Polytypic Genetic Programming

Jerry Swan[1], Krzysztof Krawiec[2], Neil Ghani[3]

[1] Computer Science, University of York, Deramore Lane, York, YO10 5GH, UK
jerry.swan@york.ac.uk
[2] Institute of Computing Science, Poznan University of Technology
60-965 Poznań, Poland
krzysztof.krawiec@cs.put.poznan.pl
[3] Department of Computer and Information Sciences University of Strathclyde
Glasgow G1 1XH, Scotland
ng@cis.strath.ac.uk

Abstract. Program synthesis via heuristic search often requires a great deal of 'boilerplate' code to adapt program APIs to the search mechanism. In addition, the majority of existing approaches are not type-safe: i.e. they can fail at runtime because the search mechanisms lack the strict type information often available to the compiler. In this article, we describe Polytope, a Scala framework that uses *polytypic programming*, a relatively recent advance in program abstraction. Polytope requires a minimum of boilerplate code and supports a form of strong-typing in which type rules are automatically enforced by the compiler, even for search operations such as mutation which are applied at runtime. By operating directly on language-native expressions, it provides an embeddable optimization procedure for existing code. We give a tutorial example of the specific polytypic approach we adopt and compare both runtime efficiency and required lines of code against the well-known EpochX GP framework, showing comparable performance in the former and the complete elimination of boilerplate for the latter.
Keywords: polytypic programming, datatype generic programming, genetic programming, functional programming, Scala.

1 Introduction

For over 20 years, Genetic Programming (GP) has been applied to a wide variety of program induction tasks, yielding an impressive list of (often human-competitive) results [1]. Most such endeavours require the domain-specific code, expressed in some *host language* (e.g. Java™ in the case of the popular ECJ GP framework[2]), to be *manipulable* by some search mechanism (e.g. an evolutionary algorithm in the case of GP). Technically, this is often achieved by mapping the host language API of interest to individual functions in the GP instruction set. For instance, in order to manipulate programs which use the API of a computer vision library (e.g., OpenCV [3], as in the GP/GI work of [4,5,6]), one could provide adaptor code for each API call of interest. This task is nowadays greatly facilitated by availability of a rich choice of domain-agnostic

software packages (ECJ [2], EpochX [7] and DEAP [8] to name a few), which offer a extensive support for the representations and operators of GP.

However, tailoring a domain-agnostic search framework to a problem/domain comes at price: the GP instructions in question have to be implemented according to the contracts mandated by a given framework. In the prevailing object-oriented paradigm, this requires extending certain framework classes (representing programs, instructions, data etc). In common domains (e.g. numeric and Boolean regression), this can be achieved at relatively low human effort. Otherwise, one is forced to realize the instructions as 'wrappers' that delegate the actual execution to the host language, which results in substantial amounts of boilerplate code, i.e. code which does little other than act as adaptor, but which is sufficiently different on a per-API basis that conventional automation approaches (e.g. C++ style macros) are insufficent. For example, Listing 1 shows some of the EpochX code required for Boolean expressions. Moreover, producing and maintaining such code may become particularly labour-intensive if a non-trivial grammar and/or type system is required, which becomes a necessity when approaching real-world program synthesis problems.

The majority of Genetic Improvement (GI) work has been in an *offline* setting, i.e. taking source- or object- code as input and producing transformed code for subsequent compilation/execution. However, the desire for systems which can respond adaptively to dynamic environments [9] has motivated a trend towards online approaches. Previous work on dynamic GI frameworks include Gen-O-Fix [10], Templar [11] and ECSELR [12]. Gen-O-Fix operates at runtime via reflection on the abstract syntax trees generated by the Scala compiler, which Templar is a wrapper for EpochX GP which makes it easy to generate the multiple variation points of a user-specified algorithm skeleton [13]. In the spirit of 'Embedded Dynamic Improvement' [14], both Gen-O-Fix and Templar can be configured via an embeddable callback mechanism which allows the training phase to take place either once, periodically or asynchronously. ECSELR extends the 'Java Agent' monitoring API to apply evolutionary operators to state snapshots of the JVM.

