Polymorphism and Genetic Programming

Tina Yu
Chevron Information Technology Company
6001 Bollinger Canyon Road
San Ramon, CA 94583 U.S.A.
tiyu@chevron.com
http://www.addr.com/~tinayu

Abstract. Types have been introduced to Genetic Programming (GP) by researchers with different motivation. We present the concept of types in GP and introduce a typed GP system, PolyGP, that supports polymorphism through the use of three different kinds of type variable. We demonstrate the usefulness of this kind of polymorphism in GP by evolving two polymorphic programs ($nth$ and $map$) using the system. Based on the analysis of a series of experimental results, we conclude that this implementation of polymorphism is effective in assisting GP evolutionary search to generate these two programs. PolyGP may enhance the applicability of GP to a new class of problems that are difficult for other polymorphic GP systems to solve.

1 Introduction

Types have been studied and implemented in many modern programming languages, e.g. Haskell, Ada and C++. The ability to support multiple types and to provide type checking for programs has made these languages more expressive and the execution of programs more efficient. Types are counterparts of programming languages.

Genetic Programming (GP) [5] automatically generates computer programs to solve specified problems. It does this by searching through a space of all possible programs for one that is nearly optimal in its ability to solve the given problem. The GP search algorithm is based on the model of natural evolution. In particular, three basic operations (selection, alteration and fitness evaluation) are applied iteratively to a population of programs. Similar to nature where the fittest would survive, the program which best solves the problem would emerge at the end of the GP search process.

The road from untyped to typed GP is led by two goals. Firstly, to enhance the applicability of GP by removing the “closure” requirement. Secondly, to assist GP searching for problem solutions using type information. Koza made the first attempt to introduce types to GP by extending GP with “constrained syntactic structures” when he realized that not all problems have solutions which can be represented in ways that satisfy the closure requirement [5]. Supporters of this argument [9,4] believe that it is important for GP to be able to handle multiple types and advocate excluding type-incorrect programs from the search space to speed up GP’s evolutionary process. Another route to promote the use of types in GP is based on the idea that types provide inductive bias to direct GP search. For example, Wong and Leung included type information in a logic grammar to bias the selection of genetic operation
location during program evolution [10]. McPhee et al. also showed that a program representation which incorporates function type can bias genetic operations and benefit GP search on some problems [8].

These two paths, although with different purposes, are actually interrelated. Types exist in the real world naturally [2]. By allowing problems to be represented in their natural ways, an inductive bias is established which selects solutions based on criteria that reflect experience with similar problems.

In this paper, we first present the concept of types in GP by defining and differentiating untyped, dynamically typed and strongly typed GP. Next, the two different implementations of strongly typed GP, monomorphic GP and polymorphic GP, are discussed. We introduce our polymorphic GP system (PolyGP) and compare it with other polymorphic GP systems. This system is then used to evolve two polymorphic programs, nth and map, which represent a class of problems that we believe to be very difficult for other polymorphic GP system to evolve. We analyze a series of experimental results to evaluate the effectiveness of PolyGP polymorphism in assisting GP evolutionary search. Finally, we give our conclusions.

2 Types in Genetic Programming

In its traditional style, GP is not capable of distinguishing different types: the term untyped is used to refer to such a system. In the case when the programs manipulate multiple types and contain functions designed to operate on particular types, untyped GP leads to an unnecessarily large search space, which contains both type-correct and type-incorrect programs. To enforce type constraints, two approaches can be used: dynamically typed GP and strongly typed GP. In dynamically typed GP, type checking is performed at program evaluation time. In contrast, strongly typed GP performs type checking at program generation time. The computation effort required for these two different type checking methods is implementation dependent. It is not valid to claim that dynamic typing is more efficient than strong typing; nor vice versa. The details of these two type checking approaches are discussed in the following sections.

3 Dynamically Typed GP

Dynamic typing performs type checking when a program is evaluated to determine whether it is a solution, i.e. type-incorrect programs are not allowed to exist in the solution space. However, the search space may contain both type-correct and type-incorrect programs. Consequently, the size of the search space is bigger than that of the solution space (Figure 1).

![Figure 1: Search Versus Solution Space in Dynamically Typed GP.](image)