The Acquisition of Optimality Theoretic Systems*

William J. Turkel
University of British Columbia

bill@hivnet.ubc.ca
turkel@unixg.ubc.ca

Draft: 12 March 1994 - Comments welcome.

Abstract

This paper explores the connection between Optimality Theoretic (OT) models of natural language and the class of computational models known as Genetic Algorithms. Genetic algorithms have been used to model the acquisition of syntax cast in a Principles and Parameters (P&P) framework. Unlike the P&P work, where the genetic algorithm is added to the theory as the acquisition component, I make the strong claim that an OT system properly construed *is* a genetic algorithm. To the extent that genetic algorithms are an adequate model of acquisition, this entails that one Optimality Theoretic system can acquire another. I briefly describe a model of language acquisition where an OT system is used to acquire the constraint rankings of other OT systems. The model crucially depends on both serial and parallel operation, suggesting that both modes have a role in the formulation of Optimality Theory. In addition, the model provides a relatively detailed description of *Gen* which is consistent with the assumptions of Optimality Theory.

Contents

*Thanks to Carl Alphonce, Henry Davis, Mark Hewitt, Douglas Pulleyblank and William K. Thompson for ideas and discussion. All errors are mine.

1 Optimality Theory and Genetic Algorithms

In this section I develop the parallels between OT models and Genetic Algorithms (henceforth GAs) by relating each to an abstract system.

1.1 An Abstract System

Consider an abstract system with the following characteristics. One part of the system is a generating function or mechanism which creates a number of

symbolic objects. These objects must be quite similar to one another, but they tend not to be identical. Call the generating mechanism a *generator*. Another part of the system is an *evaluator*, which tests each of the objects for suitability to some task. The evaluator considers each object independently of the others, and returns one or more as its best candidates. The system acts iteratively, such that the best candidates from one pass are fed into the generator and used as the seeds for the creation of the next set of candidates.

The overall architecture of the system is abstract enough that it can be used for a variety of tasks. For example, changing the nature of the generator will have an effect on the kinds of symbolic objects processed by the system. Changing the nature of the evaluator will result in different objects being designated 'best'. As described, the abstract system is a simple optimization mechanism, which generates candidates and tests them. When operated iteratively, it demonstrates evolutionary behaviour. Each generation of objects is slightly better (on average) than the previous, because the best candidates of the previous generation give rise to the candidates of the current generation.

1.2 Optimality Theory

The description of the abstract system is also a fairly good characterization of the theoretical machinery of Optimality Theory [McP 1993a, McP 1993b, PSm 1993]. In Optimality Theory, a function *Gen* creates a set of candidate outputs. "Gen contains information about the representational primitives and their universally irrevocable relations..." and "...generates for any given input a large space of candidate analyses by freely exercising the basic structural resources of the representational theory." [PSm 1993], pp. 4,5. The candidate analyses are then tested against a set of ranked constraints with the following assumptions. The constraints are violable, but the violation incurred by the optimal candidate is minimal. The notion of minimal violation is cast in terms of *harmony*, where the most harmonic candidate least violates the constraint set. The constraints are ranked on a language-particular basis, and the notion of minimal violation is defined in terms of this ranking. The constraint ranking consists of a *strict dominance hierarchy* of constraints, such that each constraint has absolute priority over all the constraints lower in the hierarchy.

The iterative operation of the abstract system best corresponds to the architectural variant of Optimality Theory known as *harmonic serialism* [PSm 1993].

> Perhaps nearest to the familiar derivational conceptions of grammar is what we might call 'harmonic serialism', by which Gen provides a set of candidate analyses for an input, which are harmonically evaluated; the optimal form is then fed back into Gen, which produces another set of analyses, which are then evaluated; and so on until no further improvement in representational Harmony is possible. (p. 4)

There are other variants, in particular one where all of the candidates are produced by *Gen* in one step and evaluated in parallel. This parallel mode tends to be assumed in most OT work. In this paper we will assume that both variants are available as a part of Universal Grammar.

The operation of an OT system is usually represented in tableaus. The constraints are ranked from left to right as columns of the table, and the candidates are listed on separate rows. At each point where a candidate violates a constraint, a star is placed in the cell at the intersection of the appropriate row and column. If a candidate violates a particular constraint `n` times, there will be `n` stars in that cell. We refer to the assignment of a star to a candidate as the *assessment of a mark*. The collection of all marks incurred by a candidate `c` is denoted *marks(*`c`*)*. As an example, say a particular candidate twice violates the requirement that syllables have onsets (the Ons constraint). Then *marks(*`c`*)* is { *ONS *ONS }.

An OT system differs from the abstract system primarily in the richness of its evaluator mechanism. Optimality Theoretic analyses tend to emphasize the role of constraint interaction, and to downplay the role of *Gen*, assuming that the generator does what is necessary under the circumstances.

1.3 Genetic Algorithms

The description of the abstract system is also a good characterization of Genetic Algorithms, a family of computational models inspired by evolution [Hol 1975/92, Gbg 1989, Wht 1993]. A genetic algorithm operates over a set of simple chromosome-like data structures. The data structures are usually bit strings which encode a proposed solution to some problem. (In the work of Clark, e.g.,

[Clk 1992a, ClR 1993], the bit strings are parameter settings for hypothesized P&P theory grammars). The GA typically starts with a randomly generated population of these chromosomes, evaluates the *fitness* of each, and gives the best a chance to reproduce. Reproduction of the chromosomes involves the recombination of the parental information in such a way as to create new chromosomes. These offspring are added to the population, the fitness of each chromosome is tested, and the best are again more likely to reproduce. Over time, the average fitness of the population rises. The process is iterated until an optimal or near-optimal chromosome is found.

Since genetic algorithms are a general technique for optimization, they do not specify the nature of the representations to be tested or the nature of the fitness function in any detail. These are assumed to be part of the problem domain.

A brief description of the operation of a typical GA follows.[1] Say we want to find an eight bit string which starts with a *0* and has no identical adjacent elements. First we generate a random population of bit strings. (For convenience, I have given each a name, although this is not part of the algorithm).