Despite recent work in GI (e.g. [4]), relatively little has been done to manipulate domain-specific functionality (as expressed in the host language) without the additional effort of re-presenting that knowledge in a form acceptable to the search framework. In this article, we describe Polytope, an embeddable Scala framework *operating directly on the host language*. The main features of Polytope are:

1. It allows the creation of manipulable (optimizable) expressions and programs with a minimum of boilerplate code.
2. It provides a strongly typed approach to GP [15], with typing rules automatically enforced *by the Scala compiler*. This is in direct contrast to other approaches, in which types must be cast/queried at runtime (see Listing 1).
3. Operates directly on language constructs, thereby easing the path to wider adoption of SBSE techniques by mainstream software developers.

The reader desiring an advance look at the resulting simplicity for the practitioner is referred to Listing 8. Crucially, the Boolean expression presented there is expressed in the host language and requires no knowledge that it is to be manipulated via Polytope, nor does it depend on Polytope in any sense typically considered in software engineering. Hence, it could have been equally well taken verbatim from an existing Scala library, without the intent of actually being manipulated by GP, GI or indeed any other synthesis approach. All the domain-specific knowledge (as implied by the grammatical structure of possible programs — see next section) that is necessary for forthcoming synthesis or improvement is automatically derived by building upon mechanisms available in standard Scala.

We proceed in Section 2 with the theoretical underpinnings of Polytope, then experimentally compare its performance with a popular GP framework in Section 3, and discuss consequences and prospects in Sections 4 and 5.

2 Background

GP can be considered to be constrained by the production rules of a user-specified grammar. For example, here is an EBNF for a grammar representing Boolean expressions:

```
<BoolEx> ::= <Const> | <Var> | <AndEx>
              | <OrEx> | <XorEx> | <NotEx> | <IfEx>
<Const>  ::= <True> |<False>
<Var>    ::= Var <Varname>
<Varname>  ::= string
<AndEx>  ::= And <BoolEx> <BoolEx>
<OrEx>   ::= Or <BoolEx> <BoolEx>
<XorEx>  ::= Xor <BoolEx> <BoolEx>
<NotEx>  ::= Not <BoolEx>
<IfEx>   ::= If <BoolEx> <BoolEx> <BoolEx>
```

In the expression trees are manipulated via traditional GP, the grammar is implicit and (as can be seen in Listing 1) describing the production rules for each entity in the grammar can require a lot of boilerplate code. In the case of Grammatical Evolution [16], the grammar rules are explicit, though they typically have some interpreted representation (e.g. as strings) which cannot be checked for validity at compile-time. In either case, the required type system must be implemented explicitly within a GP software framework, which often involves additional classes for representing particular types (cf. GPType class in ECJ [2]). This code often duplicates in part the type system of the underlying programming language, but with the attendant need for runtime type checking (again, Listing 1).

In contrast, mainstream programming languages have progressively increased in their ability to abstract across datatypes. Starting in the 1960s with subtype polymorphism [17], it became possible to use inheritance to express common

```
class AndNode extends Node {
  @Override
  public Object eval() {
    if(getChildren().length == 2) {
        Boolean b1 = (Boolean)getChild(0).eval();
        Boolean b2 = (Boolean)getChild(1).eval();
        return b1 && b2;
    }
    else throw new IllegalStateException();
  }

  @Override
  public Class<?>
  getReturnType(Class<?>... inputs) {
    return inputs.length == 2 ? Boolean.class : null;
  }
}

class OrNode extends Node {
  @Override
  public Object eval() {
    if(getChildren().length == 2) {
        Boolean b1 = (Boolean)getChild(0).eval();
        Boolean b2 = (Boolean)getChild(1).eval();
        return b1 || b2;
    }
    else throw new IllegalStateException();
  }

  @Override
  public Class<?>
  getReturnType(Class<?>... inputs) {
    return inputs.length == 2 ? Boolean.class : null;
  }
}

class ConstNode extends Node {
  public ConstNode(boolean value) { this.value = value; }
  @Override
  public Object eval() {
     if(getChildren().isEmpty)
        return value;
    else
        throw new IllegalStateException();
  }

  @Override
  public Class<?>
  getReturnType(Class<?>... inputs) {
    return inputs.isEmpty() ? Boolean.class : null;
  }
}
// Similarly for Var, Not, Xor and If.
```

Listing 1. Some of the boilerplate code required for Boolean expressions in EpochX

```
sealed trait Nat
case object Zero extends Nat
case class Succ(n: Nat) extends Nat
// Example use:
val three: Nat = Succ(Succ(Succ(Zero)))
```

