# Evolving Hierarchical and Recursive Teleo-reactive Programs through Genetic Programming

Mykel J. Kochenderfer

Department of Computer Science
Stanford University
Stanford, California 94305
`mykel@cs.stanford.edu`

**Abstract.** Teleo-reactive programs and the triple tower architecture have been proposed as a framework for linking perception and action in agents. The triple tower architecture continually updates the agent's knowledge of the world and evokes actions according to teleo-reactive control structures. This paper uses block stacking problems to demonstrate how genetic programming may be used to evolve hierarchical and recursive teleo-reactive programs.

## 1   Introduction

It is important that the control programs for intelligent agents be easy for humans to write and for machines to generate and modify through learning and planning. Nils Nilsson has proposed the teleo-reactive program structure as a possible candidate for representing these programs [4,5]. Teleo-reactive programs are hierarchical and consist of ordered lists of production rules whose conditions are continuously evaluated with reference to the agent's perceptions, thereby enabling the creation of intelligent and robust mobile robots that are highly reactive and adaptable in dynamic environments.

Teleo-reactive programs make up the "action tower" of Nilsson's triple tower architecture for autonomous agents [7]. The other two towers maintain the agent's understanding of the world, which is used by the teleo-reactive programs to evoke the appropriate actions. The "perception tower" contains rules that create an increasingly abstract description of the world based on the primitive predicates perceived by the agent. These descriptions of the world are then deposited in the "model tower" and kept faithful to the environment through a truth-maintenance system. Nilsson illustrated the performance of the triple tower architecture and teleo-reactive programs in the blocks-world domain. He created a system capable of stacking any specified tower of blocks on a table from any configuration without search[1].

---

[1] A link to an animated Java applet demonstrating the use of teleo-reactive programs and the triple tower architecture in the blocks-world domain may be found online here: `http://cs.stanford.edu/~nilsson/trweb/tr.html`.

While other learning techniques for teleo-reactive programs have been proposed [6], this paper expands upon previous work by the author [1] and is the first to demonstrate the suitability of genetic programming for the automatic creation of hierarchical and recursive teleo-reactive programs. To make it easy to compare the evolved teleo-reactive programs with Nilsson's human-designed block stacking system, the same predicates available in Nilsson's "perception tower" are used along with the same primitive actions, $pickup(x)$ and $putdown(x)$ [7]. This paper presents novel teleo-reactive programs created through genetic programming that are simpler than the solution published by Nilsson.

Some of the relevant details of teleo-reactive programs and a discussion of their incorporation into genetic programming are found in the next section. The methods used in evolving teleo-reactive programs for block stacking are explained in section 3. A report of the results follows in section 4, and conclusions are drawn from these results in section 5. Further work is discussed in the final section.

## 2   Teleo-reactive Programs

A teleo-reactive program, as proposed by Nilsson [4,5], is an ordered list of production rules, as shown below:

$$K_1 \rightarrow a_1$$
$$\vdots$$
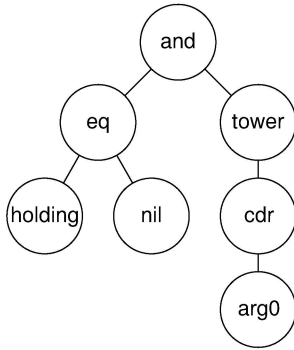$$K_i \rightarrow a_i$$
$$\vdots$$
$$K_m \rightarrow a_m$$

The $K_i$'s are conditions that are evaluated with reference to a world model, and the $a_i$'s are actions on the world. The conditions may contain free variables that are bound when the teleo-reactive program is called. Actions may be primitives, they may be sets of actions to be executed in parallel, or they may be calls to other teleo-reactive programs (thereby enabling hierarchy and recursion). Typically, $K_1$ is the goal condition, $a_1$ is the nil action, and $K_m$ is true.

The rules are scanned from top to bottom for the first condition that is satisfied, and then the corresponding action is taken. Since the rules are scanned continuously, *durative* actions are possible in addition to *discrete* actions. In the version of block stacking discussed in this paper, only *discrete* actions are used.
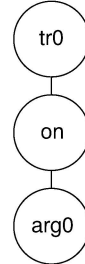
In order to use the standard genetic programming algorithms [2] to evolve teleo-reactive programs, they must be represented as trees. Teleo-reactive programs may be thought of as a list of condition trees and action trees. Fig. 1 shows a sample condition tree and action tree.

