Department of Computing Science and Mathematics
University of Stirling

Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming

Jerry Swan, Michael G. Epitropakis and John R. Woodward

Technical Report CSM-195

ISSN 1460-9673

January 2014

Department of Computing Science and Mathematics
University of Stirling

Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming

Jerry Swan, Michael G. Epitropakis and John R. Woodward

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44 1786 467 421, Facsimile +44 1786 464 551
Email jsw@cs.stir.ac.uk, mge@cs.stir.ac.uk, jrw@cs.stir.ac.uk

Technical Report CSM-195

ISSN 1460-9673

January 2014

Abstract

Genetic Improvement Programming (GIP) is concerned with automating the burden of software maintenance, the most costly phase of the software lifecycle. We describe GEN-O-FIX, a GIP framework which allows a software system hosted on the Java Virtual Machine to be continually improved (e.g. make better predictions; pass more regression tests; reduce power consumption). It is the first exemplar of a *dynamic adaptive* GIP framework, i.e. it can improve a system *as it runs*. It is written in the Scala programming language and uses reflection to yield source-to-source transformation. One of the design goals for GEN-O-FIX was to create a tool that is user-centric rather than researcher-centric: the end-user is required only to provide a measure of system quality and the URL of the source code to be improved. We discuss potential applications to predictive, embedded and high-performance systems.

1 Introduction

It is well-known that software maintenance dominates lifecycle cost [6, 16] and requires a huge investment of human effort. It often requires highly-skilled developers, since they are most likely to be able to navigate the intricacies of legacy systems. This article describes GEN-O-FIX, an embedded monitor framework that can continually improve a software system without developer intervention. A key distinguishing characteristic is that it offers *embedded adaptivity*, i.e. the improvement process can take place within the monitored system as it runs. It can therefore be deployed as part of 'always-on' systems (embedded systems, web-servers etc). As such, it is the first exemplar of a key part of the Dynamic Adaptive Search-Based Software Engineering manifesto (DAASE) [7], which "*places computational search at the heart of the processes and products it creates and embeds adaptivity into both*".

GEN-O-FIX is written in the Scala programming language [21] which runs on the Java Virtual Machine (JVM). It uses the powerful reflection facilities that were recently added to the language [5]. The reflection mechanism allows the application of *Genetic Improvement Programming* (GIP) [2,8,15,23, 26, 27] to perform online transformations on Abstract Syntax Trees (ASTs). These trees are derived at runtime from user-specified seed source code. Modified ASTs can be transformed back into source-code. A schematic overview of system functionality is given in Fig. 1, showing that a client application can be monitored by GEN-O-FIX to improve one or more criteria such as performance, energy, memory, integrity etc and the resulting improved program persisted in source (and binary) form in order to create a human-readable record of improved system state.