Randomly Generated Initial Population

Name	*Contents*
A	10110111
B	01110110
C	10001011
D	11000001
E	01010001

Now we test the fitness of each of the chromosomes. Say that the fitness function returns 100 minus 10 for each identical adjacent pair, and minus 10 for a string-initial *1*. So the fitness of chromosome C (10001011) will be (100 - (3 * 10) - 10) or 60.

Fitness of Each Chromosome

Name	*Contents*	*Fitness*
A	10110111	60
B	01110110	70
C	10001011	60
D	11000001	40
E	01010001	80
Sum		310

[1]The problem is obviously trivial, and was chosen merely to illustrate the workings of the algorithm.

So the most fit member of our randomly generated initial population is chromosome E, and the least fit is chromosome D. Now we allocate reproductive opportunities on the basis of fitness. This is usually accomplished by scaling the fitnesses so that they sum to 1, then probabilistically copying each into an intermediate generation of the same size.

Scaled Fitness of Each Chromosome

Name	*Contents*	*Fitness*	*Scaled Fitness*
A	10110111	60	0.1935
B	01110110	70	0.2258
C	10001011	60	0.1935
D	11000001	40	0.1291
E	01010001	80	0.2581
Avg		51.66	
Sum		310	1

So we have about a 19% chance of seeing a copy of chromosome A in the intermediate generation, 23% chance of seeing chromosome B, and so forth. We generate a random number for each slot in the intermediate generation, and check the number against the following cumulative probability table.

Cumulative Probability Table

Name	*Probability Range*
A	0 to 0.1935
B	0.1936 to 0.4193
C	0.4194 to 0.6128
D	0.6129 to 0.7419
E	0.7420 to 1

Say that the random numbers we generate are {0.961, 0.518, 0.012, 0.361, 0.751}. Then our intermediate population will look like that below.

Intermediate Population

Name	*Contents*
E	01010001
C	10001011
A	10110111
B	01110110
E	01010001

Note that there are two copies of chromosome E, and that chromosome D is now extinct. We can fill in the fitnesses as before. Note that the the average fitness is higher in the intermediate population than it was in the original population.

Fitnesses of Intermediate Population

Name	*Contents*	*Fitness*	*Scaled Fitness*
E	01010001	80	0.2286
C	10001011	60	0.1714
A	10110111	60	0.1714
B	01110110	70	0.2000
E	01010001	80	0.2286
Avg		58.33	
Sum		350	1

If selection were all that there was to the genetic algorithm, it would eventually converge on the best candidate in the starting population. Since there are 256 possible bit strings of length 8, the chances of finding the optimal chromosome among the members of the starting population is 5/256, or about 2%. Consequently, the GA must have a means of introducing variation into the population.

Genetic algorithms introduce variation by using mechanisms which are analagous to those found in nature, namely recombination and mutation of genetic material. Mutation is accomplished by occasionally flipping the value of a bit. Recombination is accomplished with various crossover operators. One common operator is *one point crossover* which works in the following manner.

Take a pair of members out of the intermediate population, and line them up one over the other. Pick a randomly selected point, and cut both strings at that point. Then join the head of each string to the tail of the other. If we take chromosomes E and A, and cross them over at position 5, we will get two new chromosomes, call them F and G.

One Point Crossover

Parents	*Crossover*	*Offspring*
01010001	01010 001	10110001
	X	
10110111	10110 111	01010111

Now the fitness of chromosome F (10110001) is 60 and the fitness of chromosome G (01010111) is 80. We create another population which has seven elements, by adding F and G. We then scale the fitnesses, and probabilistically select five of the seven for the next generation. If we wished, we could implement mutation at this stage by giving each bit in the population a very low probability of flipping.

Population after Reproduction

Name	*Contents*	*Fitness*	*Scaled Fitness*
E	01010001	80	0.1633
C	10001011	60	0.1224
A	10110111	60	0.1224
B	01110110	70	0.1429
E	01010001	80	0.1633
F	10110001	60	0.1224
G	01010111	80	0.1633
Avg		70	
Sum		490	1

Note that the average fitness has again risen, this time as the result of reproduction, rather than selection. Over time, the average fitness will approach the maximum fitness (in this case 100) and the population will be composed almost entirely of optimal members (01010101). At this point, we can say that the algorithm has found the optimal candidate.

1.4 Evolutionary Operation of Optimality Theoretic Systems

We have shown that Optimality Theoretic systems and Genetic Algorithms are both very similar to an abstract system which generates symbolic objects, tests them, and returns a subset of the objects which are better suited to some task. It follows that OT systems and GAs are similar to one another, and we make the following observation: *one theory is articulated precisely where the other is not.* Genetic algorithms are very detailed in their specification of the generator (postulating operations such as recombination and mutation) but they are not detailed in their specification of the fitness function or the nature of representations. On the other hand, Optimality Theoretic systems provide an explicit evaluator and set of primitives for the representations, but do not describe the generator in any detail.

I propose that we enrich our notion of an Optimality Theoretic system in the following ways:

- *Gen* is to be understood as operating in a fash-

ion similar to the generator of GAs. It can take a single symbolic object (in this case a generally well-formed linguistic representation) and modify (i.e., mutate) it in such a way as to produce another valid object. It can also take a pair of representations and recombine them in such a way as to produce a different pair of generally well-formed representations. This corresponds to Prince & Smolensky's free exercise of basic structural resources.

- Constraint hierarchies are represented in such a way that they count as well-formed linguistic representations, and can thus be manipulated by *Gen*.

- Optimality Theoretic systems are capable of two modes of operation. The first is the parallel operation which is assumed in most work. The parallel mode is demonstrably crucial for a number of linguistic analyses (see e.g., [PSm 1993] Chapters 4 & 7). The second mode of operation is serial. As we have seen, the genetic algorithm crucially relies on iterative behaviour to find an optimal or near-optimal candidate. By operating iteratively, the GA has to consider fewer alternatives in each generation, and explores only part of the search space in each cycle. In the application to which we will put the serial-mode OT system, namely acquiring constraint rankings, the search space will be too large to explore in parallel. With `n` constraints, we will have `n!` possible rankings. We need to test only a very small subset of possible rankings with each pass of the system, and converge towards the correct ranking over time.

