Appears in Congress on Evolutionary Computation, 2000

Abstract Program Evaluation and its Application to Sorter Evolution

Lorenz Huelsbergen
Lucent Technologies/Bell Labs
Murray Hill, New Jersey, USA
lorenz@research.bell-labs.com

Abstract- This paper introduces *abstract program evaluation (APE)* **that, for certain kinds of evolutionary induction problems, abstractly captures the maximal set of a problem's fitness tests. Abstraction of test cases can substantially reduce the number of such cases—fewer test cases lead to faster fitness computation. APE thereby can make some heretofore untenable induction problems solvable. Furthermore, since the computational representation (program, circuit,** *etc.***) being evolved with APE is abstractly tested on every possible input, evolved solutions which pass all abstract tests are general—APE guarantees correctness for every possible input. APE transforms operators in the representation to compute with the abstract values of the abstract test cases instead of with concrete values (***i.e.***, integers or reals); it is a form of** *symbolic evaluation* **of the executable representation. We discuss induction problems to which APE is suited as well as its limitations.**

APE, in the context of a machine-language representation, has been used to evolve general solutions to two induction problems: finding the maximum of four integers; and, sorting three integers into ascending order. General solutions to both problems are difficult for evolutionary search because large test-case sets seem necessary. Since APE drastically reduced test-set sizes, general solutions—which are correct for all possible input integers—were obtained for both problems. The sorter routines are the largest evolved machine-language programs reported to date and the first sorting programs evolved using only a low-level representation (*i.e.***, one without high-level operators for ordering and exchanging elements). Random search results confirm that evolutionary search is indeed effective on the two problems.**

1 Introduction

Induction of executable representations—programs, circuits, or arbitrary automata—by evolutionary search requires the frequent measurement of a candidate solution on a set of fitness tests to ascertain the candidate's fitness. For target functions that map large domains to some range (*e.g.*, integers to integers), a set of tests $\mathcal{T}$ that identifies a correct and general solution is likely to be very large, if not infinite. Fitness attribution with respect to such a set $\mathcal{T}$ is therefore a computationally expensive process; a new candidate requires on the order of $|\mathcal{T}|$ evaluations to precisely ascertain its fitness.

To make evolutionary search in the spaces of computation structures tractable, systems in practice consider only a subset, $\bar{\mathcal{T}} \subset \mathcal{T}$, (perhaps randomly chosen over time) against which they evaluate new candidates to attain approximate fitness measures. This strategy reduces the computational expense of fitness determination, but now admits "solutions" that may not be correct in general. Alternately, it may not be possible to design an approximate test set $\bar{\mathcal{T}}$ that provides any useful solutions at all.

This paper proposes *abstraction of test cases* from large, perhaps infinite, sets containing concrete tests to small and finite sets of abstract tests. Along with appropriate transforms of the operators of the representation being evolved, this proposal makes some heretofore untenable induction problems tractable. The central idea is to *evaluate programs symbolically* rather than concretely.

Abstract program evaluation (APE) is introduced here as a mechanism for collapsing a concrete test-case set into a smaller, but abstract, set for two purposes:

- reduction of fitness computation times
- guarantee of solution correctness

The first point is achieved because the abstract test set is smaller than the concrete set; the second point is met since the abstract test set contains the entire concrete set (with respect to some program property). Reductions due to APE may be quite large; for example, in the experiments of this paper, APE reduces infinite test sets to abstract sets containing only a few (≤ 10) elements.[1]

In this paper we describe APE in the context of generic machine-language induction (MLI; *e.g.*, [4, 5, 2, 16, 11])—software program induction over integer inputs and outputs where programs are represented as sequences of machine-language instructions operating on integer data in registers. APE is, however, representation independent and equally applicable to GP (*e.g.*, [14]) and GP variants, evolvable hardware (*e.g.*, [18]), and other forms of evolutionary automata induction, *etc*.

As an example of APE's operation, consider searching for an extremely simple program function, `max`, that compares two integer variables, `x` and `y`, and determines their maximum. C code solving this problem might look as follows:

```
if (x > y)
  y = x;
```

[1] We remark here, and further discuss later (§5), that a drastic reduction in test set size can be detrimental to the gradient necessary to learn the target function. This can possibly be mitigated by reintroducing some concrete tests.