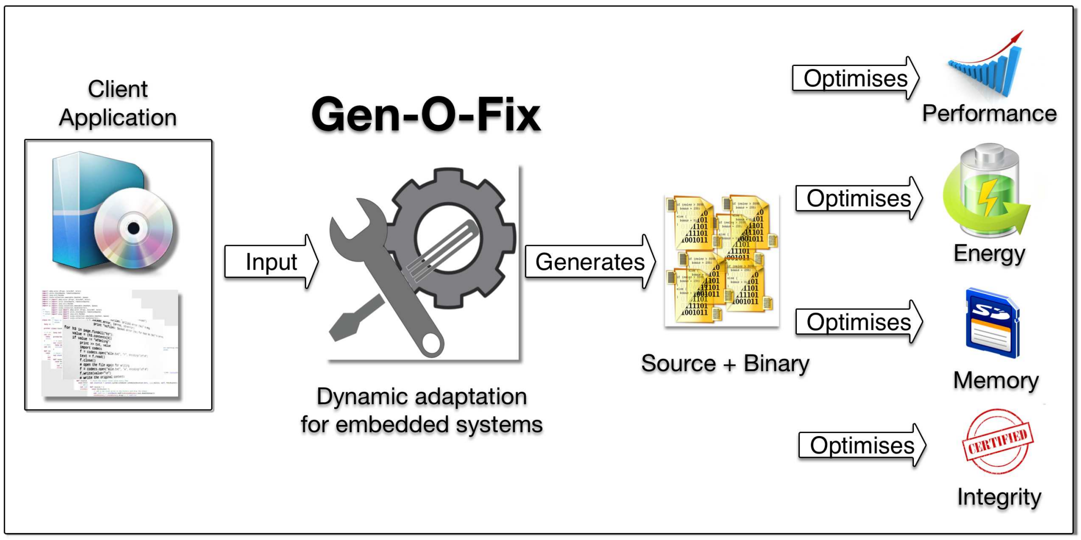

Figure 1: System Diagram for GEN-O-FIX

In summary, the main features of the GEN-O-FIX framework are:

- Embedded adaptivity, as described above.

- Minimal end-user requirements - needs only the URL of the seed functionality source code and a metric of system quality.
- Performs source-to-source transformation.
- Operates on abstract syntax trees at an arbitrarily fine level (e.g. sub-expressions).
- Tightly-integrated reflection mechanism — less brittle than existing approaches using external tools.
- Adoption of an *actor-based* approach to the execution of program variants.

The remainder of this article is as follows: Section 2 outlines related work in GIP; Section 3 discusses the particular applicability of Scala and its reflection mechanism to GIP; Section 4 describes the operation of the Gen-O-Fix framework together with the actor model that is used to evaluate program variants; Section 5 discusses possible applications and Section 6 concludes and outlines future work.

2 Genetic Improvement Programming — Related Work

Genetic Improvement Programming is the application of Genetic Programming techniques [12] to human-generated source code in order to optimize a variety of functional and non-functional properties. Although in relative infancy, GIP has been used to improve performance [15]; obtain multi-objective trade-offs between non-functional properties [8] and fix bugs [16]. In [9] it is noted that *'software is its own substrate'* and since GIP typically operates on human-generated input, it is more amenable for use in an "off the shelf" style than Genetic Programming e.g. there is no explicit need to specify terminal and nonterminal sets since they can be obtained from the seed program.

One key area of ongoing research in GIP is the issue of appropriate mutation granularity, with approaches that manipulate programs at various levels, e.g. 'line of source code' [15]; statement [16, 25]; source code [3, 26] and JVM bytecode [23]. These differing levels of granularity clearly afford a trade off between the size of the program search space (which does not *necessarily* correlate with the ease of finding solutions in such a space) and the nature of the improvements that can be expressed.

Two state-of-the-art GIP systems are [15] and [16]. In [15], the GISMOE system of Langdon and Harman is used to improve the performance of the 50,000 lines of C++ code of the widely-used DNA sequencing system Bowtie2. In [16], Le Goues et al. describe the latest version of their GenProg GIP software for bug repair in 'C' programs. GenProg employs *negative* test cases for encoding the bug and a set of *positive* test cases describing behavioural invariants. In order to conserve memory, program variants are represented as a sequence of edit operations. GenProg uses the test cases for fault-localisation and constraints the search for mutations based on the heuristic that *"a program that makes a mistake in one location often handles a similar situation correctly in another"*.

3 Scala and Reflection

Scala is a mainstream programming language that runs on the Java Virtual Machine [22]. Although a rapid proliferation of new features has gained it something of a reputation as a 'research language', it has also attracted significant interest from industry (e.g. Amazon, Cisco Systems, eBay Inc., FourSquare, IBM, NASA, Novell, SAP AG, Siemens, Sony, The Guardian, Walmart, Xerox) and it is particularly notable that the Twitter core was rewritten in Scala. This interest is not least for the following reasons:

- It is statically-typed (i.e. traps type-errors at compile-time).
- It is a unique Object-Oriented/Functional programming hybrid (generalises Hindley - Milner type-inference to [11] to include sub-typing).
- It provides strong support for Domain Specific Languages.
- It runs on the JVM and is thus interoperable with Java, Clojure etc.
- It supports expressive web-service frameworks e.g. Flint, Scalatra and Play.

- It is increasingly popular for concurrency support and reactive programming.

