proportional to weights) are returned.
3. int verify_tree_czj (lnode *tree)
   Debugging functions to verify that offspring are indeed constraint-valid.
4. int markXNodes_czj(lnode *data)
   Called from crossover to mark feasible nodes (for Function_czj and Argument_czj parent), while creating the crossover wheel as well. Returns the number of marked nodes.
5. lnode *getSubtreeMarked_czj(lnode *data, int intExt)
   Called after markXNodes_czj() to select one of the marked nodes (by spinning the crossover wheel). intExt specifies whether any (0), internal (1), or external (2) node is sought. Will switch to any (0) if no desired nodes were marked.

4.4.3 Main loop

create_czj() is called after fset is sorted.

4.4.4 Initialization

Function_czj=NumF, Argument_czj=0, Type_czj=TP_czj[NumF+NumT].retType are set at start to indicate that a tree is generated from the *Root*. All other changes are in generate functions.

4.4.5 Mutation

No explicit changes. All changes in generate() and get_subtree() functions.

4.4.6 Crossover

Crossover has been rewritten. It consists of two crossovers, where the two parents are exchanged between the two operations. This is not essential, but it increases chances of generating feasible offspring for heavily constrained problems.

4.4.7 generate() functions

generate() functions are used grow subtrees. They are used in initialization and mutation. They are rewritten so that in labeling nodes they consult MST_czj in order to create feasible labeling. Also, due to constraints, it is possible that

the function or terminal, even if needed, cannot be generated. Label is selected with a probability proportional to its weight in the appriate mutation set.

4.4.8 get_subtree()

get_subtree() is *lil-gp*'s function to select the subtree identified by an argument. It is modified so that it updates Function_czj, Argument_czj, Type_czj variables to indicate who is the parent of that subtree and what type is needed.

4.4.9 Data structure for MST_czj

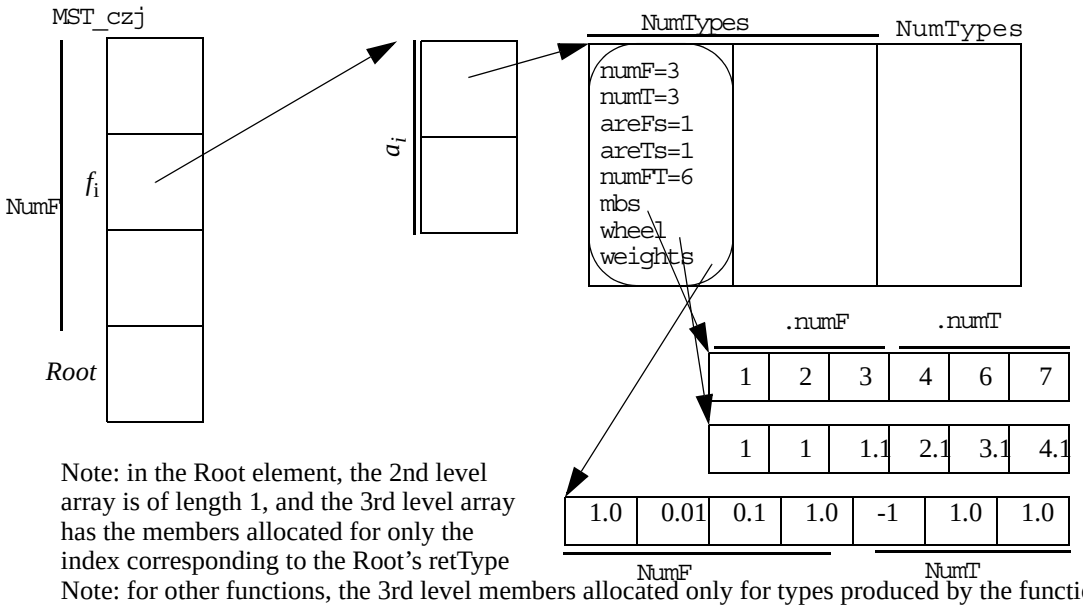
MST_czj stores the mutation sets, that is for a given function/Root (node) it gives information about what functions/ terminals can label an argument (child) if a given type is required from the child. It does not deal with the polymorphic instances (TP_czj does). . The number of elements per set may vary between zero and $|F_I|$ for F, and zero to $|F_{II} + F_{III}|$ for T. The sets are stored separately for each function (followed by the Root), each argument of the function, and each type that can be returned on this argument.

We use exactly the same ordering as that of functions of type I and terminals (type II and III) in the functions table of *lil-gp* (after sorting, which sorts by arity decreasing, then lexicographically increasing). The first is an array of $1 + |F_I|$ of pointers to other arrays dealing with individual ordinary functions of type I (plus the last entry for the *Root*). These pointers will point to other arrays of structures.

MST_czj[i] will refer to type I function at index *i* in fset. The last entry (index [$|F_I|$]) will refer to *Root*. The array for function f_i will have exactly a_i elements. which point to an array of structures, one structure per each type in the application. Each structure deals with one argument-type of the function, and represents the mutation set for this function, argument, type, along with the roulette wheel used by mutation and the weights used to construct the on-the-fly distributed roulette wheel for crossover. See Figure 6 for an example.

Weights are only read for the members, others weights are set to -1. Any input weight <MINWGHT is changed to MINWGHT. Any weight equal MINWGHT (double equality uses epsilon SMALL) does not get any area allocated on the wheel (mutation wheel is illustrated in Figure 6, crossover wheels are implicit on the trees).

Figure 6 Partially allocated and instantiated MST_czj structure for the mutation sets of Example 4. .



Note: in the Root element, the 2nd level array is of length 1, and the 3rd level array has the members allocated for only the index corresponding to the Root's retType
Note: for other functions, the 3rd level members allocated only for types produced by the function

Example 8 Figure 6 illustrates MST_czj for mutation sets pair F_1^1 and T_1^1. Assume the weights are 1.0, 0.01, 0.1, 1.0, 1.0, and 1.0 for the members (assume 0.01 is MINWGHT and thus does not get anything allocated on the wheel (and is not considered for areFs). The last element in the 3rd level array, at index NumTypes, is for untyped info, which eventually is used to initialize the remaining typed information.

For each argument-type, we have the numnber of functions (I) and terminals (II+III) in the mutation set (numF and

numT). mbs array is allocated for the total of these, and so is the wheel, while the weights array is allocated for all functions and terminals. Weight -1 indicates absence of the element in this mutation set (and absence in mbs and in wheel). Presence but with 0 weight is indicated by MINWGHT value and thus non-zero (again, these are non-zero for future processing but currently not counted in numF/T and areFs/Ts). Note that areFs != !!numFs (same for Ts). numF counts the number of functions in the mutation set. However, the functions can be set or not to MINWGHT. This is what areFs indicates.

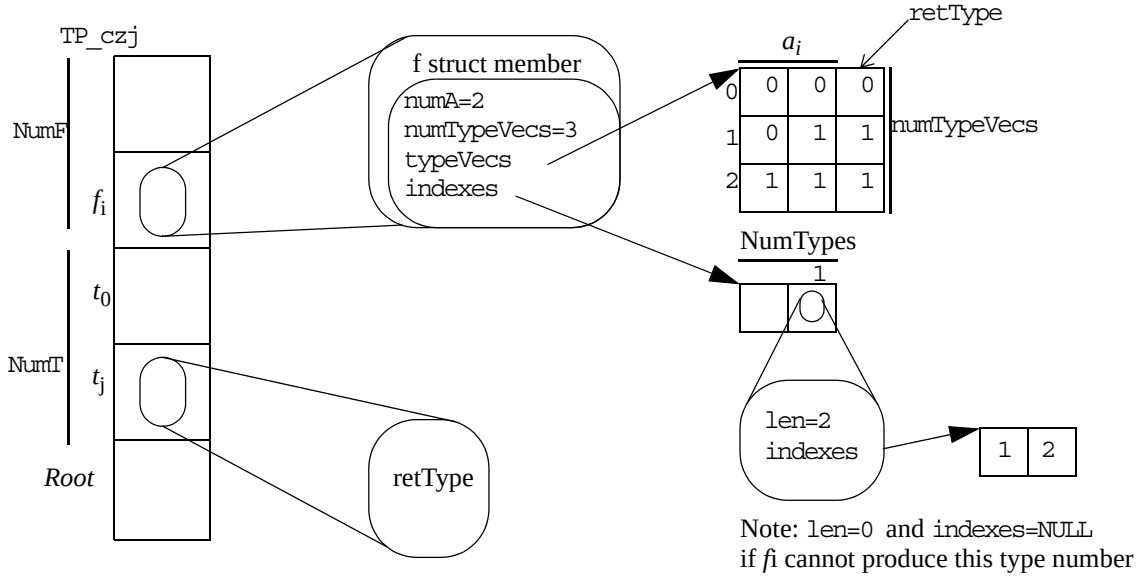
This structure will be allocated and initialized by void create_czj(void).

To redirect consulting from fset to MST_czj, we will replace the existing calls to int random_int(int) with calls to int random_F_czj(void), int random_T_czj(void), int random_F_czj(void), appropriately when a function to grow is needed, a function to terminate growth is needed, or a random function is needed. These random functions will consult MST_czj. When a particular function is not available (i.e., when the mutation set is empty, the other (T and F) set will be used (both empty is an error).

4.4.10 Data Structures for TP_czj

The TP_czj data structure is illustrated in.Figure 7. It stores information about polymorphic instances of overloaded functions, and about data types returned by terminals and tthe Root. In the part dealing with the functions, all overloaded instances are recorded. In the illustration,there are two types, and the function f_0 has two arguments. It has three polymorphic instances, represented in the 2-d array typeVecs. The first instance (.f.typeVecs[0]) produces type 0, while the two other instances (.f.typeVecs[1] and .f.typeVecs[2]) produce type 1. The number and location of the actual instances are recorded in indexes[1] (for type 1). len=2 indicates two instances, and indexes[1].indexes is the array listing the row numbers in the 2-d array typeVecs where the actual instances are represented. In the example, the first instance (.f.typeVecs[1]) requires type 0 and 1 on the arguments, respectively, while the second instance requires type 1 and 1.

Figure 7 TP_czj data structure. The first array is of a union type over functions and terminals/Root.



4.4.11 How the MST_czj and TP_czj Data Structures are used

4.4.11.1 Initialization

1. Function_czj=NumF pointing to MST[NumF] containing the Root info
2. Argument_czj=0 since Root has only one info
3. Type_czj=TP_czj[NumF+NUmT].retType fetch the Root's returning type
4. Proceed with growing a tree as in mutation, except for different parameters controlling the growth.