2 Genetic Algorithms as a Theory of Acquisition

The process of language acquisition can be seen as an *adaptive system* in that it is

> ...a problem of optimization made difficult by substantial complexity and uncertainty. The complexity makes discovery of the optimum a long, perhaps never-to-be-completed task, so the best among the *tested* options must be exploited at every step. At the same time, the uncertainties must be reduced rapidly, so that knowledge of *available* options increases rapidly. More succinctly, information must be exploited as acquired so that performance improves apace. [Hol 1975/92], p. 1

Holland developed Genetic Algorithms to model such adaptive systems. In this section I describe the work of Robin Clark [Clk 1990, Clk 1992a, Clk 1992b, ClR 1993] which uses GAs as a model of syntactic acquisition.

2.1 The Problem of Language Acquisition

If language learning is a faculty which can be usefully modelled algorithmically,[2] then the process will be inherently bounded in terms of the computational resources (space and time) that it has at its disposal. We can frame the problem as follows: how does a learning device with limited computational resources use input data to form hypotheses? One assumption that has proven to be particularly fruitful is the idea that the learner comes pre-wired to make certain kinds of generalizations. So in a Principles & Parameters system, there are a finite number of domains of finite variation, the *parameters* of the system. The learner has a certain amount of flexibility, and yet is still able to acquire the system in a reasonable amount of time.

If the parameters were all independent of one another, then a learning algorithm would simply look for evidence for the setting of each, and set it in the correct way. The usefulness and appeal of the Principles & Parameters approach comes from the fact that the parameters are not independent of one another, however. Instead the parameters interact with one another in complex ways.

Some approaches to parameter-setting rely on this interaction, and attempt to use deductive methods for acquisition. Deductive methods suffer high computational cost, and can be equivalent to brute-force enumeration in the worst case [Clk 1992a]. For any reasonable number of parameters (i.e., around 30) the search space is too large for such enumeration to be a useful strategy.

The methods of acquisition presented here are nondeductive instead. The genetic algorithm can locate a target without the computational cost of the

[2] Obviously I assume that it is. My presentation of this material is based on [Clk 1992a].

deductive approaches, and is robust in the sense that it can deal with noisy, equivocal data.

2.2 The Selection of Syntactic Knowledge

Clark [Clk 1992a] proposes that parameter-setting is accomplished with a genetic algorithm, and describes a model with the following components:

- Bit strings which represent hypothesized vectors of parameter settings
- A one point crossover operator
- A mutation operator
- A fitness function with three components
- Allocation of reproductive opportunities based on fitness

The most significant aspects of the model are those of any application of genetic algorithms, namely the representation of solutions to the problem and the fitness function. Consequently, the discussion here will focus on those components.

The representations which the GA processes are bit strings corresponding to parameter settings.

> In terms of an actual parsing framework, there would be a fixed central algorithm, corresponding to UG. Within this algorithm would be various flags, indicating the points where code must be inserted for the parser to function. The *0*s and *1*s in the hypothesis strings could be interpreted as pointers to the parameterized code. Upon receiving an hypothesis string, the machine would look up the various pieces of code indicated by the *0*s and *1*s and systematically substitute the code it finds for the flags in the main algorithm. The result would be a special parsing device designed to analyze the language enumerated by the hypothesis string. Thus a 'self-constructing' parser would be the ensemble of the core algorithm, the parameterized code, and a learning device that would select the appropriate hypothesis string in response to the input text. [Clk 1992a], p. 112

The learner is error-driven, in that hypotheses are changed on the basis of evidence from the external environment, with the requirement that the new hypothesis better account for the data.

The fitness metric consists of a summation function and two additional components. The summation function adds up the number of violations in the various modules of the parser and passes the sum to the learner. The learner does not have access to information about which components were violated, merely the gross amount of violation incurred. One of the other components rates subset hypotheses more highly than superset hypotheses, and the other prefers elegant hypotheses (i.e., those which lead to compact representations) over inelegant hypotheses. The fitness metric is designed so that it can distinguish between the performance of various hypotheses, even when none of them correctly deals with an input datum.

The overall operation of the system is as follows. An initial population of distinct hypothesis strings are generated randomly. A parsing device is constructed for each, and the parsing devices are tested against input sentences. The fitness of each hypothesis is used to allocate reproductive opportunities. Reproduction consists of crossover. Mutation is applied to the population, and the least fit elements are removed. If the target sequence has been reached, the algorithm halts, otherwise it creates parsing devices for the current population and iterates.

2.3 P-encodings

Recent work on the learnability of natural language in a P&P framework has suggested that there might be *triggers* in the PLD, where a trigger is an input sentence which causes the learner to set the value of a given parameter a certain way. Clark defines *parameter expression* as

> Parameter Expression
>
> A sentence ρ expresses a parameter p_i just in case a grammar must have p_i set to some definite value in order to assign a well-formed representation to ρ.
> [Clk 1992a]

Given parameter expression, Clark defines *p-encodings*, where the p-encoding may be thought of as a 'pure' representation of the parameters expressed in

an input sentence. He represents p-encodings as binary vectors with a don't care symbol (I will use $\diamond$). So a sentence which indicated that the first and fifth parameters should be set to *1* and the second parameter set to *0* would have a p-encoding like (1 0 $\diamond$ $\diamond$ 1 $\diamond$ $\diamond$ $\diamond$).

In principle, the sentences in an input text can be replaced with their p-encodings. This allows the frequency of parameter expression in the PLD to be studied. In [Clk 1990], this technique is used to simulate parsing for the purposes of studying the learnability of large systems of parameters.

3 A Model of Language Acquisition

In this section I discuss the problem of acquiring Optimality Theoretic systems. I first present the learnability work of Tesar & Smolensky [TS 1993] and discuss how it fits into a theory of acquisition. I then compare the problem of acquiring constraint rankings with the problem of parameter setting. Finally, I describe the way that a serial-mode OT system can acquire other (presumably parallel-mode) OT systems.

3.1 The Learnability of Optimality Theory

Tesar & Smolensky [TS 1993] describe a learning algorithm (Recursive Constraint Demotion) which takes pairs consisting of an input and its well-formed (optimal) parse, and outputs a *stratified hierarchy* of constraints.

