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
United Kingdom

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

ABSTRACT

In general, the machine learning process can be accelerated through the use of heuristic 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 produces outputs of more than one type. We also demonstrate its operation through the generation of the "map" polymorphic program.

1 Introduction

The use of heuristic knowledge to assist learning is not new in Artificial Intelligence [Mitchell, Utgoff and Banerji, 1984]. This knowledge facilitates learning in two ways: when the existence of a solution is not known, the knowledge can guide the system to search one; when a solution is known to exist, the knowledge can reduce the search space for the learning system. In the Genetic Programming (GP) [Koza, 1992] paradigm, type information is one kind of heuristic knowledge that has been adopted to assist learning [Montana, 1995; Haynes, Wainwright, Sen and Schoenefeld, 1995; Haynes, Schoenefeld and Wainwright, 1996; Clack and Yu, 1997]. With type constraints, only type-correct programs are in the search space. The search space is therefore reduced.

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/outputs of a programs can only have one type. In the later case, programs can accept inputs/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. Polymorphic functions and terminals in the function/terminal sets are presented using *dummy type variables* or *generic type variables* if the functions and terminals are program inputs/outputs related. Within program parse trees, polymorphism is expressed by *temporary type variables*. To state the polymorphic nature of the generated program, *generic type variables* are used. We have developed a formal 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.

This paper builds on our previous work in [Clack and Yu, 1997]. In this paper, we have further explored the benefits of polymorphism in GP. We also present our PolyGP system in greater details. In particular, we explain how our type system instantiates different type variables to achieve polymorphism. Moreover, the system implementation language, Haskell, is discussed.

The paper is structured as follows: Section 2 discusses type and polymorphism; Section 3 summarizes related work; Section 4 explains our system structure; Section 5 presents the type system; Section 6 discusses the implementation of the system; Section 7 demonstrates the system through the generation of the "map" polymorphic program and Section 8 concludes.

2 Type and Polymorphism

Conventional typed languages, such as Pascal, are based on the idea that arguments/return values of functions/procedures have an unique type. Such languages are called *Monomorphic* languages. By contrast, *Polymorphic* languages allow arguments and return values to have more than one type. Programs whose inputs and/or outputs have more than one type are called polymorphic programs.

[Cardelli and Wenger, 1985] has classified polymorphism as the following:

$$Polymorphism \begin{cases} universal \{ \begin{matrix} parametric \\ inclusion \end{matrix} \\ ad\ \ hoc \{ \begin{matrix} overloading \\ coercion \end{matrix} \end{cases}$$

Universal polymorphic functions work on a large number of types (all the types have a given common structure), whereas ad-hoc polymorphic functions only work on a finite set of different and potentially unrelated types. In terms of implementation, a universal polymorphic function executes the *same* code for arguments of any admissible types, whereas an ad-hoc polymorphic function may execute *different* code for each type of argument.

Our PolyGP system is currently implemented with one particular kind of polymorphism: *parametric polymorphism*. Parametric polymorphic functions can take arguments of *any* types. The functions perform the same kind of work independently of the argument types. It is the purest form of polymorphism. We will extend the system to include other kinds of polymorphism in the near future.

3 Related Work

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

In STGP, functions in a 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/return types. Instantiated generic functions are therefore monomorphic functions (even the specified argument/return types are *generic data types*, which we will discuss in more details later).

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 have *generic data types* as inputs/outputs types. During the generation of the generic programs, *generic data types* are treated as built-in types. Generic functions that are instantiated with *generic data types* are therefore also monomorphic. 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.

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

- we use a type unification algorithm rather than table-lookup mechanism to instantiate type variables.
- our type system supports higher-order functions: functions that take functions as arguments and/or return functions as outputs.
- we use *temporary type variables* to support polymorphism within program parse tree as it is being created.
- we use *generic type variables* to represent polymorphism of the generated programs. However, unlike *generic data types* in STGP, *generic type variables* are never instantiated. Polymorphic programs generated by our system are type-correct and are guaranteed to be executed without any run-time type errors.

