Creating High-level Components with a Generative Representation for Body-Brain Evolution

Gregory S. Hornby [*] Jordan B. Pollack

DEMO Lab.
Computer Science Department
Brandeis University
Waltham, MA 02454-9110
hornby,pollack@cs.brandeis.edu

Abstract

One of the main limitations of scalability in body-brain evolution systems is the representation chosen for encoding creatures. This paper defines a class of representations called *generative representations*, which are identified by their ability to reuse elements of the genotype in the translation to the phenotype. This paper presents an example of a generative representation for the concurrent evolution of the morphology and neural controller of simulated robots, and also introduces *Genre*, an evolutionary system for evolving designs using this representation. Applying *Genre* to the task of evolving robots for locomotion and comparing it against a non-generative (direct) representation, shows that the generative representation system rapidly produces robots with significantly greater fitness. Analyzing these results shows that the generative representation system achieves better performance by capturing useful bias from the design space and by allowing viable large scale mutations in the phenotype. Generative representations thereby enable the encapsulation, coordination, and reuse of assemblies of parts.

Key words: Body-brain evolution, Generative representations, Representation, Lindenmayer Systems (L-systems)

[*] corresponding author

1 Introduction

The evolution of artificial creatures has come a long way since Dawkins' evolution of two-dimensional shapes [8]. Controllers have been evolved for fixed morphologies: first with stimulus-response rules for animated, articulated creatures [28] [35]; then with neural controllers [10]; and more recently for the dynamic gait of a physical, quadruped robot [11]. More true to the spirit of artificial life is the evolution of both body and brain, starting with Sims' evolution of block creatures – for swimming, walking and light seeking [33], as well as competing for the possession of a box [32] – and Ventrella's evolution of stick figures for walking [36]. This has been followed by the evolution of walking creatures by [20] [26] and [6], summarized in [34]. For the most part, this newer work has not managed to surpass the complexity of Sims' original block creatures. We propose that one source for this difference comes from the differences in representations used to encode creatures. This paper identifies a class of representations, called *generative representations*, and investigates their impact on the problem of evolving locomoting robots.

Here we consider genotypic representations as a kind of programming language. With this analogy, the fundamental properties of programming languages can be used to understand and classify different approaches to the underlying representations of evolutionary systems. From [2], the fundamental elements of programming languages are:

- **Combination**: Languages create the framework for the hierarchical construction of more powerful expressions from simpler ones, down to atomic primitives.
- **Control-flow**: All programming languages have some form of control of execution, which permits the conditional and repetitive use of structures.
- **Abstraction**: Both the ability to label compound elements (to manipulate them as units) and the ability to pass parameters to procedures are forms of abstraction.

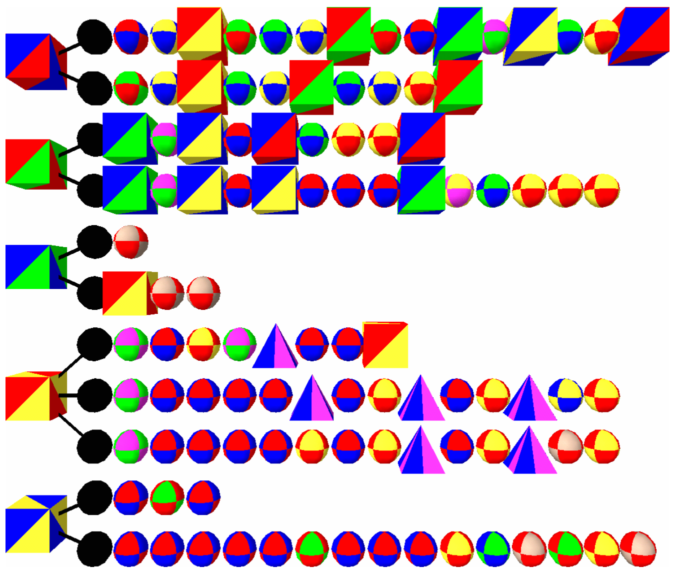
In implementation, these elements can be parceled out to different mechanisms, such as branching, variables, bindings, recursive calls, but are nonetheless present in some form in all programmable systems. Some of these basic properties have also been shown to have analogues in biological systems: phenotypes are specified by combinations of genes; the expression of one gene can be turned on/off by the expression of another gene [24]; and an upstream protein can control a downstream protein's activity through a signalling pathway [3].

We can use the properties of programming languages to understand and classify design representations. A fundamental distinction is whether a representation used in evolution is *direct* or *generative*. In a direct representation there is a one-to-one mapping from each representational element of the genotype to some component in the phenotype. A generative representation, on the other hand, is one which is capable of reusing elements of the genotype in the construction of phenotypic components. This reuse of components can come from iteration or from abstraction. Continuing with the programming language analogy, a generative representation is a kind of language such that heritable genotypic elements, together with a translation or compilation process, control the expression of phenotypic components. For instance, the processes may interpret genotypic information as constructs like loops, procedure calls, variables, parameters, etc, to control genetic expression. Thus with a generative representation, the individuals in the population are programs in a language whose instructions control the flow of construction commands for creating each design.

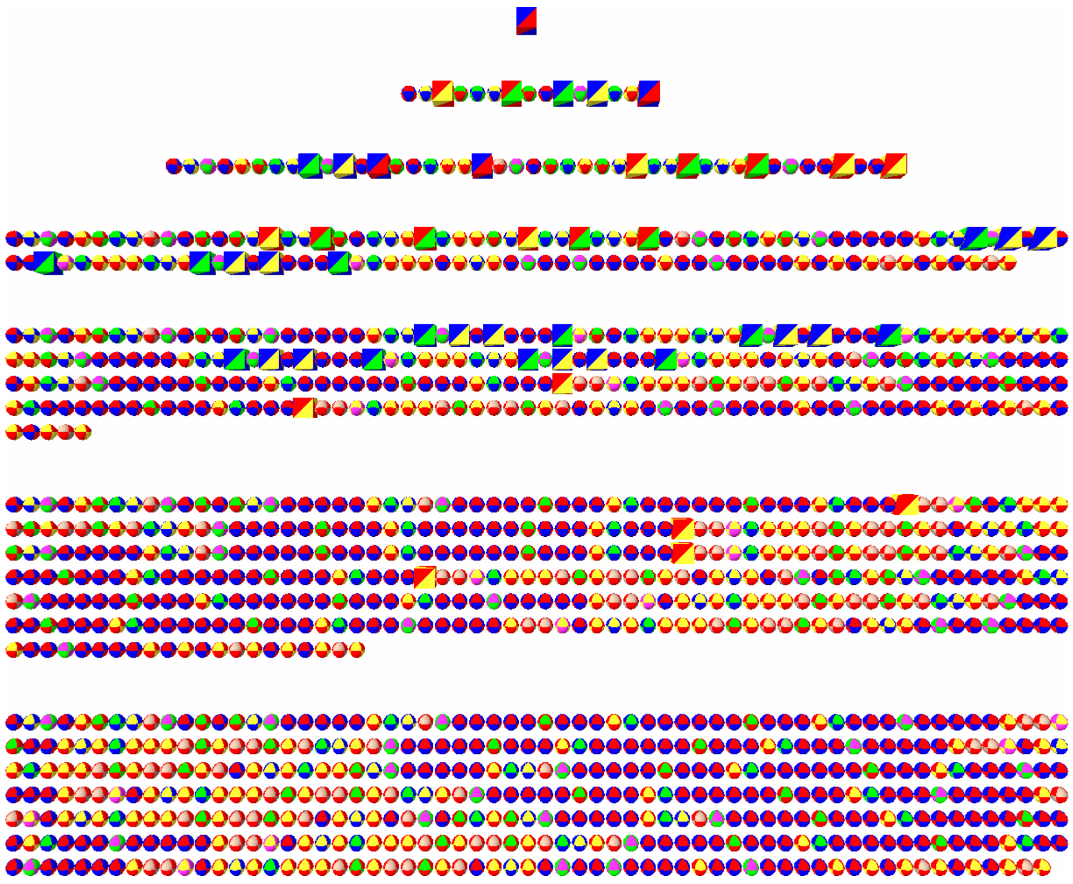
Here we use a Lindenmayer system (L-system) [25] as the generative representation to encode creatures. L-systems are a grammatical rewriting system in which the rewriting rules are applied in parallel to all symbols in a string. Figure 1.a shows a graphical representation of the rules for our generative representation. In these images cubes represent procedure calls, black spheres represent conditionals, triangles represent the repeat operator and spheres represent construction commands. Figure 1.b shows the sequence of assembly strings generated by this set of rules. The sequence begins with the first cube (here a blue and red one) and the sequence of strings below it are the strings generated after each iteration of parallel replacement. The last string of symbols is the assembly procedure used for constructing a robot.

In this paper we explain the details of this generative representation system and show that during evolution the system can incorporate useful biases from a robot design problem. This incorporation of bias comes through reuse of elements in the genotype. By continually creating new components and including these into the variations available, the unit of variation scales with the complexity of the design. Because changing the definition of a reused assembly of parts results in a change in all occurrences of that assembly, the design system has the ability to make coordinated changes in several parts of a design simultaneously.