In addition, Scala has a number of features that make it attractive for GIP, viz. homoiconicity; declarative reflection; pattern matching and implicit conversions, described in more detail as follows:

Homoiconicity via Reflection

Homoiconicity is the ability to treat code and data interoperably at runtime. An example of how this is achieved via Scala's reflection mechanism is shown in Listing 1, with *reify* transforming code into data (as given by the *expr* AST) and *eval* operating in the reverse direction by executing the AST.

```
// A. code to data:
import scala.reflect.runtime.universe._
var m = 2; var x = 3; var c = 4
val expr = reify { ( m * x ) + c }
println( "AST = " + showRaw( expr.tree ) )

// output:
AST = Apply(Select(Apply(Select(Select(Ident("m"),
  "elem"),"$times"),List(Select(Ident("x")),
  "elem"))),"$plus"),List(Select(Ident("c"),"elem")))

// B. run AST datatype as code:
println( "eval = " + expr.tree.eval() )

// output:
eval = 10
```

Listing 1: Homoiconicity in Scala

Specifically, Listing 1 gives a simple example in which an expression is transformed into an AST which is then evaluated on-the-fly. A simple arithmetic expression $((m * x) + c)$ in three integer variables is translated to an AST via the *reify* command and the AST is then evaluated via the *expr.tree.eval()* expression. The resulting AST can be seen after the first *output* comment line. It can be observed that transforming expressions to ASTs and evaluating them reflectively is a very simple procedure in the Scala programming language.

Declarative Reflection

Programming languages vary widely in their reflection capabilities. For example, in C++, one may only obtain a unique type designator for an object via the typeid operator, whereas Java allows introspection of class hierarchies, methods and attributes, together with the ability to programmatically instantiate objects and call methods on them. This is also possible in Scala, but since AST nodes are first class objects, a great deal more useful information is available in declarative form. In fact, Scala's reflection mechanism shares a large amount of code with the Scala compiler and much of the information available to the compiler is also available via reflection. In particular:

- Well-formedness of ASTs is enforced at compile-time by the type-system.
- Subtype relationships can be queried, allowing mutations to include type substitution.
- The scope of variables can explicitly be determined, thereby eliminating mutations that erroneously reference a variable prior to its declaration.

This 'tightly integrated' approach is in contrast to the use of external tools for mutation employed in [15] and [25]. We claim that the reflective approach is substantially less brittle: the separate authorship of the compiler and the mutation tool in [15] and [25] can lead to compilation issues due to a mismatch in language standards, differing 'standard' libraries, disparities in search paths for source code etc.

Pattern Matching

In common with many functional languages, Scala offers a pattern matching facility in which a conditional execution path is followed if a match is made against object structure. Scala's pattern matching extends the traditional approach by including wildcards and conditional guards[1]. As outlined in Listing 2, we can use Scala's powerful pattern matching facility to identify and operate on AST fragments. In this example, the *Apply*(...) case matches against a specific function invocation, which we could then convert into a variant AST by transforming any or all of its typed and named component parts. As observed in [8], *"type awareness reduces the search space and makes genetic operators more effective"*. The ability to specify cases as arbitrarily nested (and conditionally guarded) typed subtrees makes it easier to specify complex transformations such as context-aware recombination/mutation.

```
def transformationRule( ast : Tree ) = ast match {
  case Apply( Select( Ident( name : TermName ), t2 : TermName ), args )
    if name == "x0" && t2 == "pow" =>
      // transform AST components ...
}
```

Listing 2: Conditional pattern-matching against a nested AST fragment

Implicit Conversions

Another novel feature is the ability to provide GIP with an analog of the Open-Closed principle [19] via Scala's user-specifiable *implicit conversions*. Specifically, we can extend the range of permissible transformations on an AST according to the promotions that are in scope for the transformer. We can thereby inject domain knowledge by providing the mutation operator with knowledge of the implicit conversions we wish to permit between types, with the end-user deciding whether Complex is substitutable for Double, CamperVan for Car etc, irrespective of whether these are built-in conversions or *a priori* subtype relationships.

4 The Gen-O-Fix Framework

One of the design goals for Gen-O-Fix was to create a tool that is user-centric rather than researcher-centric, and it is therefore desirable to minimize the number of parameters that the user is required to specify. Assume that we are improving a (sub-)system that can be defined as some function $clientCallback : I \to O$, for some specific input and output types I and O respectively. For example, in the case of client functionality for predicting a univariate time-series from a history window, we could take I to be *array of Double* and O to be *Double*. The only parameters that the end-user is required to specify are:

- Initial source code: url that points to Scala source code file containing a function $seed : I \to O$ (e.g. $Double \to Double$).
- Fitness function: $f : (I \to O) \to Double$, providing a means of evaluating the quality of system functionality, i.e. how well a program variant performs.

