Theories Destroy Facts (as Programs Execute)
Version 1.0, 2014-01-13

Colin G. Johnson
Email: C.G.Johnson@kent.ac.uk

This document explores what happens to programs as they are evaluated, with the aim of developing a better understanding of partial programs so that we can synthesize them using techniques such as genetic programming (GP) and other program induction methods.

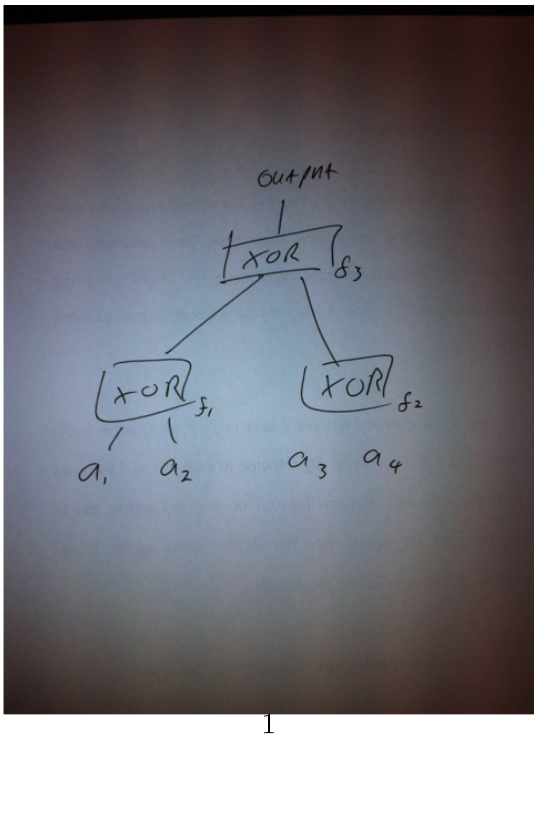
For the purposes of the argument we will look at programs represented as parse trees, with no recursion, looping or memory read/writes, and which take n inputs and return a single input. These shouldn't be particularly strong restrictions on the arguments below and as we develop the techniques we should be able to broaden the scope of these methods.

1 "Theories Destroy Facts"

"Theories Destroy Facts" is a quote from the biologist Peter Medawar, explaining how scientific progress occurs—at a certain point in scientific development, all scientists have is a bunch of unrelated facts, and as scientific development progresses, observations and experiments develop *theories* that obviate the need to store these individual facts—if we need the facts again we can reconstruct them from the theories. We use this as a loose inspiration for what happens during program evaluation.

When a program gets evaluated, we begin with a number of inputs. As the evaluation progresses—as each function in the parse tree is performed—these inputs are replaced with the results from these evaluations. In a correct program, these evaluations gradually replace the original variables with new values that contain more information relevant to solving the problem. Our big challenge here is to develop some ways of measuring this informational change.

Take, as an example, the four-bit boolean parity-checker. Call the inputs $a_1, \dots, a_4$. An example correct solution to this problem consists of three XOR gates, executing the following program:



Now, consider what is happening to the inputs as we evaluate the functions in this programs. When we evaluate the function f_1, we have extracted all of the information relevant to the problem that is available in variables a_1 and a_2. We will say that f_1 has *destroyed* a_1 and a_2, meaning that once we have calculated that function we will never again need to look at a_1 or a_2 in the remainder of the evaluation of the program. Of course, if the problem being solved were a different one, then the execution of f_1 would not necessarily destroy a_1 and a_2. This is a really important point for later on—when we want to quantify this, we will always need to do it *with regard to a problem*, we will not be able to do it in general terms.