The system we use for the evolution of robots is *Genre*, which stands for *Generative representations*. This system has been used to evolve tables [14] and oscillator controlled, robots [12] [15], demonstrating in both cases that an evolutionary algorithm (EA) using a generative representation outperforms an EA using a direct representation. Here we describe the application of *Genre* to neural network controlled robots, and we call our robots, *genobots* (for generatively encoded robots). Moving to neural networks allows us to generate more complex movement patterns and allows for later work to include sen-



(a)



(b)

Fig. 1. A graphical version of the generative representation, (a); along with the sequence of assembly strings it produces, (b).

sors in order to evolve robots with reactive controllers. With *Genre* we evolve neural-network controlled locomoting robots and compare search performance against a system that uses a direct representation. We find that the generative representation system rapidly produces robots with significantly greater fitness. Analyzing these results shows that the generative representation system achieves better performance by capturing useful bias from the design space and by allowing viable large scale mutations in the phenotype. Generative representations thereby enable the encapsulation, coordination, and reuse of assemblies of parts.

The rest of the paper is organized as follows. First we review the different representation systems used by body-brain evolution systems. Then we describe the three parts of our body-brain evolution system: the compiler for the generative representation, the body-brain constructor and simulator, and the evolutionary algorithm. Next we present the results of our experiments in evolving locomoting creatures. This is followed by a discussion of our findings and conclusions.

2 Review of other Representations

Prior methods of body-brain evolution each have their own format for representing the creatures being evolved. In this section we first describe how the properties of programming languages apply to design representations and then review several representative systems.

With the exception of combination, the elements of programming languages listed in the introduction translate directly to design representations. Two types of control-flow are conditionals and iterative expressions. Conditionals can be implemented with an `if`-statement, as in genetic programming (GP) [22], or a rule which governs the next state in a cellular automata (CA). Iteration is a looping ability, such as the repeat structure in cellular encoding [9], or embedded in the fundamental behavior of CA's. Abstraction is the ability to encapsulate part of the genotype and label it such that it can be used like automatically defined functions (ADFs) in GP [23] or automatically defined sub-networks in cellular encoding. Abstraction can be seen when subfunctions can take parameters, as with ADFs. Combination refers to the ability to create more complex expressions from the basic set of commands in the language. While GP allows explicit combinations of expressions, combination is not fully enabled by mere adjacency or proximity in the strings utilized by typical representations in genetic algorithms.

Sims used an embedded, directed graph representation to specify the construction of his creatures [33]. Nodes in the top layer of the graph represent body segments, inside of which is another graph for the body segment's neural controller. An advantage of encapsulating the neural units inside the nodes for body segments is that copying, or recombining, subgraphs automatically swaps the associated neural controller for a section of body parts. This representation is generative because cycles in the graph, along with a recursive-limit parameter, are procedural constructs that specify the number of times nodes in the cycle are to be traversed in the construction phase. But the two-layer structure does not allow a repetition of the neural processing units inside a body-segment because they are directly encoded as a design inside a body node.

The stick creatures evolved in Ventrella's work [36] [37] are encoded as fixed-length vectors of parameters for constructing a creature. Parameters specify the number of segments for a central backbone, the number of opposing limbs, the number of segments in each limb, joint angles and details for the oscillator network. While this representation is generative in the sense that it allows reuse of the genotype, the structure of what can be reused is fixed and not evolvable.

The genotypes of creatures in Framsticks [20] are encoded as a linear assembly procedure for constructing a creature, with bracketing, which turns the basic structure from a string to a tree. Commands in the command set attach sticks to existing ones as well as construct the neural controller – commands for creating a neuron attach it to the stick most recently created and is then followed by a sequence of link connections. More recently they have compared their original representation, called *recur* for direct recurrent, against the actual representation used by the simulator, *simul*, and a tree-structured representation, called *devel* for developmental [21]. Simul consists of a list of all objects (sticks, joints, neurons, sensors and actuators) that make up a creature, along with all of that object's attributes. *Devel* is a tree-structured version of *recur* with iteration through a repeat node for repeating a subtree. Of these, only *devel* is generative because it is the only representation that allows for reuse of the genotype.

In GOLEM [26], the representation of a creature is the design itself. Both the morphology and neural controller are stored as graph-based data structures with links connecting actuated joints to neurons in the network. One challenge in using a graph-based representation is in implementing meaningful recombination operators between graphs. In this case, mutation was the only variation operator implemented.

The genotypes in the work of Bongard and Pfeifer [6] are a set of gene expression rules for growing creatures under a simulated ontogenetic process. These rules determine the division of body segments based on simulating chemical concentrations inside each segment. Each segment also contains a neural controller, which is developed by the gene expression rules using cellular encoding commands [9]. They report that similarities between parts of a creature also have similar gene expression patterns, suggesting that this method can produce modular creatures. Here reuse comes about through an iterative loop external to the evolved representation; at each update iteration gene-rules are applied to the developing creature.

Table 1
Properties of the different representations.

	Control Flow		Abstraction	
System	Iteration	Conditionals	Labels	Parameters
Bongard and Pfeifer, [6]	yes	yes	no	no
Framsticks-*recur/simul*, [20]	no	no	no	no
Framsticks-*devel*, [21]	yes	no	no	no
Genre-direct	no	no	no	no
Genre-generative	yes	yes	yes	yes
GOLEM, [26]	no	no	no	no
Sims, [33]	yes	no	no	no
Ventrella, [36]	yes	no	no	no

The properties of the different representations used for the evolution of a robot's morphology and controller are summarized in table 1. Not included in this summary is a column for a representation's structure, which is left out because determining the structure of a representation is somewhat subjective and also depends on how the variation operators act on an encoding. Of the reviewed representations, only *Genre*'s generative representation (described in the next section) has reuse through both iteration and parameterized procedures. Whereas iteration produces exact copies of the repeated genotype, parameterized procedures can act as a parameterized module, with the resulting phenotype depending on the input parameters.

3 Evolutionary System

The evolutionary system used to evolve creatures consists of the robot constructor and simulator, the compiler for the generative representation, and the evolutionary algorithm. Each robot is constructed from a sequence of construction commands, called an assembly procedure, which specifies how to assemble both the morphology and the robot's neural controller. This string of construction commands is either evolved directly or produced by compiling the generative representation. Robots encoded with a generative representation are called genobots (for generatively encoded robots). Our system uses Lindenmayer systems (L-systems) as the generative representation for the genobots. The evolutionary algorithm evolves a population of these L-systems, using the fitness returned by the robot simulator. The following subsections describe each of these parts.

3.1 L-systems as a Generative Representation Language

The generative representation for each genobot is an L-system, a grammatical rewriting system introduced to model the biological development of multicellular organisms [25]. Rules are applied in parallel to all characters in the string just as cell divisions happen in parallel in multicellular organisms. A basic L-system consists of a collection of re-writing rules, such as,

$$a : \to b$$
$$b : \to a\ b$$

When started with the symbol a, this L-system produces the following sequence of strings,

$$a$$
$$b$$
$$ab$$
$$bab$$
$$abbab$$
$$bababbab$$

The class of L-systems used as the genotype for creatures in this work is context-free, parametric Lindenmayer-systems (P0L-systems). Context-free indicates that the rules for rewriting symbols do not depend on the symbol's neighbors, and parametric specifies that symbols and rewriting rules can take parameters. Production rules consist of a rule-head, which is the symbol to be replaced, followed by a number of condition-successor pairs. The condition is a boolean expression on the parameters to the production-rule, and the successor (also called the production body) consists of a sequence of characters that replace the rule-head. Rule-head symbols are re-written by testing each of their conditions sequentially, and replacing the rule-head symbol with the successor of the first condition that succeeds.

Because the P0L-system does not have the ability to directly repeat a block of symbols, iteration is added through a block-replication command. Symbols enclosed with parenthesis, { *block* }(n), are repeated n times, and is similar to *for-next* loops in computer programs. The resulting representation language has the properties of iteration, conditionals, and abstraction with parameters.

Previously EAs have been combined with L-systems to evolve neural networks [19] [5], plants [17] [29] and architectural structures [7]. For the most part, this past work has used *non-parametric* L-systems whereas here we use *parametric* L-systems. An advantage of a parametric L-system over a non-parametric L-system is that a given PL-system can produce a family of strings, with the specific string determined by the starting parameter(s). For example, the parameter to a production rule can be used as the argument to the repeat command to specify the number of times a substring is to be repeated. Furthermore, parametric L-systems naturally allow for parametric commands in the language – the parameter to a network construction command can specify the weight of a newly created link in the network.

3.2 Robot Constructor

Robots are constructed through a method that is a synthesis of LOGO-style turtle graphics [1] and cellular encoding [9]. Commands in the assembly procedure are for constructing the robot's morphology and its neural controller. So that body and brain are joined, the command for the creation of an actuated joint also creates a link to a neuron in the neural network. The following sections describe the morphology and neural network constructors separately and then how a robot's body and brain are simultaneously constructed.

3.2.1 Network Constructor

The method for constructing the neural controllers for the artificial creatures is based on that of cellular encoding [9]. The main difference is that build commands operate on the links connecting the nodes, as with edge encoding [27], instead of on the nodes of the network. With edge encoding at most one link is created with a network construction command, which allows each command to also specify the weight to attach to that link, and sub-sequences of build commands will construct the same sub-network independent their location in the assembly procedure. Another distinction between this and cellular encoding is that assembly procedures for constructing networks are linear sequences of commands (strings) and not trees. A branching ability is added to strings by using bracketed L-systems [25] with *push* and *pop* operators for storing and retrieving the current link to a stack.