The variable-length list structure of a single teleo-reactive program may be represented as a larger tree containing these condition trees and action trees. In this paper, the `trprog` function, `if` function, and `end` terminal are used to enforce this tree structure, as illustrated in Fig. 2. These functions and terminals are described in Sect. 3.1.

**Sample Condition Tree**                    **Sample Action Tree**



(and (eq holding nil) (tower (cdr arg0)))     (tr0 (on arg0))

**Fig. 1.** Sample condition and action trees and their corresponding LISP-style symbolic expressions.



$$K_0 \rightarrow nil$$
$$K_1 \rightarrow a_1$$
$$K_2 \rightarrow a_2$$
$$\vdots$$
$$T \ \rightarrow a_0$$

(trprog $K_0$ $a_0$ (if $K_1$ $a_1$ (if $K_2$ $a_2$ ...)))

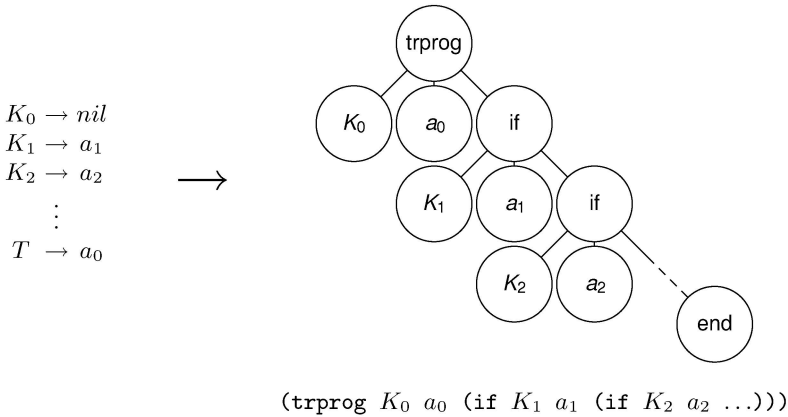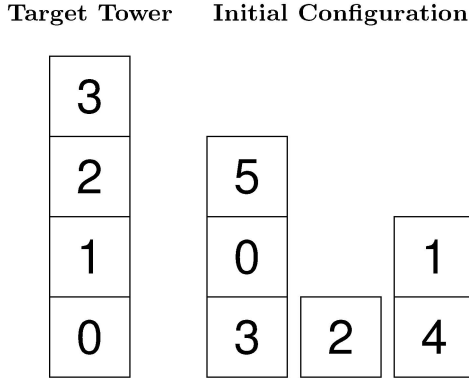**Fig. 2.** A tree representation for the variable length list of condition ($K_i$) and action ($a_i$) subtrees.

## 3   Methods

The teleo-reactive tree structures described in the previous section proved easy to implement using the strongly-typed version of LIL-GP made available by Sean Luke[2]. This section explains the setup for the genetic programming experiments.

Before discussing the functions and terminals, it is important to understand how a block stacking problem is specified. Each blocks-world problem consists

---

[2] Sean Luke's strongly-typed kernel is based on the LIL-GP Genetic Programming System from Michigan State University. The source is freely available here: http://www.cs.umd.edu/~seanl/gp/patched-gp.

**Target Tower    Initial Configuration**



**Fig. 3.** A sample block stacking problem with a target tower and initial configuration.

of a target tower that is passed as an argument to the program and an initial configuration of blocks. The target tower is simply the tower that the agent wishes to build, which is a strict ordering of a subset of the blocks. The initial configuration is the state of the world that is initially presented to the agent. The state contains $n$ blocks arranged as a set of columns. An example of a block stacking problem is shown in Fig. 3.

To allow for hierarchical and recursive programs, multiple trees are grown for each individual. The first tree (named $tr0$) is considered the "main" program, which receives the target tower as an argument, and the other trees (named $tr1$, $tr2$, etc.) are "subprograms." The main program may then call itself recursively or any program from the other trees. The subprograms may then call any other subprogram or the main program.

## 3.1    Functions and Terminals

The functions and terminals were designed to resemble the predicates available to the block stacking agent from the perception tower in Nilsson's triple-tower architecture [7]. The only significant difference in this paper is that the functions that only work on lists with single elements have an implicit "car" in their definition.

The teleo-reactive structures use four types in the strongly-typed genetic programming system: *TR*, *Boolean*, *List*, and *Action*. Every tree has as its root the function `trprog`, which takes a *Boolean*, an *Action*, and a *TR* as arguments. The `trprog` function evaluates its first argument, and if it evaluates to true the program will terminate. Otherwise, the function will evaluate its third argument. If the third argument is evaluated and returns `end`, the action specified by the second argument is taken.