4 System Structure

The system has four major components: Creator, Evaluator, Evolvor 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. Evaluator evaluates each program using test data as inputs to produce some outputs. The outputs are passed over to the fitness function which assigns 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, evolvor 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 our PolyGP system, compared with the standard GP system, is the type system. The type system is used during program creation and evolution (cross-over 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 a type signature for each function and terminal in the function and terminal sets. The type syntax and the details of the type system will be given in the next section.

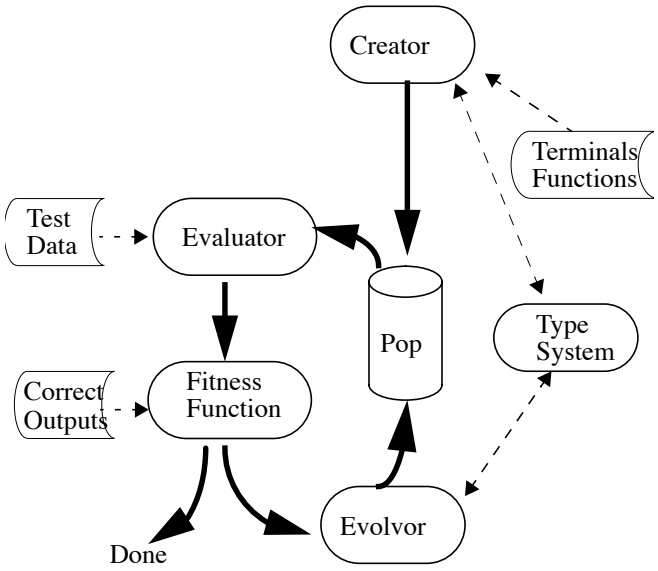

Figure 1: high-level system structure

4.1 Creator

The programs created are represented in 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, and once again the type system is used to select a function whose return type unifies with the new required type.

At any point, if the type system fails to find a function whose return type unifies with the required type, the creator stops growing the tree by calling type system to select a terminal 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. We ran an experiment using this approach to create 100 parse trees each with 32 nodes. In 10 runs of the experiment, the average number of nodes failed was 444. The success rate was 85%.

To handle the small portion of failing cases, we implement a backtracking mechanism. When the creation of a particular subtree fails, we backtrack to the particular node and regenerate a new subtree. This is an overhead of using our type system but, considering the small percentage failure rate, it's a price worth paying.

Our program parse trees are represented in "curried" form (a function is applied to one argument at a time), thus allowing partial application to be expressed. The advantages 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 solution faster. The result of our initial experiment is consistent with this conjecture [Clack and Yu 1997].

With "curried" format parse tree, each function application has two branches: a function and an argument. Figure 2 is the curried format parse tree for IF-TEST-THEN-ELSE function. 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 (THEN-exp)) and (THEN-exp). The (IF (THEN-exp) also has two branches: IF and (THEN-exp).

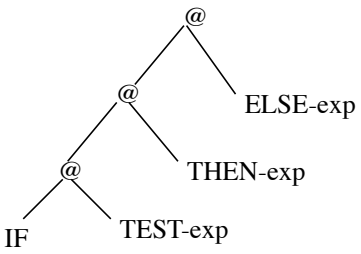

Figure 2: Curried format parse tree for the IF-TEST-THEN-ELSE function.

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 ELSE-exp subtree, then THEN-exp subtree, then TEST-exp subtree. If any of the subtrees creation fails, The creator will call the type system to select another function (other than IF-TEST-THEN-ELSE) to regenerate a new tree.

4.2 Evaluator

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

exp ::	c	*constant*
	| x	*identifier*
	| f	*function*
	| exp1 exp2	*application of one exp to another*

Constants and identifiers are provided in the terminal set while functions are provided in the function set. Application of expressions are constructed by the creator. Currently, we are working on incorporating λ abstractions in our program syntax to support hierarchical learning in polymorphic genetic programming [Yu and Clack, 1997].

The generated program is converted into a λ abstraction by wrapping it with λ notation and input variables. The evaluator then applies test data to the λ abstraction, one at a time, to produce some outputs. The evaluation of the application of the λ abstraction to the test data is a process of syntax transformation. It involves a sequence of application of some reduction rules: α, β and δ reduction rules. We first describe these reduction rules:

- α rule is a renaming rule. It simply renames a variable with a unique new name.