4.4.11.2 Mutations

1. Prepare

- `Function_czj` must be set to indicate the parent function index (`NumF` if mutating the Root).
- `Argument_czj` must be set to indicate which argument of the parent we are mutating (0 if the Root).
- Type_czj must be set to the desired type to be generate. If this is the Root node, it is `TP[NumFT].retType`. Otherwise, it we take the .typeVec member from the parent node. We then fetch `Type_czj=TP[Function_czj].f.typeVecs[parentNode.typeVec][Argument_czj]`.

2. Spin the wheel for the proper mutation set: `MST_czj[Function_czj]{Argument_czj][Type_czj].wheel` selecting say index *k*. Note that the wheel can be span seperately for selecting functions or terminals or both depending on context and parameters).
   This is done through calls to `random_F_czj()`/`random_T_czj()` and `random_FT_czj()` respectively.
3. After the index *k* to label the mutated node is selected
   - If *k* refers to a terminal, just label the node
   - If *k* refers to a function, then must select the overloaded instance of the function and contiunue recursively for all children according to the `typeVecs` information for the instance.
     - select the instance as a random element's value from the array
       n=`TP_czj[k].f.indexes[Type_czj].indexes`
       (array of length `TP_czj[k].f.indexes[Type_czj].len`)
     - select the instance represented by `TP_czj[k].f.typeVecs[n]` (and store in *node*.`typeVec`).
     - continue for all children

4.4.11.3 Crossovers

1. Crossover is prepared as in mutation, according to the parent of the destination subtree.
2. After the destination is selected, the source tree is traversed, and the wheel is constructed, using `markXNodes_czj()`. The nodes that are labeled with elements of the mutation set for the destination element (based on its parent) are used to constructed the distributed roulette wheel, according to the heuristic weights from the mutation set, and using the two extra members of the `lnode`.
3. The totals of the wheel (external or external as needed) is used to spin the roulette, and the position of the node is determined. Then a second traversal selects the node at this position from among those marked, using `getSubtreeMarked_czj()`.

4.4.12 Function/Type Information in the Tree Structure

Figure 8 The information in the tree structure in CGP1.1 (left) andA CGP2.1 (right). From among the old `lnode` members only `f` is shown. Also not shown are `wheelExt_czj`/`wheelInt_czj` members which are used for the distributed roulette. The `typeVec_czj` member is actually an index to TP_czj.

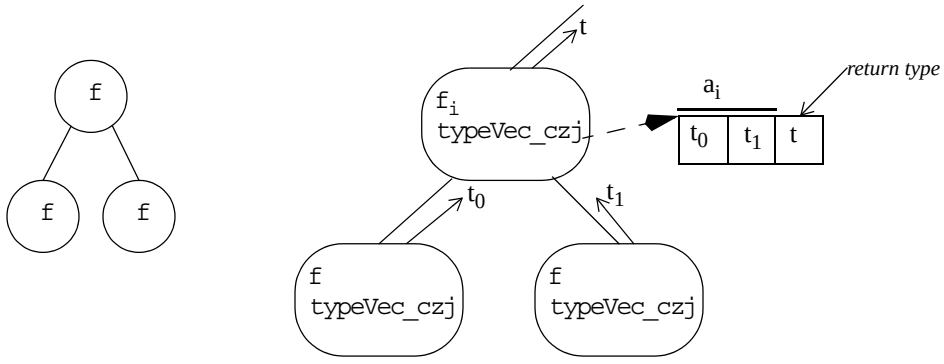


Figure 8 illustrates the function/type information maintained in the tree (in the `lnodes`). The `wheelExt_czj`/`wheelInt_czj` are the members used to construct the distributed crossover roulette wheel and not used. As seen, each node remembers not only the function (terminal) being the label, but also the actual overloaded instance (`typeVec_czj`). The instance says what are types expected of the children and then what is the type produced by the node.

5 User Manual for CGP2.1

5.1 Strong Constraints: Tspec / Fspec Constraints

Tspecs and Fspecs are sets of functions and terminals with some common characteristic.

- Tspecs are those functions/terminals which are compatible with a specific argument of a function (`T_func_`*n*), and those functions/terminals allowed to be the Root (`T_Root`). (syntactic constraint)
- Fspecs are those functions/terminals which are not compatible as a specific argument of a function (`F_func_`*n*), those functions which are not allowed to directly call a given function (`F_func`), and also those functions/terminals not allowed to be the Root (`F_Root`). (semantic constraint)

When specifying these constraints you must specify Tspecs for everything you wish to allow. The Tspecs and Fspecs are converted to their Normal form, consisting entirely of Fspecs. These Normal Form Fspecs are what are used to construct the Untyped Mutation Sets (see Section 5.4).

5.1.1 Tspec / Fspec Example

Listed here, in the order that they are requested in the interactive user interface are some example Fspecs and Tspecs.

Example 9 `F_add = sin cos`
This prevents sin() and cos() from calling add (e.g. sin(x+y) is not allowed).

Example 10 `F_add_0 = 0 PI log`
This prevents 0, PI, log() from being argument 0 of add (e.g. (PI + x) is not allowed, but (x + PI) may be).

Example 11 `T_add_0 = sin cos 0 PI`
This allows sin(), cos(), 0, and PI to be argument 0 of add. However, the Fspec, F_add_0, overrides this Tspec, and so only sin() and cos() are actually allowed.

Example 12 `F_add_1 = 0 PI`
This prevents 0, PI from being argument 1of add (e.g. (x + PI) is not allowed).

Example 13 `T_add_1 = sin cos 0 PI`
This allows sin(), cos(), 0, and PI to be argument 1of add. However, the Fspec, F_add_1, overrides this Tspec, and so only sin() and cos() are actually allowed.

Example 14 `F_ROOT = log 0 PI`
This prevents the function `log`, and the terminals `0` and `PI` from being the Root.

Example 15 `T_ROOT = add sin cos`
This allows the functions `add`, `sin`, and `cos` to be the Root.

5.2 Weak Constraints (Heuristics): Weight Constraints

Normally all functions/terminals have the same probability of being selected to be used as an argument to a function. The Weights section of the interface allows you to change this default behavior. The Weight constraints do not affect nor interact with Type constraints.

The weight of any function/terminal is in the range (0,1]. Using only the Weight constraints, you cannot absolutely prevent a function/terminal from being used. This is because any weight specified as 0 (Zero) is converted to a defined constant $MINWGHT \approx 0.00001$, defined in `kernel*/cgp_czj.c`. If you desire to absolutely prevent a function/terminal from being used, you need to specify the appropriate Fspec.

When you are given the chance to enter the weights, only those function/terminals which appear in the Untyped Mutations Set are used (see Section 5.4). So, if a function is disallowed through Fspecs, you are not asked for the weight for it.

Currently, the weights remain constant throughout the execution of the program. Work is currently underway to implement mechanisms for the automatic adjusting of the weights during program execution.

5.2.1 Weight Constraint Example

Example 16 Assume we have 4 functions(f1, f2, f3, f4) and 4 terminals(t1, t2, t3, t4). Through Fspecs, t2 was

excluded from being an argument to function f*n*. Then the weights are given as f1=1.0, f2=0.0, f3=0.1, t1=1.0, t3=1.0, t4=1.0. The total weight of all function/terminals for this argument = f1 + f2 + f3 + t1 + t3 + t4 = 4.1. The probability p(), of any function/terminal being selected as this particular argument is p((f/t)*n*) = ((f/t)*n)* / (total weight). So p(f1,t1,t3,t4) = 1.0/4.1 = 0.244, p(f2) = MINWGHT/4.1 = 2.439×10^{-6} , p(f3) = 0.1/4.1 = 0.0244, p(t2) = 0.0.

5.3 Type Constraints

Type constraints provide additional method of strongly constraining the mutation sets. Without type constraints it would be possible, with normal arithmetical and trigonometric functions, to generate a subtree which would evaluate to: `asin(x) + sin(x)`, which under normal circumstances doesn't make much sense. With the type constraints, `add()` can be instructed not to add a non-angle with an angle, etc. **(Note: The type constraints do not in any way perform type casting of functions, terminals, or arguments. If a function is to be able to accept multiple data types, then the function must be written in such a way as to make that possible.)**

It can prevent occurrences of subtrees like that shown in Example 17, or Example 18, if that kind of restriction is what you would desire.

Example 17 `sin(mult(PI,sin(sqrt(PI))))`

Example 18 If there are variables of various units, such as S(seconds), M(mass), D(distance), T(temperature), then subtrees like this could be avoided: `add(add(S,M),add(D,T))`.

5.3.1 Type Constraint Example

To begin with, you specify every data type that your problem uses. **(Note: The data types specified here do not necessarily have anything to do with the actual data types of your functions/terminals/arguments)** Then, starting with the highest arity (argument count) functions working down to the terminals, and sorted alphabetically, every instance of a function is listed. with its arguments and return type (see Example 19). After listing every instance of every function, the return types of each of the terminals and of the root of the problem are listed.

Example 19 There may be a problem needing data types: *angle*, *length*, *force*, *force-length*, and *number*. And, the `multiply()` function which takes 2 arguments could have the following instances (vectors).

<arg1>	**<arg2>**	**<return>**
number	length	length
length	number	length
number	angle	angle
angle	number	angle
number	number	number
length	force	force-length
force	length	force-length

All of this means that if `multiply()` is called with a number and a length, it will return a length. If it is called with a number and an angle, it will return an angle. If it is called with two numbers it just returns a number. And, if it is called with a length and a force, it returns a force-length (e.g. 3 lbs. * 5 ft. = 15 ft-lbs.).

The type constraints also allow you to reuse a single function is several ways. This can allow you to easily create a specific "structure" for your trees as is shown in Example 20.

Example 20 You have a defined structure (full & complete binary tree of height 4, with an `and/or` for each internal node) you're trying to create from a limited set of functions. Without type constraints you would have to create different `and` & `or` functions for each level. With type constraints, however, you can specify different types (e.g. L0, L1, L2, L3, L4). So that, all terminals return type L4, and the Root takes type L0. The `and` & `or` functions then have instances such as:

Arg1	Arg2	Return
L1	L1	L0
L2	L2	L1
L3	L3	L2
L4	L4	L3

This will cause the generation of only those trees which match the above specified structure.