Commands for constructing the network operate on links between neurons and use the most recently created link as the current one. *Push* and *pop* operators, '(' and')', are used to store and retrieve the current link-state – consisting of the from-neuron, the to-neuron and the indices of the links into these neurons – to and from the stack. This stack of edges allows a form of branching to occur in the representation: an edge can be pushed onto the stack followed by a sequence of commands and then a pop command makes the original edge the current edge again. The commands for this language are listed in table 2, for which the current link connects from neuron A to neuron B.

Table 2
Command set for constructing neural networks.

Command	Description
( )	Push/pop link-state to stack.
decrease-weight(n)	Subtracts n from the weight of the current link. If the current link is a virtual link, it creates it with weight $-n$.
duplicate(n)	Creates a new link from neuron A to neuron B with weight n.
increase-weight(n)	Add n to the weight of the current link. If the current link is a virtual link, it creates it with weight n.
loop(n)	Creates a new link from neuron B to itself with weight n.
merge(n)	Merges neuron A into neuron B by copying all inputs of A as inputs to B and replacing all occurrences of neuron A as an input with neuron B. The current link then becomes the nth input into neuron B.
next(n)	Changes the from-neuron in the current link to its nth sibling.
output(n)	Creates an output-neuron, with a linear transfer function, from the current from-neuron with weight n. The current-link continues to be from neuron A to neuron B.
parent(n)	Changes the from-neuron in the current link to the nth input-neuron of the current from-neuron. Often there will not be an actual link between the new from-neuron and to-neuron, in which case a virtual link of weight 0 is used.
reverse	Deletes the current link and replaces it with a link from B to A with the same weight as the original.
set-function(n)	Changes the transfer function of the to-neuron in the current link, B, with: 0, for sigmoid; 1, linear; and 2, for oscillator.
split(n)	Creates a new neuron, C, with a sigmoid transfer function, and moves the current link from A to C and creates a new link connecting from neuron C to neuron B with weight n.

Neurons in the network are initialized to an output value of 0.0 and are updated sequentially by applying a transfer function to the weighted sum of their inputs with their outputs clipped to the range ± 1. The different transfer functions are: sigmoid, using $tanh(sum\ of\ inputs)$; linear; and an oscillator. Oscillator units maintain a state which is increased by 0.01 after each update. The output of an oscillator unit is mapped to the range -1 to 1 by applying a triangle wave function, with a period of four, to the sum of its inputs and its state. While using oscillating neurons increases the bias for simple networks with simple oscillating patterns over the sigmoid-only networks used in [20] [26] it is a less biased model than that of [36], in which all actuators are driven by oscillators, or [33] which used a variety of transfer functions and oscillating neurons.

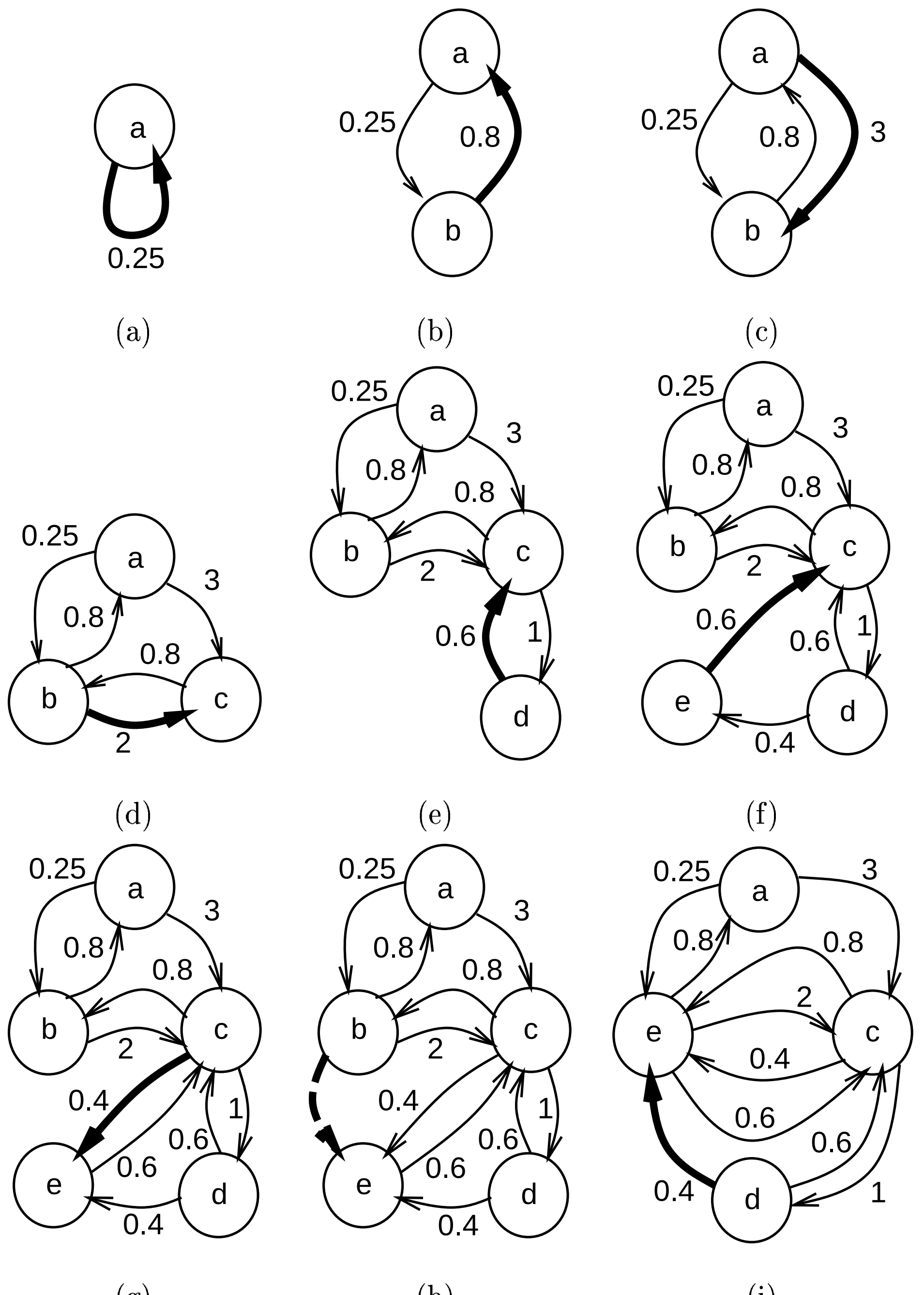
An example of the construction of a network using this system is shown in figure 2, which contains the intermediate networks in parsing the following assembly procedure,

split(0.8) duplicate(3) reverse split(0.8) duplicate(2) reverse loop(1) split(0.6) duplicate(0.4) split(0.6) duplicate(0.4) reverse parent(1) merge(1)

Networks start with a single neuron, a, which has an oscillator transfer function, and a single link of weight 0.25 feeding to itself, figure 2.a. After executing *split(0.8)*, a second neuron is created with a link of 0.8 to the oscillating neuron and the original link of weight 0.25 feeding into it, figure 2.b. Executing *duplicate(3)*, creates a second link from the second neuron to the first, which is then reversed in executing *reverse*, figure 2.c. The execution of *split(0.8) duplicate(2) reverse*, creates a third neuron, figure 2.d. A link from the third neuron to itself with weight 1 is created by *loop(1)*, with another neuron created by *split(0.6)*, figure 2.e. This is followed by *duplicate(0.4)*, which creates an additional link from neuron c to d, and then neuron e is created with *split(0.6)*, figure 2.f. Another link is created from e to c with *duplicate(0.4)*, which is then reversed, *reverse*, figure 2.g. *Parent(1)* causes a shift of link-state from the $c \rightarrow e$ link to a new "virtual link" $b \rightarrow e$, shown as a dashed line, figure 2.h. These two neurons are then joined together by the *merge(1)* command, and the final network is shown in figure 2.i.

3.2.2 Morphology Constructor

The morphology constructor uses a set of construction commands similar to that of L-system languages for creating plants [30] to build a body through the control of a LOGO-style turtle [1]. As the turtle moves, rods are created and these become the body of the robot. Commands instruct the turtle to move forward or backward, change orientation, or create an actuated joint. The method for constructing the morphology of a creature, with joints controlled by actuators, is described in [15]. The following section describes the method by which the morphology construction is combined with the neural network constructor of the previous section.



a 0.25
(a)
a b 0.25 0.8
(b)
a b 0.25 0.8 3
(c)
a b c 0.25 0.8 3 0.8 2
(d)
a b c d 0.25 0.8 3 0.8 2 0.6 1
(e)
a b c d e 0.25 0.8 3 0.8 2 0.6 0.6 1 0.4
(f)
a b c d e 0.25 0.8 3 0.8 2 0.4 0.6 0.6 1 0.4
(g)
a b c d e 0.25 0.8 3 0.8 2 0.4 0.6 0.6 1 0.4
(h)
a c d e 0.25 3 0.8 0.8 2 0.4 0.6 0.6 0.4 1
(i)

Fig. 2. Construction of a neural network.

