

# Semantic building blocks in genetic programming

Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison

Division of Science and Mathematics  
University of Minnesota, Morris  
Morris, Minnesota  
USA 56267  
{mcphee,ohsx0004,hutc0125}@morris.umn.edu  
<http://www.morris.umn.edu/~mcphee/>

**Abstract.** In this paper we present a new mechanism for studying the impact of subtree crossover in terms of semantic building blocks. This approach allows us to completely and compactly describe the semantic action of crossover, and provide insight into what does (or doesn't) make crossover effective. Our results make it clear that a very high proportion of crossover events (typically over 75% in our experiments) are guaranteed to perform no immediately useful search in the semantic space. Our findings also indicate a strong correlation between lack of progress and high proportions of fixed contexts. These results then suggest several new, theoretically grounded, research areas.

## 1 Introduction

Subtree crossover is one of the oldest and remains one of the most widely used recombination operators in genetic programming (GP). It is still unclear, however, why or how it works. It's hardly obvious that yanking a random chunk of code from one program, and plopping it unceremoniously in a random location in a second program would be a good thing. Yet it clearly works (at some level) in GP.

But why? How does subtree crossover move the population closer to the solution? Is it really just a happy accident that this simple operator provides some sort of useful recombination? Are there better operators and representations waiting to replace this strangely random process?

In this paper we present a new mechanism for studying the *semantic* effect of subtree crossover in terms of *semantic building blocks*. Subtree crossover combines two tree components: the context (the root parent with a subtree removed) and the subtree being inserted into that context. Our approach allows us to completely and compactly describe (for boolean problems) the semantics of these two key components, which allows us to completely describe the semantic action of subtree crossover. We can also enumerate the occurrences of different context and subtree semantics in a population, independent of their syntax, allowing us to perform detailed studies of the semantic components present in a population, and the opportunities this provides for subtree crossover. The resulting data strongly suggest that the distribution of context semantics are key in the success (or failure) of runs. Our results also make it clear that a very high proportion (typically over 75% in our setup) of crossover events are *guaranteed*

to perform no immediately useful search in the semantic space. These tools and results not only shed valuable new light on the operation and impact of subtree crossover, but they also suggest a number of ideas for new operations and approaches to genetic programming that are based on this new theoretical and empirical understanding.

In the next section (Section 2) we review some of previous work on GP building blocks and the behaviour of crossover. In Section 3 we present our new tools and show how the semantics of contexts and subtrees can be calculated and enumerated. In Section 4 we go over the results from empirical runs we used to collect data using new measures enabled by these ideas. We discuss those results and some of their implications in Section 5, and conclude in Section 6.

## 2 Related research

“Building blocks” have a long history in genetic algorithms (GAs), and there have been various definitions proposed for building blocks in GP. These were typically strictly syntactic in nature, and often part of an effort to adapt GA schema theory to GP (e.g., [12, 18, 14, 19, 15, 16]; see [4] for additional review). There have also been numerous studies on the impact of subtree crossover, other recombination operators, and their interactions with things like mutation [2, 1, 13, 8, 11, 4].

Many of these studies have helped us better understand important properties of GP such as code growth. None, however, have shed much light on the underlying semantic behavior of subtree crossover or provided tools to track and analyze those semantics. Perhaps the closest to the current research that we’re aware of is [2], where the proposed marking process captures useful semantic information about potential crossover points that that is related to our notion of fixed contexts discussed in Section 3.2.

## 3 Enumerating semantic “Building blocks”

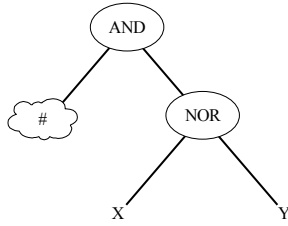
In sub-tree crossover we construct a new offspring by replacing a randomly chosen subtree from parent *A* with a random sub-tree from parent *B*. To understand the possibilities afforded by sub-tree crossover, then, we need to be able to characterize what sub-trees can be chosen from *B*, and where they can go in *A*.

We define a *context* to be a tree with some specific (but arbitrary) subtree removed (see Figure 1); we will use ‘#’ to indicate the removed subtree.<sup>1</sup> Given this definition, describing the semantic impact of sub-tree crossover reduces to describing the semantics of sub-trees, the semantics of contexts, and their interactions. We will describe these ideas in some detail below; see [10], however, for a more detailed discussion and additional examples.