5.4 Mutation Sets

Mutation Sets are the constructs that CGP uses to keep track of the structure of valid trees. A mutations set consists of the functions and terminals that are allowed to represent an argument of a function.

The initial mutation set, called the Untyped Mutation Set, is constructed from the Normal Form Fspecs. Any function/terminal listed in an Fspec is removed from the corresponding portion of the Untyped Mutation Set.

The Untyped Mutation Set is combined with the Type Constraint (see Section 5.3) information to yield the final Typed Mutation Sets, which are then used for the construction of the trees. Any function/terminals which has a type that is not compatible with the argument in which it is referenced, is removed from the Mutation Set for that argument.

Upon examining the Typed Mutation Set, the true power of it may not be apparent. In Section 5.7.1 towards the end of the listing, the example shows the Typed Mutation Set. Even though the function `add` appears in every set for every argument of every function, it is not the same instance of the `add` function. The particular instance of the function is partially determined by which Type it is under. That is to say the `add` function under a Type 'angle' section is strictly the `add` function which takes two angles as arguments and returns an angle.

5.5 Initialization and Operator Modifications

Modifications to *lil-gp* have been kept to a minimum. However some changes seemed prudent to help enable it to respond more closely to the users intent.

5.5.1 Initialization Parameters

Please refer to the *"lil-gp 1.0 User's Manual"* [11] for a complete description of the standard initialization parameters.

A couple of additional Initialization Parameter have been added. One to help control tree generation and two to provide information regarding the input files. And, like all parameters, they may be specified in a parameter file or on the command line. Please see the lil-gp Users Manual [11] for further information on using parameters and parameter files.

5.5.1.1 Tree Creation Parameter

- *`init.depth_abs`* = {true,false}, default = false.

This parameter is used to prevent the generation of initial trees shallower than that specified by the depth ramp. In the default mode, this new parameter has no effect over *lil-gp*. When *`init.depth_abs`* is set to *`true`*, *`init.depth=m-n`* will cause rejection of initial trees shallower than *`m`*. *lil-gp's* default behavior allows trees to be shallower than m. **(Technical Note: this can occur during a call to `generate_random_grow_tree()`).**

5.5.1.2 Interface Parameters

- `cgp_interface` - The file containing the constraint creation instructions.
- `cgp_input` - The file which is to be created from the constraint instructions, if the parameter `cgp_interface` is specified. This file will then act as the input to *CGP lil-gp*.

Detailed information about these parameters can be found in Section 5.8.2.

5.5.2 Mutation Operator

An additional Mutation Parameter has been added to prevent the Mutation Operator from allowing the mutated subtree from being shallower than that specified by the depth parameter.

5.5.2.1 Mutation Parameters

- `depth_abs` ={true,false}, default=false.

In the default mode, this new parameter has no effect over *lil-gp*. When `depth_abs` is set to *`true`*, `depth=m-n` will cause rejection of mutated subtrees shallower than `m` (it is a good idea to have `keep_trying` set to `true` as well). *lil-gp's* default behavior allows the Mutation operator to mutate a subtree shallower than m. **(**Technical Note: this can occur during a call to `generate_random_grow_tree()`**).**

5.5.3 Crossover Operator

Additional parameters have been added to the Crossover parameters.
The standard `internal` and `external` parameters are replaced with 2 pair of parameters which separate the functionality between the source and destination parents.

5.5.3.1 Crossover Parameters

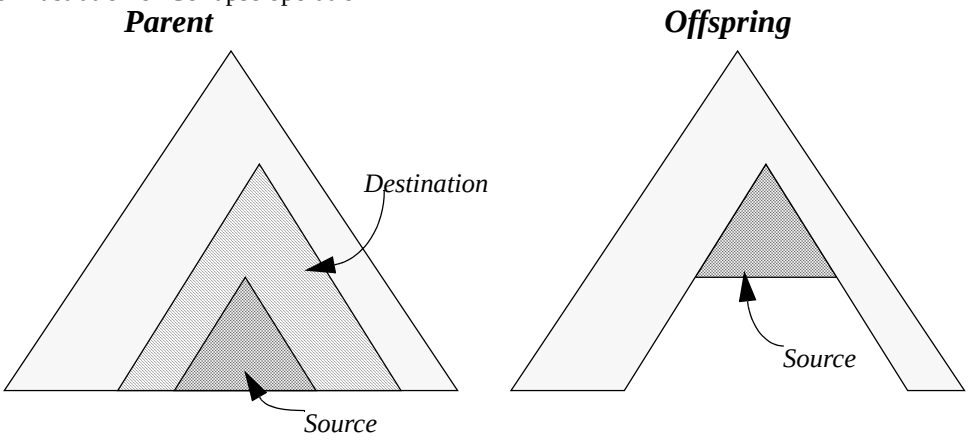
- `internal` (internal nodes in source parent), default = 0.9
- `internal_dst` (internal nodes in destination parent), default = `internal`
- `external` (external nodes in source parent), default = 0.1
- `external_dst` (external nodes in destination parent), default = `external`

These act the same way as in *lil-gp* except that there can be different frequencies for internal/external node selection in the source and destination parents.

5.5.4 Collapse Operator (New)

The Collapse operator is similar to the Mutation operator in that it causes a single parent to mutate into a single offspring. Collapse chooses a random subtree, from a given parent tree, to be the destination. Another subtree is chosen, from within the destination subtree, as the source (see Figure 9). All of the nodes in the destination subtree that are not also in the source subtree are removed. This results in the destination subtree being removed, and the source subtree moved into its place. See also Example 21.

Figure 9 Illustration of Collapse operation



Example 21 Here is a sample individual (tree) from an actual experiment. This can be read as LISP function calls, where `(atan2 y x)` corresponds to `atan2(y,x)` in the C language.

```
Parent:     (fkin (sub  (atan2 y x)
                        (acos (div (div x 2) x)))
                  (acos (add 2
                        (sub 1
                              (div (hypot x y) L1)))))
```

(The function `sub` performs subtraction, and the function `div` performs division. The function `fkin` simply causes its 2 subtrees to execute in order.) So, if *`sub`* on the first line is chosen as the beginning of the Destination subtree of collapse, and if `(div x 2)` is chosen as the Source subtree, then the resulting offspring would be:

```
Offspring:  (fkin (div x 2)
                  (acos (add 2
                        (sub 1
                              (div (hypot x y) L1)))))
```

5.5.4.1 Collapse Parameters

- `select` - same as for Mutate
- `keep_trying` - same as for Mutate

- `internal` - same as for Mutate
- `external` - same as for Mutate
- `tree` -same as for Mutate
- `tree`*`n`* - same as for Mutate

5.6 *CGP lil-gp* Interactive Interface

The interface to CGP *lil-gp* is an interactive system. Constraints are entered using function names. Listing order is irrelevant. Repetitions are disregarded. Functions are sorted by arty and then alphabetically by name.

5.7 Interactive Interface Description

An interactive interface is part of the modifications which CGP *lil-gp 2.1;1.02* has added to *lil-gp*. The interface can be broken up into the following sections:

1. Type Information (*optional*)
   a. User defined data entry section
   b. Display of Type Vectors
2. Tspecs & Fspecs (*optional*)
   a. User defined data entry section
   b. Display of Tspec & Fspec Constraints
   c. Display of Normal Constraints (Tspec's & Fspec's converted to Fspec-only)
   d. Display of Untyped Mutation Sets
3. Weights (*optional*)
   a. User defined data entry section
4. Display of Typed Mutation Sets

5.7.1 Interactive Interface Example

Note: This example is of a problem with the functions (add, asin, sin), and terminals(1, PI, x, y). All user's responses are in *italics*.

Note: Type Information Section (see Section 5.3 for more information).

```
<...>
Reading Type information...
```

Note: You are asked for the Type constraints (see Section 5.3) of the functions and their arguments.

```
Default is a single 'generic' type. Accept? [0 to change]:
   0<ENTER>
List type names: float integer angle
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :float float float<ENTER>
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :integer float float<ENTER>
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :float integer float<ENTER>
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :integer integer integer<ENTER>
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :angle angle angle<ENTER>
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :<ENTER>

Specs for 'asin' [1 arg and one ret types /<ENTER> for no more]
   :float angle<ENTER>
```

```
Specs for 'asin' [1 arg and one ret types /<ENTER> for no more]
   :integer angle<ENTER>
Specs for 'asin' [1 arg and one ret types /<ENTER> for no more]
   :<ENTER>

Specs for 'sin' [1 arg and one ret types /<ENTER> for no more]
   :angle float<ENTER>
Specs for 'sin' [1 arg and one ret types /<ENTER> for no more]
   :<ENTER>

Give ret type for terminal '1': integer<ENTER>
Give ret type for terminal 'PI': angle<ENTER>
Give ret type for terminal 'x': float<ENTER>
Give ret type for terminal 'y':float<ENTER>
Give return type for the problem: angle<ENTER>
```

Note: All valid type vectors are now displayed

```
The following types are used...
Function 'add': numArg=2, numTypeVecs=5
   vec0: 0:'float' 1:'float' -> 'float'
   vec1: 0:'integer' 1:'float' -> 'float'
   vec2: 0:'float' 1:'integer' -> 'float'
   vec3: 0:'integer' 1:'integer' -> 'integer'
   vec4: 0:'angle' 1:'angle' -> 'angle'
   Type 'float' returned from vectors: 0 1 2
   Type 'integer' returned from vectors: 3
   Type 'angle' returned from vectors: 4
Function 'asin': numArg=1, numTypeVecs=2
   vec0: 0:'float' -> 'angle'
   vec1: 0:'integer' -> 'angle'
   Type 'angle' returned from vectors: 0 1
Function 'sin': numArg=1, numTypeVecs=1
   vec0: 0:'angle' -> 'float'
   Type 'float' returned from vectors: 0
Terminal '1': -> 'integer'
Terminal 'PI': -> 'angle'
Terminal 'x': -> 'float'
Terminal 'y': -> 'float'
Root: -> 'angle'
```

Note: Tspec & Fspec section for more information).

```
Reading F/Tspec information...


Default is empty Fspecs, full Tspecs. Accept? [0 to change]:
   0<ENTER>

3 ordinary function      (Note: this is displayed for each function)
   add asin sin
4 terminals (ordinary and ephemeral):
   1 PI x y
Separate entries by [ ,;]
Hit <ENTER> for empty set
Use function names in any order

Function 'add':
   F_add (exclusions) [up to 3 names] = <ENTER>
   F_add_0 (exclusions) [up to 7 names] = <ENTER>
```

```
   T_add_0 (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>
   F_add_1 (exclusions) [up to 7 names] = <ENTER>
   T_add_1 (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>
```

Note: this is asking you for the Tspecs and Fspecs of the functions and their arguments.