3.2.3 Neural-Network Controlled Robots

A robot's morphology and neural controller are constructed by combining the command sets for constructing body and brain into one language and then building body and brain simultaneously. This command language consists of the morphology construction commands, listed in table 3, and the neural construction commands from section 3.2.1. The resulting language has two push/pop commands with two stacks: ( ), for pushing/popping the link-state to the link stack; and [ ], for pushing/popping both the morphology and link states to a stack. A robot's body and brain are joined together by attaching the current input-neuron to the newly created actuated joint each time a joint command – *revolute-1*, *revolute-2*, *twist-90*, or *twist-180* – is executed. By defining joint-creation commands in a way that affects both controller and morphology we induce a connection between body and brain.

Table 3
Command set for constructing the morphology of a robot. Neural controllers are constructed by using this language along with that of table 2.

Command	Description
[ ]	Push/pop state to stack.
forward	Moves the turtle forward in the current direction, creating a rod 10 units long if none exists or traversing to the end of the existing bar.
back	Goes back up the parent of the current bar.
revolute-1	Forward, end with a joint with range $0°$ to $90°$ about the current Z-axis that is controlled by the current neuron.
revolute-2	Forward, end with a joint with range $-45°$ to $45°$ about the current Z-axis that is controlled by the current neuron.
twist-90	Forward, end with a joint with range $0°$ to $90°$ about the current X-axis that is controlled by the current neuron.
twist-180	Forward, end with a joint with range $-90°$ to $90°$ about the current X-axis that is controlled by the current neuron.
left(n)	Rotate heading $n \times 90°$ about the turtle's Y axis.
right(n)	Rotate heading $n \times -90°$ about the turtle's Y axis.
up(n)	Rotate heading $n \times 90°$ about the turtle's Z axis.
down(n)	Rotate heading $n \times -90°$ about the turtle's Z axis.
clockwise(n)	Rotate heading $n \times 90°$ about the turtle's X axis.
counter-clockwise(n)	Rotate heading $n \times -90°$ about the turtle's X axis.

An example of an assembly procedure using this language is,

[right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward] duplicate(0.25) split(0.4) reverse revolute-1(1.0) duplicate(0.25) split(0.4) reverse revolute-1(1.0) left(1.0) right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward

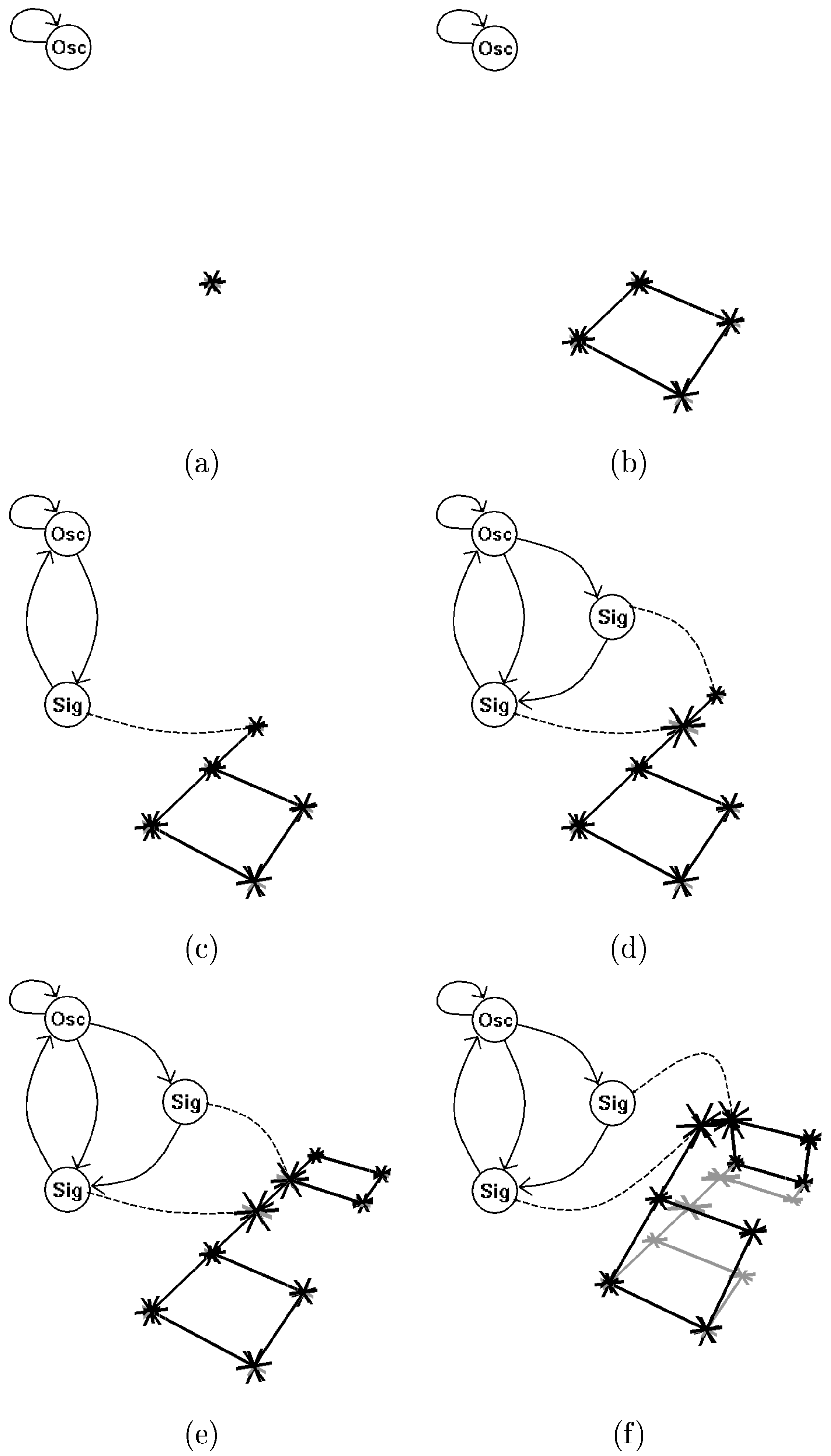


Fig. 3. Construction of a genobot.

A sequence of images showing intermediate stages in the construction of this robot is contained in figure 3. Before any commands are processed, a robot consists of a single oscillating neuron and a point, figure 3.a. After executing the commands, *[right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward]*, the robot consists of a square of four rods and the oscillating neuron, figure 3.b. After executing, *duplicate(0.25) split(0.4) reverse revolute-1(1.0)*, a second neuron is created and it is attached to the actuated joint at the end of the newly created rod, figure 3.c. The commands, *duplicate(0.25) split(0.4) reverse revolute-1(1.0)*, are repeated and a third neuron is created and it is attached to another actuated joint, figure 3.d. The last commands, *left(1.0) right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward*, attach another square onto the end of the last revolute-1 joint, figure 3.e. Figure 3.f shows the creature with the joints halfway through their movement range.

An example of a generative representation using this construction language is,

$$\begin{aligned} P0(n0): & \; n0 > 3.0 \rightarrow P1(5.0)\ P0(n0-2.0)\ left(1.0)\ P1(4.0) \\ & \; n0 > 0.0 \rightarrow \{\ duplicate(0.25)\ split(0.4)\ reverse\ revolute(1.0)\ \}(2.0) \end{aligned}$$

$$\begin{aligned} P1(n0): & \; n0 > 4.0 \rightarrow [\ P1(4.0)\ ] \\ & \; n0 > 0.0 \rightarrow \{\ right(1.0)\ forward(1.0)\ \}(n0) \end{aligned}$$

This L-system consists of two productions, each containing two condition-successor pairs and, when started with P0(4), produces the following sequence of strings, [1]

1. *P0(4)*
2. *P1(5.0) P0(2.0) left(1.0) P1(4.0)*
3. *[P1(4.0)] { duplicate(0.25) split(0.4) reverse revolute-1(1.0) }(2.0) left(1.0) { right(1.0) forward(1.0) }(4.0)*
4. *[{ right(1.0) forward(1.0) }(4.0)] { duplicate(0.25) split(0.4) reverse revolute-1(1.0) }(2.0) left(1.0) { right(1.0) forward(1.0) }(4.0)*
5. *[right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward] duplicate(0.25) split(0.4) reverse revolute-1(1.0) duplicate(0.25) split(0.4) reverse revolute-1(1.0) left(1.0) right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward*

[1] For clarity the unraveling of block replication expressions is left until the final iteration.

This last sequence of commands is the same assembly procedure as the one which produces the genobot in figure 3.

3.2.4 Simulation

Once a string of build commands has been executed, and the resulting robot is constructed, its behavior is evaluated in a quasi-static kinematics simulator similar to that used by [26]. First the neural network is updated to determine the desired angles of each actuated joint, then the kinematics are simulated by computing successive frames of moving joints in small angular increments of at most $0.06°$. After each update the structure is then settled by determining whether or not the robot's center of mass falls outside its footprint and then repeatedly rotating the entire structure about the edge of the footprint nearest the center of mass until it is stable.