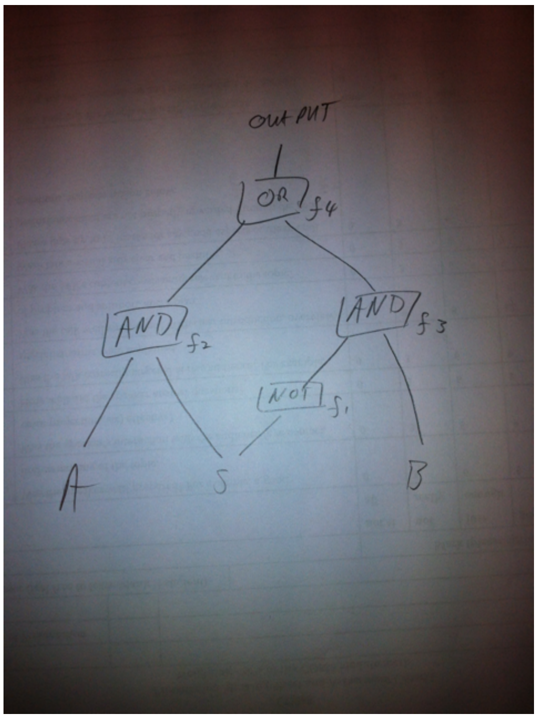
Here's another (obvious, but) interesting point. The final execution step (the root node of the parse tree) in the evaluation is just the final step in the process. Whilst it is special in one way—that it brings together all of the information required and produces the desired output—it is not where "all the magic happens". In order to be able to produce the final, correct output, a lot of information has to have been constructed prior to that. The two input paths that feed into the final execution step already contain (albeit in some subsymbolic, "holographic" form) most of the information required to solve the problem. All the final step does is just to bring them together and do a final little bit of calculation. What we want to be able to do is to quantify that information in some way.

Also, note that once we have evaluated a function, we can regard the output from that function as a new "input variable". For example, in the parity example, we can see f_1 and f_2 as being two variables, and then we can see f_3 the function that destroys those two variables. In a large program there will be a lot of this destroying one "level" of variables to make a new set of variables that "contain more information" than the previous set, and then in turn destroying those to make a new set, ...

In summary: we define *destruction* of a set of variables as the calculation of a function which means that we no longer need to look at those variables again in the program. Less formally, we have extracted (and summarised) all of the "information" that is required from those variables in a smaller number of variables.

Partial Destruction

In one way, the parity problem is an unusually good one for this framework, as we can destroy two variables with a single binary operation. When we do $a_1 \mathrm{XOR} a_2$, we remove the need ever to look at a_1 or a_2 ever again during the evaluation. But, this "total destruction" is not the only way in which information about the solution can be gained through evaluation, indeed, for some problems the original variables will need to be looked at multiple times in different contexts. Consider, for example, a very simple 3-bit multiplexer circuit (3 boolean inputs A, B, S: the circuit outputs either the value of A or the value of B depending on the value of S).



If we look at S, we note that it needs to be looked at *twice* before it can be destroyed. When we look at f_2, we have destroyed the variable A, as we will never need to revisit A again during the evaluation of the program. But, we have *not* destroyed S at that stage. It is not until we have evaluated f_2 and f_3 that we have fully destroyed S. Nonetheless, if we are going to evaluate partial programs reasonably we need some way to assign some partial credit just to something like f_2.

Note, also, that we cannot keep "all of the balls in the air" until the evaluation of the final function at the root of the tree. We *must* have destroyed a reasonable amount of the information of the original variables by the time we get to the final function evaluation. That might be partial destruction of all of the variables, or it might mean (as in the parity example) that some of the variables have been fully destroyed.

Symmetry Breaking and Dependence

It is notable that some problems have a certain amount of symmetry. For example, if we consider the parity problem again, we could choose any two pairs of the inputs for the first layer in the parse tree. If we are going to learn the functions one-by-one, then we would expect a function like $a_1 \,\mathrm{XOR}\, a_2$ to be as valuable as $a_1 \,\mathrm{XOR}\, a_3$. But, once we have created the first of these, the second is no longer as valuable, as we have already destroyed a_1. We need some way to track this, perhaps by constantly keeping a list of what variables have been created and which destroyed or partially destroyed.

Formalising this

Let's try and formalise this a bit further. What we want to do is to automatically build a program to solve a given problem p, not by a traditional GP-type approach of building full programs, but by gradually building up solutions from sub-trees.

Let us define an *input vector* i as an ordered list of variable assignments, i.e. a set of training instances aka fitness cases. A *target vector* $t_{p,i}$is defined with regard to a problem and an input vector, and lists the target values for each value for each item in the input vector (in the same order). For a particular parse tree t and input vector i, we can define the *output vector* $o_{t,i}$ is the list of outputs when we run each of the items in the input vector through the tree.