Listing 2. Algebraic datatype for Peano arithmetic in Scala

behaviours via the abstraction of a shared superclass. In the 1970s, parametric polymorphism was introduced [18], allowing the expression of functions and datatypes that do not require knowledge of the *type* of their arguments (e.g. a function to determine the length of a list is independent of the type of the elements it contains). More recently, there have been a number of developments in *polytypic programming*, whereby the specific *structure* of a datatype is abstracted away by one of a number of alternative generic mechanisms. These alternative approaches have a variety of names, e.g. *type-parametric programming* or *structural-/shape-/intensional- polymorphism*. In particular, the Haskell community tends to use the term *data-generic programming*, which should not be confused with the more populist notion of 'generic programming', since the latter refers only to parametric polymorphism.

Unlike parametric polymorphism whose strength derives from type agnosticism (e.g. as with the list length example above), polytypic programming captures a wide class of algorithms which are defined by interrogating the structure of the data type, e.g. so as to operate inductively upon it. Over the last 10 years or so, the functional programming community has shown particular interest in polytypic programming, originating a range of alternative approaches [19,20,21,22,23]. Algorithms which have been defined polytypically include equality tests, parsers and pretty printers.

3 The Polytope framework

3.1 Polytypic Programming in Scala

Languages such as Scala and Haskell achieve considerable expressive power via their support for *Algebraic Data Types* (ADTs) [4], where the creation and manipulation of ADT expressions is ubiquitous programming practice. As shown in the example in Listing 2, ADT expressions are built-up via inductive construction, which, amongst other benefits, allows them to be conveniently manipulated via sophisticated statically-checked pattern matching. POLYTOPE combines polytypic programming with an embedded search procedure that makes it possible to directly manipulate expressions of the host language (such as the last line of Listing 2) by an arbitrary combinatorial search mechanism, including GP. This in turn allows replacing the existing expressions with optimized equivalents (GI), or even synthesizing new expressions according to some specification or examples (GP). The polytypic approach we use here is essentially the Scala variant

[4] Not to be confused with the weaker notion of *abstract data types*

```
sealed trait BoolEx {
    def eval: Boolean
}

case class And(x: BoolEx, y: BoolEx) extends BoolEx {
    override def eval: Boolean = x.eval && y.eval
}

case class Or(x: BoolEx, y: BoolEx) extends BoolEx {
    override def eval: Boolean = x.eval || y.eval
}

case class Xor(x: BoolEx, y: BoolEx) extends BoolEx {
    override def eval: Boolean = x.eval ^^ y.eval
}

case class Not(x: BoolEx) extends BoolEx {
    override def eval: Boolean = !x.eval
}

case class If(cond: BoolEx, then: BoolEx, els: BoolEx) extends BoolEx {
    override def eval: Boolean = if cond.eval then.eval else els.eval
}

case class Const(override val eval: Boolean) extends BoolEx

case class Var(name: String) extends BoolEx{
    override def eval: Boolean = symbolTable.lookupVar(name)
}
```

Listing 3. Scala algebraic datatype for Boolean expressions

of Hinze's 'Generics for the Masses' [24] given by Oliveira and Gibbons [21], in which ADTs are converted to/from a universal representation.

In the following, we give a tutorial introduction to polytypic programming with a simple example, namely the polytypic calculation of size for a program tree[5]. This example is relevant to polytypic GP, as determining program size is an important part of GP workflow, allowing (for example) size-related feasibility checking. Other necessary functionality for GP, in particular mutation, is realized in a directly analogous fashion.

Boolean expressions can be represented in Scala by the ADT in Listing 3, which in the following is our example host language for either synthesis (GP) or modification (GI) of programs. We define literals of 'atomic' types such as int, char, double etc to have a size of 1. Given these atomic building blocks, the polytypic approach allows us to inductively define size independantly of any specific

[5] We focus on tree-based GP in this paper.

```
trait Size[T] {
  def size(t: T): Int
}

object Size {
  def atomicSize[T] = new Size[T] {
    def size(t: T): Int = 1
  }

  implicit def intSize: Size[Int] = atomicSize
  implicit def booleanSize: Size[Boolean] = atomicSize
  implicit def charSize: Size[Char] = atomicSize
  // ... short, long, float etc
  implicit def doubleSize: Size[Double] = atomicSize

  // syntactic sugar:
  def size[T](x: T)(implicit ev: Size[T]) = ev.size(x)
}
```

Listing 4. Size typeclass and specializations for atomic types

ADT, so that the compiler can generate code for e.g. both size(Not(Const(true))) and size(Succ(Succ(Succ(Zero)))), yielding 3 and 4 respectively.