Evolving this functionality in a generic representation—one that includes arithmetic, Boolean logic, variable assignment, comparison, and arbitrary control flow—requires test cases that contain concrete pairs of integer values for `x` and `y`. An induction system might require many such pairs to learn that the desired feature is the order relation ($<$) between `x` and `y`. Of course it is not known a priori what sets of pairs suffice for solution and it is not practical to evaluate all possible input pairs.[2] Existing systems therefore use an *ad hoc* set of tests, perhaps randomly constructed. Incomplete test-case sets however allow for the possibility that a "solution" is not general—that is, it may not produce the required result ($\mathtt{y} \leftarrow max(\mathtt{x}, \mathtt{y})$) for all `x` and `y`. This is because the evolved programs may contain essentially extraneous instructions that do not adversely influence the computation of the desired function on the test suite, but may have deleterious effects on other, non-tested, inputs. If, on the other hand, a smaller range of integers (*e.g.*, 8-bit integers) is considered in its entirety, resulting "solutions" are valid only on restricted-size inputs. Solutions found using *ad hoc* test cases therefore require an external proof of correctness, currently performed by humans (if at all). On the other hand, if it were known that a solution had passed all possible tests (perhaps infinite in number), its correctness for any input will have been established through its evolution. Tractable fitness evaluation and general solutions are the goals of APE.

In the example of the `max` function, APE can abstract the values of `x` and `y` as X_{Abs} and Y_{Abs} to yield the two-element abstract test set:

$$\{X_{Abs} > Y_{Abs}, X_{Abs} < Y_{Abs}\}$$

That is, the first test case initializes `x` to an abstract value that denotes any integer greater than `y` and initializes `y` to an abstract value that denotes integers less than `x`. Similarly, the second test case gives `x` an abstract value that denotes all integers less than `y` and gives `y` an abstract denotation of all values greater than `x`. This abstract test set—along with an abstract evaluator derived from the representation being evolved (in our case: arithmetic, Boolean logic, assignment, comparison, and control flow) that can compute with abstract values—completely specifies the target function *for all possible input values*. The abstract evaluator, for example, can compare two registers containing abstract values (of the "$<$" relation) and return a definite result: true or false. Comparing abstract values to concrete values, or performing arithmetic not defined for abstract values, may further propagate the undefined value(s) or it can produce a fault (which could identify the program as unfit, for example). If evolutionary search finds a solution that passes both test cases of this example, then it has found a completely general solution for all integers. Furthermore, such a solution is correct for integers of unbounded size. Note that APE reduces test-case size and simultaneously guarantees solution correctness. In Section 3 we give an abstract interpreter for generic machine-language instructions that handles concrete values (integers) and abstract values (of the "$<$" linear order).

APE abstracts relationships between the concrete data comprising a conventional set of tests. It is therefore applicable to programs that compute only with relationships between the data elements and not necessarily with the values of the elements. The function `max`, for example, does not require the concrete values of its arguments, but only information about their relative position in the linear order. In particular, if the function requires all of the information contained in its arguments to compute its result, abstraction via APE is not apparently useful. However, APE can cope with many types of computations. Functions that compute properties of the shape of data structures—*e.g.*, the depth of a tree, for example—are candidates for APE. Similarly, functions computing properties of numeric values, such as the sign ($+$ or $-$), stand to benefit from APE.

Using APE, we have evolved two programs: `4-max` finds the maximum among four input integers; `3-sort` sorts three input integers into ascending order. The latter is the first sorting program evolved in a representation that does not include high-level "primitives" for ordering elements (*cf.* [13, 17]). The evolved routines were synthesized solely from generic machine language instructions—no domain specific operators that could artificially simplify evolution's task were introduced. The sort routine is the largest machine-language program discovered automatically, by evolutionary search or other machine methods, to date. Since APE guarantees that solutions are correct for any integer inputs, a human proof of its correctness is unnecessary. Random search conducted for both functions confirms that the directed evolutionary approach is indeed effective for `4-max` and `3-sort`.

The next section describes *abstract interpretation* [1]—the program analysis technique on which APE is based. It also covers prior work related to program induction and MLI in particular, as well as work on sorter evolution. Section 3 describes the APE virtual register machine and contains the details of our particular APE implementation. Sections 4 and 5 contain the experimental setup and results. We summarize with a discussion.

2 Background

APE is based on ideas of *abstract interpretation* [1, 3] from programming language theory. Abstract interpretation is used in program analyses to perform compile-time optimization. To convey the idea of abstract interpretation (and of APE), we first give an example of abstractly interpreting a conventional program with respect to an abstract domain. Then, we catalog some program and circuit induction approaches that may stand to gain from APE; we also supply details of prior work on machine-language induction, the form of program induction employed in this paper's APE examples. An overview of

[2] On contemporary processors, integers typically reside in at least 32 bits; the domain for `max` is therefore of size $(2^{32})^2$, an infeasible number for exhaustive evaluation.

prior work on sorter evolution/induction concludes this section.