This paper will focus on the boolean domain, i.e., trees that represent boolean functions. Working in such a small (finite) domain is valuable because it makes it much easier to compute and catalogue the complete semantics of the sub-trees and contexts involved. (See Sec 5 for more on extending these ideas to other domains.) In the boolean

---

<sup>1</sup> This is similar to a tree schema with one ‘#’ leaf symbol from [4]. A schema, however, represents a *set* of trees, whereas for us a context is simply a syntactic construct.



**Fig. 1.** An example of a context; the # symbol represents the removed subtree.

x y	(and x y)	(or x y)	(nand x y)	(nor x y)
0 0	0	0	1	1
0 1	0	1	1	0
1 0	0	1	1	0
1 1	1	1	0	0

**Table 1.** The (sub)tree semantics for the four boolean functions used in our experiments. In a finite (e.g., boolean) domain we can fully characterize the semantics of a (sub)tree by enumerating the values of a tree (i.e., a function) on all its possible inputs.

domain. we can generate a highly compact representation of both subtree and context semantics. This allows us to enumerate the semantics of *all* the sub-trees in a given tree in the population, or even all the sub-trees of *all* the trees in a given population. We can then explore this distribution of sub-tree semantics to better understand the possibilities available to sub-tree crossover.

### 3.1 Semantics of subtrees

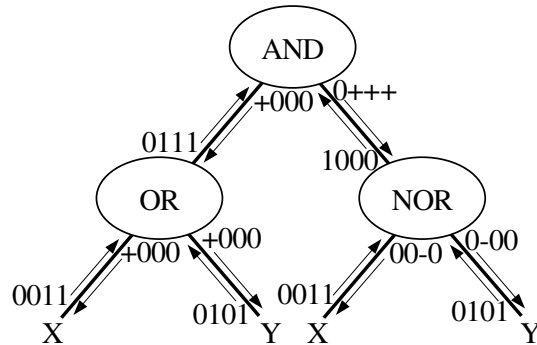
Following the ideas used in sub-machine code GP [17], we can completely specify the semantics of a boolean valued (sub)tree (or, equivalently, function) by enumerating its value on each of the possible sets of input values. Taking 0 to be *false* and 1 to be *true*, the function (and x y), for example, has the semantics 0001 corresponding to the third column in Table 1. (See Fig. 2 and [10] for additional examples of sub-tree semantics.) This, then, allows for a complete characterization of the semantics of any boolean (sub)tree in the sense that if two trees  $S_0$  and  $S_1$  have the same semantics, and tree  $T$  contains  $S_0$  as a sub-tree, we can replace the occurrence of  $S_0$  in  $T$  with  $S_1$ , and the semantics of  $T$  will remain unchanged.

### 3.2 Semantics of contexts

In general we won't know the semantics of a tree with an unspecified subtree removed, since the details of that subtree will usually affect the semantics of the entire tree. However, some contexts depend less on the details of their open subtree than others. For example, the context (and false #) is *always* going to return *false*, regardless of which subtree we insert into the open position. Further, we know from experience that genetic programming has strong tendencies towards the creation of such contexts [9, 5, 4].

We refer to a context as being *fixed* for a particular set of inputs (or a particular *position* when using strings to represent semantics) if the value of that context is completely determined (either *true* or *false*) regardless of what subtree is inserted at the open node (#). We define the entire context to be fixed if it is fixed for *every* possible set of inputs (or at every position in the semantics string).

In the boolean domain the semantics of a context depend on the details of the inserted subtree in a systematic manner. Consider, for example, the context (and true



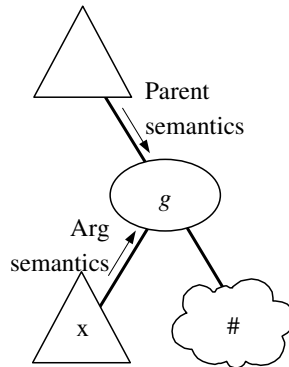
**Fig. 2.** A sample syntax tree showing both subtree and context semantics. The arrows pointing upward (on the left of the edges) are the semantics of the subtree below them, e.g., the semantics of (or x y) is 0111. The arrows pointing downward (on the right on the edges) are the semantics of the context obtained by removing the subtree below the arrow, e.g., the semantics of (and # (nor x y)) is +000.