To achieve robot designs that are made robust to transferal to the real world, error is added to evolved structures similar to the method of [18] and [16]. A robot design is evaluated by simulating it three times, once without error and twice with different error values applied to joint angles. Error is applied to all connections that are not part of a cycle and is a random rotation in the range of ± 0.1 radians about each of the three coordinate axis. The returned fitness of an evolved individual is the minimum fitness scored from the three trials. By implementing error that is fixed throughout a trial and evaluating a design with different error values, evolved designs are robust to imperfections in real-world construction. Examples of oscillator-controlled robots that were successfully transferred to the real world are in [12] [13].

3.3 Evolutionary Algorithm

The evolutionary algorithm and variation operators are described in detail in [15], here we give an overview of the system. The initial population of L-systems is created by making random production rules. Evolution then proceeds by iteratively selecting a collection of individuals with high fitness as parents and using them to create a new population of individual L-systems by applying mutation or recombination. Mutation creates a new individual by copying the parent individual and making a small change to it. Changes that can occur are: replacing one command with another; perturbing the parameter to a command by adding/subtracting a small value to it; changing a production rule's parameter equation in a successor; adding/deleting a sequence of commands in a successor; or changing the condition equation. Recombination takes two individuals, $p1$ and $p2$, as parents and creates one child individual, c, by making it a copy of $p1$ and then inserting a small part of $p2$ into it.

This is done by replacing one successor of c with a successor of $p2$, inserting a sub-sequence of commands from a successor in $p2$ into c, or replacing a sub-sequence of commands in a successor of c with a sub-sequence of commands from a successor in $p2$. Data is kept for individual L-system, specifically which production rules and successors were used, as well as the value range for each parameter. This data, similar to the environment frame of a programming language, allows variation operators to be applied only to those production rules which were used. It also allows historical-based constraints on the mutation of conditional values.

Since variations sometimes create an invalid robot (with too many/few rods, or the body parts intersect at some point while moving) variation operators are tried a second time, for a particular set of parents, if the first attempt did not create an offspring whose fitness was at least 10% of that of its parent(s).

4 Results

To compare a direct representation with a generative representation we evolved neural-network controlled robots for the task of locomotion. Fitness was a function of the distance moved by the robot's center of mass on a flat surface. In order to discourage sliding, fitness was reduced by the distance that points of the robot's body were dragged along the ground. Finally, a design was given zero fitness if it had a sequence of four or more rods in which none of the rods was part of a closed loop with other rods. This constraint was intended to keep the system from producing spindly robots which would not function well in reality.[2] The evolutionary algorithm was configured to run with a population of 200 individuals for 250 generations. The direct representation was implemented as an L-system with one production rule, no arguments, one condition-successor pair whose condition always succeeds, and without the repeat operator or the ability to call production rules. The maximum length of the production body was set to 10000 commands, allowing assembly procedures of up to 10000 commands to be evolved. The generative representation used an L-system with fifteen production rules, two condition-successor pairs, and two parameters for each production rule. For the generative representation, the maximum length of production body was set to fifteen commands and the maximum allowed length of an unraveled generative representation was set to 10000 commands – the same length as with the direct representation. Implementing the direct representation as a degenerate case of the generative representation allowed the evolutionary design system to use the

[2] A different approach would be to put a limit on the maximum torque applied on a connection, but this would require a simulator with more detailed physics then the one used here.

same variation operators on both representations so that the only difference between the two systems was the representation. All results in this section are from the same set of twenty runs, ten using the direct representation and ten with the generative representation.

4.1 Fitness Comparison

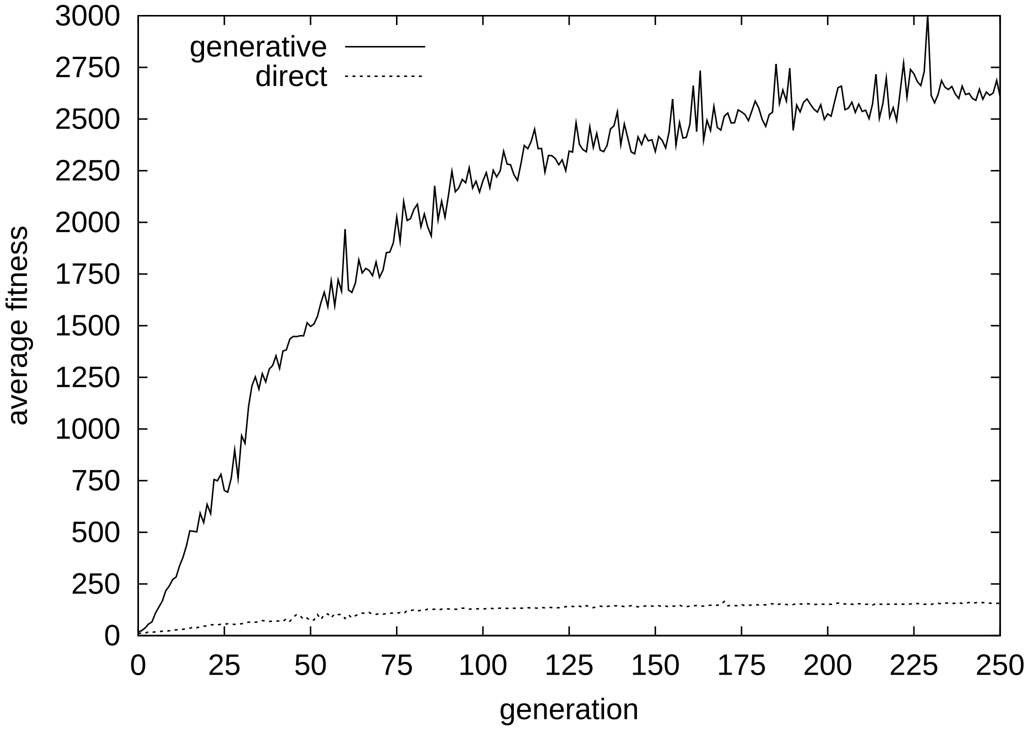


Fig. 4. Fitness comparison between the direct representation and the generative representation.

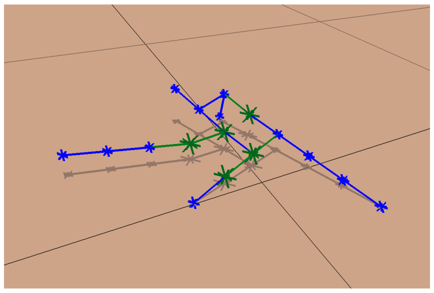
Our first graph, figure 4, plots the average fitness (over 10 runs) of the best individuals evolved with the direct representation against the best evolved with the generative representation. The final fitness values achieved were:

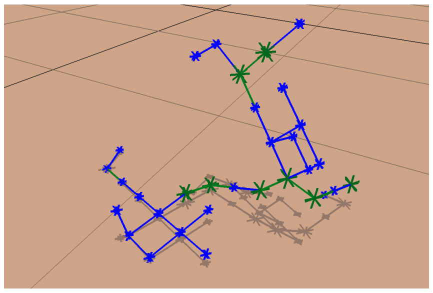
Direct: 134, 744, 42, 48, 62, 74, 42, 66, 86, 312.

Generative: 8180, 664, 2308, 3386, 696, 224, 1880, 364, 3810, 4556.

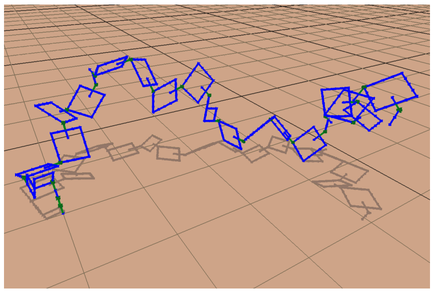
After 10 generations the generative representation achieved a higher average fitness than runs with the direct representation did after 250 generations and the final genobots evolved with the generative representation were more than 10 times faster, on average, than robots evolved with the direct representation.