A stratified hierarchy is a constraint ranking where some of the constraints are not ranked with respect to one another (members of a stratum) but each dominates the remaining constraints (which are not members of the stratum).

The key idea behind the learning algorithm is that the marks incurred by any suboptimal parse must outrank the marks incurred by the optimal parse. For every positive datum of an input and its optimal parse, any alternative analysis we may generate will be suboptimal. This allows us to determine constraint rankings from pairs of inputs and optimal parses.

From the point of view of a model of language acquisition, the assumption of having the optimal parse available as part of the input is problematic. A preferable model would be able to learn under conditions of partial information and occasional errors in the input. Nevertheless, the learning algorithm is a useful contribution, and may be integrated into models of acquisition.

3.2 The Problem of Acquiring Constraint Rankings

Tesar & Smolensky neatly summarize the acquisition problem in the context of Optimality Theory [TS 1993]. They note that under the assumption of innate knowledge of the universal constraints, the primary task of the learner is the determination of the dominance ranking of the constraints particular to the target language. They also point out that "In many respects, ranking of universal constraints in Optimality Theory plays a role analagous to parameter-setting in principles-and-parameters theory."

The problem of acquiring Optimality Theoretic systems differs from the problem of parameter setting in a couple of ways. For one thing, the search space is much larger. Assuming `n` binary parameters, there will be 2^n possible grammars. Assuming `n` constraints, there will be `n!` possible grammars. As a point of comparison, with 30 parameters, there are 1,073,741,824 possible grammars. But with 30 constraints, there are about 26,500,000,000,000,000,000,000,000,000,000 possible grammars. A theory which is equivalent to brute-force enumeration is untenable for parameter-setting, but is inconceivable for constraint ordering. As Clark has said about a similar case, "If this method were taken as a model of learning we would be legitimately disappointed by it as a theory of actual language acquisition since only immortal babies would ever find the method useful." [Clk 1992b]

Another difference is that the OT parser will be uniform across different constraint rankings. Unlike the P&P implementation, which required a new parser to be constructed for each hypothesized set of parameter settings, the OT parser is already constructed, and merely processes with different constraint rankings. Its overall operation does not need to change with each hypothesis.

Finally, note that the genetic algorithm was basically external to the machinery of the Principles & Parameters theory. Under the view presented here, the genetic algorithm *is* the machinery of Optimality Theory. Instead of grafting on a separate learning device, we say that the OT system is organized such

that it can act as its own learning device. Thus we are able to avoid the proliferation of theoretical machinery.

3.3 The Architecture of the Model

In this section I provide a top-down decomposition of the proposed model.[3]

3.3.1 Instantiating Grammars

I assume that UG includes a set of universal constraints and the machinery required to implement Optimality Theoretic systems. At the uppermost level, we have a higher-order function which instantiates specific grammars.

INSTANTIATE A GRAMMAR

(instantiate-grammar
  (lambda (*mode ranking***) ...))**
  → *a grammar*

Given a constraint ranking and a mode of operation (i.e., serial or parallel) return a grammar.

A GRAMMAR

(grammar
  (lambda (*input***)**
    ... (H-eval ... (Gen *input***))))**
  → *output*

A grammar maps a single input to a single output. Internally, it is organized into two modules, a generator and an evaluator. The input is connected to the generator, the generator is connected to the evaluator, and the evaluator is connected to the output.

The standard assumption about the generator is that it takes a single representation and returns a set of representations consisting of modifications to the input. I will assume that the generator takes a set of representations and returns a set of representations. If the input set contains one element, then the generator returns a number of variations on that element (this is the standard operation). If the input set is empty, then the generator randomly creates a set of appropriate representations and returns that. If the input set contains more than one representation, then the generator returns a set consisting of new representations built from the bits and pieces of the representations in the input set.

THE GENERATOR

(Gen
  (lambda (*input***) ...))**
  → *output set*

e.g.,
(Gen $\emptyset$ **)**
→ $\{random_1, random_2, ..., random_n\}$

(Gen *input* **)**
→ $\{modification_1, modification_2, ..., modification_n\}$

(Gen $\{input_1, input_2, ..., input_n\}$ **)**
→ $\{recombination_1, recombination_2, ..., recombination_n\}$

The evaluator takes a set of representations and evaluates them against a set of constraints. I assume that the constraints are built into the evaluator during the instantiation of the grammar. Under standard assumptions, the output of the evaluator is the single best member of the input set. We will have to assume, however, that the evaluator can sometimes return a set consisting of good members of the input set. Without getting into the details yet, let's say that when the output set contains more than one member, its cardinality will still be less than that of the input set. Intuitively, not all of the members of the input set will be equally harmonic, and we wish to return some subset whose members are more harmonic than the rest.

THE EVALUATOR

(H-eval
  (lambda (*mode input-set***) ...))**
  → *set of more/most harmonic members*

[3] I use a syntax based on the Scheme dialect of LISP to describe the basic structure of the components. `Typewriter font` is used for procedure names. *Italics* are used for formal parameters and UPPERCASE is used for actual arguments. For the interested reader, [FF 1989] is a good introduction to Scheme.

Given a set of input representations, and a mode of operation (i.e., return most harmonic member or set of more harmonic members) return a set of output representations. An empty output set corresponds to the *null parse* of [PSm 1993] Chapter 4. If the output set contains one member, it will be the most harmonic of the input set. If the output set contains more than one member they will be the more harmonic members of the input.

We now have enough machinery to describe some of the architectural variants of the theory. Consider first the standard parallel mode Optimality Theoretic system which is assumed in most work. We assume that the constraint ranking has been determined for the language. The mode of operation of the grammar is PARALLEL, and `H-eval` will return a set consisting of the single most harmonic member of its input.

Adult Parallel Mode OT System

```
(instantiate-grammar
  PARALLEL
  LANG-SPECIFIC-RANKING)
→
(grammar
  (lambda (input)
    (H-eval MOST (Gen input))))
```

The next variant which we might wish to consider is the harmonic serial mode OT system. In this case, the mode of operation is SERIAL. I assume that `instantiate-grammar` adds iterating code similar to that shown below. `H-eval` is still required to return the single most harmonic candidate found in a given pass through the loop.[4] I leave the exact formulation of the loop termination test open.