Since we wish to add this functionality in a non-intrusive manner, i.e. without requiring any change to the ADT we wish to operate on, we adopt the technique of *typeclasses*, well-known to the functional programming community. First developed in Haskell, this approach allows the *post-hoc* addition of functionality to any datatype. The essence of the approach is to provide a trait (for purposes of this article, equivalent to a Java interface) which defines the required methods, together with specialized subclasses for all types of interest.

Listing 4 shows the Size typeclass, together with specializations for atomic types. To make use of this functionality of Polytope, one uses the statement import polytope.Size._ (Listing 8). In result of this, the functions defined in the Size object are brought into scope, and automatic promotion from some atomic type A to the corresponding imported specialization of $Size[A]$ is made possible via the use of the *implicit* keyword.

For this mechanism to be fully operational, apart from the specializations of Size for atomic types in Listing 4, it is also necessary to provide specializations for ADTs. Listing 5 shows how this *could* be done manually for the first few subclasses of Ex. In Polytope, we achieve this automatically, and not just for Ex, but for any ADT. Confronting Listing 5 with Listing 3 reveals that there is a common pattern which is driven by the *shape* of the subclass constructor. Indeed, it is the ability to 'abstract over shape' that characterizes polytypic programming. In the following section, we explain how we employ this mechanism to automate creation of such specializations and avoid manually writing such boilerplate code as Listing 5.

```
implicit def constSize(implicit ev: Size[Boolean]) =
  new Size[Const] {
    def size(x: Const): Int = 1 + ev.size(x.value)
  }

implicit def andSize(implicit ev: Size[BoolEx]) =
  new Size[And] {
    def size(x: And): Int = 1 + ev.size(x.a) + ev.size(x.b)
  }

implicit def orSize(implicit ev: Size[BoolEx]) =
  new Size[Or] {
    def size(x: Or): Int = 1 + ev.size(x.a) + ev.size(x.b)
  }

// Similarly for other subclasses of BoolEx...
```

Listing 5. Manual specializations of Size for some subclasses of Ex. The equivalent functionality is achieved automatically in POLYTOPE.

3.2 Product and Coproduct Types

Automatic specialization of ADT like the one exemplified in Listing 5 requires generic mechanisms for the decomposition, transformation and reassembly of ADTs. It turns out that it is possible to provide the remaining required specializations of Size (and other operations of interest for GP) for all ADTs in terms of a generic 'sum of products' representation [21]. This requires consideration of the elementary building blocks of ADTs, viz. *products* and *coproducts*[6]. The conversion of an ADT to and from this representation is described extensively by Hinze [24] and is beyond the scope of this article, but fortunately the Scala library Shapeless [25,26] provides complete support for this and a variety of other polytypic methods (e.g. [20]).

Products will already be familiar in the guise of tuples — the type of a tuple is the 'Cartesian product' of the types it contains. The Shapeless product type is HList, a heterogeneous list with compile-time knowledge of the different types of *each of its elements*. It is actually more general than a tuple, in that it supports an 'append' constructor '::'. Thus, a HList(2.3,"hello") would have type Int :: String :: HNil, where HNil represents the type-level analog of the well-known use of Nil as a list terminator. If a Double is appended, the resulting type would be Int :: String :: Double:: HNil. As seen in the above listings, an ADT consists of a collection of subclasses implementing a given trait. Each subclass has zero or more attributes and can therefore be generically represented as a HList of these attributes.

[6] The term 'coproduct' represents a generalized notion of 'sum' inherited from Category Theory.

```
implicit val productBase = new Size[HNil] {
  def size(x: HNil): Int = 0
}

implicit def productInductionStep[H, T <: HList](
  implicit h: Size[H], t: Size[T]) =
    new Size[H :: T] {
      def size(x: H :: T) = {
        val hd = h.size(x.head)
        val tl = t.size(x.tail)
        hd + tl
      }
    }

implicit val coproductBase = new Size[CNil] {
  def size(x: CNil): Int = 0
}

implicit def coproductInductionStep[H, T <: Coproduct](
  implicit h: Size[H], t: Size[T]) =
    new Size[H :+: T] {
      def size(x: H :+: T): Int = x match {
        case Inl(l) => h.size(l)
        case Inr(r) => t.size(r)
      }
    }
```

