2007

Enumerating Building Block Semantics in Genetic Programming

Nicholas Freitag McPhee

Brian Ohs

Tyler Hutchinson

UNIVERSITY OF MINNESOTA MORRIS

Working Paper Series

Enumerating building block semantics in genetic programming

Nicholas Freitag McPhee
Professor of Computer Science

Brian Ohs
Student

Tyler Hutchison
Student

Faculty Center for Learning and Teaching
Rodney A. Briggs Library

Volume 3 Number 1
2007

Faculty and Student Research

University of Minnesota, Morris
600 East 4th Street
Morris, MN 56267

Faculty Center for Learning and Teaching
Engin Sungur, Director
Linda Pederson, Executive Administrative Specialist
Rodney A. Briggs Library
LeAnn Dean, Director
Peter Bremer, Reference Coordinator
Matt Conner, Instruction Coordinator
Steven Gross, Archivist
Michele Lubbers, Digital Services Coordinator
Shannon Shi, Cataloging Coordinator

Working Paper Series
Volume 3 Number 1
2007

Faculty Center for Learning and Teaching
Rodney A. Briggs Library
University of Minnesota, Morris

This Working Paper Series allows the broader dissemination of the scholarship of the University of Minnesota-Morris Faculty, staff, and students. It is hoped that this Series will create a broader and much more accessible forum within the borderless academic community, and will further stimulate constructive dialogues among the scholar-teachers at large.

Enumerating building block semantics in genetic programming

Nicholas Freitag McPhee
Professor of Computer Science

Brian Ohs
Student

Tyler Hutchison
Student

University of Minnesota, Morris
Morris, MN 56267, USA

Working Paper Volume 3, Number 1

Enumerating building block semantics in genetic programming

Nicholas Freitag McPhee Brian Ohs Tyler Hutchison
Division of Science and Mathematics
University of Minnesota, Morris
600 E. 4th Street
Morris, Minnesota
USA - 56267
{mcphee,ohsx0004,hutc0125}@morris.umn.edu

November 16, 2007

Abstract

This report provides a collection of definitions for the semantics of subtrees and contexts as manipulated by standard sub-tree crossover in genetic programming (GP). These definitions allow us to completely and compactly describe the exact semantics of the components manipulated by sub-tree crossover, and the semantic results of those interactions. Subsequent work shows how these definitions can be used to collect valuable data about the available diversity in a GP population and the opportunities available to sub-tree crossover.

1 Introduction

Like most evolutionary computation recombination operators, genetic programming (GP) sub-tree crossover takes components from two parents (syntax trees) and combines them to produce a new offspring. In sub-tree crossover we construct a new offspring C by replacing a randomly chosen sub-tree from parent A with a random sub-tree from parent B (see Figure 1). To understand the possibilities afforded by sub-tree crossover, we need to be able to characterize which sub-trees can be chosen from B, and where they can go in A.

The power of sub-tree crossover resides in its ability to effectively combine components that will generate new and better trees, thereby giving the evolutionary process a gradient it can capitalize on. Conversely this is also its flaw, as it is just as capable of producing a genetic mule as a gem. It would thus be valuable to be able to describe the semantics of tree components in a way which allows us to accurately and completely describe the semantic effects of sub-tree crossover.

