Evolving Strategies for the Repeated Prisoner's Dilemma Game with Genetic Programming: Studying the Effect of Varying Function Sets

By Daniel J. Scali
2006 Undergraduate Honors Thesis
Advised by Professor Sergio Alvarez
Computer Science Department, Boston College

Abstract: This thesis examines the application of genetic programming to evolving strategies for playing an iterated version of the Prisoner's Dilemma game. The study examines the evolution of strategies for a single population of players pitted against a static environment, as well as the co-evolution of strategies for two distinct subpopulations of players competing against one another. The results indicate that the strategies that can be evolved are strongly influenced by the function set provided during the design process. In co-evolutionary runs in particular, the function set shapes the environment of opponents that an individual strategy is evaluated against. Experimental runs that alter the makeup of the function set facilitate a discussion of how different function sets and environments can lead to diverse strategies with varying levels of performance.
Keywords: genetic programming, prisoner's dilemma, game theory, function set

1. Introduction

The Prisoner's Dilemma has previously been used to show the validity and effectiveness of applying genetic algorithms in a game-theoretic context. [2,5]. The current body of research consists of various methods for representing strategies for the Repeated Prisoner's Dilemma. In each case, the representation scheme that is chosen provides a framework for the evolutionary process and dictates the number of previous moves that a given strategy can consider in the calculation of its next move. For example, Miller [12] modeled strategies as bit-string representations of finite state automata whereas Axelrod's [2] more direct approach used bit strings to reflect the last three moves of the game's results history. Some of these studies evolve strategies based on their performance against a fixed environment [6,12] while others have introduced the

idea of co-evolution – the process of assessing a given stra

		Robber 2	
		Deny (Cooperate)	*Confess (Defect)*
Robber 1	*Deny (Cooperate)*	3 , 3	10 , 1
	Confess (Defect)	1 , 10	5 , 5

Table 2.1 *The Prisoner's Dilemma with payoffs as time in jail*

Economic game theory provides tools for analyzing this situation. Table 2.1 models the above scenario as a strategic game, where Robber 1's jail sentence is always listed first. The strategies available for each prisoner boil down to two options: either confess to the crime (Defect from accomplice) or deny the allegations (Cooperate with accomplice).

Each player in the game has one objective: to minimize his time in jail. Robber 1 has no knowledge of what Robber 2's move will be. However, Robber 1 knows that if Robber 2 confesses, his best response is to confess – he receives only 5 years in jail if he confesses as opposed to 10 years if he denies the allegations. He also knows that if Robber 2 chooses to deny, his best response is to confess – he receives only 1 year in jail if he confesses as opposed to 3 years if he denies. No matter what Robber 2 does, it is always in Robber 1's best interests to confess. Thus, confession is a dominant strategy for Robber 1 [4]. Since, a similar analysis holds true for Robber 2, the dominant strategy equilibrium is for both robbers to confess. Curiously, even though `[Confess, Confess]` is a dominant strategy equilibrium, both parties would be better off if the outcome was `[Deny, Deny]`.

		Player 2	
		Cooperate	*Defect*
Player 1	*Cooperate*	R = 3 , R = 3	S = 0, T = 5
	Defect	T = 5 , S = 0	P = 1 , P =1

Table 2.2 *The Prisoner's Dilemma, with payoffs as points*

The game consists of four possible payoffs, which shall be abbreviated *R*, *T*, *S* and *P*. *R* is the reward for mutual cooperation, *T* is the temptation to defect, *S* is the sucker's payoff, and *P* is the punishment for mutual defection. In order for the Prisoner's Dilemma to be present, two relationships must apply. First, it must be true that $T > R > P > S$. This ordering preserves the proper incentive structure, as the temptation to defect must be greater than the reward for cooperation, and so on. Second, the reward for cooperation must be greater than the average of the temptation to defect and the sucker's payoff – i.e. $R > .5(T+S)$. This removes the ability of players to take turns exploiting each other to do better than if they played the game egoistically [1].

A further technical analysis shows how the `[Defect, Defect]` dominant strategy equilibrium is undesirable, but not easily avoidable. `[Defect, Defect]` is not considered an efficient outcome. An outcome in a game is considered *Pareto efficient* if no other outcome exists that makes every player at least as well off and at least one

Although the possibility of mutual cooperation emerging in such a situation seems bleak, cooperation can in fact be sustained when the game is played multiple times. The resulting game is called the

and defects otherwise. The `Pavlov` strategy can either defect or cooperate on the first move[1].