Listing 6. Generic Size specialization for product and coproduct types

Regarding coproducts, each subclass in an ADT can be considered to represent a specific choice of construction step. They can therefore be represented by a disjoint union of subclass types. The canonical example of disjoint union in Scala or Haskell is the type Either[A,B], which contains an object known at compile-time to be of type A or else of type B. The corresponding 'shapeless' coproduct type for types A and B is denoted by A :+: B[7]. Hence the ADT Nat of Listing 2 can be generically represented as the type Zero :+: Succ :+: CNil, with CNil being the coproduct equivalent of HNil.

Specialization for generic product and coproduct types is defined inductively, starting with the base case, as represented by the types HNil and CNil respectively. The top two functions in Listing 6 show how this is done for product types, and the bottom two functions for coproduct types. The induction step is simplified via a recursive nesting technique: as is well-known, all n-tuples can be represented by recursive nesting of pairs, e.g. the triple (a, b, c) can be represented as $(a, (b, c))$. For purposes of building specializations one inductive step at a time, the tails of product and coproduct types are similarly nested. De-

[7] Type constructors in Scala can be infix and composed of non-alphabetic characters.

```
trait SubtreeMutate[T] {
  def mutate(t: T, index: Int): Option[T] =
    mutateImpl(t, index) match {
      case Left(t) => Some(t)
      case Right(newIndex) => None
    }

  protected def mutateImpl(t: T, index: Int): Either[T,Int] = ...
}

def mutate[T](x: T, rng: Random)(
  implicit m: SubtreeMutate[T], sz: Size[T]): T =
    ev.mutate(x,rng.nextInt(sz.size(x))).getOrElse(x)
}

// client code:
import Size._
import SubtreeMutate._

val ex = Not(Xor(Var("a"),Const(false)))
val mutated = mutate(ex)
```

Listing 7. Mutation typeclass and client code

termining the specialization for the nested tail T of the HList is dispatched to some other specialization of Size via the call to t.size(x.tail). Specialization for coproducts relies on analogous dispatching, where Inl and Inr denote left and right type-projections respectively, i.e. Inl(H :+: T) yields H, Inr(H :+: T) yields T.

The universal product and coproduct specializations in Listing 6, together with the support provided by Shapeless for conversion to/from this generic 'sum of products representation' [21] is all that is required to allow the compiler to synthesize code for size(x) for any ADT built up from the atomic specializations, automating so the functionality that would have to be otherwise implemented manually (Listing 5), for any host language expressed in standard Scala, including the example in 2, the Boolean domain in 3, and most of other common domains.

3.3 Initialization and mutation

Polytope employs the principles of polytypic programming in the design of all operators necessary to perform program synthesis or improvement, thereby allowing manipulation of arbitrary ADTs. In the current version, programs are stochastically initialized using the well-known 'full' method [27] and subtree-replacing mutation (a randomly selected subtree in a program is replaced by a random 'full' tree). The generic definitions for tree initialization and mutating a subtree follow the same general pattern as the Size example. As can be seen

Table 1. Parameters common to all Mux-6 experiments

Parameter	Value
population-size	1,000
max-generations	100
max-initial-tree-depth	5
tree-initialization-method	Full
mutation-method	Subtree

in Listing 7 (which gives the SubtreeMutate typeclass and an example of the corresponding client code), the actual mutation is performed in the method mutateImpl. The implementation of this method is slightly more complex than the Size example, since it requires additional book-keeping to keep track of the node indexing. This is represented by the Either[T,Int] return type, in which the Int value represents the index of the node for subsequent consideration. The corresponding overridden versions for atomic, product and coproduct types are too lengthy for this article, but are implemented analogously. Similar remarks apply to the initialization operator. As in the case of Size, both mutation and initialization work for any domain-specific host language expressible in Scala.

3.4 Comparison of Lines of Code

For the Boolean domain considered here, the total required by EpochX 1.4.1 is 301 lines of code (LOC) (specifically that for the classes AndFunction,OrFunction, NotFunction,XorFunction,ImpliesFunction in the org.epochx.epox.bool package). We discount the EpochX code required for Const since it is provided by built-in support for ephemeral random constants. In contrast, POLYTOPE can operate directly on the 20 LOC given in the classes of Listing 3. However, the important thing to note is that the code of Listing 3 will in general be some arbitrarily complex API that has already been implemented and that we wish to manipulate via search.

