

PolyGP: A Polymorphic Genetic Programming System in Haskell

Tina Yu and Chris Clack

Department of Computer Science
University College London

Gower Street, London WC1E 6BT, U. K.

T.Yu@cs.ucl.ac.uk C.Clack@cs.ucl.ac.uk

<http://www.cs.ucl.ac.uk/staff/t.yu> <http://www.cs.ucl.ac.uk/staff/c.clack>

ABSTRACT

In general, the machine learning process can be accelerated through the use of additional knowledge about the problem solution. For example, monomorphic typed Genetic Programming (GP) uses type information to reduce the search space and improve performance. Unfortunately, monomorphic typed GP also loses the generality of untyped GP: the generated programs are only suitable for inputs with the specified type. Polymorphic typed GP improves over monomorphic and untyped GP by allowing the type information to be expressed in a more generic manner, and yet still imposes constraints on the search space. This paper describes a polymorphic GP system which can generate polymorphic programs: programs which take inputs of more than one type and produce outputs of more than one type.

1. Introduction

Automatic programming relies on additional knowledge about the problem solution to guide code synthesis and transformation [Lenat, 1984]. Genetic Programming (GP) [Koza, 1992] automatically generates programs by using a genetic algorithm to search through the space of all possible program solutions. In its traditional style, the GP paradigm does not use type information when performing program generation due to its inability to distinguish different types: we use the term “untyped” to refer to such a system. Untyped GP leads to an unnecessarily large search space. When type constraints are enforced, the search space can be reduced since only type-correct programs are allowed in the search space. Previous work [Montana, 1995; Haynes, Schoenefeld and Wainwright, 1996; Clack and Yu, 1997] has

demonstrated that such type constraints facilitate GP in program generation.

Type constraints can be applied to GP in two different ways: Monomorphic GP and Polymorphic GP. Monomorphic GP uses monomorphic functions and terminals to generate monomorphic programs. In contrast, polymorphic GP can generate polymorphic programs using polymorphic functions and terminals. In the first instance, inputs and outputs of a program need to have the specified type. In the latter case, programs can accept inputs and outputs of more than one type. Polymorphic GP therefore generates more general solutions than those produced by monomorphic GP.

The generality of polymorphic GP is achieved through the use of different type variables (see Section 4.1). Polymorphic functions and terminals in the function and terminal sets are specified using *dummy type variables*. Within program parse trees, polymorphism is expressed using *temporary type variables*. To state the polymorphic nature of the generated program, *generic type variables* are used. We have developed a type system to handle the instantiation of all of these type variables so that both the type-correctness and the generality of the programs are maintained.

The search space of monomorphic GP (M) vs. polymorphic GP (P) vs. untyped GP (U) is as the following:

$$M \subset P \subset U$$

Monomorphic functions and terminals restrict inputs and outputs to be the specified types. The number of legal programs constructed with monomorphic functions and terminals are therefore the most limited. By contrast, type variables can be instantiated to any type in a program. However, the same type variables are restricted to be instantiated to the same type values (see Section 4.3). The search space of polymorphic GP is therefore less restrictive than monomorphic GP but more restrictive than untyped GP.

The paper is structured as follows: Section 2 summarizes related work; Section 3 explains our system structure; Section 4 presents the type system; Section 5 discusses the implementation of the system and Section 6 concludes.

2. Related Work

Generic functions in Montana's Strongly Typed Genetic Programming (STGP) system [Montana, 1995] provide a form of parametric polymorphism. Generic functions are parameterized templates that have to be instantiated with actual values before they can be used. The parameters can be type parameters, function parameters or value parameters. Generic functions with type parameters are polymorphic since the type parameters can be instantiated with many different type values.

In STGP, the function set may contain generic functions. To be used in a parse tree, a generic function has to be instantiated by specifying the argument and return types of the generic function. Instantiating a generic function can be viewed as making a new copy of the generic function with specified argument and return types. Instantiated generic functions are therefore monomorphic functions.

Montana uses a table-lookup approach to create parse trees using monomorphic functions and terminals. If a function takes an argument of type X then this implicitly constrains its child to produce a value of type X. There is a type possibility table which provides type constraints according to the depth in the tree where type matching occurs: this extra information constrains the choice of function to create nodes in the tree to ensure that the tree can grow to its maximum depth. During the creation of the initial population, each parse tree is grown top-down by choosing functions and terminals at random within the constraints of the types in the table. In this way, the initial population only consists of parse trees that are type-correct. (Similar rules are applied during the genetic operations of crossover and mutation).