  $\lambda x.E \Rightarrow \lambda y.E\ [y/x]$, y is an unique new name.

- β rule is the application rule. It substitutes the argument with test data in the program.

  $(\lambda x.E)$ test-data $\Rightarrow$ E [test-data/x].

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

Figure 3 shows the syntax transformation of our program using the rules.

```
eval (fun e1 e2) = δ-rule (eval e1)(eval e2)
eval (apply e1 e2) = β-rule e1 e2
eval (lambda x e) = (lambda x e)
```

Figure 3: syntax transformation of our program

When applying β-rule, we perform normal order evaluation: e1 is evaluated before e2. If a program terminates, the order of evaluation won't make any difference; they should reach the same result. Unfortunately, not all programs terminate. Church-Rosser Theorem II says normal order evaluation is the most likely to terminate [Rosser, 1982]. We therefore prefer normal order evaluation than other evaluation order.

Recursion: Our treatment of recursion is very simple. We give a name to the program that we are growing; at any point within the parse tree it can use its own name just as if it were a function.

Unfortunately, there will always be the problem of infinite recursion. It is impossible to know at tree-generation time which recursive call will stop and which will not (c.f. the Halting Problem) We therefore use a restricted form of recursion, which limits the number of recursive calls to avoid infinite loops [Brave, 1996]. In our experiments, we limit the number of recursive calls by the length of the input list.During the evaluation of a program, a variable is increased every time a recursive called is made. When the limit is reached, the evaluator stops evaluating the program and aborts with a flag indicating that the program may not halt. This flag is used in the computation of the fitness value of the program.

Run-Time-Errors: The evaluator handles two kinds of run-time errors. Both errors are reflected in the fitness value of the program:

- Non-terminating recursion: When this error occurs, we flag an error and cause the program evaluation to terminate immediately.
- Taking the `HEAD` or `TAIL` of an empty list: When this error occurs, we return a default type value for the expected type and keep on evaluating the program. We do so because we believe that even trees with this type of error may still contain good genetic material. The only way to reuse these good genetic building block is to complete the evaluation and score the program accordingly. The default type values are listed in [Clack and Yu, 1997].

4.3 Evolvor

The evolvor performs two genetic operations:

Crossover: The system restricts crossover to be performed on application nodes only (these include full application and partial application nodes). This is to promote the recombining of large structures. Crossover at the leaf level is more akin to point mutation rather than true crossover. By reference to our 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.

When performing crossover, we first select a crossover node from one parent. The return type with the depth of the node is passed over to the second parent to select another crossover node. In the second parent tree, a crossover node will be selected according to two criteria:

- Its return type value must unify with the given return type value. There is one extra constraint when performing type unification on crossover node. This constraint prevents type-correct but syntax-incorrect (in particular function provided with wrong number of argument) programs to be generated. We will explain the constraint using an example in section 8.
- Its depth must be such that the new tree will satisfy the maximum tree depth parameter.

Mutation: Mutation is straight-forward: we first choose a mutation node in the tree randomly; 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 satisfies the maximum tree depth parameter. The new subtree then replaces the mutation node subtree.

Like crossover, mutation can be performed on either partial or full application nodes. Our creator is capable of creating subtrees whose root returns a function type.

5 Type System

Our PolyGP system needs a type system to check that invalid parse trees 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.

5.1 Type Syntax

Our abstract type syntax is given by:

$\sigma :: \tau$	*built-in type*
$\mid \upsilon$	*type variable*
$\mid \sigma_1 \text{ -> } \sigma_2$	*function type*
$\mid [\sigma_1]$	*list of elements all of type* σ_1
$\mid (\sigma_1 \text{ -> } \sigma_2)$	*bracketed function type*

$\tau ::$ `int | string | bool | generic`$_i$
$\upsilon ::$ `dummy`$_i$ `| temporary`$_i$

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 `[`α`] -> `α);
- Applications have a type given as follows:

  if `exp1` has type (σ_1 -> σ_2)
  and `exp2` has type σ_1
  then `(exp1 exp2)` has type σ_2
  else there is a type error.

Our type system also supports higher-order functions. Higher-order functions are indicated by the use of the bracketed function type. The brackets can also be used to indicate the use of a function as a return type (though this is not strictly necessary).