3. Genetic Programming

Genetic programming (GP) is an evolutionary computation technique which is based upon the genetic algorithm (GA) developed by Holland [7]. The genetic programming paradigm, as it will be used in the scope of this paper, was popularized by John Koza [8].

Genetic algorithms take advantage of the Darwinian concept of natural selection to evolve suitable solutions to complex problems. In the typical genetic algorithm, a possible solution to a problem – called an individual – is represented as a bit string. Each individual is assigned a fitness, which is simply a determination of how effective a given individual is at solving the problem. For example, when evolving a game-playing strategy, a fitness measure might be the number of points the individual scored in the game. To kickoff the evolutionary process, an initial population of individuals is generated through a random process as detailed by Koza [8]. Next, the fitness of each individual in the population is evaluated. Favoring individuals with the highest fitness,

solutions as actual computer programs. Programming languages such as Lisp are well-

to simulate the process of Darwinian evolution and natural selection thus building stronger populations from generation to generation. In reproduction, an individual is selected from the population and then simply copied into the new generation. The crossover operation, illustrated in Figure 3.2, selects two different individuals from the population, randomly selects one node from each to be the crossover point, and then swaps the subtrees found at the crossover nodes to create two new individuals for the new generation. Lastly, mutation introduces random changes into the new generation. When mutation (Figure 3.3) is applied, an individual is selected and a mutation point is selected at random. A randomly generated subtree is inserted at the mutation point and the mutated individual is added to the new population.

