Appears in 1996 Conference on Genetic Programming, pp. 315–320.

Toward Simulated Evolution of Machine-Language Iteration

Lorenz Huelsbergen
Bell Laboratories
Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
lorenz@research.bell-labs.com

Abstract

We use a simulated evolution search (genetic programming) for the automatic synthesis of small iterative machine-language programs. For an integer register machine with an addition instruction as its sole arithmetic operator, we show that genetic programming can produce exact and general multiplication routines by synthesizing the necessary iterative control structures from primitive machine-language instructions. Our program representation is a virtual register machine that admits arbitrary control flow. Our evolution strategy furthermore does not artificially restrict the synthesis of any control structure; we only place an upper bound on program evaluation time. A program's fitness is the distance between the output produced by a test case and the desired output (multiplication). The test cases exhaustively cover multiplication over a finite subset of the natural numbers ($\mathbb{N}_{10}$); yet the derived solutions constitute general multiplication for the positive integers. For this problem, simulated evolution with a two-point crossover operator examines significantly fewer individuals in finding a solution than random search. Introduction of a small rate of mutation further increases the number of solutions.

1 Introduction

Genetic programming (GP) [8] uses principles of evolution (populations, fitness measures, recombination) to perform directed non-linear searches in the space of computer programs. GP has successfully produced solution programs in many problem domains, *e.g.* [7, 9, 8]. For the most part however, GP solutions are restricted to non-iterative (non-looping) programs. To both broaden the domain of GP applications and to scale to larger problem sizes, it is necessary to extend GP to iterative programs. To this end, we study the evolution of *unrestricted iteration* in the context of machine-language programs.

Little GP research to date addresses the problem of general iteration. Early work modeled the synthesis of machine-language programs as evolutionary processes [5, 4], but did not outperform random search. Applications of genetic programming that do use iteration do so by restricting its form to a single type of loop and by bounding its extent to a fixed number of iterations. Cramer [3] describes simulated evolution of programs in a machine language with a high-level LOOP construct parameterized by an expression e and an integer variable denoting the number of iterative executions of e; the system externally limits a program's running time. Koza [8] similarly introduces the *Do-Until* (DU) iterative language construct; parameterized by an expression e and a predicate p, DU iteratively evaluates e until p is true. Each DU operator in the program is, however, restricted in that the number of its iterations is externally bounded from above. Kinnear also uses a specialized loop construct, *dobl*, to evolve sorting algorithms [6]. As with iteration, GP experiments with recursion restrict the scope and generality of the recursion. Recent work by Brave [2] investigates restricted recursion over trees to solve state-space exploration problems. The external system again restrictively curbs the maximum depth of a recursion. With constructs such as LOOP, DU, or *dobl* as its only means of iteration, GP can only evolve programs with structured control flow.

An alternate approach to the problem of GP iteration is taken here—the automatic synthesis of general (*i.e.* unrestricted) iteration from control-flow primitives. Our program representation is an instance of a virtual register machine (VRM). The particular VRM under investigation (denoted VRM-$\mathcal{M}$) is a small integer instruction set that includes add, move, increment/decrement, set/clear, compare, and branch (conditional and unconditional) instructions. To synthesize a *for* loop in

Address	Instruction	Comment
0:	`Cmp(R2,R3)`	compare counter R_2 to zero ($R_3 = 0$)
1:	`Je(+4)`	if counter zero, then exit
2:	`Add(R0,R1)`	$R_0 \leftarrow R_0 + R_1$
3:	`Dec(R2)`	decrement counter
4:	`J(-4)`	and loop ...

Figure 1: A hand-coded VRM-$\mathcal{M}$ program that solves the multiplication problem. The input multiplier and multiplicand are initially in registers R_1 and R_2; other registers are initially zero. The product accumulates in R_0.

VRM-$\mathcal{M}$ for example, GP must construct a counter, an increment or decrement of this counter, a comparison of the counter to its terminal value, and appropriate branch instructions based on this comparison.[1] Since loops are not explicit in this representation, it is not feasible to place time bounds on individual loops (*cf. Do-Until* [8])—as in *e.g.* [3], we instead limit the total number of instructions a program may execute, *i.e.* its running time.