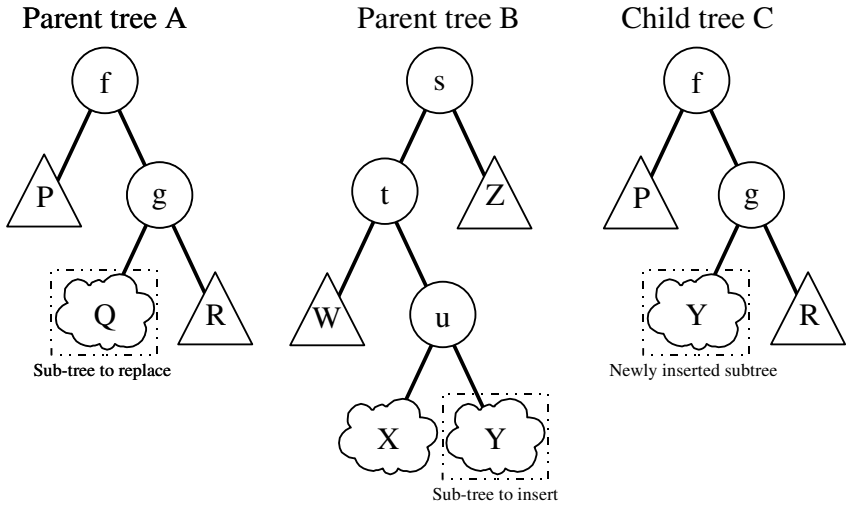

Figure 1: A randomly selected subtree from B is inserted randomly into A, replacing the existing subtree at the point of insertion. The circles nodes are internal (function) nodes, the triangles represent arbitrary subtrees, and the cloud shapes represent subtrees directly involved in the crossover operation.

To help simplify the discussion, we define a *context* to be a tree with some specific (but arbitrary) subtree removed (e.g., Fig 2).[1] Given this definition, describing the semantic effects of sub-tree crossover reduces to describing the semantics of sub-trees (e.g., Fig 1), the semantics of contexts, and their interactions.

Throughout this paper we're going to focus on the boolean domain, i.e., trees that represent boolean functions. It is valuable to work in a small (finite) domain, because it becomes much easier to compute and catalogue the complete semantics of the sub-trees and contexts involved. These ideas can be extended to other finite domains fairly easily, but the combinatorics will become computationally prohibitive quite quickly.

Sec 2 discusses how to completely describe the semantics of a subtree by enumerating its values over the set of inputs. We are able to do this for every subtree in the population. Sec 3 describes the semantics of a context independent the semantics of the subtree which will be substituted at the point of insertion. Sec 4 gives several examples and semantic descriptions of subtrees/contexts. Sec 5 shows how we can describe the effects of a sub-tree crossover using semantics, without knowledge of the underlying stuctural or syntactic properties of the trees involved. Sec 6 discusses some limitations of the method from Sec 5, due

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

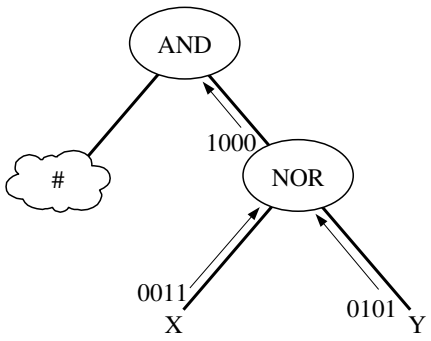

Figure 2: The ("`#`") symbol represents the unknown branch of the subtree.

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 four boolean functions. 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. Here we list the semantics for the four boolean functions used in this study; for example, the semantics of `(and x y)` can be represented with the string "`0001`".

to a rapidly increasing search space size, and we provide some conclusions in Sec 7.

2 Semantics of subtrees

Following the ideas presented by Poli and Page in sub-machine code GP [4], we can completely specify the semantics of a (sub)tree by enumerating its value across every possible set of input values. Table 1, for example, enumerates all the possible inputs and outputs for four common binary functions, taking "`0`" to be *false* and "`1`" to be *true*. The essential observation by Poli and Page was that instead of seeing, e.g., the third column as four applications of `and`, we can instead see it as a *single* application of a function that maps 2 strings of 4 bits ("`0011`" and "`0101`") to an output string ("`0001`") that represents the bitwise conjunction of the two input strings. Thus the semantics of the tree "`(and x y)`" can be completely and precisely represented by "`0001`". Similarly we can represent the semantics of *any* subtree by enumerating its values over all the possible inputs.

This allows for a complete characterization of the semantics of any boolean

(sub)tree. Assume that two trees S_0 and S_1 have the same semantics, and tree T contains S_0 as a sub-tree. We can then replace the occurrance of S_0 in T with S_1, and the semantics of T will remain unchanged. S_0 and S_1 may have very different syntactic structures, yet as long as they provide the same semantic contribution to the semantics of tree T, the semantics of T will not be changed.