Adult Serial Harmonic Mode OT System

```
(instantiate-grammar
  SERIAL
  LANG-SPECIFIC-RANKING)
→
(grammar
  (lambda (input)
    (if no-further-improvement?
      input
      (grammar
        (H-eval MOST (Gen input))))))
```

To get an OT system with evolutionary operation, we will need a serial mode grammar where `H-eval` returns a set of more harmonic members on each pass through the loop. The nature of the constraint ranking (M-RANKING) and the loop termination test (`converged?`) are discussed below. The function `FM` simply returns the foremost element of a list.[5] Since the input set has converged, any element in it is optimal, including the first.

Evolutionary Serial Mode OT System

```
(instantiate-grammar
  SERIAL
  M-RANKING)
→
(grammar
  (lambda (input-set)
    (if converged?
      (FM input-set)
      (grammar
        (H-eval MORE (Gen input-set))))))
```

3.3.2 Representations

The grammars which we instantiate will have to process linguistic representations. If we wish to use an OT system to acquire another OT system, then the grammars will also have to process constraint rankings. I assume that constraint rankings are the same kind of symbolic objects as well formed linguistic representations. As a first approximation, let us say that constraint rankings are represented as lists of symbols (much like LISP programs). Symbol lists can also be used to represent feature bundles, sets, trees and other linguistically relevant data structures.[6]

3.3.3 Mutation and Recombination

We have seen that when the generator receives a single input, it outputs a set of modifications of that input, and when it receives a set of inputs, it outputs

[4] Contrary to [PSm 1993] Chapter 5 which claims that the serial/parallel distinction pertains to *Gen*, I suggest that it is actually a characteristic of the `Gen`/`H-eval` loop.

[5] This is from [PSm 1993] p. 69. Obviously it is equivalent to the LISP `car`.

[6] The process of Genetic Programming [Kza 1992] takes a related approach; genetic algorithms operate in a search space where the chromosomes are LISP programs rather than bit strings.

a set of recombinations. We have to look at the operation of the generator when dealing with constraint rankings, and when dealing with linguistic representations. I will cover only the former in any detail, although I will suggest some ways in which the latter might be implemented.

If the generator is creating modifications of a single input, then the sorts of operators which we want to use will be analogous to the mutation operator of the traditional GA. Some examples of operators are swapping adjacent or non-adjacent pairs of constraints, reversing segments of the list, rotating the list to the left or right (so that the first element becomes the last, or vice versa) and so on. The action of a few sample operators is illustrated below.[7]

Sample Mutation Operators

`(swap-adjacent` (A B C D E F))
→ (A C B D E F)

`(swap-non-adjacent` (A B C D E F))
→ (A E C D B F)

`(reverse-segment` (A B C D E F))
→ (A E D C B F)

`(rotate-left` (A B C D E F))
→ (B C D E F A)

`(rotate-right` (A B C D E F))
→ (F A B C D E)

If the generator is creating new objects from bits and pieces of old ones, then the sorts of operators we will want to use will be analogous to the crossover operator of the traditional GA. Operators which work with two lists will have to be designed so that one constraint does not appear at two points in the list. For this reason, the one point crossover algorithm will not work, although many suitable algorithms can be devised. For example, consider one based upon *mark cancellation*.[8] Mark cancellation takes two lists of symbols and recursively cancels any common symbols, one pair at a time. Say we want to crossover two constraint ranking lists. We first pick a point on each list to serve as the crossover point. We remove the head of each list at that point, and attach it to a tail consisting of the other list from which all of the head symbols have been cancelled with mark cancellation.

Support Routines

`(adjoin-lists` (A B) (C D E F))
→ (A B C D E F)

`(head-list` 2 (A B C D E F))
→ (A B)

`(cancel-marks` (A E) (A B C D E F))
→ (B C D F)

The last routine cancels the common elements from a pair of lists, one pair at a time. Crucially, I assume that the algorithm does not affect the order of the remaining elements in the lists.

Mark Cancellation Crossover

```
(mc-crossover
  (lambda ( point list1 list2 )
    (adjoin-lists
      (head-list  point list1 )
      (cancel-marks
        (head-list  point list1 )
        list2 ))
    (adjoin-lists
      (head-list  point list2 )
      (cancel-marks
        (head-list  point list2 )
        list1 ))))
```

e.g.,

```
(mc-crossover
  2 (A E B C F D) (D E C A B F) )
```

→ (A E D C B F) (D E A B C F)

Start with two lists, *a* and *b*. Pick a point on each list and cut the list at that point, returning *head-a* and *head-b*. Use mark cancellation to remove the members of *head-a* from *b*, returning *b-prime*, and *head-b* from *a*, returning *a-prime*. Now adjoin *head-a* to *b-prime* and *head-b* to *a-prime*.

[7] Obviously, this is not a complete set. It should also be noted that some mutators can be implemented in terms of others.

[8] This algorithm is sketched in [PSm 1993].

Trace of Mark Cancellation Crossover

INPUT: (A E B C F D) (D E C A B F)

(head-list 2 (A E B C F D))
→ (A E)
(head-list 2 (D E C A B F))
→ (D E)
(cancel-marks (A E) (D E C A B F))
→ (D C B F)
(cancel-marks (D E) (A E B C F D))
→ (A B C F)
(adjoin-lists (A E) (D C B F))
→ (A E D C B F)
(adjoin-lists (D E) (A B C F))
→ (D E A B C F)

OUTPUT: (A E D C B F) (D E A B C F)

This gives us a recombination algorithm which is akin to one point crossover, but which can be implemented primarily in terms of primitives which the theory already requires for other purposes.