This paper's result is that GP can automatically synthesize multiplication routines[2] (*solutions*) by constructing a loop that has the VRM-$\mathcal{M}$'s `Add` instruction at its core. (A hand-coded solution to the *multiplication problem* is in Figure 1.) Informally, the fitness of an individual VRM-$\mathcal{M}$ program p is computed by exhaustively evaluating p with inputs (i, j), for all $i, j \in \mathbb{N}_{10}$, and computing the distance of p's result from the desired product ij. Although the test cases cover only a small finite subset of the natural numbers, they suffice to produce general multiplication (bounded only by a machine's register precision) for the positive integers. All solutions discovered to date are general.

Genetic programming finds solutions to the multiplication problem in the VRM-$\mathcal{M}$ language using a single genetic operator (two-point crossover) to recombine fit individuals during the construction of successive generations. Introduction of a mutation operator—applied with low probability—further increases the likelihood that a given GP run finds a solution. This problem is solved exactly by only a small percentage ($\approx 3\%$) of the GP runs. Even so, we find that GP solution of this problem (with and without mutation) processes significantly fewer individual programs than a random search.

The results presented here were obtained in a new system—called Finnegan and briefly described in the next section—for experimentation with simulated evolution of programs. Section 3 specifies the virtual machine language; Section 4 details the experimental setup for the multiplication problem. Results are in Section 5. We conclude with a discussion in Section 6.

[1] A high-level *if* control structure also requires such synthesis from primitive VRM-$\mathcal{M}$ instructions.

[2] Cramer [3] also studied simulated evolution of multiplication in machine languages with high-level loop constructs.

2 Finnegan System

Finnegan is a framework for experimenting with simulated evolution. It is written in the Standard ML (SML) programming language [12] and implemented using the Standard ML of New Jersey (SML/NJ) compiler [1]. The SML/NJ implementation supports parameterized modules [10] which are indispensable in an experimentation framework. SML's strong static typing ensures that module implementations match their declared interfaces (signatures). In Finnegan, one is therefore able to quickly interchange GP's variable components (*e.g.*, representations, fitness functions, selection mechanisms) by selecting modules from libraries of such components.

3 Virtual Register Machine $\mathcal{M}$

The representation used for the multiplication problem is an instance of a virtual register machine called VRM-$\mathcal{M}$. The notation VRM-$\mathcal{M}_{(m,n)}$ names a particular VRM-$\mathcal{M}$ that consists of external state (m integer registers), internal state (a program counter and flag) and a sequence of n immutable instructions.

3.1 External State: Registers

We define the *register state* as a vector

$$\vec{R} \equiv \langle R_0, \ldots, R_{m-1} \rangle$$

of m integers; the register state constitutes the machine's mutable memory. The precision of a register is inherited from the underlying implementation.[3] Most program instructions (*e.g.*, `Add`) modify registers directly. All program input and output is communicated through the register state; that is, program inputs are supplied in the initial state $\vec{R}_{initial}$ and outputs are taken from the final register state, $\vec{R}_{final}$. Registers may also be used to initially supply the program with constants. Alternately, the program can synthesize necessary constants in otherwise unused registers.

[3] Integers in the SML/NJ compiler used for Finnegan are signed and 31 bit.

3.2 Internal State: PC, Flag

In addition to the external register state, VRM-$\mathcal{M}$ maintains two pieces of internal state: a program counter (PC) and a comparison flag (*Flag*). The program counter is an integer that selects which instruction to fetch and execute. Branch instructions modify the PC to point to the branch's target; all other instructions always increment the PC to point to the next instruction. The *Flag* reflects the result of the last comparison instruction executed. It can assume the values *less*, *greater*, and *equal*. *Flag* is initially undefined which we denote as $\perp$ (bottom). Only the comparison instructions (see below) can modify the *Flag* state.