2.1 Abstract Interpretation

The canonical example of abstract interpretation applied to a conventional program—one written by humans—is that of the "rule of signs." The description presented here is a distillation of the introduction provided in Field and Harrison's book ([3], p. 526). Consider a program with expressions that perform standard arithmetic (addition, multiplication) on integers. We wish to analyze statically the program to deduce the sign ("$+$" or "$-$") that each variable or expression may take.[3] We transform the domain of integers Z into the abstract domain $Z^{\#} = \{+, -\}$. An integer variable or expression in the original program, under abstract interpretation, assumes the value "$+$" if it is known to be positive throughout its lifetime and the value "$-$" if it is always less than 0. Additionally, we use the symbol $\perp$ (bottom) to denote an unassigned value (that may during interpretation become "$+$" or "$-$" and the symbol $\top$ (top) to denote an expression or variable that can assume *both* "$+$" *and* "$-$".

Now, given the "rule of signs" that states that multiplication of two positive integers gives a positive result (as does multiplication of two negative integers) and that multiplication of opposite signs gives a negative integer, we can abstractly interpret the sign of the assignment $z = x \times y$ by knowing only the signs of x and y. If x and y are both positive (or are both negative), the result z is also positive; otherwise, the sign must be negative. Abstract interpretation of the program over $Z^{\#}$ infers z's sign by evaluating the right-hand side using the "rule of signs." All variables are initialized to $\perp$ before interpretation begins; inputs are initialized with their abstract sign.

If we introduce a sign rule for addition, we must handle the situations where the sign becomes indeterminate (either "$+$" or "$-$"). In such situations, the abstract value becomes $\top$ which represents either alternative. For example, in the assignment $u = x + y$, the sign of u is positive if x and y are both positive, negative if x and y are both negative, and $\top$ otherwise. The undefined value $\perp$ (or the overdefined value $\top$), when supplied to an arithmetic operator, typically forces the abstract result to $\top$.

Note how an implementation of the "rule of signs" could be used to evolve abstractly functions sensitive to the signs of their integer arguments without regard to the integers' actual values. In this manner, an infinite domain (Z) is approximated by a smaller finite one ($Z^{\#}$). In this paper, APE manipulates domains—somewhat more complex than ($Z^{\#}$)—that linearly relate the magnitudes of integers and thereby admit comparison (but not arithmetic in general).

Whereas abstract interpretation is a compile-time analysis technique possessing strong termination properties[4], APE operates dynamically during evaluation and, in the case of a non-terminating concrete program, need not ensure termination of the abstract program. As we shall see, APE simultaneously computes with both abstract and concrete values within the same program.

2.2 Program/Circuit Induction

Machine-language Induction (MLI) (*e.g.*, [11, 16, 2, 4, 5]) is a form of genetic algorithm (GA) [6, 9] that—analogous to genetic programming (GP) [14] of Lisp expressions—searches the space of machine-language programs. GAs use principles from evolutionary theory (populations, fitness criteria, recombination) to search large non-linear spaces. GAs typically use *crossover* [6] as the "genetic operator." Instead, we notably use *macro-mutation*—the "headless chicken" operator of Jones [12]—since we found (*e.g.*, [11]) that it consistently works better than crossover.

The early MLI experiments of Friedberg *et al.* [4, 5] were not successful when compared to random search. Friedberg *et al.* however anticipated and influenced the MLI ideas of today. Cramer [2] describes a more recent experiment that uses a representation similar to a machine language, but endowed with high-level iteration operators. Nordin's contemporary system [16] manipulates native machine-code (as opposed to the interpreted virtual machines used here) to speed fitness-test evaluation. Neither Cramer [2] nor Nordin [16] provide experimental validation that their search experiments were truly effective—by comparing to random search[5], for example. Though the prior MLI approaches [4, 5, 2, 16], can conceptually admit arbitrary control flow (forward and backward) and hence unstructured loops, *etc.*, these systems do not provide mechanisms for this; hence, the experiments are limited either to forward control flow [16, 4, 5] or to iteration via high-level loops [2]. Our approach has been to experimentally demonstrate mechanisms that can evolve general functions—free to utilize any control flow evolved from conditional branches—from generic machine-language primitives. Random search validates our results.

Another promising venue for APE is *evolvable hardware* (see [18] for examples) for here too the number of test cases can be enormous. As with the APE examples in this paper, the hardware operators (gates) must be transformed to the abstract operators (collections of gates) that manipulate abstract values.

2.3 Sorter Evolution

In his paper on co-evolution [8], Hillis tackles the problem of evolving a *sorting network*. A sorting network provides a set of parallel lines, each carrying an element to be sorted. Comparisons and exchanges between elements are affected

[3] For simplicity, we take the sign of zero to be "$+$".

[4] Typically in abstract interpretation, domains are finite and functions on abstract values are monotonically increasing [3, 1].