This compact representation allows us to enumerate the semantics of *all* the sub-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.

As a notational device in our tree diagrams we will typically associate the semantics string of a subtree with the edge extending *up* from the root of the subtree, as is done in Figure 2.

3 Semantics of contexts

Being able to enumerate the semantics of all the sub-trees in the population is arguably the simple half of characterizing the semantics of sub-tree crossover. The somewhat more interesting part is characterizing the semantics of contexts, i.e., trees with one missing sub-tree representing the contribution of the root parent in sub-tree crossover.

In general we won't know the semantics of a tree with an unspecified sub-tree, since the details of that subtree will usually affect the semantics of the entire tree. Some contexts, however, depend less on the details of their open subtree than others. The context `(and false #)` (the left diagram in Figure 3) is *always* going to return *false*, regardless of which subtree we insert for the "`#`". Further, we know from experience that genetic programming has strong tendencies towards the creation of such contexts [3, 2, 1].

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 (in the boolean case, 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 input (or at every position).

In the boolean domain the semantics of a context depends on the details of the inserted subtree in a systematic manner. Consider, for example, the context `(and true #)` (the left diagram in Figure 4). Here the value of this context will be the *same* as the value of whatever subtree we insert for the "`#`" since $true \wedge x = x$. 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 #)` (the right diagram in Figure 4). Here $\neg(true \wedge x) = false \vee \neg x = \neg x$, so 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. (Section 4 provides several examples in more detail.)

While the interactions between contexts and subtrees can be quite complex,

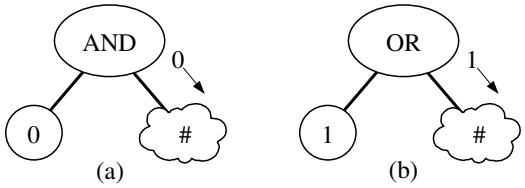

Figure 3: *These contexts have a determined output. That is, they do not derive any part of their meaning from the inserted subtree ("#").*

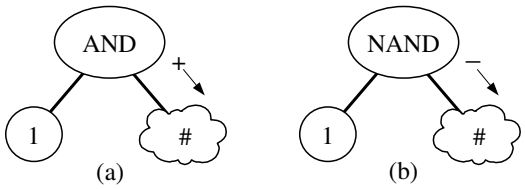

Figure 4: *These contexts do not have a determined output, so their semantics will be different given a different set of inputs.*

in the case of boolean functions there are only four options for a context on a specific set of inputs:

- The semantic value of the context is "0", regardless of the details of the inserted subtree. See Figure 3 (a) for an example.
- The semantic value of the context is "1", regardless of the details of the inserted subtree. See Figure 3 (b) for an example.
- The semantic value of the context is the same as the semantic value of the inserted subtree on those inputs; we denote this with a "+". See Figure 4 (a) for an example.
- The semantic value of the context is the logical negation of the semantic value of the inserted subtree on those inputs; we denote this with a "–". See Figure 4 (b) for an example.

One other difference between subtree semantics and context semantics is which components need to be taken into consideration when computing the semantics. In the case of subtree semantics, the semantics are a function of the (boolean) operator and the value of its arguments; they are completely independent of where that subtree might be located. In the case of context semantics, the case is slightly more complex. While we associate the semantics with the edge going down to the insertion point, they are still a function of the entire tree around that point. In particular they depend on three things (see Fig. 5):

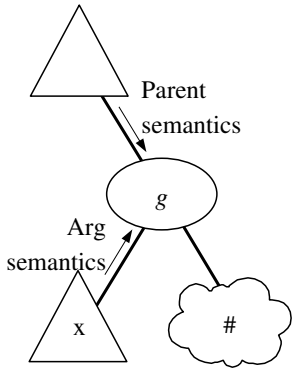

Figure 5: An illustration of the interaction of the different components in computing the context semantics. Here we have a tree with some subtree removed (the insertion point, indicated by the "#" in the lower right). g is the parent node of the insertion point, and x represents the other argument of 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.