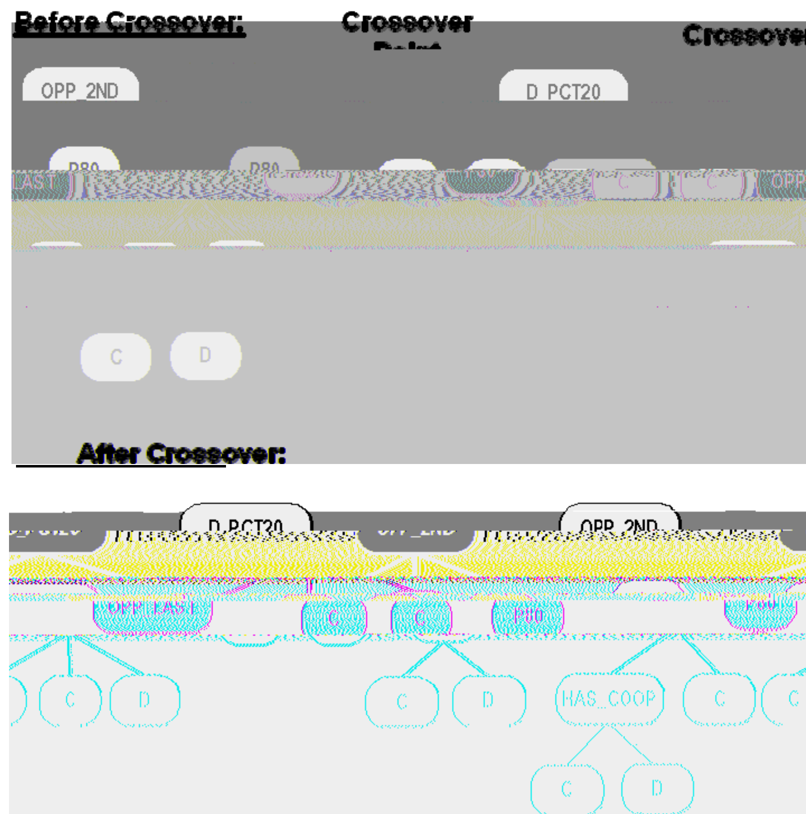

Figure 3.2 *An example of the crossover operation*

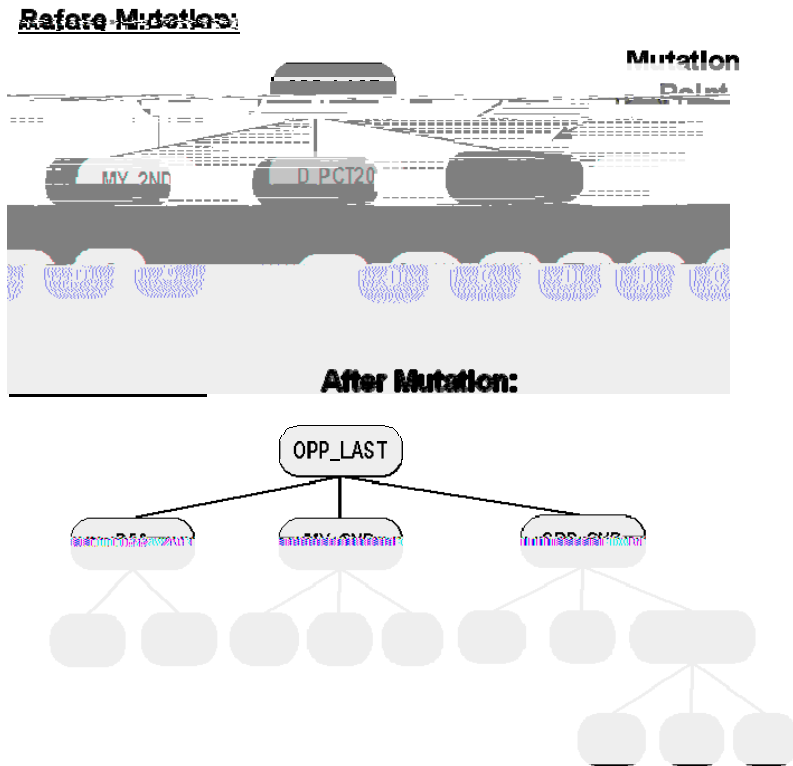

Figure 3.3 *An example of the mutation operation*

The method used to select individuals for the genetic operations can vary, but is typically based on fitness. Two popular selection methods are fitness-proportionate selection and tournament selection. For *fitness-proportionate selection*, the probability that a given individual is selected increases in proportion to

lexicographic parsimony pressure, *n* individuals are chosen at random. As in regular tournament selection, the individual with the best fitness is selected. However, in the event of tie, lexicographic parsimony pressure selects the individual with a smaller tree size. This multiobjective technique helps limit the size of the evolved trees and therefore reigns in bloat [10].

Selection methods can be supplemented by a technique known as *elitism*. The reader may have observed that the selection methods described above are probabilistic in nature – although they will select relatively more fit individuals, they do not necessarily ensure that the most fit individuals pass on to the next generation. If an individual is not

playing strategies. Co-evolution allows an individual's fitness to be calculated based on its performance playing a game against another evolving subpopulation.

5. Methodology

The Problem

In this version of the Repeated Prisoner's Dilemma, the payoff structure will be defined as [R, T, S, P] = [3, 5, 0, 1] as in Figure 1.2. Each individual plays the Prisoner's Dilemma game 100 times against the n opponents in its environment. For static environments, this means playing any number of pre-determined, well-known strategies.

most recent to least recent, respectively) while `OPP_LAST`, `OPP_2ND`, `OPP_3RD`, and `OPP_4TH` provide the same information for the opponent's moves.

Evolutionary Operators and Parameters

Unless otherwise noted, runs performed during the study used the following evolutionary operators and parameters: The evolutionary operators were crossover and reproduction, used at the probability of .7 and .3 respectively. Tournament selection with lexicographic parsimony pressure and a tournament size of 7 was used to select the individuals to be operated upon.

In runs against a static environment, a population of 500 individuals was evolved for 100 generations. For co-evolutionary environments, both subpopulations consisted of 500 individuals and were run for up to 500 generations.

The Experiments

Environment Name	Opponents in Environment
Environment 1	`All C`

Table 5.1 *Summary of the environments of pre-defined strategies used in genetic programming runs*