To generate generic programs in STGP, *generic data types* are introduced. "Generic programs" are those that have *generic data types* as input or output types. During the generation of the generic programs, *generic data types* are treated as additional built-in types. An instantiated generic function whose argument or return types are specified as *generic data types* is therefore a monomorphic function. The *generic data types* are not instantiated until the generic program is executed. Since *generic data types* can be instantiated with many different type values, the generated programs are generic programs.

The PolyGP is similar to STGP in that it supports parametric polymorphism but with the following distinctions:

- it uses a type unification algorithm rather than table-lookup mechanism to instantiate type variables.
- it uses *temporary type variables* to support polymorphism within a program parse tree as it is being created.
- it uses *generic type variables* to represent polymorphism of the generated programs. However, unlike *generic data types* in STGP, *generic type variables* are never instantiated.

- the type system supports higher-order functions (functions that take functions as arguments and/or return functions as outputs). [Yu and Clack, 1998] shows how higher-order functions can be used to support module creation and implicit recursion in GP.

A performance comparison between PolyGP and STGP can be found in [Clack and Yu, 1997].

3. System Structure

The system has four major components: Creator, Evaluator, Evolver and Type System. Figure 1 illustrates the high-level structure of the system. The creator interacts with the type system to select type-matched functions and terminals to create type-correct programs. The evaluator evaluates each program using test data as inputs to produce some outputs. The outputs are passed over to the fitness function which assigns a fitness value for the program according to the correctness of the outputs. If the fitness value satisfies the requirement, the system stops and returns the program with the satisfactory fitness value as the solution. Otherwise, the evolver is invoked to perform genetic operations to create new programs. The test-select-reproduction process continues until a satisfactory program is found.

An extra component of the PolyGP system, compared with the standard GP system, is the type system. The type system is used during program creation and evolution (crossover and mutation). The purpose of the type system is to ensure that all programs created are type-correct. To use the type system, users have to specify input and output types for each function and terminal in the function and terminal sets. The type syntax and the details of the type system are given in Section 4.

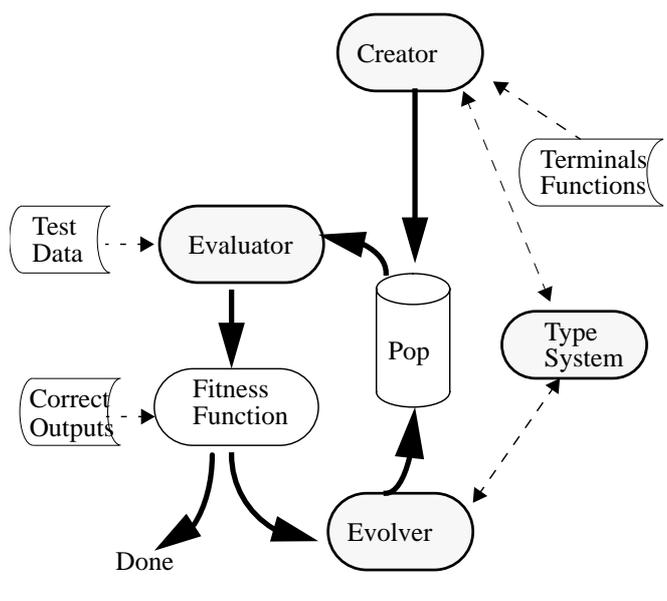


Figure 1: High-level system structure

3.1 Creator

The programs created are represented as parse trees. A parse tree is grown from the top node downwards. There is a required type for the top node of the tree. The creator invokes the type system to select a function whose return type unifies with the required type. The selected function will require arguments to be created at the next (lower) level in the tree: there will be type requirements for each of those arguments. If the argument has a function type, a λ abstraction (see next paragraph) will be created to represent it. Otherwise, the type system will randomly select a function (or terminal-see later) whose return type unifies with the new required type to construct the argument node.