Note that (in accordance with the 'scavenging' approach discussed in [8]) it suffices that the signature of *seed* matches that of *clientCallback* — there is no requirement that *seed* is actually part of the original system to be improved. By this means, we can actually improve *any* JVM-hosted system, irrespective of whether it was originally written is Scala.

Note also that many fitness measures for non-functional properties (e.g. power-consumption) could be supplied 'as standard', thereby further simplifying configuration. Figure 2 gives a high-level description of the system, showing the key entity relationships. Gen-O-Fix is coupled to the client application via the *ClientCallback* function with signature as described above, the implementation of which is replaced by any improved program variants found by Gen-O-Fix. The Gen-O-Fix framework maintains program variants as one or more *AbstractSyntaxTrees* and measures the quality of generated program variants

[1]Please refer to the Scala language documentation for more details.

with a polymorphic *FitnessFunction*, which (as described in Section 5) may be implemented in terms of (a multi-objective aggregate of) functional or non-functional properties.

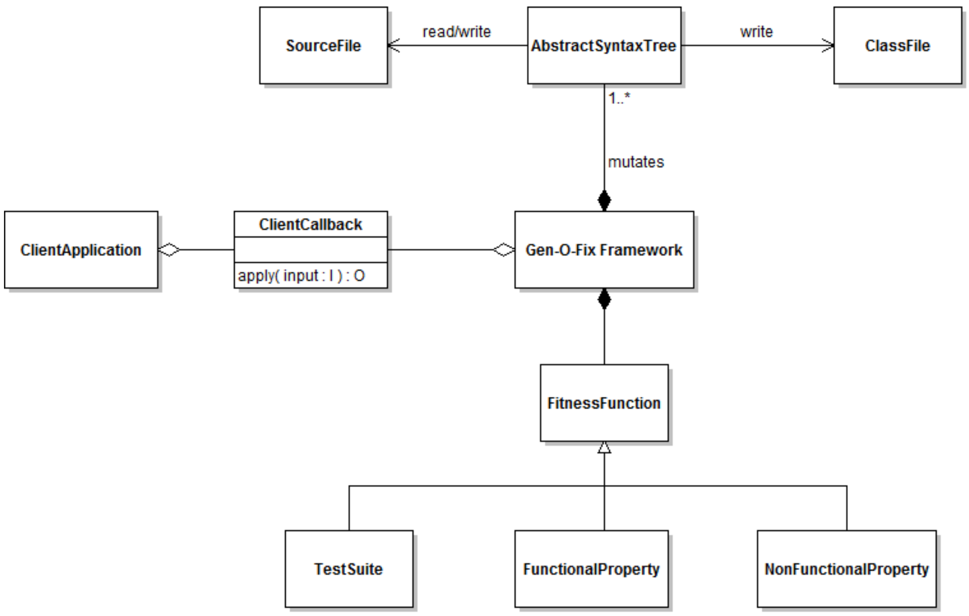

Figure 2: UML class diagram giving high-level description of GEN-O-FIX

Listing 3 outlines the operation of the core loop of GEN-O-FIX. For generality, we can assume that GEN-O-FIX runs asynchronously in a separate thread from the system to be improved. For simplicity, the core loop is shown as if it were a single-point iterative improvement algorithm [18], but any perturbative (and indeed population-based) meta-heuristic can be used. In fact, for the time-series prediction example described in Section 5, the meta-heuristic used was fully-fledged Genetic Programming with point plus subtree mutation and subtree crossover [12].

```
  params
    sourceURL  :  URL
    clientCallback : I → O
    fitness : (I → O) → Double
  var currentProgAST = compileToAST( sourceURL )
  while( true ) {
    val newProgAST = mutate( currentProgAST )
    if( newProgAST is fitter than currentProgAST ) {
      currentProgAST = newProgAST
      clientCallback.setImpl( newProgAST )
      writeSourceCode( newProgAST )
      writeClassFile( newProgAST )
    }
  }
}
```