The *TR* type consists of the three-argument `if` function and the terminal `end`. The `if` function takes a *Boolean*, an *Action*, and a *TR*. It evaluates the first argument and will then evaluate the second argument if it is true and the

third argument if it is false. The `trprog` and `if` functions enforce a linear tree structure with condition and action subtrees as explained earlier.

The *Boolean* type consists of the `true` terminal, three standard boolean functions, and three functions based on the predicates available from Nilsson's perception tower. They are explained below.

- The function `and` takes two *Boolean* arguments and returns *true* if both arguments evaluate to *true*, otherwise it returns *false*.
- The function `not` takes one *Boolean* argument and returns *false* if the argument evaluates to *true*, otherwise it returns *true*.
- The function `eq` takes two *List* arguments and returns *false* if the *car* (i.e. the first element) of both arguments are not equal, otherwise it returns *true*.
- The function `tower` takes one *List* argument and returns *true* if the argument is `ordered` and `clear`, otherwise it returns *false*.
- The function `ordered` takes one *List* argument and returns *true* if there exists a stack of blocks on the table having a substack with an order corresponding to the argument, otherwise it returns *false*. As specified in Nilsson's paper, (`ordered nil`) evaluates to *false*.
- The function `clear` takes one *List* argument and returns *true* if the first element of the list is the table or if the first element of the list is a block that does not have a block on top of it, otherwise it returns *false*.

The *List* type consists of the following terminals and functions.

- The terminal `nil` is the empty list.
- The terminal `table` is a list consisting of one element representing the table.
- The terminal `holding` is a list consisting of the block that is currently being held by the agent. If the agent is not holding a block, then this terminal is an empty list.
- The function `cdr` takes one *List* argument and returns the list with the first element removed. If the argument is `nil`, this function returns `nil`.
- The function `on` takes one *List* argument and returns the block that is directly on top of the first block in the list. If the first element in the list is not a block or if the block is clear, then this function returns `nil`.
- The function `under` takes one *List* argument and returns the block that is directly under the first block in the list. If the block is on top of the table, then the function returns `table`. Otherwise, this function returns `nil`.

In addition to the *List* terminals specified above, there are other terminals representing the arguments to the teleo-reactive programs, which are called `arg`*x*. The main program has one argument, `arg0`, which is the list specifying the target tower.

The *Action* type has one terminal, `terminate`, which simply terminates the agent. The agent has two primitive actions available, namely, `pickup` and `putdown`. Both functions take one *List* argument. The action `pickup` can be applied if no block is on the first element of the argument and the agent is not currently holding anything. The result is that the agent is holding the block.

The action `putdown` can be applied if the agent is holding a block and the first element of the argument specifies a block that is clear. The result is that the agent is no longer holding the block and that the block that was held by the agent is placed on top of the specified block.

Notice that if the agent ever takes an action that has no effect on the environment, all future actions will be the same and, hence, will have no effect on the environment. The same action will be taken in the teleo-reactive tree because only the agent may change the environment. Therefore, an experiment terminates as soon as the agent takes an action that has no effect on the world.

## 3.2    Evaluation of Fitness

The most natural way to measure fitness is to count the number of blocks that were stacked correctly by the agent by the time the agent decides to stop or within a certain number of operations. Evaluating a teleo-reactive stacking program on a single test case is unlikely to produce a solution to the general block-stacking problem. Therefore, it is best to evaluate the fitness of a particular teleo-reactive program on a collection of fitness cases involving a variety of target towers and initial configurations with varying numbers of blocks.

The number of initial configurations for a given number of blocks may be extremely large. The number of possible configurations is given by the "sets of lists" sequence, which counts the number of partitions of $\{1, \ldots, n\}$ into any number of lists [3]. Starting with $n = 2$, the sequence proceeds: 3, 13, 73, 501, 4051, 37633, 394353, 4596553, ..., according to the recursive formula:

$$a(n) = (2n - 1)a(n - 1) - (n - 1)(n - 2)a(n - 2)$$

The number of configurations grows extremely quickly. For 18 blocks, there are 588,633,468,315,403,843 possible arrangements.