Arguments of function type are created as λ abstractions. λ abstractions are local function definitions, similar to function definitions in a conventional language such as C. The following is an example λ abstraction together with an equivalent C function.

```
( $\lambda$  x (+ x 1))    ( $\lambda$  abstraction)
Inc (int x)      (C function)
{return (x+1);}
```

λ abstractions allow GP to perform module creation and reuse which are essential for GP to be effective with larger and more complex problems. By using λ abstractions, we are able to solve the Even-N-Parity problem [Koza, 1992] very efficiently [Yu and Clack, 1998].

λ abstractions are created using the same function set as that used to create the main program. The terminal set, however, consists only of the arguments of the λ abstraction to be created; no global variables are included. Argument naming in λ abstractions follows a simple rule: each argument is uniquely named with a hash symbol followed by a unique integer, e.g. #1, #2. This consistent naming style allows crossover to be easily performed between λ abstractions with the same number of arguments (see Section 3.3).

When selecting functions to construct an argument node, it is possible that there is no function in the function set whose return type can unify with the required type. In this case, the creator stops growing the tree by calling the type system to select a terminal at random whose type unifies with the required type. This approach of random selection of functions and terminals to create type-correct parse trees works well most of the time (85%). A backtracking mechanism is implemented to regenerate a new subtree when the creation of a particular subtree fails [Clack and Yu, 1997].

The program parse trees are represented in a “curried” form (a function is applied to one argument at a time), thus allowing partial application to be expressed [Peyton-Jones, 1987 Ch.10]. The advantage of such a representation is to provide more crossover locations so that more diverse new programs can be created. With more diverse programs in the population pool, we hope that GP can find a solution faster. The result of our initial experiment is consistent with this

conjecture [Clack and Yu 1997]. More detailed analysis of the partial application node crossover (see Section 3.3) is underway.

With a “curried” format parse tree, each function application has two branches: a function and an argument. Figure 2 is the curried format parse tree for the IF-TEST-THEN-ELSE function. The @ denotes an application node and is a possible crossover location. The function (IF (TEST-exp) (THEN-exp) (ELSE-exp)) has two branches: (IF (TEST-exp) (THEN-exp)) and (ELSE-exp). The first function branch, (IF (TEST-exp) (THEN-exp)), also has two branches: (IF (TEST-exp)) and (THEN-exp). The (IF (TEST-exp)) also has two branches: IF and (TEST-exp).

When creating the curried format parse tree, we expand the tree in a depth-first-right-first manner, i.e. we complete the creation of the argument subtree before start working on the function branch subtree. In the IF-TEST-THEN-ELSE parse tree example, we first create the ELSE-exp subtree, then the THEN-exp subtree, then the TEST-exp subtree. If any of the subtree creation fails, the creator will call the type system to select another function (other than IF-TEST-THEN-ELSE) to regenerate a new tree.

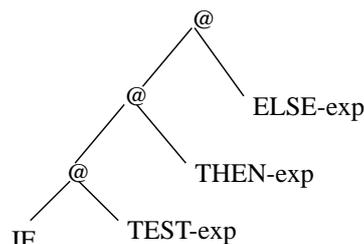


Figure 2: Curried format parse tree for the IF-TEST-THEN-ELSE function.

3.2 Evaluator

The system generates expression-based programs (λ -calculus). The benefits of using expression-based programs to represent solutions in GP are discussed in [Clack and Yu, 1997]. Here, we describe the abstract syntax of our programs:

```
exp :: c          constant
     | x          identifier
     | f          built-in function
     | exp1 exp2  application of one exp to another
     |  $\lambda$  x . exp  lambda abstraction
```

Constants and identifiers are given in the terminal set while functions are provided in the function set. Application of expressions and λ abstractions are constructed by the creator.

To evaluate a generated program, the program is first converted into a λ abstraction by wrapping it with λ notation and input variables which have been made available to the program as members of the terminal set. The evaluator then applies test data to the λ abstraction, one at a time, to produce some outputs. The application of the λ abstraction to the test data is a process of syntax transformation. It involves

a sequence of applications of β and δ reduction rules [Peyton-Jones, 1987 Ch. 2]. We first describe these reduction rules:

- β rule is the function application rule. It produces a new instance of the function body by substituting the arguments of the function with the formal parameters.

$$(\lambda x.E) M \Rightarrow E [M/x].$$

The notation $E[M/x]$ means the expression E with M substituted for free occurrences of x .