Listing 3: GEN-O-FIX monitor core loop

In more detail, the functions of Listing 3 operate as follows: The *compileToAST* function reads the file referred to by the source URL and converts it into an AST via the reflection API. The *mutate* function performs some transformation on the AST as described above. In order to ensure that improvements to the *seed* function are reflected in the system being monitored, the *clientCallback* is passed to GEN-O-FIX inside a wrapper adaptor having a *setImpl* method that allows the initial callback implementation to be replaced by an AST. The process of measuring the fitness of program variants is complicated by the possibility of runtime errors and non-termination, and is described in more detail below. The *writeSourceCode* and *writeClassFile* functions create a persistent record of the improved system state. In the event of an unexpected system outage (e.g. power loss) this could (with a little extra book-keeping by Gen-O-Fix) allow the system to be restarted in the most-recently improved state. At the time of writing, the native Scala API support for converting ASTs back into human-readable source code is incomplete, so the conversion process is supplemented by rules that are somewhat *ad hoc* in places (e.g. in the conversion of mangled arithmetic operator names such as $*plus* to their infix equivalents).

4.1 Actors and the Halting Problem

It is well-known that for any computational model more expressive than primitive recursion there is no algorithm that can establish whether a program in that model will terminate (a.k.a. the halting problem). Genetic Programming has traditionally avoided the halting problem by using restricted models such as simple mathematical expressions [28]. Unless we are applying GIP to programs in a language such as Agda [20] (in which it is guaranteed that well-formed programs terminate), we must therefore address non-termination.

When GIP is applied to a software system, the output is one or more mutated variants of the system. In order to evaluate the fitness of a variant, it is necessary to compile and execute it. If the variant is syntactically correct then it can be compiled successfully, otherwise the compilation process will fail, possibly by throwing an exception. In such a case, the GIP system has to either repair or simply discard the variant. In the execution phase there are three main outcomes:

1. The program will terminate successfully within a pre-specified time.
2. The program will fail to halt within the pre-specified time.
3. The program will crash, i.e. a runtime error will occur and exceptions will be thrown.

In the first case, GIP can successfully evaluate the quality of a variant. The last two cases are more complex, since GIP should be able to terminate its execution after a maximum pre-specified period of time and also handle any exceptions thrown by the program. In an off-line setting, GIP can easily handle the aforementioned cases, since it they do not affect the state of the original program. By contrast, in an online adaptive system it is imperative that the fitness evaluation process avoids crashing the system in which it is embedded. In the proposed system, we tackle these issues by using the "*actor model*" for concurrency computations.

Briefly, the actor model is a mathematical model for concurrent computations, introduced in [1,10]. This model has been popularized by the success of the Erlang programming language, and has been used in several other programming languages, including Scala and Groovy.

The actor model employs the *actor* as its basic computational entity and treats it as the universal primitive of concurrent computation. An actor has a mailbox and a set of behaviors that are triggered by mailbox contents. An actor can: concurrently communicate with other actors by sending or receiving messages; designate its behavior or react based on a received message, and dynamically create other actors. One of the main and important characteristics of this model is that all communications are performed asynchronously, without any assumption about the sequence of the exchanged messages. The Scala programming language adopts the actor model which is implemented in the Akka library[2].

Specifically, in the proposed system each mutated program variant will be evaluated under a new actor spawned by the Gen-O-Fix system. That actor is responsible for handling both the compilation and execution phases of the mutant variant. In the compilation phase each actor is able to catch the exceptions thrown by the compilation of the corresponding ASTs. In the current incarnation of Gen-O-Fix, if a compilation error occurs then the variant is simply discarded.

If in the execution phase the variant terminates within the pre-specified period of time, its owning actor is able to measure its fitness via the provided quality function. If an exception is thrown, the owning actor will be responsible for catching the exception and reacting accordingly (most trivially by assigning a prohibitively low fitness to the variant).