4 Experiments

With POLYTOPE's generic initialization and mutation operators, we can apply search routines to obtain an instance of any ADT, optimized to some user-specified criterion. To this end, POLYTOPE provides an implementation of the well-known Evolution Strategies (ES) metaheuristic [28], specifically, Algorithms 18 and 19 from Luke [29].

POLYTOPE can either be applied to optimize existing code (i.e. an ADT expression) or else can synthesize an ADT from scratch. Listing 8 shows the client code required to obtain an optimized expression for Mux6, the well-known 6-input multiplexer problem [27], for both *ex-nihilo* synthesis (GP-style) and improvement of existing code (GI-style). In contrast to the boilerplate of Listing 1, the only client responsibility is the implementation of the fitness function (here, the normalized sum of the zero/one errors on all possible 2^6 fitness cases).

To determine the performance relative to a traditional GP implementation, we compared our ES approach against EpochX on Mux6 over 30 runs with

```
// client code
def mux6Fitness(x:BoolEx): Double = // error of x.eval on fitness cases...

def main(args: Array[String]) = {

  // bring implicit specializations into scope
  import polytope.Size._
  import polytope.FullInitializer._
  import polytope.SubtreeMutate._

  // 1. ex−nihilo synthesis
  val opt1 = polytope.optimize(mux6Fitness)
  println( opt1, opt1.eval )

  // 2. Improvement of some existing expression
  val ex = If(Or(Var("x"),False),
    And(Var("y"),Var("z")),
    Or(Not(Var("x")),True))

  val opt2 = polytope.optimize(mux6Fitness,ex)
  println( opt2, opt2.eval )
}
```

Listing 8. Client code for Mux6 problem. Note the 'last-minute' import of POLYTOPE.

common parameters as given in Table 1. We compare against two variants of EpochX: EpochX-1 uses EpochX 'out of the box', i.e. with default parameters[8] (i.e. subtree crossover with probability 0.9, elitism count=10, max tree depth=17). EpochX-2 is intended to provide a more 'like for like' comparison with the current implementation of POLYTOPE, and therefore has no crossover and no upper bound on max-tree-depth. Although a lack of crossover is somewhat unusual in GP (Cartesian GP being a notable exception [30]) it is not so common in GI (e.g. [31]). For the ES-specific parameters, we use a $(\lambda + \mu)$-ES we take λ to be population size and $\mu = \lambda/5$. A run is terminated once a correct program is found or 100 generations elapse, whichever comes first.

Experiments were run on a Windows 10 desktop PC with 8GB of RAM and an Intel Core i5-3570 CPU @ 3.40GHz. Table 2 shows the results of the experiments, giving averaged normalized fitness, execution time in seconds, number of generations at termination, processing time per individual (elapsed time divided by the number of generations), and 0/1 success rate (defined as '1 for the optimum output, else 0'), accompanied with 0.95-confidence intervals. The rates of convergence to the optimum make it clear that the absence of crossover is detrimental to solution quality. Comparing POLYTOPE with the EpochX-2 setup, the performance of the former can be explained in part by the fact that it lacks a bloat control method [32], which fails to impose selection pressure against large

[8] http://www.epochx.org/javadoc/1.4/

Table 2. Results of Mux-6 experiment

Algorithm	Fitness	Time (s)	Generations	Time per individual (ms)	0/1 Success rate
EpochX-1	0.00 ± 0.00	2.26 ± 0.95	17.10 ± 6.71	7.14 ± 1.17	100%
EpochX-2	0.06 ± 0.11	3.60 ± 2.01	62.40 ± 31.63	17.49 ± 4.74	7.3%
POLYTOPE	0.25 ± 0.14	7.26 ± 1.96	92.03 ± 18.77	13.29 ± 1.85	6.7%

expressions and leads to trees which take longer to evaluate. However, the end-of-run fitness of POLYTOPE is not statistically significantly worse[9] than that of EpochX-2, and the 'zero or one' success rate is comparable.