We are working towards building up parse trees from the *bottom up*; that is, we want to take small tree fragments and assess how good they will be as potential sub-trees of a complete solution to the problem at hand. We *don't* want to have to build full trees before we start to assign a value or ranking to these.

Let us imagine that there was a way to evaluate how effective a small, incomplete tree was on a problem. Then, we could start by generating a number of little trees, with perhaps just two input variables as inputs, and doing this evaluation to decide which of these trees were best. We could then add the output from that set to the set of variables (and perhaps, if we have some way of deciding whether the new variable destroys fulling any of the other inputs, remove some variables from the list).

Let us construct an example. Let's let the problem be the 4-bit parity problem discussed earlier, where an even number of ones in the input returns zero and and an odd number returns one. Imagine that we generate two possible trees (in practice we would generate more): $f_1 = a_1 \,\mathrm{XOR}\, a_2$ and $f_2 = a_1 \,\mathrm{AND}\, a_2$. Define the difference between two vectors as 1 when the two inputs are the same and 0 when they are different (i.e. XNOR).

input: $a_1a_2a_3a_4$	$f_1 = a_1 \,\mathrm{XOR}\, a_2$	$f_2 = a_1 \,\mathrm{AND}\, a_2$	parity	diff f_1 and parity	diff f_2 and parity
0000	0	0	0	1	1
0001	0	0	1	0	0
0010	0	0	1	0	0
0011	0	0	0	1	1
0100	1	0	1	1	0
0101	1	0	0	0	1
0110	1	0	0	0	1
0111	1	0	1	1	0
1000	1	0	1	1	0
1001	1	0	0	0	1
1010	1	0	0	0	1
1011	1	0	1	1	0
1100	0	1	0	1	0
1101	0	1	1	0	1
1110	0	1	1	0	1
1111	0	1	0	1	0

How are we going to decide which of these two examples to use? We want to find some way of getting at the intution that f_1 is better because the difference between f_1 and the parity has more structure, that is, the calculation of f_1 has captured more of the structure of the problem, or, in a different way, there is less computation to do to get from the output vector of f_1 to the target vector.

We can use the idea that compression is roughly correlated to the complexity of calculation needed to be done to transform one sequence into another. If we compress the two difference vectors (using gzip), we find that the difference between f_1 and parity, we find that is is of length 38, whereas the difference between f_2 and parity compresses to length 40. So, we would prefer f_1.

Will this work in general? I'm implementing something at the moment to do some proper testing of this.

Another approach might use "mutual information".

A Contrary Example

Here is a contrary example. Let's say we are trying to evolve a program that adds four numbers together, where the function set is the basic arithmetic operators. I can't see any way in which a partial evaluation could give any useful information, as the result is so dependent on all four summands. But, perhaps, we need to think about the ordering of the training cases?

Some Final Notes

Some other topics to be explored further:

- Traditional computer science theory has concentrated on correct programs. When a program is incorrect most aspects of theory don't say much about it, other than "it is wrong". We have a very weak set of theoretical work about how "close" a program is to a correct one; this contrasts with data mining/machine learning, which has a really good set of ideas about how good or bad a model is of a dataset. What is the semantics of wrong programs?
- There is a flavour here which might be captured by some kind of Bayesian reasoning, perhaps. Or perhaps some kind of "signal processing" approach to the program strings.
- Is there a link here with Bob McKay's "partial functions" work?
- Traditional GP has placed great importance on getting as much right as possible and building on those programs that get a lot right. By contrast, we might argue that this is the wrong approach; a good step-on-the-way to solving a problem solves a certain part of the problem and leaves the rest "untouched" in some way so that later generations can build on it.
- Has GP taken a wrong direction by concentrating on "whole programs" rather than ways of evaluating partial programs (w.r.t. a problem). A while ago I did some experiments with "dynamic baseline" GP where the algorithm consisted of running lots of little GP runs, each of which added to the best result from the previous GP run. This seems to capture the essence of what we want from mutation better than traditional syntactic mutation.
- Does the ordering of the input set matter? It shouldn't, but. . . .
- We need to get a good set of terminology here. I don't really like the word "destroyed"; what is better?