[5] Our experience [10] indicates that for small problems it is often possible for random search to find solutions in time comparable to evolutionary methods.

by connecting a pair of lines. Minimizing comparisons is the goal of Hillis' paper. In sharp contrast to ours, Hillis' system is specialized for sorting. On the other hand, our sorters—though evolved from a generic and general machine-language representation—currently handle only small inputs (three elements) in contrast to Hillis' networks which process 16 elements. Koza *et al.* [15] implemented a system similar to Hillis' in reconfigurable hardware constructed from an FPGA and evolved optimal 7-element sorting networks.

O'Reilly and Oppacher [17] first reported results of evolving a general sort using a GP framework that included high-level looping and a `swap` operator. The experiments did not result in perfect sorters. Their notion of generality is slightly broader than that of this paper; not only did they stipulate that any integer sequence of fixed length be sorted (as we do in this paper with length-3 sequences), but that the input sequence be of variable length as well. Kinnear [13] extended O'Reilly and Oppacher's work [17] with an additional operator tailored for sorting (`order`) that exchanges two elements into ascending order. Although GP was able to find general sorting solutions when given the `order` "primitive" for sorting two elements, the resulting solutions were proved general only by human inspection; our sorters are automatically proved general by virtue of APE. Unfortunately, the GP experiments on sorting [17, 13] lack the validation of random search.

3 Virtual Register Machine $\mathcal{A}$

Here we define an APE virtual register machine (VRM) that operates on abstract domains suitable for solving the `4-max` and `3-sort` problems. This VRM is called VRM-$\mathcal{A}$ ("Abstract") and is very similar to the VRM-$\mathcal{S}$ that we defined previously [10]. Specifically, VRM-$\mathcal{A}$'s instructions are those of VRM-$\mathcal{S}$ with the omission of the latter's I/O instruction and the inclusion of the `Mod` arithmetic instruction for completeness. We refer the reader to the definition of VRM-$\mathcal{S}$ [10] for details of VRM evaluation (branching, termination, exceptional arithmetic conditions, *etc.*) as well as for complete descriptions of the instructions.

The notation VRM-$\mathcal{A}_{(n,m)}$ names a particular VRM-$\mathcal{A}$ that consists of external state (m registers containing either an integer or an abstract value), internal state (a program counter and comparison flag) and a sequence of n immutable abstract instructions.

3.1 External State: Registers

Let the *register state* be a vector

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

of m values; a value may either be an integer or an abstract value denoting some property of the input test data. We define the function $Abs(R_i) \rightarrow Bool$ so that the VRM interpreter can determine if the content of register R_i is abstract.

Program inputs (abstract or concrete) are placed in the initial state $\vec{R}$. Outputs are taken from the final register state $\vec{R}'$. Non-input registers are initialized concretely to zero.

3.2 Internal State: PC, Flag

In addition to the external register state, VRM-$\mathcal{A}$ maintains two pieces of internal state: a program counter (*PC*) and a comparison flag (*Flag*). The program counter is an integer, $0 \leq PC < n$, 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 *PC* is initially set to zero.

The *Flag* reflects the result of the last comparison instruction executed. It can assume the values *less*, *greater*, and *equal*. *Flag* is initially undefined. Only the comparison instruction (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 past the end of the program. (Our evaluation strategy also limits the maximum number of instructions evaluated; see Section 3.5.) Figure 1 contains the essential VRM-$\mathcal{A}$'s instructions and their operational semantics. Additionally, VRM-$\mathcal{A}$ also includes instructions to clear a register to zero (`Clr`, similar to `Set`), to decrement a register (`Dec`, similar to `Inc`) and the arithmetic functions `Sub`, `Mul`, `Div`, and `Mod`. (See [10] for explicit definitions.) In total, VRM-$\mathcal{A}$ contains 17 different instructions and its instruction set forms the core of the instruction sets of contemporary processors. Note that VRM-$\mathcal{A}$ does *not* contain domain-specific instructions (such as `swap` [17] or `order` [13]) that could abet sorter evolution, for example.

New to VRM-$\mathcal{A}$ is the abstract operation of the instructions under APE. The `Mov` instruction simply transfers the content of its source to the destination regardless of whether source (or destination) are concrete or abstract. Similarly, `Set` and `Clr` overwrite the register contents with concrete integer constants. The functions `Inc`, `Dec`, and `Neg`, `Add`, *etc.*, first check if the register is abstract or concrete. If it is abstract, the destination register is made undefined ($\perp$); otherwise, the operation on the concrete integer(s) is performed and the destination register updated.