The actor model provides an essential means for handling these issues efficiently. In general terms, there are several ways to terminate an actor (though the specifics may depend on the design of the system at hand) such as using *Futures* [29] or the 'Reaper' pattern[3]. In the current Gen-O-Fix system,

[2]http://akka.io/
[3]http://letitcrash.com/post/30165507578/shutdown-patterns-in-akka-2

we have elected to use a simple actor-based "fork-join" execution, making it easy to terminate an actor after a pre-specified time.

It is important to note that the actor model also makes it simple to execute an actor in a distributed fashion. Thus, each actor can run either in the same execution node as its parent, or in any node available to the computational system at hand (e.g. on some remote host). This is clearly an important factor for scalability.

5 Applications

As discussed above, much of the work in GIP to date has been concerned with bug-fixing. If we choose to adopt the perspective of "The GISMOE Challenge" [8], we can consider the number of unit tests passed by a program variant to be merely one of several multi-objective criteria to be optimized. In particular, non-functional criteria to optimize for may include:

- **Energy consumption**. In the light of our comments above regarding the value of strongly-typing, it is interesting to note the use by Cohen et al. of a type-system to annotate different profiles of energy usage within a program [4].
- **Memory Usage**. The ability to trade memory against execution time has been extensively studied since the inception of computer science. We therefore anticipate that optimizing for memory-usage may bring renewed interest in the automatic generation of data structures [14].

There are a number of application areas in which the dynamic adaptive nature of Gen-O-Fix might be used to particular advantage, viz. isolated/embedded systems; High-Performance Computing and predictive systems. We discuss each in more detail below:

Isolated or Embedded Systems

In certain circumstances, it may be desirable to use Gen-O-Fix as an alternative to the centralized deployment of software updates. This is particularly true for systems that are isolated (i.e. with low or no connectivity) and/or embedded systems. In both cases, the local knowledge that the system possesses about the environment in which it operates can be used to directly optimize the performance of that system. Even assuming that centralized redeployment of software updates is possible, each locally-improved variant program is likely to outperform any "average case" program deployed *en masse*.

High Performance Computing

The applicability of GIP to automated parallelization has already been demonstrated [13] at the level of CUDA kernels. Of more general interest is the ability to perform multi-objective optimization on alternative parallelization strategies. An interesting avenue in this respect is to search the space of functions provided as arguments to the ubiquitous and highly-parallelizable higher-order function *reduce*.

Predictive Systems

As a 'proof-of-concept' demonstrated at TODO[4], we have embedded Gen-O-Fix within a web application for stock-price prediction. The starting point for the evolved functionality is a predictor extracted from the web-application source code, and this function is transformed by GIP via subtree crossover and point/subtree mutation.

6 Conclusion and Future Work

In this article, we have described a system for software self-improvement in which compilation and mutation are tightly integrated via reflection. We believe that the declarative information made available by this approach is an essential ingredient in addressing longer-term scalability concerns for GIP.

[4]Name of international workshop omitted for 'double-blind' purposes.

Approaches such as [15] and [17] have been termed ‘plastic surgery’, in that code is ‘scavenged’. Such ‘scavenging’ might be from other parts of the program under evolution [17] or (as suggested in [8]) from some user-specified repository or via an online semantic similarity search. It is our belief that such ‘plastic surgery’ approaches will ultimately not be able to scale to meet the wider ambitions of ‘programming as iterated debugging’ alluded to in [16], since these relatively poorly-informed methods either severely limit the nature of the mutations that can be created or else have a higher chance of creating ill-formed source code. It is common currency in the meta-heuristics community that domain knowledge is the key to scalability. For GIP, such domain knowledge is well-represented by the use of strong typing and context-driven rules for mutation and crossover. By ‘context’ in this sense, we mean, as described above, the ability to pattern match typed AST fragments *at the subexpression level* and conditionally manipulate them accordingly. This is a more general approach than the ‘context-sensitive’ mutation of [17], which we hypothesise will not scale well beyond cases such as the introduction of missing guard conditions that happen to have been included elsewhere in the program.

The main body of future work is therefore concerned with extending the nature of the AST transformation rules (in particular, the context-aware matching aspects), thereby increasing the structural aspects of ASTs that can be improved in a less-stochastic fashion. We are currently investigating automated repair of (Scala versions of) well-known successes of other GIP systems such as the Zune and GCD bugs [17], which will allow further investigation of the hypothesis in the preceding paragraph.