- **F_add = Fspec for the add function (List all functions which cannot directly call add)**
- **F_add_0 = Fspec for Argument 0 of the add function (List all functions which cannot be argument 0 of the add function)**
- **T_add_0 = Tspec for Argument 0 of the add function (List all functions which can be argument 0 of the add function)**
- **F_add_1 = Fspec for Argument 1of the add function (List all functions which cannot be argument 1of the add function)**
- **T_add_1 = Tspec for Argument 1of the add function (List all functions which can be argument 1of the add function)**

```
<...>
Function 'asin':
   F_asin (exclusions) [up to 3 names] = <ENTER>
   F_asin_0 (exclusions) [up to 7 names] = <ENTER>
   T_asin_0 (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>

<...>
Function 'sin':
   F_sin (exclusions) [up to 3 names] = <ENTER>
   F_sin_0 (exclusions) [up to 7 names] = add <ENTER>
   T_sin_0 (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>

<...>
Root:
   F^Root (exclusions) [up to 7 names] = asin<ENTER>
   T^Root (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>
```

Note: The Tspecs and Fspecs for the Root are slightly different.

- **F^Root = Fspec for the Root (List all functions which cannot be the root function)**
- **T^Root = Tspec for the Root (List all functions which can be the root function)**

Note: based on conditional compilation, constraints may be echoed. This section displays the constraints as you entered them, and then as converted into the Normal Constraints. The || separates functions from terminals.

```
Read the following constraints...

      CONSTRAINTS
Function 'add' [#0]:
   F_add [#Fs=0:#Ts=0] = ||
   F_add_0 [#Fs=0:#Ts=0] = ||
   F_add_1 [#Fs=0:#Ts=0] = ||
   T_add_0 [#Fs=3:#Ts=4] = 'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'
   T_add_1 [#Fs=3:#Ts=4] = 'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'
Function 'asin' [#1]:
   F_asin [#Fs=0:#Ts=0] = ||
   F_asin_0 [#Fs=0:#Ts=0] = ||
   T_asin_0 [#Fs=3:#Ts=4] = 'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'
Function 'sin' [#2]:
   F_sin [#Fs=0:#Ts=0] = ||
   F_sin_0 [#Fs=1:#Ts=0] = 'add' ||
   T_sin_0 [#Fs=3:#Ts=4] = 'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'
```

```
Root:
   F_Root [#Fs=1:#Ts=0] =  'asin' ||
   T_Root [#Fs=3:#Ts=4] =  'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'

The normal constraints are...

      CONSTRAINTS
Function 'add' [#0]:
   F_add_0 [#Fs=0:#Ts=0] = ||
   F_add_1 [#Fs=0:#Ts=0] = ||
Function 'asin' [#1]:
   F_asin_0 [#Fs=0:#Ts=0] = ||
Function 'sin' [#2]:
   F_sin_0 [#Fs=1:#Ts=0] = 'add' ||
Root:
   F_Root [#Fs=1:#Ts=0] =  'asin' ||
```

Note: This section displays the mutation sets as if the generic type were used. F[] = functions, T[] = terminals.

```
The following untyped mutation sets are used...

Function 'add': arity 2
   Argument 0
      Type unconstrained mutation set
         F [3 members] = 'add' 'asin' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
   Argument 1
      Type unconstrained mutation set
         F [3 members] = 'add' 'asin' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
Function 'asin': arity 1
   Argument 0
      Type unconstrained mutation set
         F [3 members] = 'add' 'asin' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
Function 'sin': arity 1
   Argument 0
      Type unconstrained mutation set
         F [2 members] = 'asin' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
Root:
      Type unconstrained mutation set
         F [2 members] = 'add' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
```

Note: Weights Section (see Section 5.2 for more information).

Note: Entering 0 here will prompt you to enter all weights, similar to entering constraints above.

```
Setting initial weights for mutation set members...
Initial weights are all equal. Do you accept [0 to change]:
   0<ENTER>
Function 'add': 2 mutation set pairs
Argument 0:
F [3 members] = 'add' 'asin' 'sin'
   T [4 members] = '1' 'PI' 'x' 'y'


   Reading the weights for type I functions...
```

```
   Function 'add': give weight (0,1]: 0.25<ENTER>
   Function 'asin': give weight (0,1]: 0.25<ENTER>
   Function 'sin': give weight (0,1]: 0.5<ENTER>
   Reading the weights for type II/III terminals...
   Terminal '1': give weight (0,1]: 0.2<ENTER>
   Terminal 'PI': give weight (0,1]: 0.2<ENTER>
   Terminal 'x': give weight (0,1]: 0.3<ENTER>
   Terminal 'y': give weight (0,1]: 0.4<ENTER>
Argument 1:
   F [3 members] = 'add' 'asin' 'sin'
   T [4 members] = '1' 'PI' 'x' 'y'


   Reading the weights for type I functions...
   Function 'add': give weight (0,1]: 0.25<ENTER>
   Function 'asin': give weight (0,1]: 0.25<ENTER>
   Function 'sin': give weight (0,1]: 0.5<ENTER>
   Reading the weights for type II/III terminals...
   Terminal '1': give weight (0,1]: 0.2<ENTER>
   Terminal 'PI': give weight (0,1]: 0.2<ENTER>
   Terminal 'x': give weight (0,1]: 0.3<ENTER>
   Terminal 'y': give weight (0,1]: 0.4<ENTER>


Function 'asin': 1 mutation set pairs
Argument 0:
   F [3 members] = 'add' 'asin' 'sin'
   T [4 members] = '1' 'PI' 'x' 'y'


   Reading the weights for type I functions...
   Function 'add': give weight (0,1]: 0.25<ENTER>
   Function 'asin': give weight (0,1]: 0.25<ENTER>
   Function 'sin': give weight (0,1]: 0.5<ENTER>
   Reading the weights for type II/III terminals...
   Terminal '1': give weight (0,1]: 0.2<ENTER>
   Terminal 'PI': give weight (0,1]: 0.2<ENTER>
   Terminal 'x': give weight (0,1]: 0.3<ENTER>
   Terminal 'y': give weight (0,1]: 0.4<ENTER>


Function 'sin': 1 mutation set pairs
Argument 0:
   F [2 members] = 'asin' 'sin'
   T [4 members] = '1' 'PI' 'x' 'y'


   Reading the weights for type I functions...
   Function 'asin': give weight (0,1]: 0.5<ENTER>
   Function 'sin': give weight (0,1]: 0.4<ENTER>
   Reading the weights for type II/III terminals...
   Terminal '1': give weight (0,1]: 0.6<ENTER>
   Terminal 'PI': give weight (0,1]: 0.4<ENTER>
   Terminal 'x': give weight (0,1]: 0.3<ENTER>
   Terminal 'y': give weight (0,1]: 0.1<ENTER>


Root:
      F [2 members] = 'add' 'sin'
      T [4 members] = '1' 'PI' 'x' 'y'
   Reading the weights for type I functions...|
```

```
Function 'add': give weight (0,1]: 1<ENTER>
Function 'sin': give weight (0,1]: 1<ENTER>
Reading the weights for type II/III terminals...
Terminal '1': give weight (0,1]: 1<ENTER>
Terminal 'PI': give weight (0,1]: 0.2<ENTER>
Terminal 'x': give weight (0,1]: 1<ENTER>
Terminal 'y': give weight (0,1]: 1<ENTER>
```

Note: End of Weights Section and start of the Display of Typed Mutation Sets Section

Note: This section displays the typed mutation sets. It shows the valid types of each argument, for every function, along with the functions and terminals of that type which can be used in that argument.

```
The following typed mutation sets are used...

Function 'add': arity 2
   Argument 0
      Type 'float'
         F [2 members] = 'add' 'sin'
         T [2 members] = 'x' 'y'
      Type 'integer'
         F [1 members] = 'add'
         T [1 members] = '1'
      Type 'angle'
         F [2 members] = 'add' 'asin'
         T [1 members] = 'PI'
   Argument 1
      Type 'float'
         F [2 members] = 'add' 'sin'
         T [2 members] = 'x' 'y'
      Type 'integer'
         F [1 members] = 'add'
         T [1 members] = '1'
      Type 'angle'
         F [2 members] = 'add' 'asin'
         T [1 members] = 'PI'
Function 'asin': arity 1
   Argument 0
      Type 'float'
         F [2 members] = 'add' 'sin'
         T [2 members] = 'x' 'y'
      Type 'integer'
         F [1 members] = 'add'
         T [1 members] = '1'
Function 'sin': arity 1
   Argument 0
      Type 'angle'
         F [1 members] = 'asin'
         T [1 members] = 'PI'
Root:
      Type 'angle'
         F [1 members] = 'add'
         T [1 members] = 'PI'
Send 1 to continue, anything else to quit cgp-lil-gp: 1<ENTER>
```

5.8 Interface File

The Interface File, described below, uses a simple language to describe the constraints you wish to specify. It takes this information, and converts it into an input file, which CGP then reads from instead of reading from the keyboard during

use of the Interactive Interface (Section 5.6).

5.8.1 Interface File Overview

There are 4 sections in the Interface File. The first three sections may appear in any order, and may even be repeated. No section has any effect on the other sections. This file must contain at least the End of File Marker. The sections are:

1. Fspec/Tspec Constraints *(optional)*
2. Weight Constraints *(optional)*
3. Type Constraints *(optional)*
4. End of File Marker *(required)*

5.8.2 Interface File Parameters

Two new parameters have been added to all flexibility in the use of this new interface. They are:

- `cgp_interface` - The file containing the constraint creation instructions.
- `cgp_input` - The file which is to be created from the constraint instructions, if the parameter `cgp_interface` is specified. This file will then act as the input to *CGP lil-gp*.

And, like all parameters, they may be specified in a parameter file or on the command line. Please see the lil-gp Users Manual [11] for further information on this. Depending on how you specify these parameters, several things may happen:

1. Nothing specified - Use Interactive Interface
2. Specify cgp_input only - The specified file is used for the Interactive Interface instead of the keyboard.
3. Specify cgp_interface only - A temporary file is created for the input file, and deleted when finished.
4. Specify both - The interface file is used to create the input file. The input file, may later be used as in option 2.