in the current evolutionary generation. Figure 5.2 shows a high level view of the experimental procedure. In Step 1, the function set was sequentially set equal to three predefined function sets: FS1, FS2, and FS3. The function set FS1 consisted of eight functions (MY_LAST, MY_2ND, MY_3RD, MY_4TH, OPP_LAST, OPP_2ND, OPP_3RD, and OPP_4TH) that could check the last four moves of each player in the history. After this, FS2 was constructed by adding the HAS_COOP and HAS_DEFECTED functions to FS1. The evolutionary process was again set in motion in all environments. Finally, the function set was reduced to FS3, a function set consisting of just two functions: MY_LAST and OPP_LAST.

```
Procedure:
   1. Set F = FS1 = {MY_LAST, MY_2ND, MY_3RD, MY_4TH, OPP_LAST,
      OPP_2ND, OPP_3RD, OPP_4TH}
   2. Run against all environments
   3. Set F = FS2 = {MY_LAST, MY_2ND, MY_3RD, MY_4TH, OPP_LAST,
      OPP_2ND, OPP_3RD, OPP_4TH, HAS_COOP, HAS_DEFECTED }
   4. Run against all environments
   5. Set F = FS3 = {MY_LAST, OPP_LAST }
   6. Run against all environments
```

Figure 5.2 *The experimental procedure and various function sets*

6. Results and Discussion

Static Environments

A summary of fixed (i.e. static) environment results can be found in Table 6.1. One of the most interesting static environments was Environment 3, which consisted of the opponents

Evolved Strategy:

`(OPP_LAST D (MY_LAST D C D) (OPP_2ND D C (MY_LAST C C D)))`

Evolved Strategy: **DD**DDDDDD**...**

`All D`: **DD**DDDDDD**...**

Environment Number	Function Set 1 (FS1)	Function Set 2 (FS2)	Function Set 3 (FS3)
	Evolved Strategy		

Table 6.1 *Summary of results for genetic programming runs against environments of pre-defined opponents*

Additionally, the results show that changes to the function set can limit this diagnostic ability and thus produce less-effective strategies. This became apparent when the function set FS3 was used to evolve strategies in the same environment (Environment 3). The results of this run are summarized in Figure 6.2.

Evolved Strategy:

```
(MY_LAST D C (OPP_LAST C D C))
```

D. For example, the best individual in Generation 5 was (MY_LAST D D D). As the graph reveals, no significant

strategy. In fact at one point (Generation 48), once the population was dominated by `Grim Trigger` strategies, the strategy `All C` emerged as the best in Subpopulation 2 since it was a smaller individual than `Grim Trigger` and performed just as well as `Grim Trigger`

however, the data shows that the Grim Trigger strategy performed comparably to Tit-for-Tat and the best-of-generation strategy.

The fact that Grim Trigger emerged from co-evolution speaks to the ability of genetic programming to evolve human-competitive solutions to game theoretic problems like the Repeated Prisoner's Dilemma. By using the populatr -0.T(i) -0(h) -0.1(e) -0.1 u

In co-evolution, the pre-defined opponents were replaced by a strategy's peers in the evolving population. `Grim Trigger`, a strategy regarded as a good general strategy in previous work, emerged as the most effective strategy. `Grim Trigger` was shown to perform similarly to `Tit-for-Tat` in this environment. The results suggest that changing the function set has an increased effect in co-evolution since the changes alter the environment of opponents.

The application of genetic programming to evolutionary game theory discussed in this paper suggests some interesting directions for future research. From a game-theoretic perspective, the results of the study might be analyzed further. Are strategies that emerge during evolution well-generalized strategies, evolutionary stable, or both? This study also points out that the results of genetic programming runs are sensitive to changes made to the function set. Additional research might build on these findings to provide a more complete understanding of the general characteristics of functions that, when added, tend to improve results.

8. Acknowledgements

Thank you to my advisor Professor Sergio

```
class GrimTrigger extends Strategy
{
    public char getMove()
    {
       if(hasDefected){
              return 'D';
       }else{
              return 'C';
       }
    }
}
```

```
class Pavlov extends Strategy
{
    public char getMove()
    {
       if(moveNumber == 0) return 'C';

       if(oppHistory[moveNumber-1] == myHistory[moveNumber-1]){
              return 'C';
       }else{
              return 'D';
       }
    }
}
```

```
class TitForTat extends Strategy
{
    public char getMove()
    {
       if(moveNumber == 0) return 'C';

       if(oppHistory[moveNumber-1] == 'C'){
           return 'C';
       }else{
           return 'D';
       }
    }
}
```

```
class CBTFT extends Strategy
{
    public char getMove()
    {
       if(moveNumber == 0) return 'C';

       if(oppHistory[moveNumber-1] == 'C'){
           return 'D';
       }else{
           return 'C';
       }
    }
}
```

```
class DBTFT extends Strategy
{
    public char getMove()
    {
       if(moveNumber == 0) return 'D';

       if(oppHistory[moveNumber-1] == 'C'){
           return 'D';
```

```
        }else{
            return 'C';
        }
    }
}
```

```
class DPC extends Strategy
{
    public char getMove()
    {
        if(hasCooperated){
            return 'C';
        }else{
            return 'D';
        }
    }
}
```

```
class DTFT extends Strategy
{
    public char getMove()
    {
        if(moveNumber == 0) return 'D';

        if(oppHistory[moveNumber-1] == 'C'){
            return 'C';
        }else{
            return 'D';
        }
    }
}
```