- δ rules are rules associated with the functions in the function set. For example, the IF-THEN-ELSE function has one rule to describe how it should be transformed.

When applying the β -rule, we perform normal order evaluation: the *leftmost outermost* expression is evaluated first, i.e. $exp1$ is evaluated before $exp2$. Intuitively, this means the body of a function is evaluated first and the arguments are evaluated when necessary. If a program terminates, the order of evaluation should not make any difference; they should reach the same result. Unfortunately, not all programs terminate. The Church-Rosser Theorem II says normal order evaluation is the most likely to terminate [Rosser, 1982]. We therefore prefer normal order evaluation rather than any other evaluation order.

The following is an example to demonstrate the operation of the program evaluation; here x is a program input variable existing in the terminal set.

```
(λ x (+ x (λ #1 (* #1 #1)) 5)) 10
β=> (+ 10 (λ #1 (* #1 #1)) 5)
β=> (+ 10 (* 5 5))
δ=> (+ 10 25)
δ=> 35
```

3.3 Evolver

The evolver performs three kinds of genetic operation: application node crossover, λ modular crossover and mutation. We first explain the crossover location selection scheme and the point-typing constraints the system uses. The three operators are then presented.

3.3.1 Crossover Location Selection

A crossover location selection scheme which biases toward root crossover is used. This selection scheme is designed to accommodate the premature convergence of the root nodes which we have observed during our experiments and has been reported in [Gathercole and Ross 1996]. In brief, due to the restriction of tree depth, the standard crossover operator is not able to swap all possible pairs of subtrees between two parents and still produce “legal” trees. Instead, most of the genetic exchanges take place near the leaf nodes, with nodes near the root left unchanged. The premature convergence of the root node can severely impair GP performance if the behavior of a program depends highly on the program root node. This is the case with the “map” program [Clack and Yu 1997]. We modified the crossover selection scheme so that

program root nodes have more opportunities to be replaced with new nodes.

To select a crossover point, program trees are traversed in a depth-first, right-to-left manner. Moreover, the possibility of the node selection decreases exponentially, i.e. 50%, 25%, 12.5%, 6.25% and so on. Once the crossover node in the first program tree is selected, the same scheme is used to select a crossover location in the second parent program tree. This selection scheme has produced the expected effect of more diversified root nodes in the program populations. Currently, we are investigating other effects the scheme might have brought into the GP process.

3.3.2 Point-Typing Structure-Preserving Crossover

We perform the “point-typing” structure-preserving crossover [Koza, 1994, pp. 532] in our programs: a point is first selected from the first parent program; depending on the source of the node (the main program or a λ abstraction), a node with the same source is selected from the second parent program. This assures the produced offspring has valid syntax.

Two other constraints are imposed when performing crossover: type constraint and maximum tree depth constraint. After a node is selected from the first parent program using the described scheme, the type and the depth of the node are used to select a crossover point in the second parent; we use the same selection scheme to find a node whose type value unifies with the given type value and whose depth is such that the new tree will satisfy the maximum tree depth.

3.3.3 Application Node Crossover

Application node crossover can be performed on full application or partial application nodes appeared in either the main program body or any λ abstractions. By reference to the abstract syntax, this means that any occurrence of the application expression ($exp1\ exp2$) appearing at any place in one tree is a possible crossover location. We do not perform crossover on leaf level because it is more akin to point mutation rather than true crossover.

3.3.4 λ Modular Crossover

λ modular crossover swaps a λ abstraction in one program with a λ abstraction in another program. We view λ abstractions as structured building blocks and crossover is allowed to be performed on them like other program segments. This operation is similar to the modular crossover in [Kinnear, Jr., 1994]. However, with the benefit of the type system, there is no problem of arguments mismatching in our implementation. Each λ abstraction is annotated with a type (see Section 4) which indicates the number and type of its arguments. The type system assures that λ modular crossover is only performed using two λ abstractions with the same number and type of arguments.

3.3.5 Mutation

Mutation is straight-forward: we first use the described scheme to choose a mutation node in the tree; a new subtree

is created whose root has the same return type as the mutation node, with tree depth that would make the new tree satisfy the maximum tree depth parameter. The new subtree then replaces the mutation node subtree.