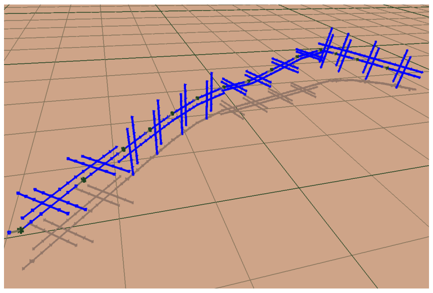
Figure 5 shows the two best individuals evolved with the direct representation (a and b) and the two best evolved with the generative representation (c and d). From the images it can be seen that the robots evolved with the direct representation are irregular, and have few components, whereas the robots



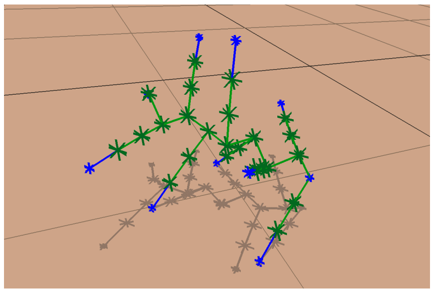


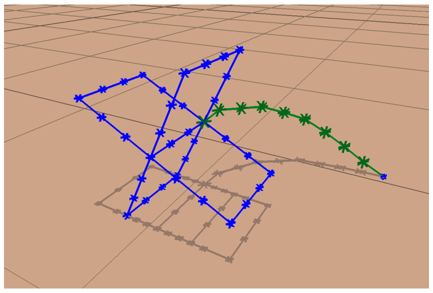
(a) (b)



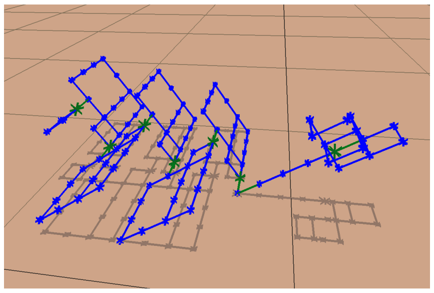


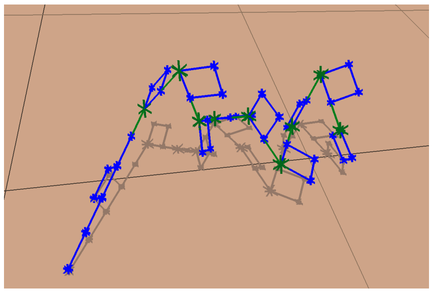
(c) (d)





(e) (f)





(g) (h)

Fig. 5. The two best individuals evolved with: (a) and (b), the direct representation; and (c) and (d), the generative representation. Genobots (e)-(h) are evolved with the generative representation and no constraints on limb lengths.

evolved with the generative representation are more regular and, in some cases, have two or more levels of reused assemblies of components. Furthermore, the network morphologies constructed from the generative representation also contain some reuse of subnetworks. The neural network controller shown in figure 6 is the controller for the genobot in figure 5.h. In addition to its reuse of components, its linear sequence of outputs corresponds to the linear sequence of joints in the genobot's morphology.

4.2 Reuse and Evolvability

In our introduction we argued that one advantage of a generative representation is its ability to create more complex components from simpler ones. The graph in figure 7.a shows, for each generation, the average length of the genotype for both representations, and the length of the assembly procedure produced by the generative representation. In the initial populations that used the generative representation the average length of the genotype was 126 symbols and the average length of the generated assembly procedure was 534 symbols. This means that on average each symbol in the genotype was being used 4.2 times in creating the assembly procedure. After 250 generations, this evolved to an average length of the genotype of 208 symbols and an average length of the resulting assembly procedure of 2387 symbols, which is an average reuse of 11.5. The average number of parts (rods only) used in a design is plotted in the graph in figure 7.b. As the genotype sizes for both direct and generative representations are about the same, the increased number of parts used in designs constructed from the generative representation suggests that the multiple expression of genotype produced a reuse of parts. Further support comes from the images in figures 5.c-h, which show that designs evolved with the generative representation have the same assemblies of parts occurring multiple times in a genobot.

The second part of our argument for a generative representation is that, through evolution, useful bias of the of search space becomes embedded in a design encoded with a generative representation, resulting in better performance of the variation operators. To compare the performance in variation operators between the two representations we compare the change in fitness between a parent and its child (from mutation) and plot it against the difference between parent and child's assembly procedures. For the direct representation, the assembly procedure is the same as the genotype, for the generative representation, the assembly procedure is the last string produced by the L-system. In the case where the assembly procedures are the same length, the command difference between two assembly procedures is the number of locations for which the parent and child have different symbols. When strings are different lengths, the number of occurrences of each symbol is counted and

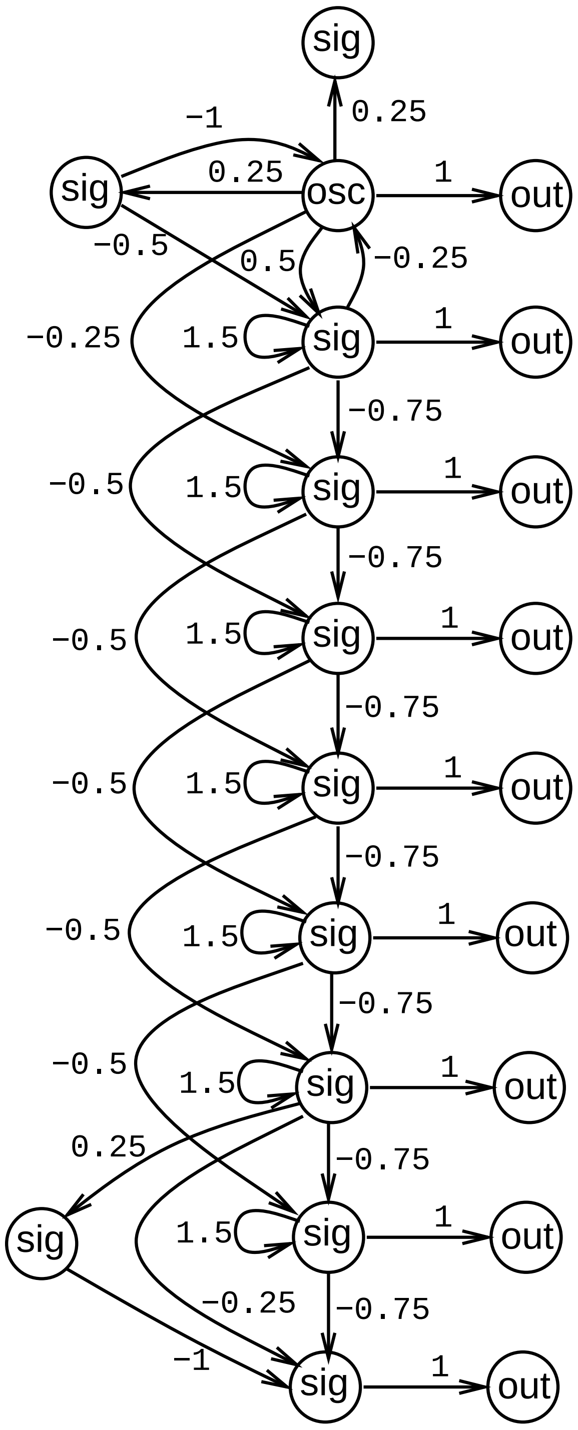
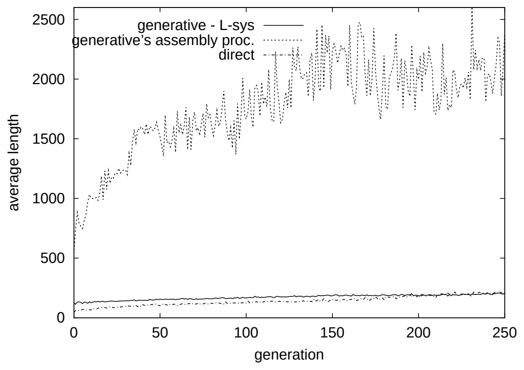


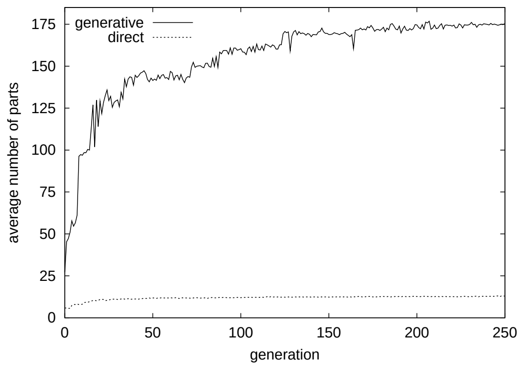
Fig. 6. Evolved neural network controller for the genobot in figure 5.h.

the command difference is the sum of the differences between these values. [3]

[3] An alternative distance metric which could be used is edit distance, but this is an expensive computation for bracketed strings (effectively trees) and would be prohibitive [31].

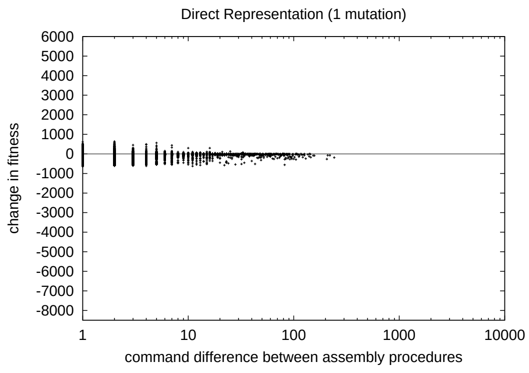


(a)

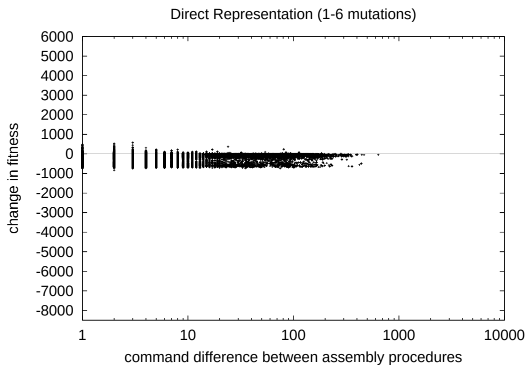


(b)

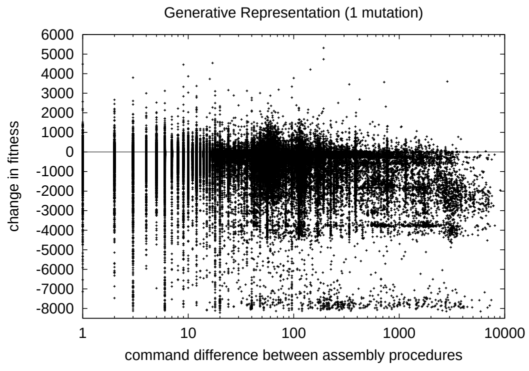
Fig. 7. Graph of (a) length of the genotypes, and command string produced by the generative representation, against generation, and (b) graph of number of parts against generation.