Our type system supports 3 kinds of type variables:

Generic Type Variables: The "generic" types are used when evolving polymorphic programs such as `MAP`, which has type `(G1->G2) -> [G1] -> [G2]` where `G1` and `G2` are generic type variables. While the program is being evolved the 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 those which express the 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). 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 5.3. Typically, the constraints imposed by the target type of the program being evolved 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 delay 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. One tricky situation involves the unification of a dummy type variable and a temporary type variable. In this case, the dummy type variable is first instantiated to a unique temporary type variable before unifying with the other temporary type variable. 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.

5.2 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. We now give a brief explanation of the unification algorithm.

The unification algorithm takes two types 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 flags an error.

- A substitution, θ, is a finite set (possible empty) of pairs of the form (X_i, t_i) where X_i is a type variable and t_i is a type value or a type variable. For example: θ = `{(a, int)}`.
- The result of applying a substitution to a type A, denoted by $A\theta$, is the type obtained by replacing every occurrence of X in A by t, for each pair (X, t) in θ. For example:
  α `->` α `{(`α`, int)}` = `int -> int`
- Two types A and B unify if there exists a substitution θ which makes the types identical $A\theta = B\theta$. For example, if A = `T1-> int` and B = `[string] -> T2`, A and B unify with θ = `{(T1, [string]),(T2, int)}`
- There may be more than one substitution which unifies two types. The "most general unifier" of two types A and B is a substitution θ that unifies A and B such that $A\theta$ is more general than any other common instance of A and B. For example, if $A\theta_1$ = `T1 -> int` and $A\theta_2$ = `int -> int`, then $A\theta_1$ is more general than $A\theta_2$ and θ_1 is the most general unifier.

5.3 Contextual Instantiation

Type expressions which contain several occurrences of the same type variable, like in α -> α, 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 value. This is done through the process of *contextual instantiation*: applying 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.

6 Implementation

In this section we give details of our implementation of the system. First we describe our genetic algorithms and then we discuss the language we used to implement the system.

6.1 Genetic Algorithms

We use steady-state replacement [Syswerda, 1989] to perform population updates. Initially, we create a population with a specified size. Within the population, every tree is unique. During evolution, we select two trees to perform crossover. The system ensures that the newly created tree is unique before putting it back into the same population pool to replace the tree with a lowest fitness score. The size of the population therefore remains constant. The advantage of steady-state replacement is that a tree with a good fitness score is immediately available as a parent for reproduction rather than having to wait until the next generation.

The scheme we use to select parents for reproduction is exponential fitness normalization [Cox, Davis and Qiu, 1991]. This means:

- We use rank selection instead of fitness-proportionate selection, and
- The probability of selecting the *n*-th best individual is `Parent-Scalar` times the probability of selecting the *(n-1)*-th best individual. The `Parent-Scalar` is a parameter provided by users.

6.2 Programming Language

The system is implemented in Haskell 1.4 using 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). When there is run-time error (such as recursion error) in a parse tree, not-yet-needed expressions would never be computed. The "purity" feature of Haskell, however, slows down our GP system because it does not support global storage for population pool. Instead, the population pool (implemented as a list) has to be passed around as an argument for updating during the GP run.

Haskell is also a typeful programming language [Hudak, Peterson and Fasel, 1997]. We have benefited from Haskell's rich type system in the following ways:

- Our expression-based language and type language are defined using "User-Defined Recursive Type". With recursive types, we can implement recursive functions that use the types. Figure 5 is the recursive function "applySub", which performs the contextual instantiation operation for a type expression.
- A user-defined type can declare to be a derived type of any existing type classes supported by the Haskell language. Haskell will *automatically* generate codes to perform the type classes associated operation for the user-defined type. Figure 4 is our TypeExp type declaration. TypeExp is an user-defined recursive type which is also derived from "Eq" and "Text" type classes. Haskell would generate "==" and "print" functions for our TypeExp type.

```
data TypeExp = IntNum |
               Boolean |
               Str |
               ListType TypeExp |
               Arrow TypeExp TypeExp |
               Brackets TypeExp |
               TempType String |
               DummyType String |
               GenType String |
               deriving (Eq, Text)
```

Figure 4: TypeExp type declaration

```
applySub :: Theta -> TypeExp -> TypeExp
applySub theta typeExp =
  case typeExp of {
  (TempType v) -> replaceVar v theta;
  (DummyType v) -> replaceDummy v theta;
  (ListType t) -> ListType (applySub theta t);
  (Arrow t1 t2) -> Arrow (applySub theta t1)
                         (applySub theta t2);
  (Brackets e) -> Brackets (applySub theta e);
  _ -> typeExp
  }
```

Figure 5: the applySub recursive function

7 The MAP Polymorphic Program

This section presents a worked example, the MAP polymorphic program, to demonstrate the operation of our system.