Mutation can be performed on partial or full application nodes. Our creator is capable of creating subtrees whose root returns a function type.

4. Type System

Our PolyGP system employs a type system to perform type checking so that invalid programs are never created. We define a syntax of valid types which annotate the nodes and leaves of program parse trees. This type information is used by the type system to validate program parse trees.

4.1 Type Syntax

Our abstract type syntax is given by:

$$\begin{aligned} \sigma &:: \tau && \text{built-in type} \\ &| v && \text{type variable} \\ &| \sigma_1 \rightarrow \sigma_2 && \text{function type} \\ &| [\sigma_1] && \text{list of elements all of type } \sigma_1 \\ &| (\sigma_1 \rightarrow \sigma_2) && \text{bracketed function type} \end{aligned}$$

$$\begin{aligned} \tau &:: \text{int} \mid \text{string} \mid \text{bool} \mid \text{generic}_i \\ v &:: \text{dummy}_i \mid \text{temporary}_i \end{aligned}$$

Every expression in the program may be annotated with a type:

- Constants such as 0 and identifiers such as x have a type pre-defined by the user;
- Functions also have pre-defined types (for example, the function HEAD has the type $[a] \rightarrow a$) where a is a dummy type variable;
- Applications have a type given as follows:
 - if exp1 has type $(\sigma_1 \rightarrow \sigma_2)$
 - and exp2 has type σ_1
 - then $(\text{exp1 } \text{exp2})$ has type σ_2
 - else there is a type error;
- λ abstractions have the following type:
 - if x has type σ_1
 - and exp has type σ_2
 - then $(\lambda x. \text{exp})$ has type $\sigma_1 \rightarrow \sigma_2$.

Our type system also supports higher-order functions whose types are indicated by the use of the bracketed function type.

The type system supports three kinds of type variables:

Generic Type Variables: The “generic” types are used to specify the polymorphic feature of evolved programs such as LENGTH, which has type $[G1] \rightarrow \text{Int}$ where $G1$ is a generic type variable. While the program is being evolved a generic type variable must *not* be instantiated: it therefore takes on the role of a built-in type.

Dummy and Temporary Type Variables: Dummy types are used to express polymorphism of functions in the function set and terminals in the terminal set: whenever they are used in a parse tree they must be instantiated to some other type (and the type must not involve a dummy type, but it may be a generic or temporary type). Note that if a dummy type variable occurs more than once in the type, then when the dummy type is instantiated it is necessary to instantiate all occurrences to the same type. This is done through the process of *contextual instantiation* which will be discussed in section 4.3. Typically, the constraints imposed by the return type of the functions mean that the dummy type will be instantiated as a known type. However, there are also situations where there are no such constraints and so the dummy type is instantiated as a new temporary type variable. This delayed binding of temporary type variables provides greater flexibility and generality; essentially it supports a form of polymorphism within the parse tree as it is being created.

Within a parse tree, temporary type variables must be instantiated consistently to maintain the legality of the tree. A global type environment is maintained for each parse tree during the process of tree generation: this environment records how each temporary type variable is instantiated. Once a temporary type variable is instantiated, all occurrences of the same variable in the parse tree are instantiated to the same type value.

4.2 Unification Algorithm

Our type system uses Robinson’s unification algorithm [Robinson, 1965] to select functions and terminals whose return types “unify” with the required type. The unification algorithm takes two type expressions and determines whether they unify, i.e. whether they are equivalent in the context of a particular set of instantiation of type variables (called a “substitution”, or a “unifier”). If the two type expressions unify, it returns their most general unifier, otherwise, it indicates the two type expressions do not unify. A more detailed description of the unification algorithm can be found in [Clack and Yu, 1997].

4.3 Contextual Instantiation

Type expressions which contain several occurrences of the same type variable, like in $a \rightarrow a$, express *contextual dependencies* [Cardelli, 1987]. Whenever an occurrence of a type variable is instantiated, all the other occurrences of the same type variable must be instantiated to the same type value. This is done through the process of *contextual instantiation*: applying a substitution θ , which contains the instantiation of type variables, to the type expression. The process is applied in two places in the type system:

- During the instantiation of dummy type variables of a polymorphic function. In this case, dummy type variables get instantiated and bound to type values.

- During the instantiation of temporary type variables in a parse tree. In this case, temporary type variables get instantiated and bound to type values.