The `Cmp` instruction is central to APE for evolving the functions of interest, `4-max` and `3-sort`. It can examine the order implied by two abstract values. Note that if both of `Cmp`'s registers are concrete, an integer comparison is performed. If both are abstract, they are compared *with respect to their position in the abstract linear order*. All elements of the linear order are contained in every test of the test-case set and the test set consists of all permutations of this order (§4.1). If one register is abstract and the other concrete, the comparison is undefined and $\perp$ is placed in the *Flag*. The re-

$$
\begin{aligned}
\texttt{Mov}(R_{dst}, R_{src}) &\equiv \left(\begin{array}{l} PC \longleftarrow PC + 1 \\ R_{dst} \longleftarrow R_{src} \end{array} \right. \\
\texttt{Set}(R_a) &\equiv \left(\begin{array}{l} PC \longleftarrow PC + 1 \\ R_a \longleftarrow 1 \end{array} \right. \\
\texttt{Inc}(R_a) &\equiv \left(\begin{array}{l} PC \longleftarrow PC + 1 \\ R_a \longleftarrow \begin{cases} \bot & \text{if } Abs(R_a) \\ R_a + 1 & \text{otherwise} \end{cases} \end{array} \right. \\
\texttt{Neg}(R_a) &\equiv \left(\begin{array}{l} PC \longleftarrow PC + 1 \\ R_a \longleftarrow \begin{cases} \bot & \text{if } Abs(R_a) \\ 0 - R_a & \text{otherwise} \end{cases} \end{array} \right. \\
\texttt{Add}(R_{dst}, R_{src}) &\equiv \left(\begin{array}{l} PC \longleftarrow PC + 1 \\ R_{dst} \longleftarrow \begin{cases} \bot & \text{if } Abs(R_{dst}) \vee Abs(R_{src}) \\ R_{dst} + R_{src} & \text{otherwise} \end{cases} \end{array} \right. \\
\texttt{NOP} &\equiv \left(\; PC \longleftarrow PC + 1 \right. \\
\texttt{Cmp}(R_a, R_b) &\equiv \left(\begin{array}{l} PC \longleftarrow PC + 1 \\ \mathit{Flag} \longleftarrow \begin{cases} \bot & \text{if } (Abs(R_a) \wedge \neg Abs(R_b)) \vee (\neg Abs(R_a) \wedge Abs(R_b)) \\ \mathit{less} & \text{if } R_a < R_b \\ \mathit{greater} & \text{if } R_a > R_b \\ \mathit{equal} & \text{otherwise} \end{cases} \end{array} \right. \\
\texttt{J}(\mathit{offset}) &\equiv \left(\; PC \longleftarrow \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) \right. \\
\texttt{Jl}(\mathit{offset}) &\equiv \left(PC \longleftarrow \begin{cases} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } \mathit{Flag} = \mathit{less} \\ PC + 1 & \text{otherwise} \end{cases} \right. \\
\texttt{Jg}(\mathit{offset}) &\equiv \left(PC \longleftarrow \begin{cases} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } \mathit{Flag} = \mathit{greater} \\ PC + 1 & \text{otherwise} \end{cases} \right. \\
\texttt{Je}(\mathit{offset}) &\equiv \left(PC \longleftarrow \begin{cases} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } \mathit{Flag} = \mathit{equal} \\ PC + 1 & \text{otherwise} \end{cases} \right.
\end{aligned}
$$

Figure 1: Partial operational semantics for the virtual register machine VRM-$\mathcal{A}$. Not shown are `Clr` (similar to `Set`), `Dec` (similar to `Inc`), and arithmetic instructions `Sub`, `Mul`, `Div`, `Mod` (similar to `Add`).

sulting *Flag* state governs the conditional branch instructions `Jl`, `Jg`, and `Je`. In this manner, dynamic control flow may be effected by the results of comparisons of abstract values.

3.4 Abstract Domain $Z^<$

The abstract values of VRM-$\mathcal{A}$ specify a finite linear order of size n on the relation "$<$" among integers. Here, n corresponds to the number of input integers to a program. More formally, let

$$Z^<(n) = \left\{X_1^<, \ldots, X_n^<\right\}$$

denote an abstract domain of n abstract values, $X_1^<, \ldots, X_n^<$, where $X_i^< < X_j^<$ if $i < j$.

In VRM-$\mathcal{A}$, the `Cmp` instruction, operating on two abstract values $X_i^<$ and $X_j^<$, will compare their indices to ascertain whether or not the "$<$" relation holds. As described in Section 4.1, a test set contains all permutations of $Z^<(n)$.