The mutation and recombination operators which we have provided allow the generator to easily explore the possibility space of constraint rankings. When it has to explore the space of linguistic representations, however, its operation will need to be both more powerful, and more tightly constrained. It will have to be more powerful in the sense that it can explore permutations of lexical and structural objects, and can manipulate nested lists. But it will be more constrained because not every list of linguistic symbols will be wellformed, unlike every list of unique constraints.[9]

The power of the generator can be increased by giving it recombination and mutation operators which work with nested lists. These would probably consist of tree-rearranging and tree-pruning mechanisms, facilities for adjoining trees and cancelling common subtrees and so forth.[10] The generator can be constrained by adding filters which rule out symbolic lists that do not correspond to valid (i.e., generally well-formed) linguistic objects. There is no reason for such filters not to be implemented as an OT system of very high level constraints.

3.3.4 Operational Overview

We are now in a position to ask how the system is supposed to work.

At the beginning of the acquisition process, the language learner has an evolutionary serial mode OT system instantiated. There are no inputs to the generator at first, so it randomly creates a set of constraint rankings.[11] (This is what I meant by 'appropriate' in the initial discussion of the generator). The system will be exposed to primary linguistic data (PLD) and will test the fitness of each of its candidate constraint rankings against the M-Ranking. The most fit constraint rankings (i.e., the most harmonic, with respect to the M-Ranking) will be passed back into the generator for the next iteration of the system. The generator will apply its recombination (and possibly its mutation) operators to the rankings to produce new rankings. That set of rankings will again be evaluated against the M-Ranking, and the system will eventually converge to the target constraint ranking. At that point, the adult parallel mode system is instantiated with the target ranking.

3.3.5 What is the M-Ranking?

The crux of the system is the fitness testing. I will build up the notion of M-Ranking one step at a time.

Consider a single candidate tested against three binary constraints. Instead of creating the familiar tableau, we can write this as a binary number, where *1* is equal to a star and *0* to an empty cell. The best candidate possible would be one which did not violate any constraints *000*. The next best candidate would

[9] Actually, there may be some constraint lists which are not well-formed. For example, the system might not allow the so-called undominated constraints to be dominated. It might also rule out any ranking which does not maintain the relative order of the peak and margin hierarchies [PSm 1993] Chapter 8. If such well-formedness requirements hold of constraint lists, they can be easily added to the system presented here.

[10] Again the reader is referred to [Kza 1992], which discusses such processes for LISP programs. For a related proposal, see the discussion of *treebot ecology* in [Clk 1992b]. See also the discussion in [PSm 1993] p. 79, footnote 49.

[11] It may be the case that there is some default order to the constraints which is also innate. In this model, such information could be reflected in the initial population (e.g., the majority of the initial set of rankings could be the default ranking) or it could be reflected in the recombination operators of the generator. In the latter case, the system might be predisposed towards those recombinations which would lead to a default ranking. I set aside the matter here, because it does not affect the formulation of the model at this level. Note the parallels between the Superset Principle of [TS 1993] and the space of hypotheses processed at each point during acquisition. At first, any ranking may be hypothesized, but as the system starts to converge toward the target ranking, some orderings will no longer be considered. Unlike the Recursive Constraint Demotion algorithm, however, individual hypotheses consist of total rankings and not stratified hierarchies.

violate only the lowest ranked constraint *001*. In order from best to worst, the candidates are *000*, *001*, *010*, *011*, *100*, *101*, *110*, *111*. Let's call this measure *absolute harmony*. The absolute harmony of a candidate for `n` constraints will be equal to $(n-x)/n$ where `x` is the binary value corresponding to the candidate's row in the tableau. This measure captures the nature of candidates assessed against a strict dominance hierarchy in an absolute rather than relative fashion.[12] The basic idea is that of Tesar & Smolensky; "The fact that surface forms are optimal means that every positive example contains a great number of implicit negative examples: for any given input, every candidate output other than the correct form is ill-formed." [TS 1993]. Unlike Tesar & Smolensky, however, I do not use that insight as the basis for a deductive system. Rather, I note that, all other things being equal, the forms output by the grammar will *tend to have a low absolute harmony*. So we want our idea of the fitness of hypothesized constraint rankings to reflect this tendency.

Note that a system which uses absolute harmony as the core measure of fitness only has access to how good something is overall, and not to which constraints have been violated. In this respect, the model is similar to the P&P implementation.

How does absolute harmony help us? Say that we are able to instantiate a parallel mode grammar with a hypothesized constraint ranking. When we test the PLD against this constraint ranking, we get absolute harmony values. For now, assume that we have perfect information about which constraints are obeyed by a surface form and which constraints are violated.

Now we wrap our parallel mode grammar in some kind of an averaging function, which returns the average absolute harmony of PLD surface forms when assessed against a particular hypothesized constraint ranking.

The M-Ranking is almost like an inverse of the *encapsulated constraints* of [PSm 1993] Chapter 8. Instead of packaging up the results of a number of constraints into a composite constraint, the M-Ranking takes a single value (the average absolute harmony of the PLD assessed against a hypothesized constraint ranking) and *returns it as if it were a row in a constraint tableau*. In other words, the M-Ranking is a single function which simulates the effect of a candidate tested against a number of constraints. This means that the evaluator can use the standard tableau evaluation mechanism to assess the output of the M-Ranking. M-Ranking stands for *Meta-Ranking...* it is not really a ranking, it just simulates one for the purposes of assessing hypothesized constraint rankings.

Let's look again at the core notion of absolute harmony. There are two questions we might ask. The first is, does it work? Consider some circumstances under which it would fail. One possibility is that the correct (target) grammar would have the most highly ranked constraints uniformly violated, and would relegate all meaningful constraint interplay to the lowest ranked constraints. This would compete against incorrect grammars whose higher ranked constraints occasionally (but inconsistently) licensed the data. Such a situation would cause the model presented here to go awry. Whether or not systems with uniformly violated high ranking constraints can exist for natural languages is another question. To me, it seems unlikely. Another case where absolute harmony would fail (pointed out to me by Douglas Pulleyblank, personal communication) is where the constraints are independent, or not crucially ranked with respect to one another. In this case, a single independent constraint would tend to travel towards the bottom of the constraint hierarchy. Multiple independent constraints would also travel to the bottom of the hierarchy, but would not converge on some ranking with respect to one another. As long as the child can press one of these rankings into service in the adult grammar, however, it doesn't matter which is chosen.