#). Here the value of this context will be the *same* as the value of whatever subtree we insert for the #. We will denote the semantics in such a case with a +, indicating that the value of the subtree passes through unchanged. The alternative case is represented by a context like (nand 1 #). Here the value of the context is going to be the *negation* of the value returned by the inserted subtree. We will use a - to denote the semantics in this case. ([10] provides several examples in more detail.) Thus while the interactions between contexts and subtrees can be quite complex, in the case of boolean functions there are only four options for a context on a specific set of inputs: the fixed semantics (0 and 1), the “unchanged semantics” (+), and the “negation semantics” (-).

A key difference between subtree semantics and context semantics is which components need to be taken into consideration when computing the semantics. The semantics of a subtree are solely a function of the operator and the value of its arguments; they are completely independent of where that subtree might be located. For context semantics, the case is slightly more complex. While we associate the semantics with the edge above the insertion point, they are still a function of the entire tree around that point. In particular they depend on three things (see Fig. 3):

- The operator  $g$  immediately above the insertion point.
- The semantics of the context obtained by removing the subtree rooted at  $g$  (the “Parent semantics” in Figure 3).
- The subtree semantics of the other argument ( $x$ ) of the operator  $g$  (the “Arg semantics” in Figure 3).

The one exception is when the insertion point (#) is in fact the root of the context, in which case there is no parent node. In this case the context semantics are simply defined to be + since the value returned by the tree is going to be the value of the inserted subtree.



**Fig. 3.** Illustration of the interaction of the different components in computing the context semantics. Here we have a tree with some subtree removed (the insertion point, indicated by the # in the lower right).  $g$  is the parent node of the insertion point, and  $x$  represents the other argument of  $g$  (i.e., the sibling subtree of the insertion point). Note that  $x$  is not necessarily a leaf but can represent an arbitrarily complex node. The semantics of this context is then a function of the specific operator  $g$ , the semantics of the context obtained by removing the subtree rooted at  $g$ , and the subtree semantics of  $x$ .

Table 2 lists the cases for the boolean functions used in this work: *and*, *or*, *nand*, and *nor*. In the last line of Table 2, for example, if the parent semantics of a *nand* node is  $-$ , and the argument semantics of the sibling is  $1$ , then the context semantics is  $+$ .

Notationally it is convenient to associate context semantics with the edge extending *down* to the insertion point (i.e., the # symbol), as this allows us to indicate the semantics of all the possible contexts in a tree on a single diagram as is done in Figure 2. It's important to realize, however, that even though they are attached to specific edges, these semantics describe the *entire* context, i.e., the entire tree minus the subtree below the edge in question.

## 4 Empirical results

To see how subtree crossover affects the distribution of both context and subtree semantics, we did multiple runs on five different problems: Even Parity problems with 2, 3, 4, and 6 bits (2-EP, 3-EP, 4-EP, and 6-EP), the 6-bit multiplexer (6-MUX) problem, and flat fitness on four bits (4-Flat).

### 4.1 Parameters and data collected

For each problem we did 38 independent runs using the parameters listed in Table 3. Since we weren't particularly interested in maximizing our chances of solving the problems, no effort was made to tune our parameter choices.

Parent semantics	Arg semantics (x)	(and x #)	(or x #)	(nand x #)	(nor x #)
0	0	0	0	0	0
0	1	0	0	0	0
1	0	1	1	1	1
1	1	1	1	1	1
+	0	0	+	1	-
+	1	+	1	-	0
-	0	1	-	0	+
-	1	-	0	+	1

**Table 2.** The context semantics for *and*, *or*, *nand*, and *nor*. See the text for details.

Parameter	Value
Function set	Binary AND, OR, NAND, and NOR
Terminal set	$x_0, x_1, \dots, x_{n-1}$ , where $n$ is the number of variables (or bits) in the problem.
Control strategy	Generational
Population size	1000
Initialization	PTC2 [7], with equal proportions of sizes 50, 70, and 100 nodes and maximum initial depth of 10
# of generations	500
Tournament size	2
XO Probability	1
XO bias away from leaves	None (all nodes are equally likely)
Maximum size after XO	500 (If the resulting child is too large, then new parents are chosen independently and process begins again.)