Problem Description: The `MAP` program takes two arguments, a function `F` and a list `L`, and returns the list obtained by applying `F` to each element of `L`.

Output Type: The output has generic type `[G2]`.

Arguments Type: The argument `F` has generic type `G1->G2` and the argument `L` has generic type `[G1]`.

Terminal Set:

```
T = {L:: [G1], NIL:: [α], F:: (G1->G2)}
```

Function Set:

```
F = {HEAD:: [α] -> α,
     IF-THEN-ELSE:: bool -> α -> α -> α,
     TAIL :: [α] -> [α],
     CONS :: α ->[α] -> [α],
```

```
NULL :: [α] -> bool,
F :: G1 -> G2,
MAP :: (G1->G2)-> [G1] -> [G2]}
```

Notice that polymorphic functions and terminals are normally expressed using *dummy type variables*, such as α. However, if they are program arguments/outputs, they use *generic type variables*, such as G1, to express polymorphism. During the program parse trees creation, *dummy type variables* are instantiated while *generic type variables* are never instantiated.

Maximum Tree Depth: 3.

Program Creation: Creator calls the type system to select a function whose return type unifies with the MAP program return type: [G2]. The IF-THEN-ELSE function is selected, α is instantiated to [G2] and the contextual instantiation process is applied to its arguments type:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG1}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ \mathrm{ARG2}^{[G2]})^{[G2]\to[G2]}\ \underline{\mathrm{ARG3}})^{[G2]}$$

The system first expands the right branch of the curried program, i.e. ARG3. The type system selects MAP whose return type [G2] unifies with the type of ARG3:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG1}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ \mathrm{ARG2}^{[G2]})^{[G2]\to[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{ARG4}^{(G1\to G2)})^{[G1]\to[G2]}\ \underline{\mathrm{ARG5}}^{[G1]})^{[G2]})^{[G2]}$$

The depth-first-right-first approach makes ARG5 the next node to be expanded. Since we have reached the maximum tree depth, we can only select terminals. L is selected by the type system:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG1}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ \mathrm{ARG2}^{[G2]})^{[G2]\to[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \underline{\mathrm{ARG4}}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}$$

The next node to be expanded is ARG4 whose type is a higher-order function type (G1->G2). We are in leaf level now. The only terminal that unifies with the expected type is F:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG1}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ \underline{\mathrm{ARG2}}^{[G2]})^{[G2]\to[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{F}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}$$

The next node to be expanded is ARG2. The type system selects IF-THEN-ELSE whose dummy type variable α is instantiated to [G2]:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG1}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ (((\mathrm{IF\text{-}THEN\text{-}ELSE}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG6}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ \mathrm{ARG7}^{[G2]})^{[G2]\to[G2]}\ \underline{\mathrm{ARG8}}^{[G2]})^{[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{F}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}$$

The next node to be expanded is ARG8 whose type is [G2]. We have again reached the maximum tree depth and have to select terminals. The only choice is NIL:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG1}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ (((\mathrm{IF\text{-}THEN\text{-}ELSE}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG6}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ \mathrm{ARG7}^{[G2]})^{[G2]\to[G2]}\ \mathrm{ARG8}^{[G2]})^{[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{F}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}\ \underline{\mathrm{ARG6}}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ \underline{\mathrm{ARG7}}^{[G2]})^{[G2]\to[G2]}\ \mathrm{NIL}^{[G2]})^{[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{F}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}$$