The second question we might ask is, does it require the power of arithmetic to compute the absolute harmony for a datum or the average absolute harmony for a hypothesized constraint ranking? I think not. The measure of absolute harmony was cast in terms of binary numbers for expository purposes. We can get essentially the same functionality after recasting it in terms of the cancellation of marks. Say that we keep a mark list for a datum assessed against the hypothesized constraint ranking. Furthermore, suppose that both negative and positive marks are kept in the mark list, so if the datum is twice licensed by Ons, the marks { $\surd$ONS $\surd$ONS } will be added to the list. Now if we test a number of PLD against the constraint ranking, we can keep adding the positive and negative marks they incur to the same mark list.

[12] I make two simplifying assumptions: all constraints can be cast as binary constraints, and constraint violation is all-or-none. So each cell in this tableau can only contain one star at most. As will be seen, the system presented here can be readily generalized to more complex constraints and tableaus.

So the list will contain all of the marks accrued by data that system is exposed to.

When we have accumulated the marks of enough data[13] we can compute an 'average' in the following way. For each mark in the list, we recursively attempt to cancel its opposite from the rest of the list. What's left when we are finished will be a measure of how the constraint ranking performed against the data overall. I describe the algorithm below, and work through its operation for a simple set of constraints and PLD.

Miscellaneous Support Routines

`(adjoin-lists` () (A B C))
→ (A B C)

`(adjoin-lists` (A B C) ())
→ (A B C)

`(empty-list?` (A B C D E F))
→ *false*

`(empty-list?` ())
→ *true*

`(in-list?` (C) (A B C))
→ *true*

`(in-list?` (E) (A B C))
→ *false*

`(rest` (A B C D E F))
→ (B C D E F)

`(rest` (F))
→ ()

`(opposite` *ONS)
→ √ONS

`(opposite` √ONS)
→ *ONS

[13] See [Clk 1992b] for arguments that a hypothesized grammar should evaluate individual examples in the context of the input stream of data. Following Clark, I assume that the system can simulate memory for past examples, and I implement this by allowing the system to test each hypothesized constraint ranking against a number of PLD. Note that since individual hypothesized grammars are independent of one another, they can be tested in parallel.

Overall Performance

```
(overall-performance
  (lambda (accrued-marks)
    (let* ((current (FM accrued-marks))
      (op-current (opposite current))
      (tail (rest accrued-marks)))
    (if (empty-list? accrued-marks)
      accrued-marks
      (if (in-list? op-current tail)
        (overall-performance
          (cancel-marks op-current tail))
        (adjoin-lists
          current
          (overall-performance tail)))))))
```

We can now work through a simple example. Say that we have a set of three constraints which we are trying to find a ranking for, call them A, B, and C. We also have three PLD, call them D_1, D_2, and D_3. The marks incurred by each of the data are as follows.

Marks Incurred by Each

marks(D_1) = { *A *A √B *C *C *C }
marks(D_2) = { *A √A √B √C }
marks(D_3) = { *A *B √B *C √C }

Under the assumption that we are accruing marks for all the PLD in a single list, we will have

Marks Accrued by All PLD

accrued-marks = { *A *A *A *A √A
*B √B √B √B
*C *C *C *C √C √C }

When we apply the function to test the overall performance of the accrued marks, we get

Overall Performance for PLD

`(overall-performance` *accrued-marks*)
→ { *A *A *A √B √B *C *C }

Deductively, we could now conclude that the ranking (B C A) will give the best results with this PLD. Let's look at why. Assuming all-or-none constraint violation, there are eight possible sets of marks a candidate could be assessed.

Logical Possibilities for Assessed Marks

{ √A √B √C }
{ √A √B *C }
{ √A *B √C }
{ √A *B *C }
{ *A √B √C }
{ *A √B *C }
{ *A *B √C }
{ *A *B *C }

But we have already noted that the optimal candidate violates constraints A and C. So the question which we want to ask is, for a given constraint ranking, how many candidates could the optimal form compete against, and still be optimal? Let's look at each of the six rankings in turn. For the rankings (A B C) and (C B A) the optimal candidate is going to be assessed marks for the first and last column, and so will have a row in the tableau which looks like $*\surd*$. Of the eight possible candidates, the only two which the optimal form could defeat are the ones whose rows will look like $**\surd$ and $***$. On the other hand, there are five candidates who would defeat the optimal candidate if the correct ranking were one of the above two. Those are the candidates whose rows will look like $\surd\surd\surd$, $\surd\surd*$, $\surd*\surd$, $\surd**$, and $*\surd\surd$. If the correct constraint ranking is (A B C) or (C B A), then we have to prevent our generator from creating those other five candidates somehow, because otherwise one of them would be optimal.

Using a similar logic, we see that the rankings (A C B) and (C A B) each have an optimal form with a $**\surd$ row. This means that the optimal form could defeat only the worst candidate ($***$) in direct competition, and that we would have to find a way to prevent the other six logically possible constraints from even entering the picture.

So we know that the best ranking to explain our PLD is either (B A C) or (B C A). To deductively distinguish between the two, we need to abandon our all-or-none logic, and note that constraint A is violated more frequently than constraint C. Thus we conclude that (B C A) is the correct ranking, in that it gives the grammar the most latitude in generating the form which we observe as part of the PLD.

This is not a deductive model, however. Instead of using logic similar to that above to select a constraint ranking, we allocate reproductive opportunities to the rankings which will maximize the number of candidates which the optimal candidate can defeat. We do this by testing each hypothesized constraint ranking in a tableau.

Say that we have the rankings (C B A), (A C B) and (B A C) as our hypotheses. We put the overall performance of each into a tableau, and pass the whole thing back to `H-eval` to be evaluated. Since we are operating with the assumption that `H-eval` can carry a population of candidates from one iteration to the next, the constraint rankings are going to get better over time. The tableau which is fed into `H-eval` by the M-Ranking function will look like this:

What M-Ranking Returns to H-eval

Hypothesis	C_1	C_2	C_3
(C B A)	**	$\surd\surd$	***
(A C B)	***	**	$\surd\surd$
(B A C)	$\surd\surd$	***	**

H-eval will, of course find (B A C) to be the most harmonic, then (C B A), and finally (A C B). This is exactly what we want.