Concerning the time for processing a program, POLYTOPE performs slightly better than the 'like for like' comparator EpochX-2, which might be explained by the amount of compile-time support afforded by our chosen polytypic approach. Since there are no theoretical obstacles to adding polytypic equivalents for crossover and bloat control to POLYTOPE, it would then be expected to behave comparably to (or even slightly better than) the 'out of the box' version of EpochX.

5 Discussion and Conclusion

We have described how polytypic programming (specifically Oliveira and Gibbons [21] variant of Hinze's 'Generics for the Masses' [24]) can be used to provide initialization and mutation operators for arbitrary datatypes, and have implemented POLYTOPE, a Scala framework which uses embedded optimization to perform synthesis and improvement of Scala code with minimal end-user effort.

Using POLYTOPE as a GP system frees the end-user from having to write the significant amounts of instruction-set specific code that is necessary when using most popular GP frameworks (Listing 1). Although previous work [33,10] has used runtime reflection as a means of reducing this burden, we describe how this can be via compile-time techniques. We explain how methods from polytypic programming can achieve this via the automatic and non-intrusive derivation of the grammatical structure of datatypes.

In addition to manipulating existing datatypes, POLYTOPE resembles GP in that it also supports *ex-nihilo* synthesis of expressions involving these datatypes. This is in contrast to some other GI frameworks [31,34] that manipulate programs by 'plastic surgery' (i.e. moving around pre-existing expressions modulo variable re-naming). With the historical emphasis on GI being 'offline, top-down', POLYTOPE can therefore be considered to occupy an intermediate position between traditional notions of GP and GI.

The current version of POLYTOPE lacks both crossover and any built-in mechanism for bloat control. The experiments in Section 4 show that that both are desirable. There is no intrinsic technical obstacle to the implementation of either and they are suitable subjects for further work. Regarding bloat-control,

[9] as determined by the nonparametric Wilcoxon Signed Rank test

irrespective of the provision of a general mechanism for this, reducing expressions to some minimal-size form via domain-specific rewrite rules [35] can be implemented very naturally on ADTs using pattern-matching. Nevertheless, even without these extensions, we anticipate two distinguishing uses of POLYTOPE in its current form: as a 'rapid prototyping tool' for GP, in which development time is more of an issue than raw speed, and as a background optimization process in long-running systems, continuing to adapt to an operating environment that changes over time.

6 Acknowledgements

J. Swan would like to thank Miles Sabin and the contributors to the Scala 'shapeless' library. His work on this paper is funded by EPSRC grant EP/J017515/1 (DAASE). K. Krawiec acknowledges support from National Science Centre, Poland, grant 2014/15/B/ST6/05205.

References

1. Karthik Kannappan, Lee Spector, Moshe Sipper, Thomas Helmuth, William La Cava, Jake Wisdom, and Omri Bernstein. Analyzing a Decade of Human-Competitive ("HUMIE") Winners: What Can We Learn? In Rick Riolo, William P. Worzel, and Mark Kotanchek, editors, *Genetic Programming Theory and Practice XII*, Genetic and Evolutionary Computation, pages 149–166, Ann Arbor, USA, 8-10 May 2014. Springer.
2. Sean Luke. The ECJ Owner's Manual. `http://www.cs.gmu.edu/~eclab/projects/ecj`, 2010.
3. Kenneth Dawson-Howe. *A Practical Introduction to Computer Vision with OpenCV*. Wiley Publishing, 1st edition, 2014.
4. William B. Langdon, David R. White, Mark Harman, Yue Jia, and Justyna Petke. API-constrained genetic improvement. In Federica Sarro and Kalyanmoy Deb, editors, *Proceedings of the 8th International Symposium on Search Based Software Engineering, SSBSE 2016*, volume 9962 of *LNCS*, pages 224–230, Raleigh, North Carolina, USA, 8-10 October 2016. Springer.
5. Krzysztof Krawiec and Bir Bhanu. Visual Learning by Coevolutionary Feature Synthesis. *IEEE Transactions on System, Man, and Cybernetics – Part B*, 35(3):409–425, June 2005.
6. Simon Harding, Juergen Leitner, and Juergen Schmidhuber. Cartesian Genetic Programming for Image Processing. In *Genetic Programming Theory and Practice X*, Genetic and Evolutionary Computation. Springer, Ann Arbor, 2012.
7. Fernando Otero, Tom Castle, and Colin Johnson. EpochX: Genetic Programming in Java with Statistics and Event Monitoring. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 93–100, New York, NY, USA, 2012. ACM.
8. Flix-Antoine Fortin, Franois-Michel De Rainville, Marc-Andr Gardner, Marc Parizeau, and Christian Gagn. DEAP: Evolutionary Algorithms Made Easy . *Journal of Machine Learning Research*, 2171–2175(13), jul 2012.
9. Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. Genetic Improvement for Adaptive Software Engineering (Keynote). In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 1–4, New York, NY, USA, 2014. ACM.