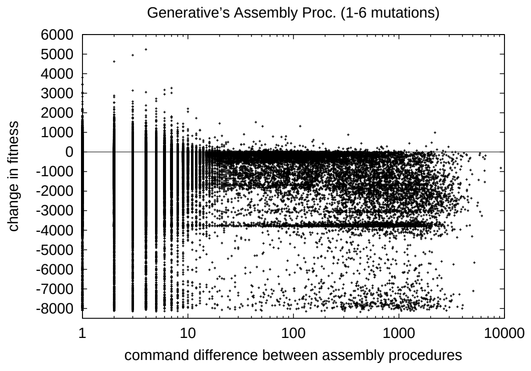
(a)



(b)

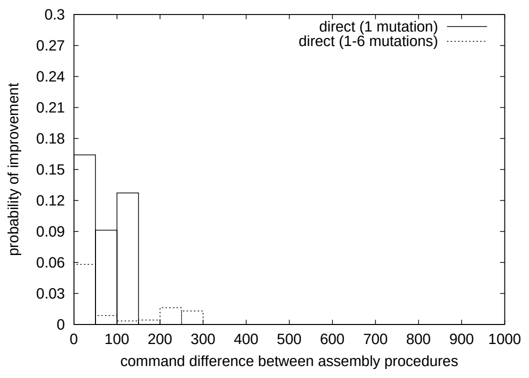


(c)

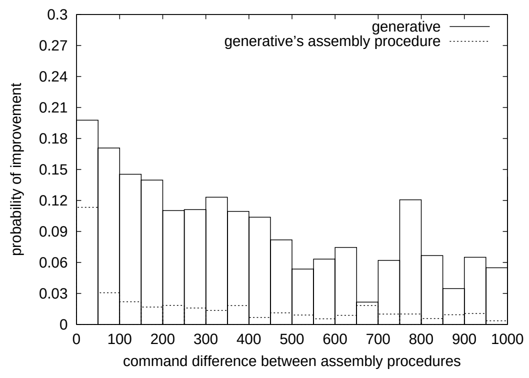


(d)

Fig. 8. Plot of amount of change in genotype from parent to child against change in fitness.



(a)



(b)

Fig. 9. Probability of success (child is more fit than parent) comparison between different representations, for ranges 1-50, 51-100, 101-150, ...

Figure 8 shows four different change-in-fitness against command-difference plots. The first graph, figure 8.a, is a plot of change-in-fitness against command difference for a single mutation operator applied to a direct representation. Most mutations were close to the parent, and most successful mutations were less than 10 commands apart. As offspring under the generative representation will tend to be further from their parent (because of reuse of the genotype), we also plot change in fitness against command difference for the direct representation with 1-6 mutations (chosen with uniform probability) applied. From the graphs it can be seen that mutations on the direct representation were usually only successful when the change in command difference between assembly procedures was small (less than 10), and even then improvements were not large. The graph in figure 8.c is a plot change-in-fitness against command difference between assembly procedures and shows that with the generative representation there was a larger variation in assembly procedure distance between a parent and its child than with the direct representation. This graph also shows offspring were more likely to have higher fitness than their parents with the generative representation than with the direct representation.

To determine if this improved performance under variation was only a result of the types of strings generated by the generative representation we also applied 1-6 mutations to the assembly procedure produced by the generative representation. The plot of figure 8.d shows that variation on the generative representation's assembly procedure was not as successful as on the generative representation itself, suggesting that the structure with the generative representation had captured some useful bias of the design problem over the course of evolution.

As a way of normalizing for the higher average fitness achieved with the generative representation, we next show the success rate (a child's fitness is greater than its parent) for the mutation operator for different command differences between parent and child, figure 9. Again we include a comparison with 1-6 mutations applied to the direct representation as well as 1-6 mutations applied to the assembly procedure produced by the generative representation. With the direct representation, the success rate of the mutation operator quickly dropped to zero as the difference between parent and child's assembly procedures increased. In contrast, the success rate of mutation more gracefully decayed with command difference when applied to the generative representation – even when parent and child were 500 construction symbols apart the success rate of mutation was 10%. The higher success rate of mutation, especially with larger differences in assembly procedures, and greater average increase in fitness with the generative representation provides strong evidence that the generative representation captured meaningful bias of the design problem.

4.3 Summary of Results

Table 4
Summary of results for evolving neural-network controlled robots.

Summary of experiments	Direct	Generative
Average final best fitness	157	2609
Number of mutations with distance > 0	184667	188749
Success rate of mutations	16.4%	18.1%
Average fitness change	-27	-526
Average fitness change of successful mutations	19	178
Average distance of mutations	3	118
Average distance of successful mutations	3	44
Number of large mutations (distance > 100)	65	27899
Success rate of large mutations (distance > 100)	10.8%	10.5%
Average fitness change of large mutations (distance > 100)	-74	-1396
Average fitness change of successful, large mutations (distance > 100)	25	357
Average distance of large mutations (distance > 100)	131	693
Average distance of successful, large mutations (distance > 100)	113	344

The results of our experiments are summarized in table 4. While the overall success rate of mutations is similar between the two representations, there is a difference in the average distance of mutations. Mutations on the direct representation were considerably smaller in assembly-procedure command difference than those on the generative representation. Examining the probability of success of a mutation for similarly sized distances (the graphs in figure 9, which plot the rate of success for distance in bins of size 50), it can be seen that for all distances the rate of success was higher with the generative representation. In addition, considering only mutations that are successful (the child has higher fitness than its parent), the average increase in fitness was significantly higher with the generative representation than with the direct representation.

5 Discussion

By allowing the inclusion of subprocedure-like structures (here, the L-system's production rules) a generative representation can create more complex building-blocks from simpler ones. Since these production rules are a single character that can be inserted/removed from the genotype with a single mutation, variation operators can scale with design complexity because new assemblies of components become possible unit variations. In addition, reusing code in the genotype to reuse parts in the actual design makes certain types of design changes easier. The images in figure 10 show examples of reuse through variations applied to the individual of figure 5.d. Changing the genotype to add rods to an assembly of parts results in the change to all occurrences of that part in the design, figure 10.b, and a single change to the genotype can cause the addition/subtraction of a large number of parts, figure 10.c. In both cases the same change would be harder to make with a direct representation. For example, even though recombination can duplicate assemblies of parts in a direct representation, a later application of variation will only change one instance of this assembly. As designs become more complex, the likelihood of the same change happening simultaneously to all uses of this assembly becomes increasingly unlikely with a direct representation, yet remains constant with a generative representation.

Of the representations described in section 2, the generative representation of *Genre* has similarities to those of Framsticks and the one used by Sims. The method of specifying a creature's morphology by a sequence of commands, with parentheses and brackets used for branching is almost identical to the Framsticks-*recur* representation [20]. One difference is in specifying the neural controller. In Framsticks, this is done by listing the links immediately after the neuron whereas in our system a cellular encoding language is used. The other difference is that the Framsticks-*recur* representation does not have any properties of reuse. Both Framsticks-*devel* [21] and Sims' [33] allow for parts of the genotype to be reused through a looping ability. This looping is like the repetition blocks of the generative representation defined in section 3 and is the beginning of a procedural description. The representation of *Genre* extends on

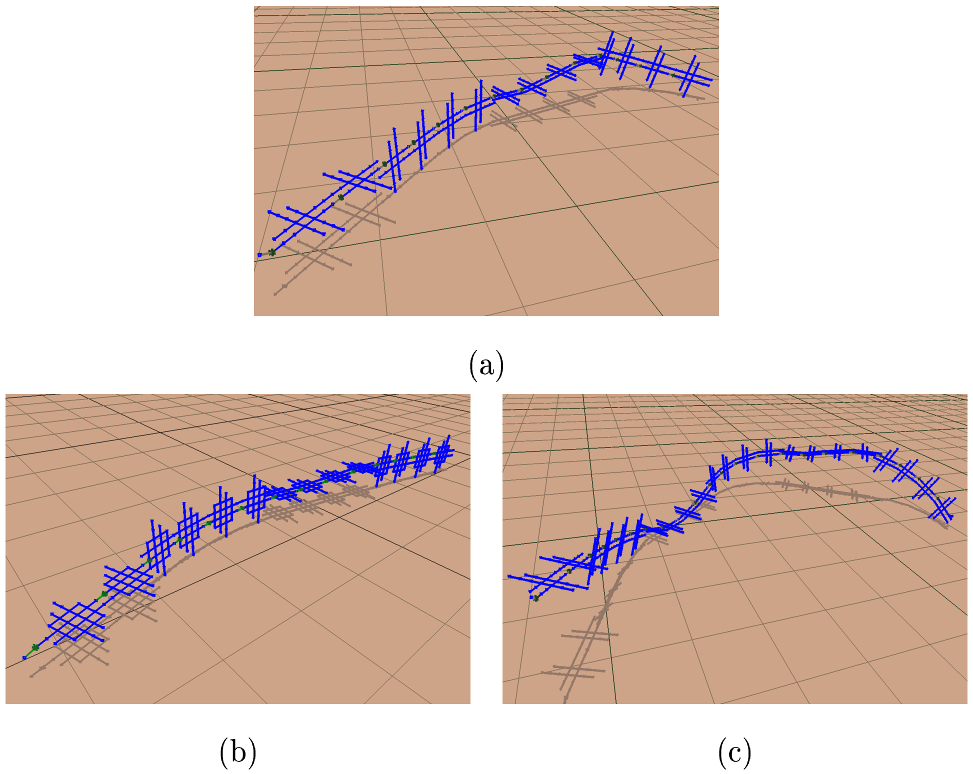


Fig. 10. Mutations of a genobot: (a), the genobot from figure 5.d; (b), a change to a low-level component of parts results in all occurrences of this part to have the change; (c), a single change to the genotype changes the number of high-level components in the genobot from four to six.