3.3 Instruction Set

A *program* is a vector of n instructions:

$$\vec{I} \equiv \langle I_0, \ldots, I_{n-1} \rangle$$

The program counter naturally corresponds to an index of $\vec{I}$. A program *terminates* when $PC = n$; that is, when evaluation steps off the end of the program. (Our evaluation strategy also limits the maximum number of instructions evaluated; see Section 5.) Figure 2 contains VRM-$\mathcal{M}$'s instructions and their operational semantics. The instruction set consists of a single nullary instruction `Nop` which does nothing, a single arithmetic instruction `Add`, a register move instruction `Mov`, a register comparison instruction `Cmp`, an unconditional branch `J`, conditional branches (`Jl`, `Jg`, `Je`), instructions to initialize registers (`Set` and `Clear`), and instructions to increment (`Inc`) and decrement (`Dec`) a given register.

Since any finite implementation of integer arithmetic may result in overflow or underflow, we define the arithmetic operators $\oplus$ and $\ominus$ (used in Figure 2) to be conventional integer addition and subtraction respectively, except that they produce zero when the result would exceed the implementation's precision.

Branches are always relative to the program counter. We chose to forego absolute branch addressing since relative addressing readily admits code relocation, which is in essence what the application of genetic operators (§4.4 below) to VRM-$\mathcal{M}$ code does. Note that the operational semantics rewrites a branch to an address < 0 as a branch to I_0 (*i.e.* $PC \leftarrow 0$) and a branch past the end of the program ($n-1$) as termination (*i.e.* $PC \leftarrow n$). A jump instruction I_j can therefore branch to any one of $n+1$ distinct addresses.

The number of syntactically-distinct instructions for VRM-$\mathcal{M}_{(m,n)}$ is

$$S(m,n) \equiv 3m^2 + 4m + 4(n+1) + 1 \qquad (1)$$

because there are three binary register instructions, four unary register instructions, four relative branch instructions, and one nullary instruction (`Nop`). The number of possible (syntactic) programs in a such a machine is then:

$$S(m,n)^n \qquad (2)$$

3.4 Evaluation Function

Evaluation of an n-instruction VRM-$\mathcal{M}_{(m,n)}$ program $\vec{I}$ is with respect to an m-register input state $\vec{R}$ and an integer number ($k > 0$) of evaluation steps (instructions). The evaluation function $\mathcal{E}$ maps a triple to a pair:

$$\mathcal{E} : \left(\vec{I}, \vec{R}, k\right) \rightarrow \left(\vec{R}', PC\right) \qquad (3)$$

$\mathcal{E}$ produces a pair consisting of the final register state $\vec{R}'$ and final PC after evaluation of at most k instructions. A program that terminates within k instructions has $PC = n$. Since our VRM-$\mathcal{M}$ evaluator is an interpreter (essentially Figure 2), it can be easily halted after evaluation of k instructions.

4 Experimental Setup

In addition to a program representation, a GP experiment is parameterized by a number of variable components: a set of test cases, a fitness function, a population selection mechanism, and a set of genetic operators. This section fixes the components used in our experiments with the multiplication problem. We defer elaboration of the quantitative settings (*e.g.*, population size) to the next section (§5).

4.1 Test Cases

A test case t for the multiplication problem is a pair (i,j). The *answer* for t is the product ij. Over any infinite domain, the set of all test cases is also infinite; we therefore restrict ourselves to test cases over a finite subset of the natural numbers, $\mathbb{N}_{10}$.[4] Denote the set of all such test cases as T:

$$T \equiv \{(i,j) \,|\forall i, \forall j \in \mathbb{N}_{10}\} \qquad (4)$$

4.2 Fitness Function

We define registers R_1 and R_2 as input registers (multiplier and multiplicand) and register R_0 as the output register (product).[5] The raw fitness of an n-instruction program $\vec{I}$ for test-case $t \equiv (i,j)$ is given by

$$\mathcal{F}_{(i,j)}\left(\vec{I}\right) \equiv |R'_0 - ij| + |PC - n| \qquad (5)$$

where, using (3), $\mathcal{E}(\vec{I}, \vec{R}, k) = (\langle R'_0, \ldots \rangle, PC)$, input registers $\vec{R} \equiv \langle 0, i, j, 0, \ldots, 0_{m-1} \rangle$, and k is the maximum

[4] We note that GP driven by this set finds exact solutions to the multiplication problem; we have not ascertained whether a smaller test-case set would also suffice (see §5 and §6).