Appendix B: Java Source Code for Functions and Terminals

For brevity, the source code for the functions and terminals used in static environment experiments is not included. The source code shown here was used for co-evolution. It contains modifications to the classes used in the static environments which facilitate co-evolution and add the ability to specify a pre-defined strate

```
                            final int tree,
                            final GPIndividual typicalIndividual,
                            final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=0)
           state.output.error("Incorrect number of children for node " +
                           toStringForError() + " at " +
                           individualBase);
    }

    public void eval(final EvolutionState state,
                  final int thread,
                  final GPData input,
                  final ADFStack stack,
                  final GPIndividual individual,
                  final Problem problem)
    {
       PDdata data = ((PDdata)(input));
       data.x = 'D';
    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;

import ec.app.pd_compete.*;

public class HAS_COOP extends GPNode {
    public String toString() { return "HAS_COOP"; }

    public void checkConstraints(final EvolutionState state,
                            final int tree,
                            final GPIndividual typicalIndividual,
                            final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=2)
           state.output.error("Incorrect number of children for node " +
                           toStringForError() + " at " +
                           individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
    {
        PD pd = (PD)problem;

       boolean oppHasCooperated;
       if (pd.whosTurn == pd.P1){
           oppHasCooperated = pd.p2HasCooperated;
       }else if(pd.whosTurn == pd.P2){
           oppHasCooperated = pd.p1HasCooperated;
       }else{
           oppHasCooperated = pd.p3HasCooperated;
       }
```

```
        if(oppHasCooperated){
            children[0].eval(state,thread,input,stack,individual,problem);
        }else{
            children[1].eval(state,thread,input,stack,individual,problem);
        }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;

import ec.app.pd_compete.*;

public class HAS_DEFECTED extends GPNode {
    public String toString() { return "HAS_DEFECTED"; }

    public void checkConstraints(final EvolutionState state,
                                 final int tree,
                                 final GPIndividual typicalIndividual,
                                 final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=2)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
```

```
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class MY_2ND extends GPNode {
    public String toString() { return "MY_2ND"; }

    public void checkConstraints(final EvolutionState state,
                                final int tree,
```

```
public class MY_3RD extends GPNode {
    public String toString() { return "MY_3RD"; }

    public void checkConstraints(final EvolutionState state,
                                final int tree,
                                final GPIndividual typicalIndividual,
                                final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=3)
           state.output.error("Incorrect number of children for node " +
                            toStringForError() + " at " +
```

```
                              final int tree,
                              final GPIndividual typicalIndividual,
                              final Parameter individualBase)
{
```

```
                       individualBase);
}

public void eval(final EvolutionState state,
                 final int thread,
                 final GPData input,
                 final ADFStack stack,
                 final GPIndividual individual,
                 final Problem problem)
{
    PD pd = (PD)problem;

   char temp;
   if (pd.whosTurn == pd.P1){
       temp = (pd.p2Moves.get(1)).charValue();
   }else if(pd.whosTurn == pd.P2){
       temp = (pd.p1Moves.get(1)).charValue();
   }else{
       temp = (pd.p3Moves.get(1)).charValue();
   }

   switch(temp){
   case 'C':
       children[1].eval(state,thread,input,stack,individual,problem);
```

```
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
    {
         PD pd = (PD)problem;

        char temp;
        if (pd.whosTurn == pd.P1){
            temp = (pd.p2Moves.get(2)).charValue();
        }else if (pd.whosTurn == pd.P2){
            temp = (pd.p1Moves.get(2)).charValue();
        }else {
            temp = (pd.p3Moves.get(3)).charValue();
        }

        switch(temp){
        case 'C':
            children[1].eval(state,thread,input,stack,individual,problem);
            break;

        case 'D':
            children[2].eval(state,thread,input,stack,individual,problem);
            break;

        default:  // assume undefined
            children[0].eval(state,thread,input,stack,individual,problem);
            break;
        }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class OPP_4TH extends GPNode {
    public String toString() { return "OPP_4TH"; }

    public void checkConstraints(final EvolutionState state,
                            final int tree,
                            final GPIndividual typicalIndividual,
                            final Parameter individualBase)
    {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=3)
            state.output.error("Incorrect number of children for node " +
                            toStringForError() + " at " +
                            individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
```

```
                  final Problem problem)
{
    PD pd = (PD)problem;

   char temp;
   if (pd.whosTurn == pd.P1){
       temp = (pd.p2Moves.get(3)).charValue();
   }else if (pd.whosTurn == pd.P2){
       temp = (pd.p1Moves.get(3)).charValue();
   }else {
       temp = (pd.p3Moves.get(3)).charValue();
   }

   switch(temp){
   case 'C':
       children[1].eval(state,thread,input,stack,individual,problem);
       break;

   case 'D':
```