the ability to define loops by including labeled sub-procedures with parameters in the genotype – similar to the modules of GLiB [4], automatically defined functions (ADFs) of GP [22] and automatically defined sub-networks (ADSNs) of [9] – and conditionals on the parameters.

In our comparison robots evolved with a generative representation were, on average, ten times faster than those evolved with a direct representation. These results differ from those of [21], in which their comparison produced little difference between a generative representation, *devel*, and a direct representation, *recur*. Since their generative representation had only the property of iteration – whereas our generative representation also has conditionals and abstraction with parameters – it suggests that these additional properties can make a significant difference on the performance of an evolutionary design system.

6 Conclusion

The concurrent evolution of bodies and brains has been limited by the representations used to encode them [36] [20] [26] [6]. Here we have defined the class of generative representations and presented *Genre*, a generic system for evolving designs with this class of representations. Previous work has shown how this system can be used to evolve table designs [14], two-dimensional oscillator-controlled genobots [12], and three-dimensional oscillator controlled genobots [15]. Since *Genre* treats command sets and assembly procedures as symbols and strings, by replacing one command set with another and/or replacing the design constructor, this evolutionary system can be used on any design domain in which a design can be constructed from a linear assembly procedure.

In this paper we have described a method for evolving the morphology and neural controller of three-dimensional robots. We have showed that robots evolved with the generative representation reach higher fitnesses than those evolved with a direct representation. This improved performance has been shown to be the result of the ability of the generative representation to reuse parts of the design. In summary, generative representations accelerate evolution by learning useful problem bias over the course of the evolution and by encapsulating, in heritable elements of the genotype, assemblies of phenotypic components, thereby allowing mutation to scale with design complexity.

Generative representations capture the fundamental elements of general purpose programming languages - combination, control flow, and abstraction. However, we note that the linear representation is somewhat limiting, even though primitives like "push" and "pop" add tree-like constructions. As continuing work expands the range and power of generative representations, while maintaining evolvability, we expect to see ever more progress towards general purpose evolutionary design.

Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Administration (DARPA) Grant, DASG60-99-1-0004. The authors would like to thank the members of the DEMO Lab (especially J. Rieffel, S. Viswanathan, and R. Watson), H. Lipson, M. Xiu, and the anonymous reviewers.

References

[1] H. Abelson and A. A. diSessa. *Turtle Geometry.* M.I.T. Press, 1982.

[2] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* McGraw-Hill, second edition, 1996.

[3] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of The Cell.* Garland Publishing, New York, 4th edition, 2002.

[4] P. Angeline and J. B. Pollack. Coevolving high-level representations. In C. Langton, editor, *Proceedings of the Third Workshop on Artificial Life*, Reading, MA, 1994. Addison-Wesley.

[5] E. J. W. Boers, H. Kuiper, B. L. M. Happel, and I.G. Sprinkhuizen-Kuyper. Designing modular artificial neural networks. In H.A. Wijshoff, editor, *Proceedings of Computing Science in The Netherlands*, pages 87–96, SION, Stichting Mathematisch Centrum, 1993.

[6] J. C. Bongard and R. Pfeifer. Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In *Genetic and Evolutionary Computation Conference*, pages 829–836, 2001.

[7] P. Coates, T. Broughton, and H. Jackson. Exploring three-dimensional design worlds using lindenmayer systems and genetic programming. In P. J. Bentley, editor, *Evolutionary Design by Computers*, 1999.

[8] R. Dawkins. *The Blind Watchmaker.* Harlow Longman, 1986.

[9] F. Gruau. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Ecole Normale Supérieure de Lyon, 1994.

[10] R. Grzeszczuk and D. Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 63–70, Los Angeles, California, August 1995. In *Computer Graphics* Annual Conf. Series, 1995.

[11] G. S. Hornby, M. Fujita, S. Takamura, T. Yamamoto, and O. Hanagata. Autonomous evolution of gaits with the sony quadruped robot. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1297–1304. Morgan Kaufmann, 1999.

[12] G. S. Hornby, H. Lipson, and J. B. Pollack. Evolution of generative design systems for modular physical robots. In *IEEE Intl. Conf. on Robotics and Automation*, pages 4146–4151, 2001.

[13] G. S. Hornby, H. Lipson, and J. B. Pollack. Generative representations for the automatic design of modular physical robots. Technical report, Computer Science Dept., Brandeis University, Waltham, MA, 2002.

[14] G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *Congress on Evolutionary Computation*, pages 600–607, 2001.

[15] G. S. Hornby and J. B. Pollack. Evolving L-systems to generate virtual creatures. *Computers and Graphics*, 25(6):1041–1048, 2001.

[16] G. S. Hornby, S. Takamura, O. Hanagata, M. Fujita, and J. Pollack. Evolution of controllers from a high-level simulator to a high dof robot. In J. Miller, editor, *Evolvable Systems: from biology to hardware; Proc. of the Third Intl. Conf. (ICES 2000)*, Lecture Notes in Computer Science; Vol. 1801, pages 80–89. Springer, 2000.

[17] C. Jacob. Evolution programs evolved. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature PPSN-IV*, Lecture Notes in Computer Science 1141, pages 42–51, Berlin, 1996. Springer-Verlag.

[18] N. Jakobi. *Minimal Simulations for Evolutionary Robotics.* PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, May 1998.

[19] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[20] M. Komosinski. The world of framsticks: Simulation, evolution, interaction. In *Virtual Worlds 2*, Lecture Notes in Artificial Intelligence 1834, pages 214–224. Springer-Verlag, 2000.

[21] M. Komosinski and A. Rotaru-Varga. Comparison of different genotype encodings for simulated 3d agents. *Artificial Life*, 7(4):395–418, 2001.

[22] J. R. Koza. *Genetic Programming: on the programming of computers by means of natural selection.* MIT Press, Cambridge, Mass., 1992.

[23] J. R. Koza. *Genetic Programming II : Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, Mass., 1994.

[24] B. Lewin. *Genes VII.* Oxford University Press, 2000.

[25] A. Lindenmayer. Mathematical models for cellular interaction in development. parts I and II. *Journal of Theoretical Biology*, 18:280–299 and 300–315, 1968.

[26] H. Lipson and J. B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978, 2000.

[27] S. Luke and L. Spector. Evolving graphs and networks with edge encoding: Preliminary report. In J. Koza, editor, *Late-breaking Papers of Genetic Programming 96*, pages 117–124. Stanford Bookstore, 1996.

[28] J. T. Ngo and J. Marks. Spacetime constraints revisited. In *SIGGRAPH 93 Conference Proceedings*, pages 343–350, 1993. Annual Conference Series.

[29] G. Ochoa. On genetic algorithms and lindenmayer systems. In A. Eiben, T. Baeck, M. Schoenauer, and H. P. Schwefel, editors, *Parallel Problem Solving from Nature V*, pages 335–344. Springer-Verlag, 1998.

[30] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag, 1990.

[31] D. Sankoff and J. B. Kruskal, editors. *Time warps, string edits and macromolecules: the theory and practice of sequence comparison.* Addison-Wesley, Reading, Mass., 1983.

[32] K. Sims. Evolving 3d morphology and behavior by competition. In R. Brooks and P. Maes, editors, *Proceedings of the Fourth Workshop on Artificial Life*, pages 28–39, Boston, MA, 1994. MIT Press.

[33] K. Sims. Evolving Virtual Creatures. In *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 15–22, 1994.

[34] T. Taylor and C. Massey. Recent developments in the evolution of morphologies and controllers for physically simulated creatures. *Artificial Life*, 7(1):77–87, 2001.

[35] M. van de Panne and E. Fiume. Sensor-actuator Networks. In *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 335–342, 1993.

[36] J. Ventrella. Explorations in the emergence of morphology and locomotion behavior in animated characters. In R. Brooks and P. Maes, editors, *Proceedings of the Fourth Workshop on Artificial Life*, Boston, MA, 1994. MIT Press.

[37] J. Ventrella. Animated artificial life. In J.-C. Heudin, editor, *Virtual Worlds: Synthetic Universes, Digital Life, and Complexity.* Perseus Books, 1999.