5.8.3 Interface File Definitions

- *`funclist`* = list of applicable functions
- *`termlist`* = list of applicable terminals
- *`functermlist`* = list of applicable functions & terminals
- *`arglist`* = list of applicable argument numbers (0...*arity*) (i.e. the arglist for sin() is 1 item long.)
- *`weightlist`* = list of applicable weights (user defined in interface file),
  $length(weightlist) \leq length(functermlist)$
- *`typelist`* = list of valid types (user defined in interface file)
- *`type`* = a single valid type from *`typelist`*
- *`argtypelist`* = list of the valid type for each argument in a particular instance (length defined by arity; i.e. the argtypelist for sin() is 1 item long.)
- *`null`* = empty list (not actually typed in, just hit the <ENTER> key)
- * = wildcard, include all elements from applicable list. Any item appearing after a wildcard is ignored.
- *#* = begin of comment. Comments continue until end-of-line. Valid characters are "*#" + space + alpha-numerics.

5.8.4 Interface File Sections

There are 3 sections corresponding to Fspecs/Tspecs, Weights, and Types. These sections may appear in any order. If a section appears, even if empty, it will override the default behavior of CGP, so care must be taken to ensure that enough information is present to allow CGP to have enough constraint information.

5.8.5 Fspec/Tspec Constraints

The Section Header and Footer must be present if this section is to be used. If this section appears, even if no specs are listed, then any items not appearing in a Tspec will be placed in the appropriate Fspec of the Normal Constraints. If an item appears in both an Fspec and the corresponding Tspec, then the Fspec will take precedents. Once a Spec is specified it cannot be overridden, with the exception of an Fspec overriding a Tspec.

5.8.5.1 Syntax

```
FTSPEC                    (Note: Section Header)
F_(funclist | *) = funclist | * | null
F_(funclist | *)[arglist | *] = functermlist | * | null
T_(funclist | *)[arglist | *] = functermlist | * | null
F_ROOT = functermlist | * | null
T_ROOT = functermlist | * | null
ENDSECTION                (Note: Section Footer)
```

5.8.5.2 Default Behavior

If this section is omitted, then all FSpecs are left empty and all TSpecs are full. If the section header is given, then all Fspecs and Tspecs default to the empty set. If no TSpecs are entered, this will result in normal constraints being generated with full Fspecs, yielding no possible tree growth.

5.8.6 Weight Constraints

The Section Header and Footer must be present if this section is to be used. If this section appears, even if no specs are listed, then any items not appearing in a Weight specification will be given a weight of 1.0. Any weight may be overridden by specifing a new weight.

Note: if there are fewer elements in *weightlist* than in *functermlist,* the last element in *weightlist* will be used for the remaining elements in *functermlist.*

5.8.6.1 Syntax

```
WEIGHT                    (Note: Section Header)
(funclist | *)[arglist | *](functermlist | *) = weightlist
ROOT(functermlist | *)= weightlist
ENDSECTION                (Note: Section Footer)
```

5.8.6.2 Default Behavior

Whether this section is used or not, the default behavior is to set all the weights to 1.0.

5.8.7 Type Constraints

The Section Header and Footer must be present if this section is to be used. If this section appears, TYPELIST must be the first item following it. Also, there must be an entry for every function, terminal, and the Root. Once an instance of a function has been specified, it cannot be overridden. However, terminals and the Root may be specified multiple times but the last entry for each will be the one used.

5.8.7.1 Syntax

```
TYPE                      (Note: Section Header)
TYPELIST = typelist       (Note: Define valid types)
(funclist)(argtypelist) = type
(termlist | *) = type
ROOT = type
```

```
ENDSECTION              (Note: Section Footer)
```

5.8.7.2 Default Behavior

If this section is omitted, then only a generic type is used. If the section header is given, then all types, functions, terminals, and the Root are undefined.

5.8.8 End of File Marker

```
ENDFILE                 (Note: Section Footer)
```

5.8.9 Interface File Example

The Interactive Interface Example, Section 5.7.1, can be duplicated in the Interface File by the following Interface File example.

```
FTSPEC
F_(*)=                  #not required since it's empty
F_(*)[*]=               #not required since it's empty
F_(sin)[0]=add          #prevent sin(_+_)
F_ROOT=asin             #prevent asin() from being the root
#must specify some TSpecs
T_(*)[*]=*              #allow all TSpecs
T_ROOT=*                #allow all functions/terminals for Root
ENDSECTION

WEIGHT
#All unspecified weights default to 1.0
#If I desire to reduce the odds of everything but that which
# I explicitly specify, I could do the following
#(*)[*](*)=0
#ROOT(*)=0
#
#Set the weights for the functions: add asin sin 1 PI x y,
#as the arguments for the add & asin functions.
(add asin)[*](*)=.25 .25 .5 .2 .2 .3 .4
#similarly for the sin function
(sin)[0](*)=.5 .4 .3 .6 .4 .3 .1
ROOT(*)=1                   #not really needed as default is 1.0
ROOT(PI)=.2
ENDSECTION

TYPE
TYPELIST = float integer angle      #list of all valid types
(add)(float float)=float            #add() instances
(add)(integer float)=float
(add)(float integer)=float
(add)(integer integer)=integer
(add)(angle angle)=angle
(asin)(float)=angle                 #asin() instances
(asin)(integer)=angle
(sin)(angle)=float                  #sin() instance
(1)=integer                         #terminal types
(PI)=angle
(x y)=float
ROOT=angle                          #Root return type
ENDSECTION

ENDFILE
```

5.9 Input File Interface Redirection

If you intend to run multiple experiments, or if the data entry section is overly long, you may wish to create an input file which contains all user inputs and which can be redirected into the program at run-time. At the present time, generation of this file is a manual process, and must match **exactly** what you intend for the input. Work is underway to provide an easier-to-use user interface.

5.9.1 Interface Redirection File Contents

The Interactive Interface shown in Section 5.7.1 could be duplicated in an input file as follows. This file can be created from the example in Section 5.8.9 (Interface File Example). The format is very important, as this file is used instead of the user having to type all of this in. Also, since this file is simply redirected into the program, no form of comments are allowed in it. (Note: The *<ENTER>* is only shown for clarity, and the **(Note: ...)** are not part of the file.

```
0<ENTER>                                (Note: start of Types)
float integer angle<ENTER>                      (Note: valid types)
float float float<ENTER>                        (Note: add()Types)
integer float float<ENTER>
float integer float<ENTER
integer integer integer<ENTER>
angle angle angle<ENTER>
<ENTER>
float angle<ENTER>                              (Note: asin() Types)
integer angle<ENTER>
<ENTER>
angle float<ENTER>                              (Note: sin() Types)
<ENTER>
integer<ENTER>                                  (Note: terminal Types)
angle<ENTER>
float<ENTER>
float<ENTER>
angle<ENTER>                                    (Note: Root return Type)
0<ENTER>                                (Note: start of F/TSpecs)
<ENTER>                                         (Note: F/T_add Constraints)
<ENTER>
add asin sin 1 PI x y<ENTER>
<ENTER>
add asin sin 1 PI x y<ENTER>
<ENTER>                                         (Note: F/T_asin Constraints)
<ENTER>
add asin sin 1 PI x y<ENTER>
<ENTER>                                         (Note: F/T_sin Constraints)
add<ENTER>
add asin sin 1 PI x y<ENTER>
asin<ENTER>                                     (Note: F/T_Root Constraints)
add asin sin 1 PI x y<ENTER>
0<ENTER>                                (Note: start of Weights)
0.25 0.25 0.5 0.2 0.2 0.3 0.4<ENTER>            (Note: add[0]() Weights)
0.25 0.25 0.5 0.2 0.2 0.3 0.4<ENTER>            (Note: add[1]() Weights)
0.25 0.25 0.5 0.2 0.2 0.3 0.4<ENTER>            (Note: asin[0]() Weights)
0.5 0.4 0.6 0.4 0.3 0.1<ENTER>                  (Note: sin[0]() Weights)
1 1 1 0.2 1 1<ENTER>                            (Note: Root Weights)
1<ENTER>          (Note: end of Weights Section. Finished, enter 1 to continue)
```

Note: anything after this point will not be read by the program

5.9.2 Interface Redirection Example

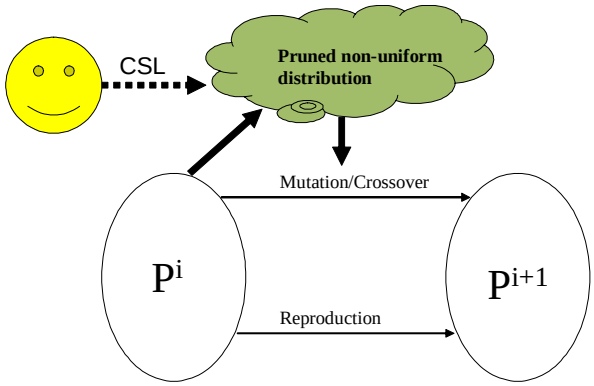
Once the input file has been created, it is a simple matter to use it. If the file were called `experiment.input` and the executable file is called `gp` then using that file would be a matter of:

```
gp parameters < experiment.input
```

This is the standard way of redirecting a file into a program in Unix and DOS, but there may be other variations on your system. Please refer to your system documentation for further information in this area.

6 ACGP Technical Elements

Figure 10 Working envorinment for ACGP.

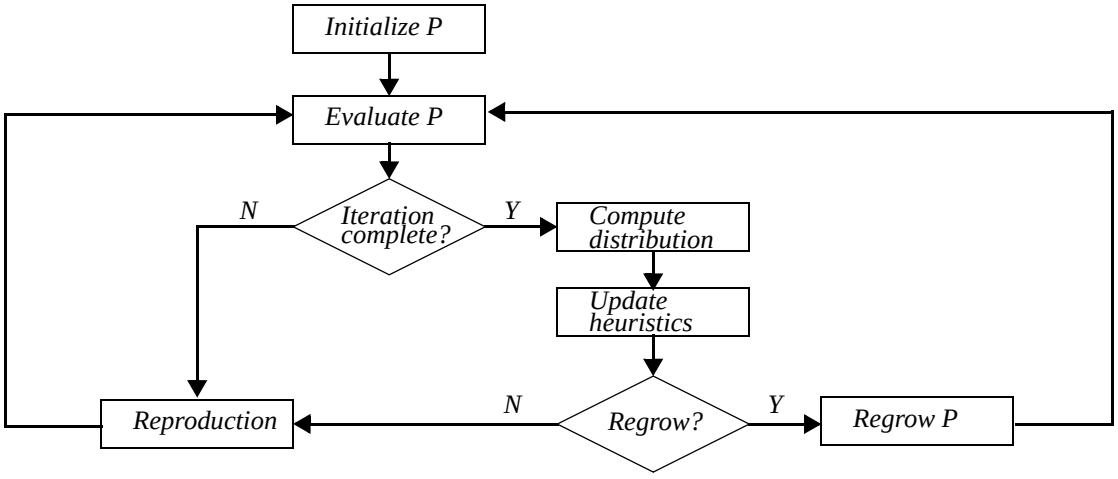


The working envornment for ACGP is the same as for CGP, except that the non-uniform probabilistic distribution space is being adjusted during the run of the GP system.