- 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 5).
- The subtree semantics of the other argument (x) of the operator g (the "Arg semantics" in Figure 5).

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.

Table 2 enumerates the cases for the boolean functions *and* and *or*. Table 3 enumerates the cases for the boolean functions *nand* and *nor*. In the last line of Table 3, 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 "+".

Unlike the subtree case (Table 1), we can't simply read the context semantics off these tables. Instead the tables enumerate the cases necessary to *compute* the context semantics for each of the 2^N possible input values (assuming N input variables).

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

Parent semantics	Arg semantics (x)	`(and x #)`	`(or x #)`
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	1
+	0	0	+
+	1	+	1
-	0	1	-
-	1	-	0

Table 2: The context semantics for *and* and *or*. See the text for details.

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

Table 3: The context semantics for *nand* and *nor*. See the text for details.

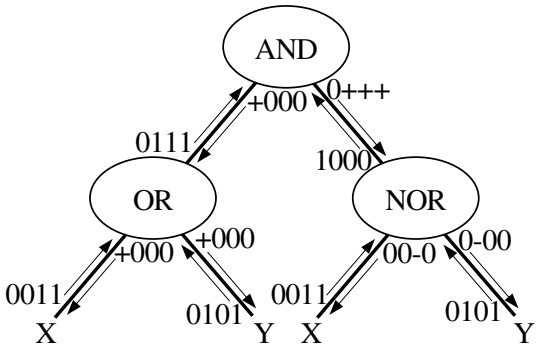

Figure 6: A sample syntax tree showing both subtree and context semantics. The arrows pointing upward are the semantics of the subtree below them, e.g., the semantics of `(or x y)` is "`0111`". The arrows pointing downward are the semantics of the context obtained by removing the subtree below the arrow, e.g., the semantics of `(and # (nor x y))` is "`+000`". See Sec 4 for details.

indicate the semantics of all the possible contexts in a tree on a single diagram as is done in Figure 6. 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 Examples

In this section we will go over some examples of the computation of both subtree and context semantics in more detail.

As our first example, consider the tree in Figure 6. If we remove `(or x y)` we obtain the context $C =$ `(and # (nor x y))`. The subtree semantics of `(nor x y)` is "`1000`"; this means `(nor x y)` will return *true* when both `x` and `y` are *false*, and *false* in the other three cases. Because the root of C is an `and` node, the value of C is guaranteed to be *false* for those last three possible inputs regardless of what subtree we insert as the left argument of the root node. Thus the semantics for this context are completely determined for these last three input cases. What then about the first input, where both `x` and `y` are *false*? In that case the value of C is going to be affected by the value of the subtree at "`#`", but in a very specific way, namely the value of C on those inputs will be *equal* to the value of the inserted subtree on those inputs, denoted with a "`+`". Thus the semantics of the context C are "`+000`".

As a second example, consider again the tree in Fig. 6, but this time with the rightmost `x` removed, yielding the context $C' =$ `(and (or x y) (nor # y))`. The subtree semantics of the rightmost `y` is "`0101`", and the context semantics

of `(and (or x y) #)` is "`0+++`". These semantics, combined with those of the `nor` function (Table 3), yield an overall context semantics for C' of "`00-0`". This means that C' will yield *false* in all cases except when `x` is *true* and `y` is *false*, regardless of what subtree is inserted in place of the "`#`". If, however, `x` is *true* and `y` is *false*, the value of C' will be the logical negation of the value of the inserted subtree.

5 Semantic effect of sub-tree crossover

Given these definitions, we can compute the (semantic) results of an application of sub-tree crossover using only the semantics of the sub-tree and context and without any knowledge of the structural or syntactic properties of the trees involved.