Appendix C: Java Source Code for the Prisoner's Dilemma Problem

The following Java source code is the formulation of the Prisoner's Dilemma problem that was used to co-evolve solutions in ECJ.

```
package ec.app.pd_compete;

import ec.util.*;
import ec.*;
import ec.gp.*;
import ec.gp.koza.*;
import ec.simple.*;
import ec.coevolve.*;

import java.util.*;

public class PD extends GPProblem implements GroupedProblemForm{
```

```
//Define payoff values here to make it easier to adjust them if needed
public static final int MUTUAL_COOP = 3;
public static final int MUTUAL_DEFECT = 1;
public static final int TEMPTATION = 5;
public static final int SUCKER = 0;

//A variable used in the functions from the function set to determine who
//is P1 and who is P2
public int whosTurn;
public static final int P1 = 1;
public static final int P2 = 2;
public static final int P3 = 3;

//comparison strategies
private Strategy p3;

//for printing p3 info
private int p3score = 0;
private int oppCount = 0;
private int maxScore = 0;

public Object protoClone() throws CloneNotSupportedException
{
    PD newobj = (PD) (super.protoClone());
    newobj.input = (PDdata)(input.protoClone());
    return newobj;
}
public void setup(final EvolutionState state,final Parameter base)
{
    // very important, remember this
    super.setup(state,base);

    input =(PDdata)state.parameters.getInstanceForParameterEq(
            base.push(P_DATA), null, PDdata.class);
    input.setup(state,base.push(P_DATA));

     p3 = new GrimTrigger();

     //set up random number generator
     rand = new MersenneTwisterFast(3252354);
}


public void preprocessPopulation( final EvolutionState state,
                                  Population pop )
{
   int opps = (state.parameters).getInt(
                 new Parameter("eval.subpop.0.num-rand-ind"));
   maxScore = opps*TEMPTATION*100;

    for( int i = 0 ; i < pop.subpops.length ; i++ )
        for( int j = 0 ; j < pop.subpops[i].individuals.length ; j++ )
             ((KozaFitness)(pop.subpops[i].individuals[j].fitness)).
                 setStandardizedFitness(state, (float)opps*TEMPTATION*100 );

}

public void postprocessPopulation( final EvolutionState state,
                                   Population pop )
{
    for( int i = 0 ; i < pop.subpops.length ; i++ )
        for( int j = 0 ; j < pop.subpops[i].individuals.length ; j++ )
        {
```

```
          state,threadnum,input,stack,((GPIndividual)ind[0]),this);
p1Move = input.x;
if(p1Move == 'C'){
   p1HasCooperated = true;
}else{
   p1HasDefected = true;
   p1defectCount++;
}
whosTurn = P2;
//Evaluate the individual to get Player 2's move
((GPIndividual)ind[1]).trees[0].child.eval(
          state,threadnum,input,stack,((GPIndividual)ind[1]),this);
p2Move = input.x;
if(p2Move == 'C'){
   p2HasCooperated = true;
}else{
   p2HasDefected = true;
   p2defectCount++;
}

//calculate each individual's payout based on given moves
result1 = getPayout(p1Move, p2Move);
result2 = getPayout(p2Move, p1Move);

//keep a tally for how each player is doing
sum1 += result1;
sum2 += result2;

//Update both players' move history
p1Moves.addFirst(new Character(p1Move));
p2Moves.addFirst(new Character(p2Move));

//--- Play TFT (or another Strategy) against subpopulation 2 ---//
```