**Table 3.** Parameters used in our runs. The crossover probability of 1 means that subtree crossover was the *only* recombination operator used in these runs, i.e., there was no mutation and no reproduction.

For each test problem except the flat fitness case (4-Flat) the fitness was the number of test cases handled correctly, with higher values being better. For 4-Flat the fitness was constant for all individuals, so there was no selection bias in those runs.

Along with traditional data such as fitnesses and tree sizes, we also tracked two kinds of data specific to building block semantics:

**Proportion of fixed contexts** The percentage of contexts (over all contexts in every individual in the population) that are completely fixed, i.e., all the positions are either a 0 or 1. This means that any crossover using this context is going to be a semantic no-op, yielding an offspring with the same semantics as the context regardless of the subtree inserted. High proportions of fixed contexts suggest that a run has essentially stalled, with very little effective search going on anymore. Note that this is not *necessarily* a bad thing – if the run has found the target, for example, then fixing strongly is not necessarily problematic. However, if the target has yet to be found then a large proportion of fixed contexts is probably undesirable.

**Proportion of compatible contexts** The percentage of contexts (over all contexts in every individual in the population) that are *compatible* with the target context. A compatible context is a context that has the possibility of producing a target solution, meaning that any fixed values in the context must match the corresponding values in the target. (A fully fixed context that will always produce a target solution is also considered a compatible context.) If a context is incompatible, it is guaranteed to not produce a solution if it is used in a crossover event. Therefore low proportions of compatible contexts suggest that a run is unlikely to succeed, at least in the near term.

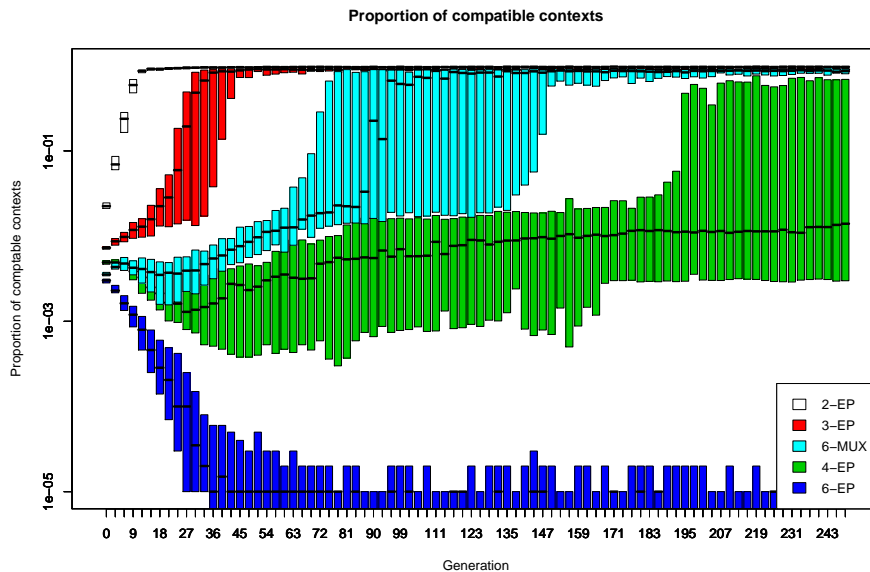
## 4.2 Results

Not surprisingly, the 2-EP and 3-EP problems were quite easy and had 100% success rates. The 4-EP runs found a solution 20 out of 38 times; two of these 20 runs, however, later lost their successful solution (we weren't using any form of elitism) and ultimately converged on functions with fitness 15. None of the 6-EP runs solved the problem, while all of the 6-MUX were successful, supporting the idea that 6-MUX is generally much easier to solve than 6-EP with this function set. We weren't particularly interested in the success of the runs, but the relative difficulty of these problems (as demonstrated by these success rates) is clearly reflected in many of the results below.