The user may or may not provide any initial stroing constraints and/or heuristics. ACGP will adjust the heuristics (or the uniform space if no heuristics are given apriori), favoring heuristics leading to better fitness and smaller trees (subject to parameters).

Note that apriori strong constraints cannot be undone by ACGP. On the other hand, heuristics can be adapted, and new strong constraints can be discovered.

Figure 11 Main loop for *ACGP2.1*.



ACGP1.1 adjusts the heuristics by observing the distribution of such local heuristics in the population. The heuristics can be updated, or completely replaced by new ones. Following *lilgp1.02*, *ACGP1.1* can run multiple populations (but not ADFs). However, only a single set of heuristics is extracted from all the populations.

The main generation loop for *ACGP1.1* is presented in Figure 11. *ACGP1.1* operates in *iterations*, where an iteration is a unit learning interval. An iteration can be anywhere from one generation to many generations. When iteration is not complete, the current loop is the same as one generation in the GP loop. When the current generation completes

and iteration, heuristics are extracted (from the current population(s)), updated (or raplaced), and then new generation is produced either by the same operators as during a regular generation, or by regrowing a new population (Regrow=Y).

Also, mutation can be redone like crossover, not explicit roulette, but one built on the fly. This way can easily use the new heuristics, and also redo CGP2.2 (levels) without expandsing to the varioous levels but rather by comuting for each mutation set member what is the min and max level on which this can be used. The advantage is that fewer weights on various levels must be precomputed after each update, but the must do more on the fly? TBD once we decide what counters to use and how.

6.1 Distribution Counters

Figure 12 The context information in the tree structure in ACGP1.1.1 (left) and ACGP2.1 (right), as expressed in the tree and as explored..

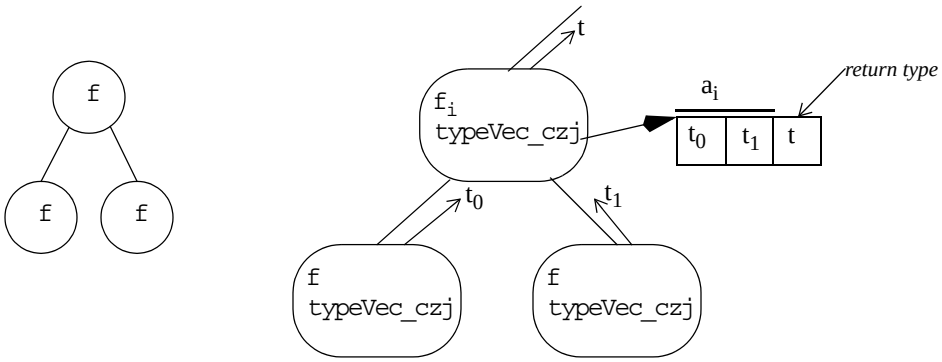
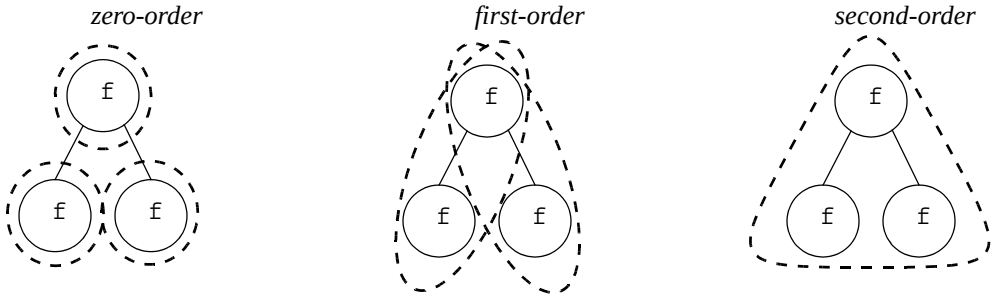


Figure 12illustrates the context information that is explored for the duistribution infrmation in ACGP1.1.1 and ACGP2.1. In v1.1.1, the information consisted of the node labels (functions and terminals), and the distribution was collected by exploring one parent-child at a time. In v2.1 the information is much richer, as explained in Section 4.4.12.

ACGP2.1 will use a few distribution counters to collect various heuristics: on functions/terminals, on types, and in combination.

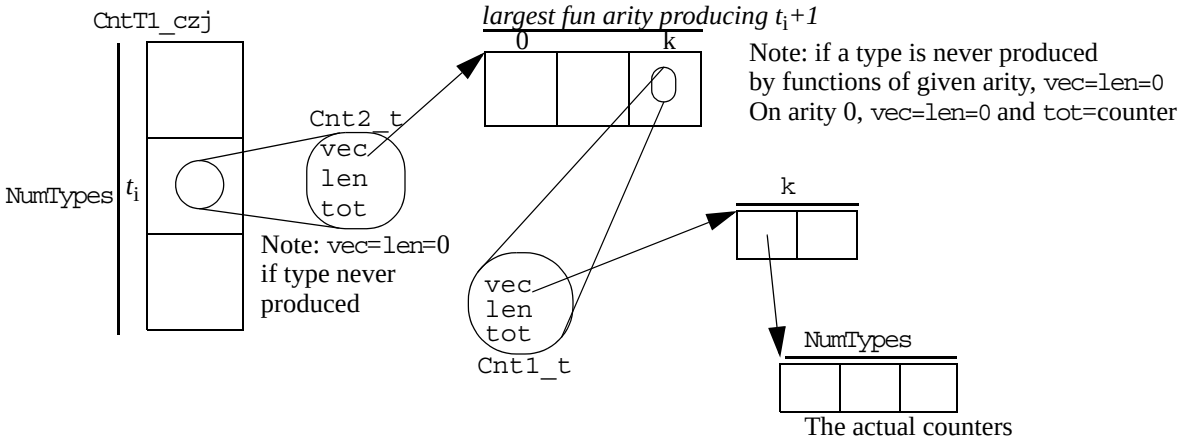
Figure 13 Order of the heuristics.



6.1.1 First-order Heuristics on Types: Counter `CntT1`

The counter `CntT1` collects first-order heuristics on types only. For every type available, it considers all function arities separately. For terminals (zero arity), it simply collects the number of times this type has been used. For a given non-zero arity, it considers all functions of this arity producing this type, counting the types used only. If there are any, it counts the type used on each argument, separately. This is illustrated in Figure 14.

Figure 14 Partially allocated and instantiated `CntT1_czj` structure.

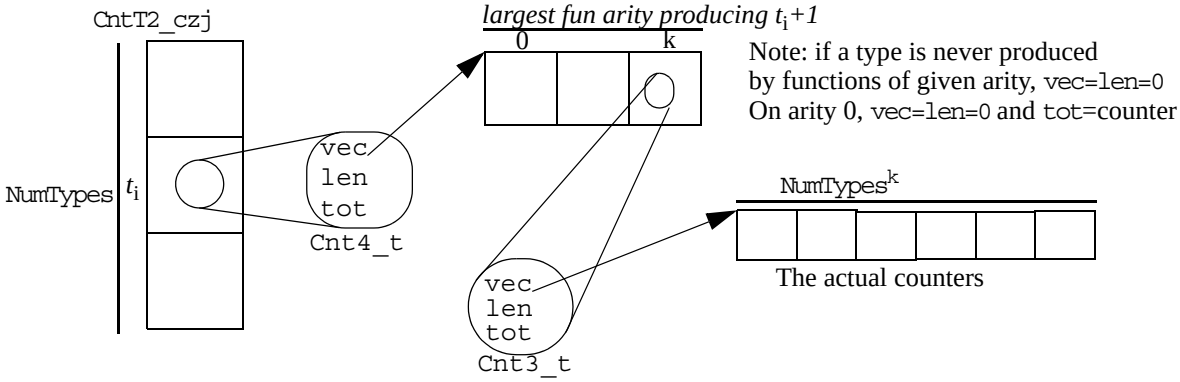


This counter is not very useful for typeless applications but it will be used for statistics.

6.1.2 Second-order Heuristics on Types: Counter `CntT2`

The counter `CntT2` collects second-order heuristics on types only. For every type available, it considers all function arities separately. For terminals (arity zero), it simply collects the number of times this type has been used. For a given non-zero arity, it considers all cominations of types producin the given type (cobinations over all arguments). If there are any, it counts the type used on each argument, separately.

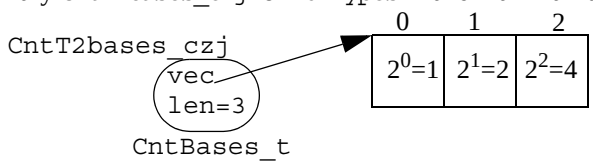
Figure 15 Partially allocated and instantiated `CntT2_czj` structure.



This counters collects the following: $(\forall a = arity)(\prod_{a} type \rightarrow type)$. This is illustrated in Figure 15. This counter is not used for typeless applications. Even though this coungter expresses second-order heuristics, the information needed is contained in a single node (using the `typeVec`).

