Genetic Programming for Entire Applications

Gene Michael Stover

created Wednesday, 21 May 2003
updated Wednesday, 21 May 2003

Contents

1 Disclaimer

This essay can accurately be described as "Gene's thoughts about this topic as he thinks, writes, & re-works them". It may be badly composed, inaccurate, & rambling. Comments are not solicited.

2 Here Goes

Genetic programming has been applied succesfully to small problems, but it can be used to write much of the code in entire applications. It is possible to do this due to some current trends in software development, & I predict that it will be done within a few years.

For genetic programming to work on the larger scales than it has in the academic literature, you need to employ a few tricks: formal requirements, software components, & restricted implementation languages.

3 Formal Requirements

By "formal requirements", I mean formal in the sense of formal languages, languages that can be recognized by an automaton. The term "formal requirements" is laughably applied to requirements written in a natural language. Better terms for those requirements would be "definitive requiremenets" or "official requirements".

For genetic programing to work, your requirements must be expressed so a program (the genetic programming framework) can compare candidated programs, which are being evolved, to the requirements. The genetic programming engine/framework needs to evaluate the programs it is evolving according to the requirements. So the requirements must be specified in a formal language.

The trend of converting requirements from natural language to formal language is already under way & in the vogue. It is often called "test-driven design", but to use it to evolve most of the code in an application, you must recognize that a single natural language requirement leaves much unsaid. I wouldn't be surprised if the average natural language requirement should convert to 100 or 1,000 formal requirements.

Requirements in natural language are still necessary because formal languages are usually imperative. Those languages describe how to verify the requirement, not what the requirement is. Computers need to know how to verify requirements, but humans need to know what the requirements are. Also, it is easier to derive the "how to verify it" (formal) requirements from the "what must be fulfilled" (natural) requirements, so the natural language requirements really are the definitive requirements.

If a real world application contains hundreds or thousands of definitive requirements, & if the mean definitive requirement converts to 100 to 1,000 formal requirements, then the number of formal requirements in an application will range from the low thousands to the low millions. Someone might complain that the work of writing all those requirements would exceed the work of developing the application in the first place.

Not so. Programs that embody formal requirements are smaller than applications. They are generally small enough that errors in them are rare (though such errors do happen). What's more, many formal requirements will be unchanged between applications. These might be requirments like "the application can be launched from the Unix command line" & "it has a graphical, windowed user interface" (expressed in test programs, of course). So many of your formal requirements would come in libraries of requirements. Also, the common complaint about software is that it is buggy; I'm describing a technique for evolving less buggy applications.

Many of the formal requirements in a test suite used by an evolver will appear banal to humans, but every tiny little requirement counts in evolution.

The trick to generating whole applications (or large chuncks of them) is in how you apply the formal requirements. Academic demonstrations of genetic programming use few test functions, & most of the functions are not pass/fail but have conceptually continuus success ranges. Some requirements are most

conveniently expressed as pass/fail. So you convert your definitive (natural language) requirements into tons & tons of formal requirements. Where possible, a formal requirement should have a continuus range of success values, but where that's difficult, go ahead & make a pass/fail (binary) formal requirement. If you have tons & tons of requirements, testing every tiny little thing, then it's okay if some of them are binary because the evolver can still detect small improvements in the population of programs it's evolving.

4 Restricted Implementation Language

One of the main tricks to making genetic programming work in the first place was realizing that you don't need to evolve programs in terms of the same languages humans use when they program. You don't evolve a program in the CPU's native code or in the characters that compose C.

Instead, you evolve a program in terms of the linguistic building blocks of those languages. Even better, evolve a program in terms of the linguistic building blocks of more appropriate languages.

By "linquistic building blocks", I mean the semantic parts of a language, the pieces of the parse trees. Contrast this with the individual characters that make up a C program. It'd take a long time, just to get a program to compile correctly if you evolved it as a string of characters, but if you evolve it in terms of parse trees, you can guarrantee that most or all of the programs are syntactically correct. In that case, you only need to evolve appropriate semantics.

That is what allows genetic programming to work.

When applied to applications, we take it a step farther. We carefully choose the terms that the evolved programs use.

If I need to evolve a communications protocol, the primitives I supply to the evolver have to do with sending & receiving packets & maybe parsing or encoding entire fields; I do not provide primitives that deal with scanning or copying individual bytes to make packets.

If I need to evolve a program that queries a database, I provide it with primitives that can be composed into SQL queries. I do not provide it with individual characters as the primitives in the hopes that it will eventually evolve syntactically correct SQL statements.

It's all in the primitives you select (& the suite of test cases).

5 Software Components

In theory, an evolver with an appropriate suite of test programs could create an entire application, but it'd take a long time. It is faster if the programmers act as architects to break the application into many discrete, unambiguously defined components.

One reason you do this is to make the problem more manageable to the evolver. You don't ask the evolver to create an entire application. You ask it

to create a well-defined, testable, automatically verifiable component. If you define your application in terms of a manageable number of components, you can set evolvers running to write each of those in parallel. Once they are all done, humans write a small amount of code to glue them together.

This is nothing new, of course. The "divide & conquer" technique of software development is simultaneously the most universally appropriate & most forgotten design methodology. So you should divide your programs into components, anyway. Now, you have an automaton to write them for you; you just define them (with formal requirements), & then glue them together with a (hopefully) small amount of human-generated code.

References