The next node to be expanded is ARG7 whose type is [G2]. We have again reached the maximum tree depth and have to select terminals. The only choice is NIL:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG1}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ (((\mathrm{IF\text{-}THEN\text{-}ELSE}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \underline{\mathrm{ARG6}}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ \mathrm{NIL}^{[G2]})^{[G2]\to[G2]}\ \mathrm{NIL}^{[G2]})^{[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{F}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}$$

The next node to be expanded is ARG6 whose type is bool. Unfortunately there is no terminal whose type unifies with bool. We have to back track to select another function to replace IF-THEN-ELSE. The system selects HEAD:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \mathrm{ARG1}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ (\mathrm{HEAD}^{[[G2]]\to[G2]}\ \underline{\mathrm{ARG9}}^{[[G2]]})^{[G2]})^{[G2]\to[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{F}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}$$

The next node to be expanded is ARG9 whose type is [[G1]]. The only terminal whose type unifies with the required type is NIL whose dummy type variable α is instantiated to [G2]:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ \underline{\mathrm{ARG1}}^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ (\mathrm{HEAD}^{[[G2]]\to[G2]}\ \mathrm{NIL}^{[[G2]]})^{[G2]})^{[G2]\to[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{F}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}$$

The next node to be expanded is ARG1 which has type bool. The only function whose return type unifies with the required type is NULL. Because there is no contextual dependency between the argument and return type, the dummy type variable α is instantiated to a *temporary type variable* T1 which is allowed to be bound to other type later:

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ (\mathrm{NULL}^{[T1]\to\mathrm{bool}}\ \underline{\mathrm{ARG10}}^{[T1]})^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ (\mathrm{HEAD}^{[[G2]]\to[G2]}\ \mathrm{NIL}^{[[G2]]})^{[G2]})^{[G2]\to[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{F}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}$$

The next and last node to be expanded is ARG10. We are in the leaf level and both L and NIL unify with the required type [T1]. The type system selects NIL. Note that *dummy type variable* is not allowed to perform unification algorithm with *temporary type variable*. The *dummy type variable* α is first instantiated to an unique *temporary type variable* T2. T2 then unifies with T1, i.e. T1 is bound to T2.

$$(((\mathrm{IF}^{\mathrm{bool}\to[G2]\to[G2]\to[G2]}\ (\mathrm{NULL}^{[T2]\to\mathrm{bool}}\ \mathrm{NIL}^{[T2]})^{\mathrm{bool}})^{[G2]\to[G2]\to[G2]}\ (\mathrm{HEAD}^{[[G2]]\to[G2]}\ \mathrm{NIL}^{[[G2]]})^{[G2]})^{[G2]\to[G2]}\ ((\mathrm{MAP}^{(G1\to G2)\to[G1]\to[G2]}\ \mathrm{F}^{(G1\to G2)})^{[G1]\to[G2]}\ \mathrm{L}^{[G1]})^{[G2]})^{[G2]}$$

The generated program is polymorphic since its arguments L and F can be of more than one type. The program also contains a temporary type variable T2. One important note is that after the completion of the program generation, neither of the type variables will ever be instantiated. As long as the

inputs L and F exhibit the constraints imposed by the generic type variables, i.e. L has the same type as the argument type of F, MAP will generate output that is the same type as the return type of the function F. Programs generated by the PolyGP system are guaranteed to have no run-time type errors.

Crossover: We use the parse tree generated in the previous section as the first parent:

```
(((IF^bool->[G2]->[G2]->[G2] (NULL^[T2]->bool NIL^[T2])^bool) ([G2]->[G2]->[G2]) (HEAD^[[G2]]->[G2] NIL^[[G2]])^[G2])^[G2]->[G2] ((MAP^(G1->G2)->[G1]->[G2] F^(G1->G2))^[G1]->[G2] L^[G1])^[G2])^[G2]
```

The second parent is given below. We made the program with return type T1 to help to demonstrate our crossover operation. Note the double-underlined type in both programs indicates the partial application node where the crossover operation occurs.

```
(((IF^bool->T1->T1->T1 (NULL^[G1]->bool L^[G1])^bool) (T1->T1->T1) (HEAD^[T1]->T1 NIL^[T1])^T1)^T1->T1 (HEAD^[T1]->T1 NIL^[T1])^T1)^T1
```