The index in the actual counter array is computed as follow: take the children types (left child the most significant), read the number as a number in base `NumTypes`, and take the decimal equivalence. For example, for `NumTypes`=2 and functions of arity 3, the counter is of size 2^3=8. The context where the hildren are labeled (felt to right) as giving type 011_2 will be counted at index 3_{10}. The index is computed by the function `int acgp_indexCntT2(arity,typeVec)`. The actual computation is helped with an array of base `NumTypes` multipliers allocated and initialized in `CntT2bases_czj`, as illustrated in Figure 16 for the same example (0112 is 0*1+1*2+1*4

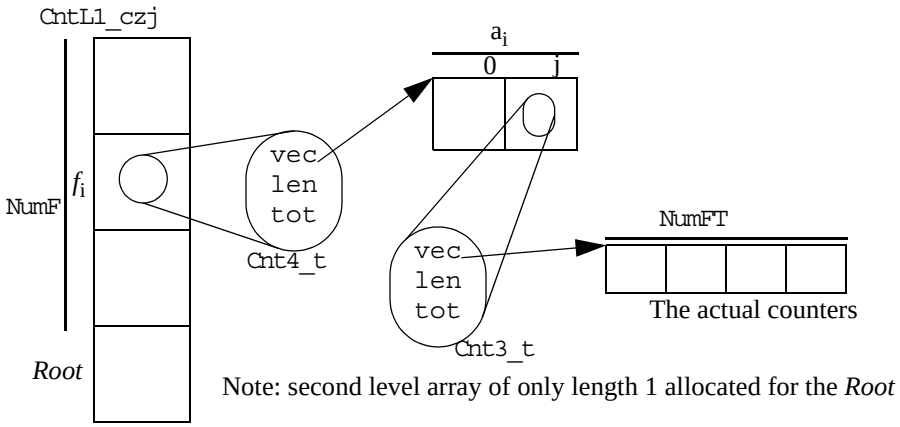
Figure 16 Auxilary `CntT2bases_czj` for `NumTypes`=2 and maximal function arity 3 in an application.



6.1.3 First-order Heuristics on Labels Only: Counter `CntL1`

The counter `CntL1` collects simple first-order typeless heuristics as the following distribution information: $function \times arg \rightarrow function$ (equivalent to the counters in *ACGP1.1.1*). That is, for every function and every of its argument it counts what labels the child. This is illustrated in Figure 17.

Figure 17 Partially allocated and instantiated `CntL1_czj` structure.

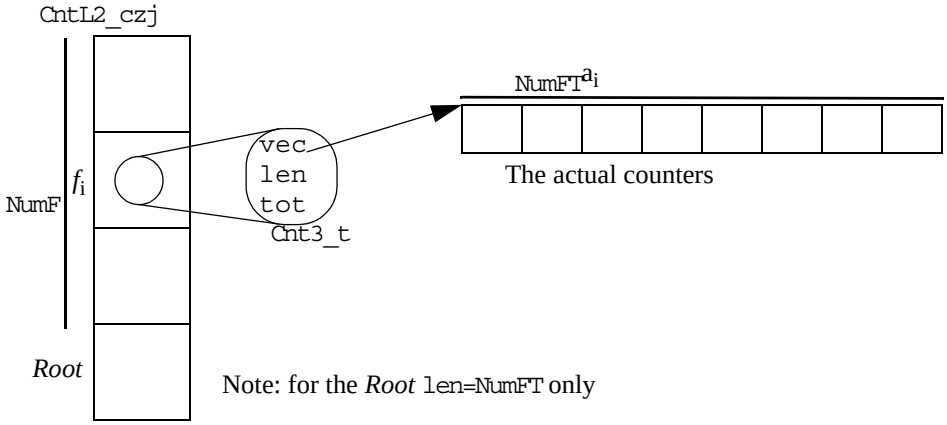


Note: second level array of only length 1 allocated for the *Root*

6.1.4 Second-order Heuristics on Labels Only: Counter `CntL2`

The counter `CntL2` collects simple first-order typeless heuristics as the following distribution information: $\left(\prod_{a = arity} function \right) \rightarrow function$. That is, for every function and every of its argument it counts what labels the child. This is illustrated in Figure 18.

Figure 18 Partially allocated and instantiated `CntL2_czj` structure.



Note: for the *Root* `len=NumFT` only

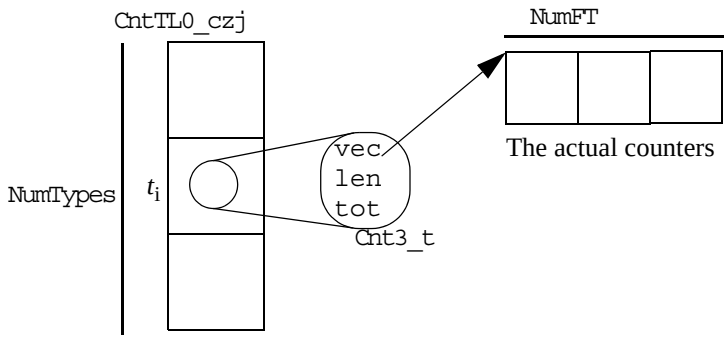
Note that this array can get long. For the 11-multiplexer problem, `not` will have the array of length 15, `and` and `or` will have it of length 225, while `if` will have it of length 3375 (15^3).

Indexing is done similarly to that for the counter `CntT2_czj`, through the function `int acgp_intexCntL2(int arity, int *children)`. The `children` array is allocated and initialized in the `acgp_count_czj()` function. The bases are precomputed in `CntL2bases_czj` for the largest arity and used as needed. For example, for the `if` function having the three children a_0 (FT index 4), `if` (0), d_2 (9), respectively, the counter index will be $4*15^2+0*15+9*1=909$.

6.1.5 Zero-order Combined Heuristics: Counter `CntTL0`

The counter `CntTL0` collects simple heuristics information: $type \rightarrow label$. That is, for every type produced, what functions produce it. This is illustrated in Figure 19.

Figure 19 Partially allocated and instantiated `CntTL0_czj` structure.

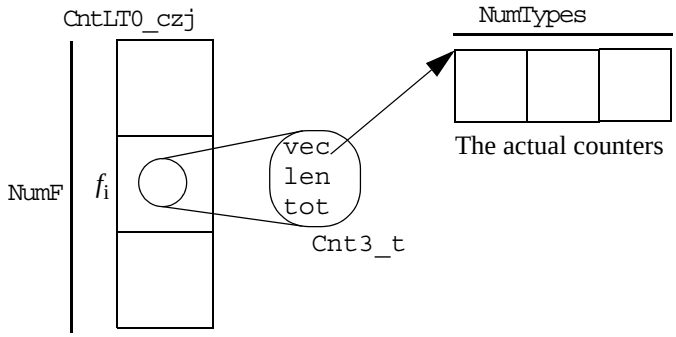


This counter is not very useful for typeless applications but will be used for statistics.

6.1.6 Zero-order Combined Heuristics: Counter `CntLT0`

The counter `CntLT0` collects simple heuristics information: $function \rightarrow type$. That is, if a function needs to be used, what types whould it produce (note that terminals are fixed type and thus no heuristics needed). This is illustrated in Figure 20.

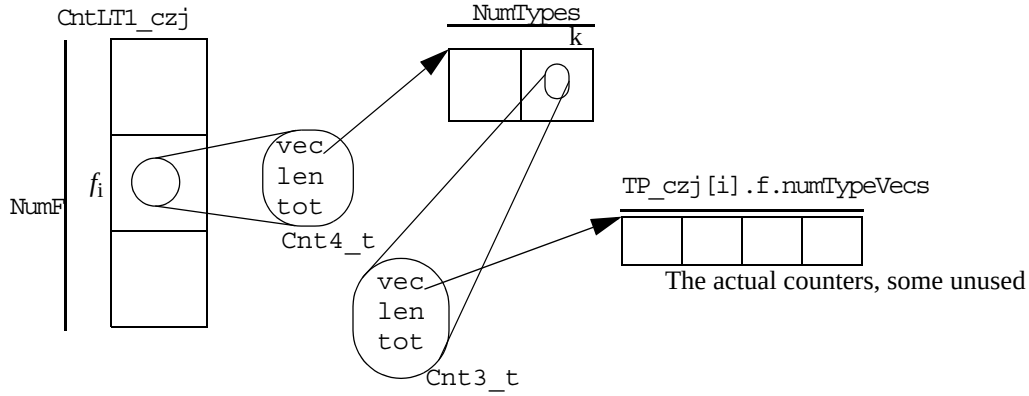
Figure 20 Partially allocated and instantiated `CntLT0_czj` structure.



This counter is not used for typeless applications.

6.1.7 First-order Combined Heuristics: Counter `CntLT1`

Figure 21 Partially allocated and instantiated `CntLT1_czj` structure.



The counter `CntTL1` collects combined heuristics information, while handling one node at a time (not looking at parent-child). The following distribution information is collected: $function \times type \rightarrow$ `typeVec`. That is, for every function that has to generate a specific type, this is the counter of what function instance (`typeVec`) is used. This is a limited-context heuristic, but in links information among three levels: the given node, the parent who determines what type the node has to generate, and the children, based on the instances of the function. Therefore it is very simple yet very rich

information. This information can apparently be used to put heuristics on the different function instances.

This heuristics can be read as either:

- if a function of a given type needs to generated, what instance to use
- if a type from a given function needs to be generated, what instance to use

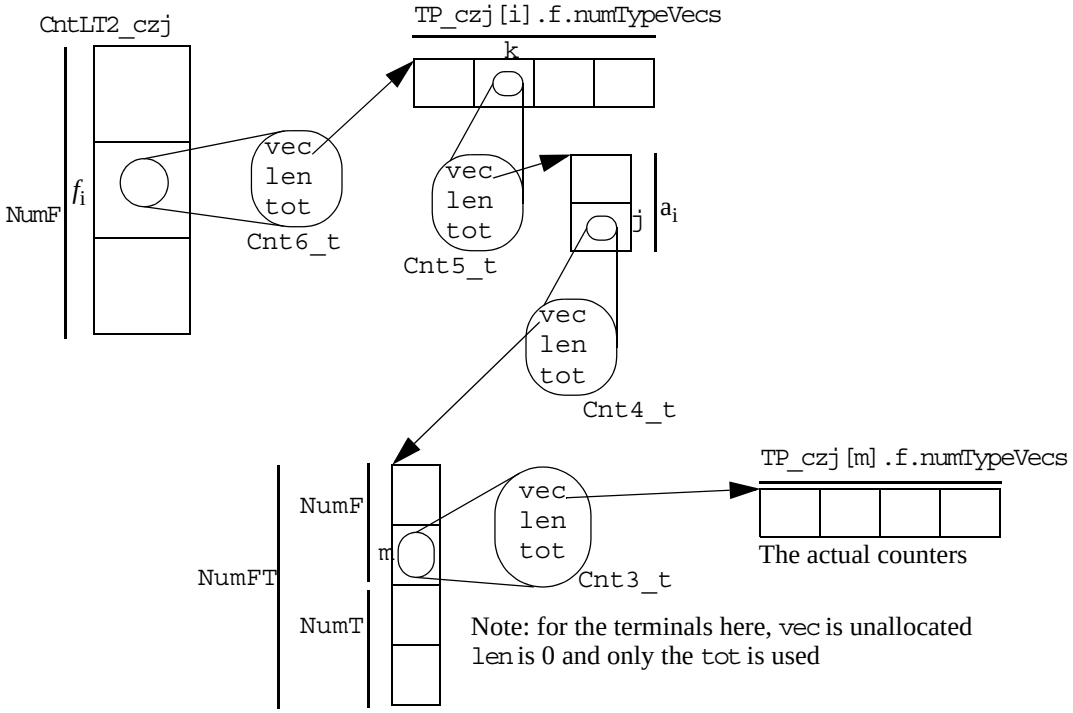
This counter is not used for typeless applications. It is really a zero-order heuristics yet quite rich in contents.