**Compatible contexts** Figure 4 plots the proportion of compatible contexts for all five non-flat problems (Flat fitness is not included here because it does not make sense to talk about compatible contexts when there is no target to be compatible with.) The proportion of compatible contexts for the relatively easy 2-EP and 3-EP problems quickly jumps to nearly 1 as solutions are found, and subsequent bloat leads to large trees with many (correctly) fixed contexts. 6-MUX, which also has a very high success rate, shows a similar behavior, although it takes a little longer for it to find a solution so the proportion of compatible contexts does not rise as soon.

The proportion of compatible contexts for most 6-EP runs quickly drops to effectively zero, indicating that those runs have converged on local optima that are inconsistent in a significant way with the target. This suggests that those runs are very unlikely to ever find a solution, as it would presumably take a significant jump to move from the peak they've converged on to the target peak. What's not indicated in the plot (because the whiskers and outliers are suppressed) is that there are handful of runs (4 of 38) with considerably higher proportions of compatible contexts. The proportions in these runs are around 0.01, putting them in the lower range of the plotted data for the 4-EP runs. None of the 6-EP runs succeed in finding a solution in the 500 generations we used, but it seems plausible that this small group of runs would be the most likely to eventually find a solution if given more time.

The higher persistent variance in the percentages for the 4-EP runs presumably reflects the fact that several of the runs have succeeded, while others remain stuck at local optima. The fact that many of the (so far) unsuccessful 4-EP runs have percentages that are well above zero (around 0.01) suggests, however, that those runs may still have some chance of eventually jumping to the solution. In fact 6 (of 38) 4-EP runs go on to find



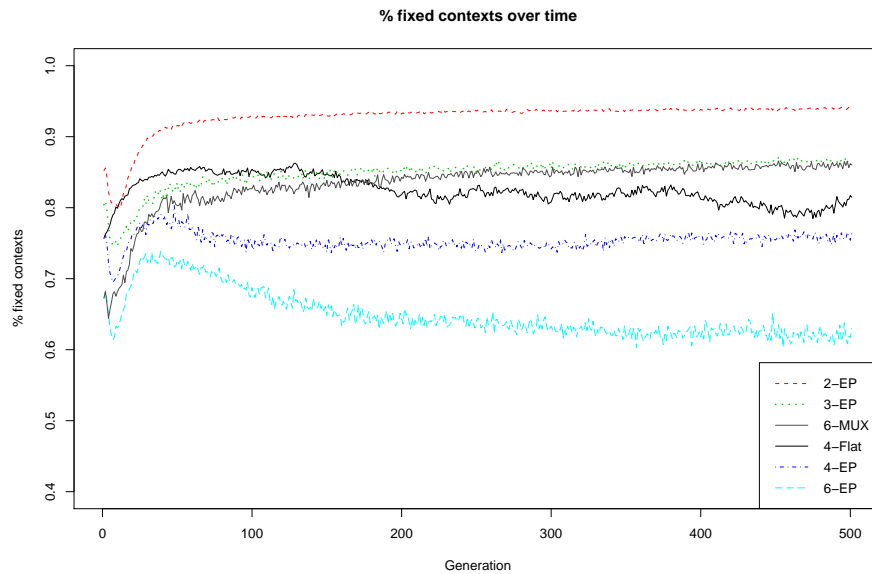
**Fig. 4.** Boxplots of the percentage of compatible contexts for the first 250 generations across the 38 runs for all five problems. The whiskers and outliers have been suppressed, so only the median and two middle quartiles are plotted. Note the log scale on the y-axis.

solutions after the final generation (250) plotted in Figure 4, although the distribution of compatible contexts isn't changed significantly by those successes.

The percentage of compatible contexts provides an upper bound on the probability of constructing a target solution via sub-tree crossover (since doing so will require both a compatible context and an appropriate subtree). Our empirical data shows, at least for these test conditions, that the probability of constructing a target is in fact almost the same. (See [10] for more.)

**Proportion of fixed contexts** Figure 5 shows the median proportion (over the 38 runs) of contexts that are *completely* fixed for each of our six test environments. Remember that a completely fixed context is one where the return value *in all cases* is completely determined and will be unaffected by the details of the particular subtree inserted into the context. Thus a crossover using a completely fixed context is guaranteed to generate an offspring with the same semantics as the root (or context) parent, and no semantic exploration will have occurred.