[5] A VRM-$\mathcal{M}$ that can solve the multiplication problem must therefore contain at least $m = 3$ registers.

$$
\begin{aligned}
\texttt{Nop} &\equiv \left(PC \leftarrow PC + 1 \right. \\
\texttt{Mov}(R_{dst}, R_{src}) &\equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{src} \end{array} \right. \\
\texttt{Add}(R_{dst}, R_{src}) &\equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{src} \oplus R_{dst} \end{array} \right. \\
\texttt{Cmp}(R_a, R_b) &\equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ \mathit{Flag} \leftarrow \left\{ \begin{array}{ll} \mathit{less} & \text{if } R_a < R_b \\ \mathit{greater} & \text{if } R_a > R_b \\ \mathit{equal} & \text{otherwise} \end{array} \right. \end{array} \right. \\
\texttt{J}(\mathit{offset}) &\equiv \left(PC \leftarrow \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) \right. \\
\texttt{Jl}(\mathit{offset}) &\equiv \left(PC \leftarrow \left\{ \begin{array}{ll} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } \mathit{Flag} = \mathit{less} \\ PC + 1 & \text{otherwise} \end{array} \right. \right. \\
\texttt{Jg}(\mathit{offset}) &\equiv \left(PC \leftarrow \left\{ \begin{array}{ll} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } \mathit{Flag} = \mathit{greater} \\ PC + 1 & \text{otherwise} \end{array} \right. \right. \\
\texttt{Je}(\mathit{offset}) &\equiv \left(PC \leftarrow \left\{ \begin{array}{ll} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } \mathit{Flag} = \mathit{equal} \\ PC + 1 & \text{otherwise} \end{array} \right. \right. \\
\texttt{Inc}(R_a) &\equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_a \leftarrow R_a \oplus 1 \end{array} \right. \\
\texttt{Dec}(R_a) &\equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_a \leftarrow R_a \ominus 1 \end{array} \right. \\
\texttt{Set}(R_a) &\equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_a \leftarrow 1 \end{array} \right. \\
\texttt{Clear}(R_a) &\equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_a \leftarrow 0 \end{array} \right.
\end{aligned}
$$

Figure 2: Operational semantics for the virtual register machine VRM-$\mathcal{M}$ used to solve the multiplication problem. Evaluation commences with $PC = 0$ and $\mathit{Flag} = \perp$ (uninitialized). The arithmetic operators $\oplus$ and $\ominus$ are defined to be conventional integer addition and subtraction respectively, except that when their result exceeds the implementation's precision (overflow or underflow), they produce zero.

number of evaluation steps. The total *raw fitness* of a program $\vec{I}$ with respect to the test-case set T is the sum of the raw fitness values of the individual test cases:

$$\mathcal{F}\left(\vec{I}\right) \equiv \sum_{t \in T} \mathcal{F}_t\left(\vec{I}\right) \tag{6}$$

The fitness function has two parts: the distance of a computed result from the desired product and the distance of the program counter (after k steps) from the halting position n. The $|PC - n|$ term favors—in some sense—programs that after k steps are "close" to the proper termination point, n. In this manner, $\mathcal{F}$ selects for programs that resemble multiplication and that terminate.

4.3 Population Selection

Population selection, for the construction of successive generations, is performed via *proportional selection* (see, *e.g.*, [8, 11]). Let P be a population (set) of N programs. Our selection mechanism first normalizes an individual's raw fitness (6) to the unit interval:

$$\widehat{\mathcal{F}}\left(\vec{I}\right) \equiv \frac{1}{\mathcal{F}\left(\vec{I}\right) \sum_{p \in P} \mathcal{F}\left(p\right)^{-1}} \tag{7}$$

We then select a N random reals, $X \equiv \{x_1, \ldots, x_N\}$, in the unit interval. For each $x \in X$, we select program $\vec{I}$ for the next generation if $x < \widehat{\mathcal{F}}\left(\vec{I}\right)$ and there does not exist an $\vec{I}'$ such that $x < \widehat{\mathcal{F}}\left(\vec{I}'\right) < \widehat{\mathcal{F}}\left(\vec{I}\right)$.