5. Implementation

Both steady-state [Syswerda, 1989] and generational replacement are implemented in the system.

The system is implemented in Haskell 1.4 [Peterson and Hammond 1997] using the Glasgow Haskell Compiler version 2.02. Haskell is a non-strict purely functional programming language. Non-strict languages evaluate expressions by need. This is beneficial for our GP system because program parse trees normally contain “redundant expressions” (introns). Not-needed expressions would never have to be computed. The “purity” feature, however, is a disadvantage to our system because no side-effects are allowed in a program. The population pool (implemented as a list) in the system has to be passed around as an argument for updating during the GP run. Each time the population is updated, a new copy has to be made. This is a computationally expensive process.

6. Conclusion

We have presented a PolyGP system which utilizes type information to generate general purpose and type-correct solutions. The solutions are presented as polymorphic programs: programs that can take inputs of more than one type and produce outputs of more than one type. The PolyGP system uses 3 different kinds of type variables to represent polymorphism: dummy type variables, temporary type variables and generic type variables. We demonstrate our type system which handles the instantiation of all of these type variables so that both the type-correctness and the generality of the programs are maintained. The code for PolyGP can be found at <http://www.cs.ucl.ac.uk/staff/t.yu/PolyGP.tar.z> and an example of the system applied to the LENGTH function can be found at <http://www.cs.ucl.ac.uk/staff/t.yu/Example>.

Acknowledgments

We thank Benjamin Goldberg for his implementation of the unification algorithm in SML, which the type system is based on. The first author also likes to thank Amanda Clare for her comments on the paper.

Bibliography

Cardelli, L. 1987. Basic polymorphic type checking. *Science of Computer Programming*. Vol. 8, pp. 147-172.

Clack, C., and Yu, T. 1997. Performance enhanced genetic programming, *Proceedings of the Sixth International Conference on Evolutionary Programming*, Angeline, P.J., Reynolds, R., McDonnell, J., and Eberhart, R. (eds.), Springer-Verlag, Berlin, pp.87-100.

Gathercole, C., and Ross, P. (1996). An adverse interaction between crossover and restricted tree depth in genetic programming. *Genetic Programming 1996: Proceedings of the First Annual Conference Genetic Programming*. Koza, J.R., Goldberg, D.E., Fogel, D.B., and Riolo, R.L. (eds.), MIT Press, Cambridge, MA. pp. 291-296.

Haynes, T.D., Schoenefeld, D.A., and Wainwright, R.L. 1996. Type inheritance in strongly typed genetic programming. *Advances in Genetic Programming II*, Angeline, P.J. and Kinnear, Jr., K.E.(eds), MIT Press, Cambridge, MA, pp. 359-376.

Kinnear, Jr., K. E. 1994. Alternatives in automatic function definition: A comparison of performance. *Advances in Genetic Programming*, K.E. Kinnear, Jr.(ed.), MIT Press, Cambridge, MA, pp. 119-141.

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

Koza, J. R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

Lenat, D. 1984. The role of heuristics in learning by discovery: Three case studies. *Machine Learning: An Artificial Intelligence Approach*, Michalske, R., Carbonell, J., and Mitchell, T. (eds.), chapter 9, Springer-Verlag, pp. 243-306.

Montana, D. J. 1995. Strongly typed genetic programming. *Evolutionary Computation*, Vol. 3:3, pp. 199-230.

Peterson, J., and Hammond, K. editors 1997. Report on the programming language Haskell, a non-strict purely functional language (version 1.4). Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, 1997.

Peyton-Jones, S. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall International.

Rosser, J.B. 1982. Highlight of the history of the lambda calculus. *Proceedings 1982 ACM Conference on LISP and Functional Programming*. ACM, pp. 216-225.

Robinson, J.A. 1965. A machine-oriented logic based on the resolution principle. *Journal of ACM*. Vol. 12:1, pp. 23-49, January.

Syswerda, G. 1989. Uniform crossover in genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, Schaffer, J.D. (ed.), Morgan Kaufmann, San Mateo, CA, pp. 2-9.

Yu, T., and Clack, C. 1998. Recursion, lambda abstractions and genetic programming. *Genetic Programming 1998: Proceedings of the Third Annual Conference Genetic Programming*. (to appear)