3.5 Evaluation Function

An interpreter evaluates an n-instruction VRM-$\mathcal{A}_{(n,m)}$ program $\vec{I}$ with respect to an m-register input state $\vec{R}$ and an integer number of evaluation steps (instructions), $K > 0$. $\mathcal{E}_\mathcal{A}$ maps a triple to a singleton:

$$\mathcal{E}_\mathcal{A} : \left(\vec{I}, \vec{R}, K\right) \rightarrow \vec{R}' \tag{1}$$

$\mathcal{E}_\mathcal{A}$ produces the final register state $\vec{R}'$ after executing at most K instructions.[6]

We overload the notation: $\mathcal{E}_\mathcal{A}(\vec{I}, t) \rightarrow \vec{v}$ means that the evaluation of test case t on program $\vec{I}$ produces answer vector $\vec{v}$; the initial and final register state, the layout of $\vec{v}$, and K are implied from the context; $\mathcal{E}_\mathcal{A}(\vec{I}, t)[i]$ denotes the i-th answer element.

4 Experimental Setup

This section describes the search methods—population-based macro-mutation (GA-MM), and random search—used in finding solutions to `4-max` and `3-sort`. Before describing the individual methods, we first define their test-case sets and fitness functions. The quantitative settings (*e.g.*, test-case and program size) are given in the next section (§5).

4.1 Test Cases

The abstract test-case set for a program of n input integers is

$$T^<(n) = Permute(Z^<(n))$$

where $Permute(S)$ denotes the set of all permutations of the finite set S. Note that $Permute(S)$ contains $n!$ elements. For `3-sort` and `4-max` which operate on three and four input integers respectively, the test case sets are $T^<(3)$ and $T^<(4)$.

[6] Since our VRM-$\mathcal{A}$ evaluator is an interpreter (essentially Figure 1), it can easily be halted after evaluation of K instructions.

For example, `3-sort` is tested on the six sequences:

$$\begin{aligned} T^<(3) \quad = \quad & \{\left\langle X_1^<, X_2^<, X_3^<\right\rangle, \left\langle X_1^<, X_3^<, X_2^<\right\rangle, \\ & \left\langle X_2^<, X_1^<, X_3^<\right\rangle, \left\langle X_2^<, X_3^<, X_1^<\right\rangle, \\ & \left\langle X_3^<, X_1^<, X_2^<\right\rangle, \left\langle X_3^<, X_2^<, X_1^<\right\rangle\} \end{aligned}$$

Similarly, the tests for `4-max` contain the 24 permutations of $Z^<(4)$.

Although APE reduces the number of test cases via abstraction, the size of the test case set remains exponential in n. This means that APE evolution of programs with large inputs remains intractable.

4.2 Fitness Functions

For the fitness functions for `3-sort` and `4-max`, lower values indicate better fitness and a fitness of zero indicates a perfect program.

The fitness function for evolving `3-sort` is

$$\mathcal{F}_{\text{sort}}(\vec{I}) = n! - \sum_{t \in T^<(n)} \sum_{i=1}^{n} Equal(\mathcal{E}_\mathcal{A}(\vec{I}, t)[i],\ X_i^<)$$

where $n = 3$ and $Equal(x, y)$ gives 1 if x equals y and 0 otherwise. The `3-sort` fitness is a measure of how close the program is to sorting the test sequence into ascending order; every incorrect value in the result registers imposes a unit penalty. Note that concrete values in the output are considered incorrect; the entire abstract sequence must occur in linear order in the output registers for the sort to be correct for all possible inputs.

The fitness function for evolving `4-max` is

$$\mathcal{F}_{\max}(\vec{I}) = \sum_{t \in T^<(n)} max(t) - \mathcal{E}_\mathcal{A}(\vec{I}, t)$$

where $n = 4$, max gives an abstract sequence's maximum, and $\mathcal{E}_\mathcal{A}(\vec{I}, t)$ is an element of $Z^<(n)$. Again, concrete integer answers are incorrect and taken to be $-\infty$ (*i.e.*, $-maxint$ in VRM-$\mathcal{A}$). Here, the fitness is a measure of the distance of the program's abstract answer to the correct abstract answer.

4.3 Search Methods

Two search methods are compared: population-based macro-mutation (GA-MM) and random search.[7] Point-wise mutation was not used.

Population Selection Population selection, for the construction of successive generations, is performed via k-tournament selection (see, *e.g.*, [7]). Let P be the population (set) of VRM-$\mathcal{A}$ programs. To select a single individual from P, tournament selection examines k individuals in P and selects the one with best fitness. When the fitness function cannot distinguish a single best, one of the best is chosen at random.

[7] Preliminary experimentation indicated that conventional GA crossover finds solutions to `3-sort` and `4-max`; however, we found GA-MM to do so somewhat faster. Comparison of macro-mutation to crossover for MLI can be found elsewhere [11].