4.4 Genetic Operators

We use a single recombination operator that performs two-point crossover. For some experiments, we augment this operator with mutation.

4.4.1 Two-Point Crossover

Crossover of two n-instruction programs $\vec{I}_i$ and $\vec{I}_j$ first selects a subsequence of instructions starting at a random point $0 \leq p_i < n$ in program $\vec{I}_i$. The length l of the subsequence is chosen randomly such that $p_i < p_i + l < n$. A random point p_j, $0 \leq p_j + l < n$, is then chosen in program $\vec{I}_j$. Finally, the l instructions in $\vec{I}_i$ starting at p_i are interchanged with the l instructions in $\vec{I}_j$ starting at p_j.

Crossover in a population P is performed by first selecting a subset $P' \subseteq P$ of programs from the population; an individual program is randomly selected for P' with probability $Prob_{xover}$. The programs in P' are then randomly paired and the crossover operator is applied to each pair. The pairs resulting from crossover replace the corresponding original pairs in the population.

4.4.2 Mutation

With probability $Prob_{mutate}$, a program instruction in the population is changed to a random instruction. A random VRM-$\mathcal{M}_{(m,n)}$ instruction (§3) is generated by first selecting an opcode at random and then, when necessary, randomly generating the opcode's operands (*i.e.* registers and relative addresses).

5 Results

We performed two sets of experiments for the multiplication problem using the GP configuration of the previous section (§4)—two-point crossover (Experiment I) and two-point crossover with mutation (Experiment II). We also compared the number of programs examined by GP solution of this problem to the number required for a successful random search. All experiments used Finnegan compiled with SML/NJ version 108.15 and the pseudorandom number generator from the SML/NJ library.

For experiments I and II, we studied VRM-$\mathcal{M}_{(4,8)}$; that is, VRM-$\mathcal{M}$ with four registers and eight instructions. Each experiment consisted of 2000 GP runs with $Prob_{xover} = 0.25$ on a population of 1024 (initially random) programs. Evaluation of an individual was arbitrarily[6] limited to $k = 100$ steps. A run was deemed complete when the fitness of the best individual remained constant for 50 consecutive generations (stasis). Experiment I did not use mutation ($Prob_{mutate} = 0.0$); experiment II had a slight mutation rate ($Prob_{mutate} = 0.001$).

Experiment I evaluated 6.2×10^7 programs overall with a yield of 28 exact solutions to the multiplication problem, or an exact solution approximately every 71 runs (2.2×10^6 evaluations per solution).[7] Experiment II evaluated 6.8×10^7 programs overall and gave 60 exact solutions with an exact solution appearing approximately every 33 runs (1.1×10^6 evaluations per solution).

Some solutions discovered by experiment I are in Figure 3. Solutions (a) and (b) closely resemble a human's coding of this problem (*cf.* Figure 1) with (b) exhibiting unstructured control flow. Solution (c) always executes at least one loop iteration and, in the case of a zero multiplier, then proceeds to undo this incorrect subcomputation. Solution (d) is interesting because it accumulates *two* separate products—in $\mathtt{R}_0$ and $\mathtt{R}_2$. In (d), $\mathtt{R}_2$ is used as multiplier input, loop counter, and product accumulator. Unlike most solutions, (d) requires only

[6] A system might find suitable k values automatically by performing runs using $k = 2^i$ for successive integer values of i until a run yields a solution.

[7] Finnegan eliminates some redundant evaluations; a program that moves unchanged from generation g to generation $g+1$ is not reevaluated. Eliminated evaluations do *not* appear in reported totals.