It is worrying, then, that in each of the six cases, the proportion of fixed contexts in the population exceeded 60% at all times, and was typically greater than 75%. This means that a great majority of all crossover events are (at least in the short term) useless,



**Fig. 5.** Median proportion of contexts that are fixed vs. generation for all six test environments. Note the y-axis doesn't continue all the way down to 0. Also, the 3-EP and 6-MUX plots almost completely overlap for the second half of the plots, so they almost look like a single line in the graph.

as they don't explore any new semantic space.<sup>2</sup> The two harder problems (4-EP and 6-EP) had the lowest percentage of fixed contexts, but even in these problems well over half of all crossovers were guaranteed to be ineffective, regardless of the subtree chosen for crossover.

All the problems with non-trivial fitness (everything but the 4-Flat) show a small drop in the proportion of fixed contexts in the first few generations, with the proportion of fixed contexts then climbing again from around generation 10 to around generation 30. After 200 generations the median proportion of fixed contexts become fairly flat with the exception of 4-Flat, where there appears to be a certain amount of drift.

Figure 5 also shows that the proportion of fixed contexts in the initial generations are driven (for these problems) by the number of inputs. 4-EP and 4-Flat for example, both start at the same proportion, but diverge almost immediately, with 4-Flat's proportion of fixed contexts growing gently instead of dipping initially as is the case with 4-EP. Similarly 6-EP and 6-MUX start out similarly for a few generations, and then diverge as the 6-MUX runs begin to gain traction on the problem while the 6-EP runs continue to flounder.

<sup>2</sup> It is possible that such a crossover creates new structure that, when sampled in later generations, will lead to an important discovery. Given the lack of any immediate semantic effect, however, such benefit is quite random and unguided by the fitness function.

## 5 Discussion

**Approximation and extension to non-boolean domains** As mentioned in Section 3, these techniques are currently constrained to boolean problems, and even for boolean problems they scale badly to large number of variables. Of the new measures mentioned above (proportions of compatible contexts and proportion of fixed contexts), the proportion of fixed contexts is probably the easiest to generalize to problems with more variables and non-boolean domains. Given that the proportion of fixed contexts appears to have potential as an indicator of problem difficulty (see Figure 5), being able to at least estimate it might have value even if it can't be computed exactly.

One could estimate the proportion of fixed contexts on larger boolean problems, for example, by randomly sampling contexts (perhaps as part of the existing crossover process), and checking to see if they're fixed. One could, for example, insert each of the  $2^{2^N}$  different subtree semantics at the crossover point in the context to see if any changed the semantic value of the context. It is sufficient, however, to only check any two complementary subtree semantics (e.g., the constants *true* and *false*). If the (boolean) context is in fact not completely fixed, then it must contain at least one '+' or '-', which means it will have different values for at least one set of inputs when complementary subtree semantics are inserted. One would still need to check all  $2^N$  possible inputs to know for certain if the context is fixed, but for large  $N$  one could further approximate by sampling the set of possible inputs.

For non-boolean domains the problem becomes more complex, especially in continuous domains like symbolic regression over the reals. With real-valued functions, for example, there is potentially a whole spectrum of fixation. A completely fixed context might (as in the boolean case) be completely independent of the inserted subtree, while a "nearly fixed" context might change, but only by very small amounts. There's also no simple analogy to the complementary subtree semantics (such as *true* and *false*) to simplify the sampling of the subtree semantics. Still, it seems likely that sampling a few constant values at the insertion point across several sets of input values would provide a useful approximation of the "fixedness" of a context, even in a real-valued problem.

**Designing new operators and models** In many ways the high proportion of fixed contexts (Figure 5) is quite disheartening, as it suggests that the majority of crossover applications are accomplishing nothing (at least in the short term). We could, therefore, use these results to guide the design of new recombination operators that would deliberately work to reduce the proportion of fixed contexts, hopefully increasing the exploratory power of our system.

Experiments, for example, with a crossover operator that avoids choosing approximately fixed contexts (essentially the same as the approach taken in [2]) don't appear to improve the likelihood of finding solutions and can significantly slow down the evaluation of individuals. It does, however, provide a very effective bloat control mechanism, and it's possible that modifications of this idea, or combinations with other operators, could improve performance.