3.3.6 Noisy Data

Up until this point we have been operating under the simplifying assumption that we have perfect information about the marks which a particular input datum is going to be assessed by each candidate. It is more likely, however, that input forms are going to contain partial and contradictory information about the constraints which they violate, and the constraints which license them. We have already taken care of contradictory information with our measurement of overall performance. How does the model handle partial information?

We can make use of a notion similar to p-encoding: every input datum will express one of three things about every constraint. Either the datum will violate a given constraint, or it will be licensed by that constraint, or it will be indifferent to it. I will represent such indifference with a prefixed diamond. We now want `H-eval` to be sensitive to the three way distinction. So we say that marks (i.e., stars) are antiharmonic (after [PSm 1993] Chapter 5), that positive marks (i.e., checks) are harmonic, and that indifferent marks (i.e., diamonds) are neither.

$$\emptyset \succ^* *C$$
$$\emptyset \approx^* \diamond C$$
$$\surd C \succ^* \emptyset$$

3.3.7 Converging on the Target Constraint Ranking

The last element of the model which needs to be covered is convergence on the final ranking. It is possible that the system will not stop testing hypotheses until it reaches the exact ranking which it is attempting to acquire. In this case, each of the members of the set of constraint rankings will be equal to the target ranking.

Since the process is error-driven, however, it will settle into its final state when it can no longer discriminate between pairs of constraint rankings in terms of their ability to generate the target language. Thus, the final ranking may or may not correspond to the exact ranking of the target. Clark and Roberts [ClR 1993] use this as a model of diachronic change.

It gives us a picture of language variation as well. Two idiolects may differ in the ordering of a few low level constraints. Two dialects differ in terms of a few intermediate level constraints, and so forth.

3.3.8 Loose Ends

There are many details of the model which need to be elaborated on. For example, the population size has to remain relatively constant from one iteration to the next. So there has to be some mechanism by which the generator and evaluator can work independently on the population without changing its size too much.

Another area which needs work is the allocation of reproductive opportunities. This will have to be specified for `H-eval`. How does the evaluator pick the more harmonious members of an input set? How many copies of each are passed into the generator on the next iteration? These issues are related to the problem of maintaining a constant population. Ideally, they will be implemented without recourse to the power of arithmetic.

Finally, the nature of the mutation and recombination operators for linguistic representations are of interest, because it is these which will determine whether or not the generator presented here is a good model of the *Gen* of Optimality Theory.

3.4 Traps

Certain formal relationships between parameters and between parameter values can lead to what Clark [Clk 1992a] has called *traps*. One example is the *subset condition* of [Bwk 1985] which states that the learner must guess the smallest language compatible with the input. Failure to do so can lead to a situation where the learner hypothesizes a superset language, and is unable to retreat from this incorrect guess because no positive evidence bears on the issue. Clark has also described a trap known as *shifting* where the learner can "...obey the Subset Condition on the microscopic level (with respect to a single parameter) while violating it on the macroscopic level (due to shifting interactions between parameters)" [Clk 1992a], p. 102; original discussion in [Clk 1990].

In the P&P implementation of a GA learning theory, the fitness function was specifically designed so that the system would avoid subset condition violations and other 'traps' that can arise in parameter setting [Clk 1992a].

Can constraint rankings lead to violations of the subset condition? In other words, is it possible to order two constraints such that the learner will overgenerate, and there will be no positive evidence that he or she could use to correctly reorder the constraints? To my knowledge, the relevant work has not been done for Optimality Theory yet. When we have a knowledge of the kinds of traps which must be avoided in constraint ranking, then we can adjust the fitness function to penalize those kinds of hypotheses which will lead the learner into such traps. Until then, I merely note that this remains an area of potential difficulty.

4 Conclusion

In conclusion, we have shown that Optimality Theoretic systems are capable of acting as genetic algorithms under certain cirmcumstances. This provides us with a robust and efficient mechanism for acquiring OT systems, and does not require that we add anything to the theory which has not already been proposed. Furthermore, the model of acquisition presented in this paper provides some internal structure for *Gen* and suggests that both serial and parallel modes of operation are available as a part of Universal Grammar.

References

[Bwk 1985] Berwick, R. *The acquisition of Syntactic Knowledge* Cambridge, MA: MIT Press.

[Clk 1990] Clark, R. *Papers on Learnability and Natural Selection.* Technical Reports in Formal and Computational Linguistics, No. 1, Université de Genève.

[Clk 1992a] Clark, R. The Selection of Syntactic Knowledge. *Language Acquisition* 2:85-149.

[Clk 1992b] Clark, R. Finitude, Boundedness and Complexity: Learnability and the Study of First Language Acquisition, ms., University of Pennsylvania.

[ClR 1993] Clark, R. & I. Roberts. A Computational Model of Language Learnability and Language Change. *Linguistic Inquiry* 24(2).

[FF 1989] Friedman, D. P. & M Felleisen. *The Little LISPer.* New York, NY: Macmillan.

[Gbg 1989] Goldberg, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading, MA: Addison-Wesley.

[Hol 1975/92] Holland, J. H. *Adaptation in Natural and Artificial Systems.* Cambridge, MA: MIT Press.

[Kza 1992] Koza, J. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

[McP 1993a] McCarthy, J. J., & A. Prince *Prosodic Morphology I: Constraint Interaction and Satisfaction*, ms., University of Massachusetts, Amherst and Rutgers University.

[McP 1993b] McCarthy, J. J., & A. Prince *Generalized Alignment*, ms., University of Massachusetts, Amherst and Rutgers University.

[PSm 1993] Prince, A., & P. Smolensky *Optimality Theory: Constraint Interaction in Generative Grammar*, Rutgers University Center for Cognitive Science Technical Report 2.

[TS 1993] Tesar, B., & P. Smolensky. The Learnability of Optimality Theory: An Algorithm and Some Basic Complexity Results, ms., University of Colorado, Boulder.

[Wht 1993] Whitley, D. A Genetic Algorithm Tutorial, Colorado State University Computer Science Technical Report CS-93-103.