Appendix D: Sample ECJ parameter file

This is the ECJ parameter file that was used for co-evolutionary runs with FS2

```
parent.0 = ../../gp/koza/koza.params

generations = 500
evalthreads = 1
```

```
pop.subpop.1.species.ind = ec.gp.GPIndividual

pop.subpop.0.species.ind.numtrees = 1
pop.subpop.0.species.ind.tree.0 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.0.tc = tc0
pop.subpop.1.species.ind.numtrees = 1
pop.subpop.1.species.ind.tree.0 = ec.gp.GPTree
pop.subpop.1.species.ind.tree.0.tc = tc0

pop.subpop.0.species.pipe = ec.breed.MultiBreedingPipeline
pop.subpop.0.species.pipe.generate-max = false
pop.subpop.0.species.pipe.num-sources = 3
pop.subpop.0.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.7
pop.subpop.0.species.pipe.source.1 = ec.breed.ReproductionPipeline
pop.subpop.0.species.pipe.source.1.prob = 0.3
pop.subpop.0.species.pipe.source.2 = ec.gp.koza.MutationPipeline
pop.subpop.0.species.pipe.source.2.prob = 0.0

pop.subpop.1.species.pipe = ec.breed.MultiBreedingPipeline
pop.subpop.1.species.pipe.generate-max = false
pop.subpop.1.species.pipe.num-sources = 3
pop.subpop.1.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline
pop.subpop.1.species.pipe.source.0.prob = 0.7
pop.subpop.1.species.pipe.source.1 = ec.breed.ReproductionPipeline
pop.subpop.1.species.pipe.source.1.prob = 0.3
pop.subpop.1.species.pipe.source.2 = ec.gp.koza.MutationPipeline
pop.subpop.1.species.pipe.source.2.prob = 0.0

breed.reproduce.source.0 = ec.parsimony.LexicographicTournamentSelection
gp.koza.mutate.source.0 = ec.parsimony.LexicographicTournamentSelection
gp.koza.xover.source.0 = ec.parsimony.LexicographicTournamentSelection
gp.koza.xover.source.1 = ec.parsimony.LexicographicTournamentSelection
select.lexicographic-tournament.size = 7

#add elitism
breed.elite.0 = 10
breed.elite.1 = 10

# We have one function set, of class GPFunctionSet
gp.fs.size = 1
gp.fs.0 = ec.gp.GPFunctionSet
# We'll call the function set "f0".  It uses the default GPFuncInfo class
gp.fs.0.name = f0
```

```
gp.fs.0.func.7.nc = nc3
gp.fs.0.func.8 = ec.app.pd_compete.func.OPP_3RD
gp.fs.0.func.8.nc = nc3
gp.fs.0.func.9 = ec.app.pd_compete.func.OPP_4TH
gp.fs.0.func.9.nc = nc3
gp.fs.0.func.10 = ec.app.pd_compete.func.HAS_DEFECTED
gp.fs.0.func.10.nc = nc2
gp.fs.0.func.11 = ec.app.pd_compete.func.HAS_COOP
gp.fs.0.func.11.nc = nc2

eval.problem = ec.app.pd_compete.PD
eval.problem.data = ec.app.pd_compete.PDdata

# The following should almost *always* be the same as eval.problem.data
# For those who are interested, it defines the data object used internally
# inside ADF stack contexts
eval.problem.stack.context.data = ec.app.pd_compete.PDdata
```

References

[1] Axelrod, Robert. *The Evolution of Cooperation*. New York: Basic Books, 1984.
[2] Axelrod, Robert. *The Complexity of Cooperation*. Princeton: Princeton University Press, 1997.
[3] Dacey, Raymond and Norman Pendegraft. "The Optimality of Tit-For-Tat." *International Interactions* 15, no. 1 (1988): 45-64.
[4] Dixit, Avinash and Susan Skeath. *Games of Strategy*. 2nd ed. New York: W.W. Norton, 2004.
[5] Fujiki, Cory and John Dickinson. "Using the Genetic Algorithm to Generate Lisp Source Code to Solve th