In addition to the GIP-enabling features described in this article, Scala is a highly developed programming language that is undergoing continual principled evolution. We anticipate that further language developments (e.g. *quasiquotes* [24]) will make reflective software modification easier and more powerful than it is at present.

References

[1] Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)

[2] Arcuri, A., White, D.R., Clark, J., Yao, X.: Multi-objective improvement of software using co-evolution and smart seeding. In: Li, X., Kirley, M., Zhang, M., Green, D., Ciesielski, V., Abbass, H., Michalewicz, Z., Hendtlass, T., Deb, K., Tan, K.C., Branke, J., Shi, Y. (eds.) Simulated Evolution and Learning, pp. 61–70. No. 5361 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (2008)

[3] Cody-Kenny, B., Barrett, S.: Self-focusing genetic programming for software optimisation. In: Proceeding of the Fifteenth Annual Conference Companion on Genetic and Evolutionary Computation Conference Companion. pp. 203–204. GECCO ’13 Companion, ACM, New York, NY, USA (2013)

[4] Cohen, M., Zhu, H.S., Senem, E.E., Liu, Y.D.: Energy types. In: OOPSLA. pp. 831–850 (2012)

[5] Coppel, Y.: Reflecting Scala. Semester project report, Laboratory for Programming Methods. Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland (2008)

[6] Glass, R.L.: Frequently forgotten fundamental facts about software engineering. IEEE Software 18(3), 112–111 (2001)

[7] Harman, M., Burke, E., Clark, J., Yao, X.: Dynamic adaptive search based software engineering. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–8. ESEM ’12, ACM, New York, NY, USA (2012)

[8] Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE ’12) (Keynote). ACM, Essen, Germany (2012)

[9] Harman, M., McMinn, P., de Souza, J.T., Yoo, S.: Search-based software engineering: Techniques, taxonomy, tutorial. In: Meyer, B., Nordio, M. (eds.) Empirical Software Engineering and Verification, Lecture Notes in Computer Science, vol. 7007, pp. 1–59. Springer (2011)

[10] Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence. pp. 235–245. IJCAI’73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1973)

[11] Hindley, J.R.: The principal type-scheme of an object in combinatory logic. Transactions of the American Mathematical Society 146, 29–60 (1969)

[12] Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). A Bradford Book, 1 edn. (1992)

[13] Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In: Sobrevilla, P. (ed.) 2010 IEEE World Congress on Computational Intelligence. pp. 2376–2383. IEEE, Barcelona (18-23 Jul 2010)

[14] Langdon, W.B.: Genetic Programming + Data Structures = Automatic Programming!, Genetic Programming, vol. 1. Kluwer, Boston (1998)

[15] Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation (2013), accepted

[16] Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Software Quality Jornal 21, 421–443 (2013)

[17] Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering 38, 54–72 (2012)

[18] Luke, S.: Essentials of Metaheuristics. Lulu, second edn. (2013), available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/

[19] Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edn. (1988)

[20] Norell, U.: Dependently typed programming in agda. In: Lecture Notes from the Summer School in Advanced Functional Programming (2008)

[21] Odersky, M.: The scala experiment – can we provide better language support for component systems? In: Proc. ACM Symposium on Principles of Programming Languages. pp. 166–167 (2006)

[22] Odersky, M.e.a.: An overview of the scala programming language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland (2004)

[23] Orlov, M., Sipper, M.: Flight of the FINCH through the java wilderness. IEEE Transactions on Evolutionary Computation 15(2), 166–182 (2011)

[24] Shabalin, D., Burmako, E., Odersky, M.: Quasiquotes for Scala. Tech. rep. (2013)

[25] Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Communications of the ACM 53(5), 109–116 (2010)

[26] White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation 15(4), 515–538 (2011)

[27] Wilkerson, J.L., Tauritz, D.R., Bridges, J.M.: Multi-objective coevolutionary automated software correction. In: Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference. pp. 1229–1236. GECCO '12, ACM, New York, NY, USA (2012)

[28] Woodward, J.R.: Evolving turing complete representations. The 2003 UK Workshop on Computational Intelligence, University of Bristol (2003)

[29] Wyatt, D.: Akka concurrency. Artima Inc, [S.l.] (2013)