As an example, consider a crossover where `(or x y)` in the tree in Figure 6 is replaced by `(nand x y)`. Thus the context is `(and # (nor x y))`, with semantics "`+000`", and the sub-tree to be inserted is `(nand x y)`, with semantics "`1110`". We can, then, compute the semantics of the new tree `(and (nand x y) (nor x y))` entirely via a combination of the semantics of the components with no reference to syntax of the new tree. In particular, the context semantics ("`+000`") provide the value of the new tree ("`0`") for the last three input cases, regardless of the inserted subtree. Since the first character in the context semantics is "`+`", the value of the new tree for the first input case will be the value of the inserted subtree for that case, or (in our example) "`1`". Thus the resulting semantics for the new tree are "`1000`", meaning that the tree returns true for the first input case (where `x` and `y` are both *false*), and *false* in all other cases. Note that both the context and inserted sub-tree in this example could have been much more structurally complex, but as long as they had the same semantics the *semantic* result would have been the same.

In general we can compute the semantics of trees generated via sub-tree crossover by simply combining the semantics of the context and the inserted sub-tree as above. Fixed positions in the context semantics remain fixed at those values, and non-fixed positions take on either the value (for "`+`") or the negated value (for "`-`") of the inserted sub-tree at that position.

6 Context semantics and search space size

Assume we have N binary inputs. Then there are 2^N possible input patterns. The semantics of a boolean function of N inputs can be represented by a string of 2^N output bits. Subtree semantics are binary strings of length 2^N, so there are $2^{(2^N)}$ possible subtree semantics. Since context semantics are based on an alphabet of four characters, there exist $4^{(2^N)}$ possible context semantics.

For a six-bit problem, for example, there are over 10^{38} possible contexts, showing that any reasonably sized population is only able to explore a very limited subset of this extremely large search space. Since the size of both the

subtree and context search spaces are doubly exponential in the number of inputs, this disparity between the search space size and number of subtrees and contexts in the population will continue to increase as the number of inputs rises.

7 Conclusions

In this report we extended Poli and Page's ideas from sub-machine code GP [4] to provide the ideas of both sub-tree and context semantics. These provide a compact and exact mechanism for describing the semantics of the components used in sub-tree crossover and its results.

These definitions give us a useful mechanism for studying sub-tree crossover and its effects. These ideas could also be used to define crossover in new ways, where we see individuals in a population not as syntax trees, but instead as collections of semantics. Sub-tree crossover then becomes a process whereby semantics are chosen from the two parent trees, and combined to generate the semantics for the resulting offspring.

Note, however, that these ideas only work on small, finite domains such as boolean functions, making them perhaps better suited for theoretical investigations than production systems.

8 Acknowledgments

Nic would like to thank Riccardo Poli for early conversations on sub-machine code GP, which led to the early ideas regarding context semantics. Nic would also like to thank the participants at Dagstuhl Seminar 06061 on the Theory of Evolutionary Algorithms (Feb. 2006) for their many ideas in response to an early version of these ideas.

References

[1] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[2] 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.

[3] 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.

[4] 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.

University of Minnesota Morris

Working Paper Series

Volume 1

Ritual and Ceremony In a Contemporary Anishinabe Tribe, Julie Pelletier

The War for Oil or the American Dilemma of Hegemonic Nostalgia?, Cyrus Bina

The Virgin and the Grasshoppers: Persistence and Piety in German-Catholic America, Stephen Gross

Limit Orders and the Intraday Behavior of Market Liquidity: Evidence From the Toronto Stock Exchange, Minh Vo

Volume 2

Specialization of Java Generic Types, Sam DeVier, Elena Machkasova

A Call-by-name Calculus of Records and its Basic Properties, Elena Machkasova

Computational Soundness of a Call by Name Calculus of Recursively-scoped Records, Elena Machkasova

Genetic Memory and Hermaphroditism: Trans-Realism in Eugenides's Middlesex, Edith Borchardt

Limit Cycle Displacement Model of Circadian Rhythms, Van D. Gooch

Volume 3

Enumerating building block semantics in genetic programming, Nicholas Freitag McPhee, Brian Ohs, Tyler Hutchison