It is necessary, therefore, to rely on random samples of this space for the fitness cases. Using the BWSTATES program made available by John Slaney[3], samples were selected randomly from a uniform distribution of blocks-world states. It is also important that the fitness cases include target towers consisting of varying numbers of blocks. The target towers ranged from one block to all of the blocks. Two fitness case collections were generated for the experiments, each consisting of 100 blocks-world problems. One collection consists of problems with the number of blocks ranging from three to seven, and the other collection consists of problems with the number of blocks ranging from three to twelve.

During the genetic programming run, the teleo-reactive programs were evaluated on each fitness case. The agent is allocated a certain number of steps to solve each problem or to give up. The agent was also allowed a maximum recursion depth that varied with the number of blocks in the world. The raw fitness is the sum of the number of blocks stacked correctly in all the fitness cases, minus half a point for every block on top of an otherwise correct tower. The standardized fitness is simply the sum of all the target tower sizes minus the raw fitness. The number of hits is the number of problems that were solved by the agent.

---

[3] Freely available here: `http://arp.anu.edu.au/~jks/bwstates.html`.

### 3.3   Parameters

The experiments used a population size of 60,000. The "grow" method [2] of generating the initial random population was used with a depth ramp ranging from 3 to 9 for each of the trees.

There were two breeding phases: crossover (90%) and mutation (10%). Breeding was fitness proportionate. The maximum depth for evolved trees was 12.

The individuals were allowed to have four program trees, meaning that an individual may use up to three subprograms in addition to its main program. One of the subprograms was allowed two arguments, but the other subprograms were allowed only one. All subprograms were allowed to call each other and the main program.

### 3.4   Human-Designed Solution

A version of Nilsson's hierarchical teleo-reactive program that solves the block-stacking problem [7] is shown in Fig. 4. This program uses the functions and terminals defined in this paper to make it easy to compare the evolved programs presented later.

## 4   Results and Discussion

The experiments were run on shared dual-processor Sun UltraSPARC III+ 900MHz workstations. Several completely fit individuals evolved within 300 generations, which took a few days of processing time. To determine whether the fully fit individuals could solve block stacking problems they had not seen before, the evolved programs were tested on a collection of 11,500 randomly generated problems with the number of blocks ranging from three to twenty-five. Most evolved programs solved at least 99% of these new problems. Two individuals were able to solve all of them, and they are believed to be completely general solutions for block stacking.

The evolved programs generally consisted of a lot of code that never gets executed for a variety of reasons. Frequently, most likely because of crossover, there are redundant conditions. The evolved programs listed in this section have been stripped of their "dead code" and some of the conditions were replaced by equivalent, more reader-friendly conditions.

The first individual that solved the collection of 11,500 test problems was evolved in generation 272. It used the 100 fitness cases with the number of blocks ranging from three to seven. The individual consists of only the main program and one subprogram, and contains fewer rules than the human designed solution. The individual is reproduced in Fig. 5. The program effectively terminates when it reaches the goal by making infinite recursive calls between the two program trees. Although infinite recursion might not be desirable, it only occurs when the agent has completed its task.

Another individual that solved all of the test problems used the 100 fitness cases with the number of blocks ranging from three to twelve. It was produced

**tr0(arg0):**

$$tower(arg0) \rightarrow nil$$
$$ordered(arg0) \rightarrow tr3(arg0)$$
$$cdr(arg0) = nil \rightarrow tr1(arg0)$$
$$tower(cdr(arg0)) \rightarrow tr2(arg0, cdr(arg0))$$
$$T \rightarrow tr0(cdr(arg0))$$

**tr1(arg0):**

$$under(arg0) = table \rightarrow nil$$
$$holding \neq nil \rightarrow putdown(table)$$
$$clear(arg0) \rightarrow pickup(arg0)$$
$$T \rightarrow tr3(arg0)$$

**tr2(arg0, arg1):**

$$on(arg1) = arg0 \rightarrow nil$$
$$holding = arg0 \wedge clear(arg1) \rightarrow putdown(arg1)$$
$$holding \neq nil \rightarrow putdown(table)$$
$$clear(arg0) \wedge clear(arg1) \rightarrow pickup(arg0)$$
$$clear(arg1) \rightarrow tr3(arg0)$$
$$T \rightarrow tr3(arg1)$$

**tr3(arg0):**

$$clear(arg0) \rightarrow nil$$
$$T \rightarrow tr1(on(arg0))$$

**Fig. 4.** A version of Nilsson's hierarchical teleo-reactive program that solves the block stacking problem using the function and terminals defined in this paper.