4-max

	#Solns	#Evals	$\frac{\#\text{Solns}}{\#\text{Evals}}$
GA-MM	4	6.2×10^{9}	6.5×10^{-10}
Random	0	1×10^{10}	0

Parameters: program size, $N = 16$; registers, $M = 12$; instruction limit, $K = 10N$

3-sort

	#Solns	#Evals	$\frac{\#\text{Solns}}{\#\text{Evals}}$
GA-MM	3	1.37×10^{10}	2.2×10^{-10}
Random	0	1.5×10^{10}	0

Parameters: program size, $N = 64$; registers, $M = 12$; instruction limit, $K = 8N$

Table 1: Search results for the `4-max` and `3-sort` problems. Common parameters are: population size, $|P| = 4096$; $Prob_{MM} = 0.25$. The ratio #Solns/#Evals is a measure of search efficiency.

Operator: Two-Point Macro-mutation Population-based macro-mutation is a search method that, instead of recombining fit individuals, randomly mutates a subsequence of consecutive instructions in a fit individual. Our GA-MM uses the "headless chicken" crossover operator proposed by Jones [12]. No information "crosses over" between individuals under this operator.

To effect macro-mutation, GA-MM first selects a subsequence of instructions from a program $\vec{I}$ starting at a random point $0 \leq p_i < n$ in program $\vec{I}$. The length $k > 0$ of the subsequence is chosen randomly such that $p_i < p_i + k \leq n$. The instructions p_i to $p_i + k - 1$ are replaced with random instructions.

4.3.1 Random Search

The role of random search is to indicate whether a heuristic search (*e.g.*, GA-MM) is effective (better than guessing) for a particular problem.

Random search randomly generates an individual p, evaluates p and computes its fitness, and (optionally) records p's fitness as the best if p improves on the current best fitness. This process continues until a sufficient number of global solutions are found or until the number of program evaluations exceeds a predetermined threshold.

5 Results

The APE VRM-$\mathcal{A}$ was implemented in our custom GA framework (*cf.* [11]). Table 1 contains the results for finding `4-max` and `3-sort` solutions using macro-mutation (GA-MM) and random search. The table also contains the parameter settings of the experiments. GA-MM searches were halted when the best fitness remained constant for 50 generations (stasis). In addition to the number of general solutions found, the table lists the total number of fitness evaluations performed and the search efficiency.

Initially, we attempted to evolve `2-max` (maximum of two input integers) using the APE test sets $T^{<}(2)$. However, GA-MM did not improve on random search and was therefore ineffective. We suspect that the small APE test set of only $2!$ elements does not provide enough of a gradient for the GA to learn the desired function. Therefore, we turned to `4-max` with its larger test suite $T^{<}(4)$ of 24 elements. (Another way of increasing the gradient is to increase the number of test cases by adding concrete tests to the abstract set; we have not verified whether or not this approach can be used to solve `2-max`.) On the `4-max` problem, random search found no solutions whereas GA-MM found four; therefore, evolutionary search is effective for this problem. A sample `4-max` solution is in Figure 2. The four abstract input values are supplied in $R_2, \ldots, R_5$, and the problem size in R_1,[8] The program result is computed into R_0.

Solutions to `3-sort` were harder to come by. Twice the computational effort over `4-max` produced 3 general solutions. No perfect solutions were found during the initial searches with stasis at 50 generations. For each good approximate solution (fitness ≤ 4) found during the initial searches, we repeated GA-MM search with a stasis setting of 10000 generations. Out of the initial searches' five good solutions, three yielded general `3-sort` algorithms after the extended search.[9] An evolved `3-sort` is shown in Figure 3. The three input integers are placed in registers $R_2, \ldots, R_4$ and the problem size (concrete integer 4) in R_1. The result is taken from registers $R_5, \ldots, R_7$.

Note that since APE fitness computation complexity scales with $O(n!)$, searching for n`-max` and n`-sort`, for large n, is intractable (*cf.* §4.1).

6 Summary

The main contribution of this paper is the idea of abstractly representing test cases—and abstractly evaluating representations with respect to the abstract tests—for induction problems. This abstract approach has two significant benefits: it can drastically reduce the number of test cases, thereby lowering fitness evaluation costs; and, it can guarantee the correctness of discovered solutions. For evolutionary induction of programs, we presented an implementation of this idea. Abstract program evaluation enabled evolution of correct and general solutions to two problems: finding the maximum of four integers and sorting three integers. We believe that ab-

[8] The problem-size input is superfluous in the `4-max` and `3-sort` experiments; it was inherited from prior experiments.

[9] The three solutions were found in searches requiring 479, 2231, and 7521 generations.

stract treatment of executable representations and test cases can enable solution of yet harder problems via automatic induction in many domains.

Acknowledgments

Thanks to the anonymous reviewers for helpful comments.

Bibliography

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. Association for Computing Machinery, 1977.