We could see the data reported here as the result of a co-evolutionary system where there is a serious problem of *disengagement* [3], where one population (the contexts)

“beats” the other (the subtrees). In this case the population of contexts reaches such a high proportion of fixedness that the subtrees are essentially frozen out of the process. The restricted crossover operator defined above, then, could be seen as a means of combating disengagement by increasing the chances that a context distinguishes among subtrees instead of simply dominating them [3]. One could extend this observation to build an explicit co-evolutionary model of subtree crossover in GP. Obviously in standard GP the “population” of contexts and the “population” of subtrees are linked on several levels (any particular node is a component of numerous contexts and numerous subtrees at the same time), but given the apparent dominance of contexts in determining the likelihood of success, detaching the two might in fact prove helpful rather than problematic.

Alternatively, one could see our distributions of context and subtree semantics as the basis for a co-evolutionary estimation of distribution (EDA) algorithm [6].

## 6 Conclusions

In this paper we have presented a novel means of exactly and compactly describing (for boolean problems) the semantics of the two tree components combined by subtree crossover: the context (the root parent with a subtree removed) and the subtree being inserted into that context. This allows us to completely describe the semantic action of subtree crossover, and enumerate in a syntax independent fashion the occurrence of different context and subtree semantics in a population. The resulting data strongly suggest that the distribution of context semantics are key in the success (or failure) of runs. Our results also make it clear that the proportion of fixed contexts in these problems is very high (typically over 75%), indicating that the substantial majority of subtree crossover events actually perform no search in the semantic space.

As well as shedding valuable new light on the impact of subtree crossover, these tools and results suggest a number of ideas for new operations and approaches to genetic programming that would be based on theoretical and empirical understanding rather than simple guesswork.

## Acknowledgments

Nic would like to thank the organizers and participants in Dagstuhl Seminar 06061 for providing a venue to present early versions of this work, for valuable feedback and ideas. Nic would also like to thank Riccardo Poli and the University of Essex for providing such a productive sabbatical environment.

## References

1. P. J. Angeline. Subtree crossover: Building block engine or macromutation? In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

2. T. Blickle and L. Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik (MPI-I-94-241).
3. E. D. de Jong and J. B. Pollack. Ideal evaluation from coevolution. *Evolutionary Computation*, 12(2):159–192, 2004.
4. W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.
5. W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O’Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.
6. P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, 2002.
7. S. Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, Sept. 2000.
8. S. Luke and L. Spector. A revised comparison of crossover and mutation in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213, University of Wisconsin, Madison, Wisconsin, USA, 22–25 July 1998. Morgan Kaufmann.
9. N. F. McPhee and J. D. Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA, 15–19 July 1995. Morgan Kaufmann.
10. N. F. McPhee, B. Ohs, and T. Hutchison. Enumerating building block semantics in genetic programming. Technical report, University of Minnesota, Morris, 2007. Available at [http://www.morris.umn.edu/academic/fclt/Working Papers/Morris.WP\\_3.1.pdf](http://www.morris.umn.edu/academic/fclt/Working Papers/Morris.WP_3.1.pdf).
11. N. F. McPhee and R. Poli. Using schema theory to explore interactions of multiple operators. In W. B. Langdon, et al, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 853–860, New York, 9–13 July 2002. Morgan Kaufmann Publishers.
12. U. M. O’Reilly and F. Oppacher. The troubling aspects of a building block hypothesis for genetic programming. Working Paper 94-02-001, Santa Fe Institute, 1399 Hyde Park Road Santa Fe, New Mexico 87501-8943 USA, 1992.
13. R. Poli. Is crossover a local search operator? Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97, 20 July 1997.
14. R. Poli and W. B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998.
15. R. Poli and N. F. McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evolutionary Computation*, 11(1):53–66, Mar. 2003.
16. R. Poli and N. F. McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003.
17. R. Poli and J. Page. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. *Genetic Programming and Evolvable Machines*, 1(1/2):37–56, Apr. 2000.
18. J. P. Rosca. Analysis of complexity drift in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA, 13–16 July 1997. Morgan Kaufmann.
19. K. Sastry, U.-M. O’Reilly, D. E. Goldberg, and D. Hill. Building block supply in genetic programming. In R. L. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practice*, chapter 9, pages 137–154. Kluwer, 2003.