6.1.8 First-order Combined Heuristics: Counter `CntLT2`

The counter `CntLT2` collects more detailed *first-order* heuristics as the following distribution information: $function \times \texttt{typeVec} \times arg \rightarrow function \times \texttt{typeVec}$. That is, for every function that is using a specific instance, and for every child argument, it collects the information about what function/terminal labels the child if the child is a function, what instance of the function it is. This information is much richer. Note that for the child we collect the instance not the type information because the type information of this child is already implied by the instance of the parent and thus is not subject to modification. The `total` field is the only one used for terminals (`typeVecs=NULL`) and it may be used for functions in updfating the heuristics if there is no desire to differentiate between different child's instances.This is illustrated in Figure 22.

This counter is not used for typeless applications.

Figure 22 Partially allocated and instantiated `CntLT2_czj` structure.



6.2 Using the Counters to Update Heuristics

The counters can be used directly, by modifying initialization/mutation/crossover, or by modifying the underlying CGP2.1 mechanisms. TBD. At present, we only analyze the counters afterwards. Counters can be used directly to affect mutation/crossover, or they can be used to modify the existing CGP2.1 weights (probably after changing the function overloading from boolean to probabilistic instances with a roulette wheel).

6.3 New Output Files

On every ACGP run (`acgp.acgp`>0), the *`output.basename`*`.acgp` file is generated, with all the independent and dependent acgp-paramaters, and the initial weights. The counters are computed and printed according to the acgp.file_interval and acgp.pop_sampling parameters. Each of the eight counters is printed in a different file. Moreover, each counter may be printed into between one and four different files. The general form of a counter filename is

output.basename.??.*XXX* where

- *XXX* is the name of the counter w/o the _czj
- ?? is of the form [g|i][b|w], with
  - g - generation data
  - i - iteration data
  - b - sampling of best trees
  - w - statistics from the whole population

No weight modifications are printed at present as the weights are not adjusted yet. When they are, some weight files will have to be added.

The counter files are as follow:

1. If acgp.acgp==0 then none
2. If acgp.acgp>0 then for every counter the following files (the ?? above) are produced:

Counter files generated		acgp.pop_sampling	
		0	1
acgp.file_interval	0	ib	ib+iw
	1	ib+gb	ib+iw+gb+gw

located in the following indexes in the file array for each counter:

File kind	File array index
ib	0
iw	1
gb	2
gw	3

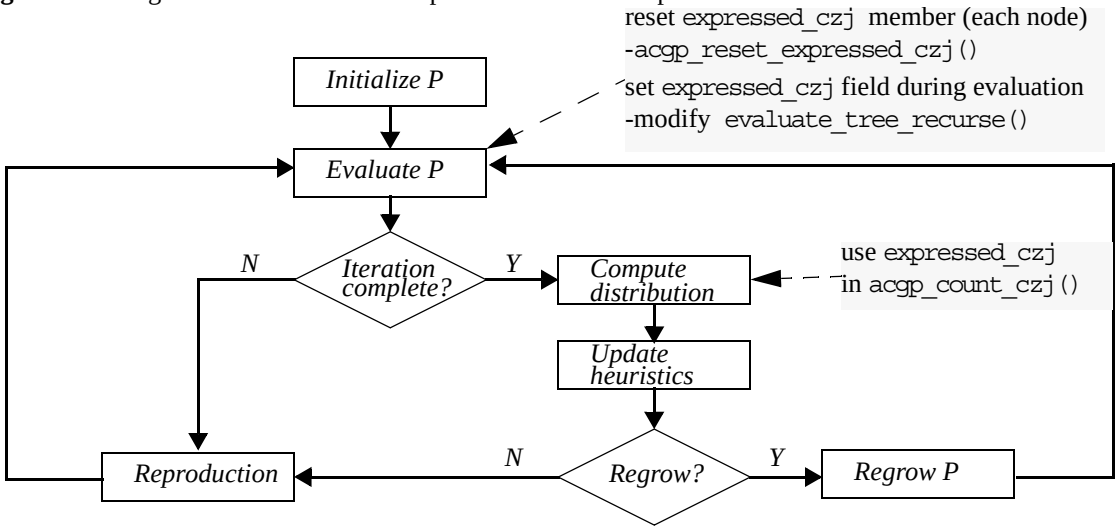
6.4 Implementation Changes

There are new functions and data structures in cgp_czj.c, to count the distribution and print them into a file. All functions and data structures have the acgp prefix.

gp.c is modified in three ways. Communication into this file is accomplished through two new global variables Acgp_regrow and Acgp_stop_on_term.

1. The generation loop is modified to take Acgp.stop_on_term paramater into account.
2. The *lil-gp* function generation_information(), called in the generation loop at the end of each loop, calls acgp_czj() on every generation, which in turn sets the current value of Acgp_regrow, and computes and prints all counters as needed.
3. At the end of the generation loop, reproduction is redirected to regrowing as needed according to Acgp_regrow (see Figure 11).

Figure 23 Changes in the basic ACGP loop to accommodate expressed info.



The data structure for the node is changed to include the floating field `expressed_czj`, which counts the number of times a given node is visited during evaluation. Note that ifd the evaluation is done only once, then the field is going to be 0 or 1. If the evaluation is done more than once (as to evaluate for different data), this will be the actual count. How this information is utilized in counting the distribution depends on the new parameter acgp.use_expreessed.

The changes needed in the basic ACGP loop are shown in Figure 23.

7 ACGP 2.1 User's Manual

The following are *ACGP2.1* specific parameters (some are from *ACGP1.1.1*). They can be specified the same way as any *ligp1.02* (or *CGP2.1*) parameters. The following gives the parameter's name, range, default value, and interpretation.

- `acgp.use_trees_prct`
  (0..1]
  0.1
  The effective rate for distribution sampling, that is this prct of all trees (from all populations if running multiple populations) will be taken for sampling. Note that the actual number is taken by applying the ceil() function and thus the actual number of trees may happen to be somehow larger than predicted.
- `acgp.select_all`
  [0,1]
  1
  1 - Extract `acgp.use_trees_prct` best (after sort) out of each population then take them all for sampling
  0 - Extract `sqrt(acgp.use_trees_prct)` best of each pop then resort and take again `sqrt(acgp.use_trees_prct)` resulting in `acgp.use_trees_prct` effective rate
- `acgp.extract_quality_prct`
  [0..1]
  0.99
  Two trees with fitness diff by no more than (`1-acgp.extract_quality_prct`) is considered same fitness and thus compared on size
- `acgp.gen_start_prct`
  [0..1]
  0.0
  Start extracting at generation `acgp.gen_start_prct*MaxGen`
- `acgp.gen_step`

[1..MaxGen]
1.
After starting extracting, extract at this gen interval (this is the iteration length)

- acgp.gen_slope
  [0,1,2]
  1
  0 - Use extracted heuristics to update the old heuristics at the constant rate of acgp.gen_slope_prct, but if this is 0 then use constant rate of $\sqrt{1/numIterations}$
  1 - Use extracted heuristics to update the old heuristics with rate increasing with iteration number
  The heuristics formula is $newWeight = oldWeight(1-r) + counterRatio \cdot r$
  where $counterRatio$ is the frequency of a heuristic to all heuristics of a given parent, and r is the rate of change.
- acgp.gen_slope_prct
  [0..1]
  0.10
  See above.
- acgp.0_threshold_prct
  [0..1]
  0.025
  If a weight drops to weight such that weight/mutSetSize less than acgp.threshold_prct, then drop weight to 0.
- acgp.acgp
  [0,1,2]
  0
  0 - CGP run, no statistrics no ACGP outputs
  1 - ACGP run, no regrow
  2 - ACGP run, with regrow
- acgp.file_interval
  [0,1]
  0
  0- produce the statistics and counter files at every iteration only
  1- produce the statistics and counter files at every generation and iteration
- acgp.pop_sampling
  [0,1]
  0
  0 - count from the selected best trees according to acgp.use_trees_prct and acgp.select_all
  1 - count also from the whole population
- acgp.stop_on_term
  [0,1]
  1
  0 - Continue remaining generations even on solving (term)
  1 - Stop generations upon solving
- acgp.use_expressed
  [0,1,2]
  0
  0 - collect distribution from all nodes
  1 - skip over subtrees which are not expressed (not used in evaluation) and use the other subtrees with the same weight
  2 - use distribution proportional to the number of visitis in a node (equivalent to option 1 if a node is visitied 0 or 1 times only)

If you desire to use regrowing (acgp.acgp must be 2), you must include regrow operator as one of the *lilgp* oper-

ators, as the last one listed (the operator is defined with *ACGP 1.1*). If you dont want to use regrow during regular reproduction, set its probability to a very low value.

8 Bibliography

[1] Lawrence Davis (ed.). *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[2] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1989.

[3] John Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[4] Cezary Z. Janikow. *"Constrained Genetic Programming"*. Submitted to *Evolutionary Computation*.

[5] Cezary Z. Janikow. *"A Methodology for Processing Problem Constraints in Genetic Programming"*. *Computers and Mathematics with Applications*, Vol. 32, No. 8, pp. 97-113, 1996.

[6] Cezary Z. Janikow. *"Adapting Representation in Genetic Programming"*. *Proceedings of GECCO 2004*, 6/2004, TBP

[7] Cezary Z. Janikow. *"ACGP: Adaptable Constrained Genetic Programming"*. *Proceedings of GPTP04*, 5/2204, TBP

[8] Kenneth E. Kinnear, Jr. (ed.). *Advances in Genetic Programming*. The MIT Press, 1994.

[9] John R. Koza. *Genetic Programming*. The MIT Press, 1992.

[10] John R. Koza. *Genetic Programming II*. The MIT Press, 1994.

[11] Douglas Zongker & Bill Punch. *"lil-gp 1.0 User's Manual"*. `zongker@isl.cps.msu.edu`.