Address	Instruction
0:	Cmp(R1,R3)
1:	Je(+7)
2:	Add(R0,R2)
3:	Dec(R1)
4:	Jg(-4)
5:	~~Mov(R2,R3)~~
6:	~~Jg(+2)~~
7:	~~Add(R1,R1)~~

(a) Generation 30

Address	Instruction
0:	J(+6)
1:	~~Jl(+7)~~
2:	~~Je(-2)~~
3:	Inc(R3)
4:	~~Nop~~
5:	Add(R0,R1)
6:	Cmp(R2,R3)
7:	Jg(-6)

(b) Generation 15

Address	Instruction
0:	Dec(R1)
1:	Cmp(R3,R1)
2:	Add(R0,R2)
3:	Jl(-3)
4:	Je(+4)
5:	~~Jl(0)~~
6:	Mov(R0,R1)
7:	Inc(R0)

(c) Generation 13

Address	Instruction
0:	Cmp(R0,R2)
1:	Dec(R2)
2:	Je(+5)
3:	~~Je(+3)~~
4:	Cmp(R0,R2)
5:	Add(R0,R1)
6:	Add(R2,R1)
7:	Jl(-6)

(d) Generation 28

Figure 3: Solutions to the multiplication problem discovered by GP using the VRM-$\mathcal{M}_{(m,n)}$ representation and halted after $k = 100$ steps. Instructions that do not affect a program's result have a ~~line~~ through them.

three VRM-$\mathcal{M}$ registers (but seven instructions).

Examination of 10^9 random VRM-$\mathcal{M}_{(4,8)}$ programs found 104 general solutions to the multiplication problem.[8] On average, a solution via random search required examination of $\mu = 9.5 \times 10^6$ individuals per solution with sample variance $\sigma^2 = 7.2 \times 10^{13}$. In this regard, GP's directed search strategy is quite effective. However, examination of a program by random search is usually much faster than examination by GP. This is because a random search can discard the current program as soon as a single test case fails whereas GP must construct fitness values by evaluating programs against all test cases. With respect to the multiplication problem, it is yet unclear how the absolute efficiences of GP and random search relate.

6 Conclusion

We have shown that—using a machine-language representation with addition as its sole arithmetic operator—genetic programming can effectively synthesize unstructured loops from primitive instructions (comparisons and relative branches) to evolve general multiplication. Furthermore, the number of unique programs processed by GP solving this problem is significantly less than the number processed by a random search, on average.

Questions concerning the absolute efficiency of a GP solution relative to random search (as noted in the previous section) are of foremost importance. To this end, we are pursuing a two-pronged approach. First, we are designing optimizations for the Finnegan system that further remove redundant program evaluations and that reduce the costs of such evaluations. This can directly increase GP's efficiency. Second, we are turning our attention toward harder problems (synthesis of multiplication over $\mathbb{Z}$ instead of $\mathbb{N}$, for example), the assumption being that unrestricted-iterative GP may effectively scale to harder problems for which random search is intractable.

To establish unrestricted iteration as a valuable device for genetic programming, future work must successfully apply it to richer representations and solve problems of higher complexity.

[8] Combined with equation 2, these empirical data suggest that multiplication is surprisingly dense in the space of VRM-$\mathcal{M}_{(4,8)}$ programs.

Acknowledgment

Thanks to Chris Fraser for helpful comments on drafts of this paper.

References

[1] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[2] S. Brave. Evolving recursive programs for tree search. In P. Angeline and K. E. Kinnear Jr., editors, *Advances in Genetic Programming II*, chapter 10. MIT Press, Cambridge, MA, 1996. (To appear).

[3] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceeding of the International Conference on Genetic Algorithms and their Applications*, pages 183–187. Texas Instruments, July 1985.

[4] R. M. Friedberg. A learning machine: Part I. *IBM Journal of Research and Development*, 2:2–13, 1958.

[5] R. M. Friedberg, B. Dunham, and J. H. North. A learning machine: Part II. *IBM Journal of Research and Development*, 3:282–287, 1959.

[6] K. E. Kinnear Jr. Evolving a sort: Lessons in genetic programming. In *International Conference on Neural Networks*, New York, NY, 1993. IEEE.

[7] K. E. Kinnear Jr., editor. *Advances in Genetic Programming*. MIT Press, Cambridge, MA, 1994.

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

[9] J. Koza. *Genetic Programming II*. MIT Press, Cambridge, MA, 1994.

[10] D. B. MacQueen. Modules for Standard ML (revised version). *Polymorphism*, II(2), October 1985.

[11] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Verlag, Berlin, 1992.

[12] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.