10. Jerry Swan and Nathan Burles. Templar - A Framework for Template-Method Hyper-Heuristics. In *Genetic Programming - 18th European Conference, EuroGP 2015, Copenhagen, Denmark, April 8-10, 2015, Proceedings*, pages 205–216, 2015.
11. Swan, J. and Epitropakis, M. G. and Woodward, J. R. Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming. Technical Report CSM-195, Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, Scotland, January 2014.
12. Kwaku Yeboah-Antwi and Benoit Baudry. Embedding adaptivity in software systems using the ECSELR framework. In *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference*, pages 839–844. ACM, 2015.
13. John R. Woodward and Jerry Swan. Template Method Hyper-heuristics. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1437–1438, New York, NY, USA, 2014. ACM.
14. Nathan Burles, Jerry Swan, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, and Nadarajen Veerapen. Embedded Dynamic Improvement. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, pages 831–832, 2015.
15. David J. Montana. Strongly Typed Genetic Programming. *Evol. Comput.*, 3(2):199–230, June 1995.
16. Conor Ryan, JJ Collins, and Michael O. Neill. *Grammatical evolution: Evolving programs for an arbitrary language*, pages 83–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.
17. J.C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.
18. R. Milner, L. Morris, and M. Newey. A Logic for Computable Functions with Reflexive and Polymorphic Types. In *Proceedings of the Conference on Proving and Improving Programs*, Arc-et-Senans, 1975.
19. Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing Approaches to Generic Programming in Haskell. In *Proceedings of the 2006 International Conference on Datatype-generic Programming*, SSDGP'06, pages 72–149, Berlin, Heidelberg, 2007. Springer-Verlag.
20. Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM.
21. Bruno C. d. S. Oliveira and Jeremy Gibbons. Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming. *J. Funct. Program.*, 20(3-4):303–352, July 2010.
22. Jeremy Gibbons. Origami programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 41–60. Palgrave, 2003.
23. Adriaan Moors, Frank Piessens, and Wouter Joosen. An Object-oriented Approach to Datatype-generic Programming. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming*, WGP '06, pages 96–106, New York, NY, USA, 2006. ACM.
24. Ralf Hinze. Generics for the Masses. *J. Funct. Program.*, 16(4-5):451–483, July 2006.
25. Miles Sabin et al. shapeless: generic programming for Scala, 2011–2016. http://github.com/milessabin/shapeless.

26. Dave Gurnell. *The Type Astronauts Guide to Shapeless*. Underscore Consulting LLP, http://underscore.io/books/shapeless-guide, 2016. ISBN 978-1-365-61352-4.
27. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
28. I. Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Number 15 in Problemata. Frommann-Holzboog, Stuttgart-Bad Cannstatt, 1973.
29. Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.
30. Julian F Miller and Peter Thomson. Cartesian genetic programming. In *Genetic Programming*, pages 121–132. Springer Berlin Heidelberg, 2000.
31. William B. Langdon and Mark Harman. Optimising Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, February 2015.
32. Sean Luke and Liviu Panait. A Comparison of Bloat Control Methods for Genetic Programming. *Evol. Comput.*, 14(3):309–344, September 2006.
33. Simon M. Lucas. *Genetic Programming: 7th European Conference, EuroGP 2004, Coimbra, Portugal, April 5-7, 2004. Proceedings*, chapter Exploiting Reflection in Object Oriented Genetic Programming, pages 369–378. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
34. Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, January 2012.
35. Jerry Swan, Zoltan A. Kocsis, and Alexei Lisitsa. The 'Representative' Metaheuristic Design Pattern. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1435–1436, New York, NY, USA, 2014. ACM.
36. Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, Iterated Race for Automatic Algorithm Configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.
37. Geoffrey Neumann, Jerry Swan, Mark Harman, and John A. Clark. The Executable Experimental Template Pattern for the Systematic Comparison of Metaheuristics. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1427–1430, New York, NY, USA, 2014. ACM.