**tr0(arg0):**

$$holding \neq nil \wedge on(arg0) = nil \wedge \neg(under(arg0) = table \wedge tower(cdr(arg0))) \rightarrow tr1(cdr(arg0))$$
$$holding \neq on(arg0) \rightarrow tr1(on(arg0))$$
$$tower(arg0) \rightarrow tr1(arg0)$$
$$under(arg0) = table \wedge \neg tower(cdr(arg0)) \rightarrow tr1(cdr(arg0))$$
$$T \rightarrow pickup(arg0)$$

**tr1(arg0):**

$$arg0 = holding \rightarrow tr0(arg0)$$
$$holding = nil \rightarrow tr0(arg0)$$
$$tower(arg0) \rightarrow putdown(arg0)$$
$$under(arg0) = table \rightarrow tr0(arg0)$$
$$T \rightarrow putdown(table)$$

**Fig. 5.** An evolved program that solves arbitrary block stacking problems.

in generation 286. This hierarchical and recursive individual uses only the main program, and it is reproduced in Fig. 6. The program terminates when it reaches the goal by taking the action $putdown(arg0)$, which does not change the state of the world.

$$\textbf{tr0}(\textbf{arg0}):$$

$$\neg ordered(arg0) \wedge holding = nil \wedge clear(arg0) \wedge under(arg0) \neq table \rightarrow pickup(arg0)$$
$$tower(cdr(arg0)) \wedge holding = nil \wedge clear(arg0) \wedge arg0 \neq nil \rightarrow pickup(arg0)$$
$$arg0 = nil \rightarrow putdown(table)$$
$$\neg tower(cdr(arg0)) \wedge \neg clear(arg0) \rightarrow tr0(on(arg0))$$
$$tower(arg0) \rightarrow putdown(arg0)$$
$$tower(cdr(arg0)) \wedge clear(arg0) \rightarrow tr0(on(arg0))$$
$$holding = nil \wedge tower(cdr(arg0)) \rightarrow tr0(on(arg0))$$
$$T \rightarrow tr0(cdr(arg0))$$

**Fig. 6.** An evolved program that solves arbitrary block stacking problems.

## 5   Conclusions

This paper has demonstrated how genetic programming may be used to evolve teleo-reactive programs. Teleo-reactive programs may be represented as symbolic expressions that are recombined and mutated according to fitness through the standard strongly-typed genetic programming procedures. The block stacking problem has demonstrated how genetic programming is capable of evolving hierarchical and recursive teleo-reactive programs.

It is remarkable that hierarchical and recursive programs are able to successfully evolve when both the calls to other programs and the programs themselves are evolving in parallel with only the guidance of fitness-proportionate selection. The standard genetic programming technique was able to evolve novel solutions to the block stacking problem for an arbitrary number of blocks in any configuration. Not only do the evolved solutions not resemble the human-designed solution, they use completely different approaches. The evolved programs are simpler and have fewer rules and subprograms than the one produced by a human programmer.

It is rather surprising that only one hundred fitness cases selected randomly from the extremely vast state space of problems can guide genetic programming to evolve general plans for stacking blocks. The preliminary results presented in this paper indicate that genetic programming is well suited for learning teleo-reactive programs.

## 6   Further Work

Further work will be done using genetic programming to evolve teleo-reactive programs. Certainly, it would be interesting to see if genetic programming can evolve teleo-reactive programs for use in other, more complex domains.

The obvious next step is to evolve the rules that produce the higher-order predicates, such as *clear* and *ordered*, from the primitive predicate *on*. These rules would be evolved in parallel with the teleo-reactive programs that use these predicates, just as the main program and the three subprograms were evolved in parallel. In other words, the perception tower and action tower of the triple tower architecture would be evolved together.

## Acknowledgments

## References

1. M. Kochenderfer. Evolving teleo-reactive programs for block stacking using index-icals through genetic programming. In J. R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford*, pages 111–118. Stanford Bookstore, Stanford University, 2002.
2. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.
3. T. S. Motzkin. Sorting numbers for cylinders and other classification numbers. *Combinatorics, Proceedings of Symposia in Pure Mathematics*, 19:167–176, 1971.
4. N. Nilsson. Toward agent programs with circuit semantics. Technical Report STAN-CS-92-1412, Department of Computer Science, Stanford University, 1992.
5. N. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.
6. N. Nilsson. Learning strategies for mid-level robot control. Unpublished memo, Department of Computer Science, Stanford University, May 2000.
7. N. Nilsson. Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, 5:99–110, 2001.