The following program is generated by the crossover operation. Note that T1 is instantiated to [G2].

```
(((IF^bool->[G2]->[G2]->[G2] (NULL^[G1]->bool L^[G1])^bool)^[G2]->[G2]->[G2] (HEAD^[[G2]]->[G2] NIL^[[G2]])^[G2])^[G2]->[G2] ((MAP^(G1->G2)->[G1]->[G2] F^(G1->G2))^[G1]->[G2] L^[G1])^[G2])^[G2]
```

When performing crossover, we don't permit *temporary type variables* to unify with function types. This is to prevent the generation of syntax incorrect programs. For example, if we perform crossover on nodes whose types are in strike-through style (T1 is instantiated to [G1]->[G2]), a type-correct yet syntax incorrect program will be generated:

```
(((IF^bool->[G2]->[G2]->[G2] (NULL^[T2]->bool NIL^[T2])^bool)^[G2]->[G2]->[G2] (HEAD^[[G2]]->[G2] NIL^[[G2]])^[G2])^[G2]->[G2] ((HEAD^[(G1->G2)]->(G1->G2) NIL^(G1->G2))^[G1]->[G2] L^[G1])^[G2])^[G2]
```

The underlined expression is syntactically incorrect. HEAD should only take one argument but is provided with two. Our evaluator will report a run-time error.

Mutation: Again, we use the same parse tree to demonstrate our mutation operation:

```
(((IF^bool->[G2]->[G2]->[G2] (NULL^[T2]->bool NIL^[T2])^bool)^[G2]->[G2]->[G2] (HEAD^[[G2]]->[G2] NIL^[[G2]])^[G2])^[G2]->[G2] ((MAP^(G1->G2)->[G1]->[G2] F^(G1->G2))^[G1]->[G2] L^[G1])^[G2])^[G2]
```

The creator generates a subtree whose root has the double-lined type [G2]->[G2]:

```
(CONS^G2->[G2]->[G2] (HEAD^[G2]->G2 NIL^[G2])^G2)^[G2]->[G2]
```

The new tree is used to replace the mutation node subtree.

```
(((CONS^G2->[G2]->[G2] (HEAD^[G2]->G2 NIL^[G2])^G2)^[G2]->[G2] ((MAP^(G1->G2)->[G1]->[G2] F^(G1->G2))^[G1]->[G2] L^[G1])^[G2])^[G2]
```

8 Conclusion

We have presented a PolyGP system which utilizes type information to generate general 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. Our 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.

Acknowledgments

The authors would like to thank Robin Hirsch and Jonny Farringdon for their valuable comments on the paper. We also thank Benjamin Goldberg for his implementation of the unification algorithm in SML, which our type system is based on.

References

Brave, S. (1996). Evolving recursive programs for tree search. *Advances in Genetic Programming II*, P.J.Angeline and K.E.Kinnear, Jr.(eds.), MIT Press, Cambridge, MA, pp.203-220.

Cardelli, L., and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism, *Computing Surveys*, Vol. 17:4, pp.471-522.

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*, P.J. Angeline, R. Reynolds, J. McDonnell and R. Eberhart (eds.), Springer-Verlag, Berlin, pp.87-100.

Cox, A.L. Jr., Davis, L. and Qiu, Y. (1991). Dynamic anticipatory routing in circuit-switched telecommunications networks. *Handbook of Genetic Algorithms*. L. Davis (ed.), Van Nostrand Reinhold, NY, pp.124-143.

Haynes, T.D., Wainwright, R., Sen, S., and Schoenefeld, D. (1995), Strongly typed genetic programming in evolving cooperation strategies. In *Proceedings of the Sixth International Conference on Genetic Algorithms,* Eshelman, L. (ed), Morgan Kaufmann Publishers, Inc. pp. 271-278.

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

Hudak, P., Peterson, J., and Fasel, J.H. (1997), A gentle introduction to Haskell. version 1.4. http://haskell.org/tutoral/index.html.

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

Mitchell, T., Utgoff, P., and Banerji, R. (1984). Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning: An Artificial Intelligence Approach*, chapter 6, pp. 163-190. Springer-Verlag.

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

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

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

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

Yu, T., and Clack, C. (1997). Hierarchical learning using λ abstractions in polymorphic genetic programming. In preparation.