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

[3] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

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

[7] D. E. Goldberg and K. Deb. A comparitive analysis of selection schemes used in genetic algorithms. In G. Rawlins, editor, *Foundations of Genetic Algorithms*. Morgan Kauffman, 1991.

[8] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In *Aritifical Life II*, volume X, pages 313–324. Addison-Wesley, 1992.

[9] J. H. Holland. *Adapation in Natural and Artifical Systems*. University of Michigan Press, 1975.

[10] L. Huelsbergen. Learning recursive sequences via evolution of machine-language programs. In *Proceedings of the Second Conference on Genetic Programming*, pages 186–194, July 1997.

[11] L. Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In *Proceedings of the Third Conference on Genetic Programming*, pages 158–166, July 1998.

[12] T. Jones. Crossover, macromutation, and population-based search. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 73–80, 1995.

[13] K. E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294. Morgan Kaufmann, 1993.

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

[15] J. R. Koza, F. H. Bennett III, J. L. Hutchings, S. L. Bade, M. A. Keane, and D. Andre. Evolving sorting networks using genetic programming and rapidly reconfigurable field-programmable gate. In T. Higuchi, editor, *Workshop on Evolvable Systems. International Joint Conference on Artificial Intelligence*, pages 27–32, 1997.

[16] P. Nordin. A compiling genetic programming system that directly manipulates the machine-code. In K. Kinnear Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[17] U.-M. O'Reilly and Franz Oppacher. An experimental perspective on genetic programming. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 331–340. Elsevier, September 1992.

[18] M. Sipper and D. Mange, editors. *Transactions on Evolutionary Computation; Special Issue: From Biology to Hardware and Back*. IEEE, September 1999.

Address	Instruction	Address	Instruction
0:	Mov(R_0,R_3)	8:	Jg(1)
1:	Cmp(R_2,R_0)	9:	Mov(R_2,R_5)
2:	Jl(+4)	10:	Mov(R_5,R_4)
3:	Neg(R_{11})	11:	Jl(-10)
4:	Cmp(R_3,R_4)	12:	Jg(-11)
5:	Mov(R_0,R_2)	13:	Cmp(R_{10},R_6)
6:	Clr(R_{10})	14:	Sub(R_2,R_2)
7:	Mul(R_1,R_7)	15:	Div(R_5,R_3)

Figure 2: Sample evolved `4-max` program.

Address	Instruction	Address	Instruction
0:	Mov(R_6,R_4)	32:	Cmp(R_9,R_7)
1:	Neg(R_9)	33:	Sub(R_3,R_8)
2:	Mov(R_4,R_3)	34:	Mul(R_5,R_8)
3:	Sub(R_9,R_{11})	35:	Clr(R_5)
4:	NOP	36:	Mul(R_5,R_8)
5:	Mov(R_5,R_4)	37:	Clr(R_5)
6:	Mov(R_7,R_2)	38:	NOP
7:	Set(R_0)	39:	Cmp(R_2,R_7)
8:	Mov(R_4,R_2)	40:	Jl(-43)
9:	Add(R_8,R_0)	41:	Mov(R_2,R_3)
10:	Mov(R_2,R_6)	42:	Dec(R_5)
11:	Cmp(R_3,R_7)	43:	Dec(R_0)
12:	Jg(+10)	44:	Add(R_9,R_4)
13:	Mov(R_6,R_4)	45:	Mov(R_3,R_7)
14:	Neg(R_9)	46:	Jg(-28)
15:	Mov(R_4,R_3)	47:	Je(-53)
16:	Sub(R_9,R_{11})	48:	Dec(R_5)
17:	NOP	49:	Set(R_3)
18:	Mov(R_5,R_4)	50:	Mod(R_8,R_5)
19:	Mov(R_7,R_2)	51:	Add(R_4,R_8)
20:	Cmp(R_{10},R_{11})	52:	Add(R_6,R_{11})
21:	Mov(R_4,R_2)	53:	Div(R_9,R_{10})
22:	Add(R_8,R_0)	54:	Set(R_7)
23:	Mov(R_2,R_6)	55:	Set(R_{11})
24:	Cmp(R_3,R_7)	56:	Neg(R_4)
25:	Jg(+10)	57:	Cmp(R_1,R_2)
26:	Cmp(R_7,R_2)	58:	Inc(R_3)
27:	Jg(-1)	59:	Cmp(R_1,R_3)
28:	Neg(R_8)	60:	Dec(R_5)
29:	Add(R_{11},R_4)	61:	Mov(R_7,R_6)
30:	J(-20)	62:	Mov(R_3,R_7)
31:	Add(R_6,R_9)	63:	Jg(-28)

Figure 3: Sample evolved `3-sort` program.