ReMath Milestone 3 Report: REFUSR

Olivia Lucca Fraser, Anthony Di Franco, Steve 'Dillo' Okay

August 10, 2021

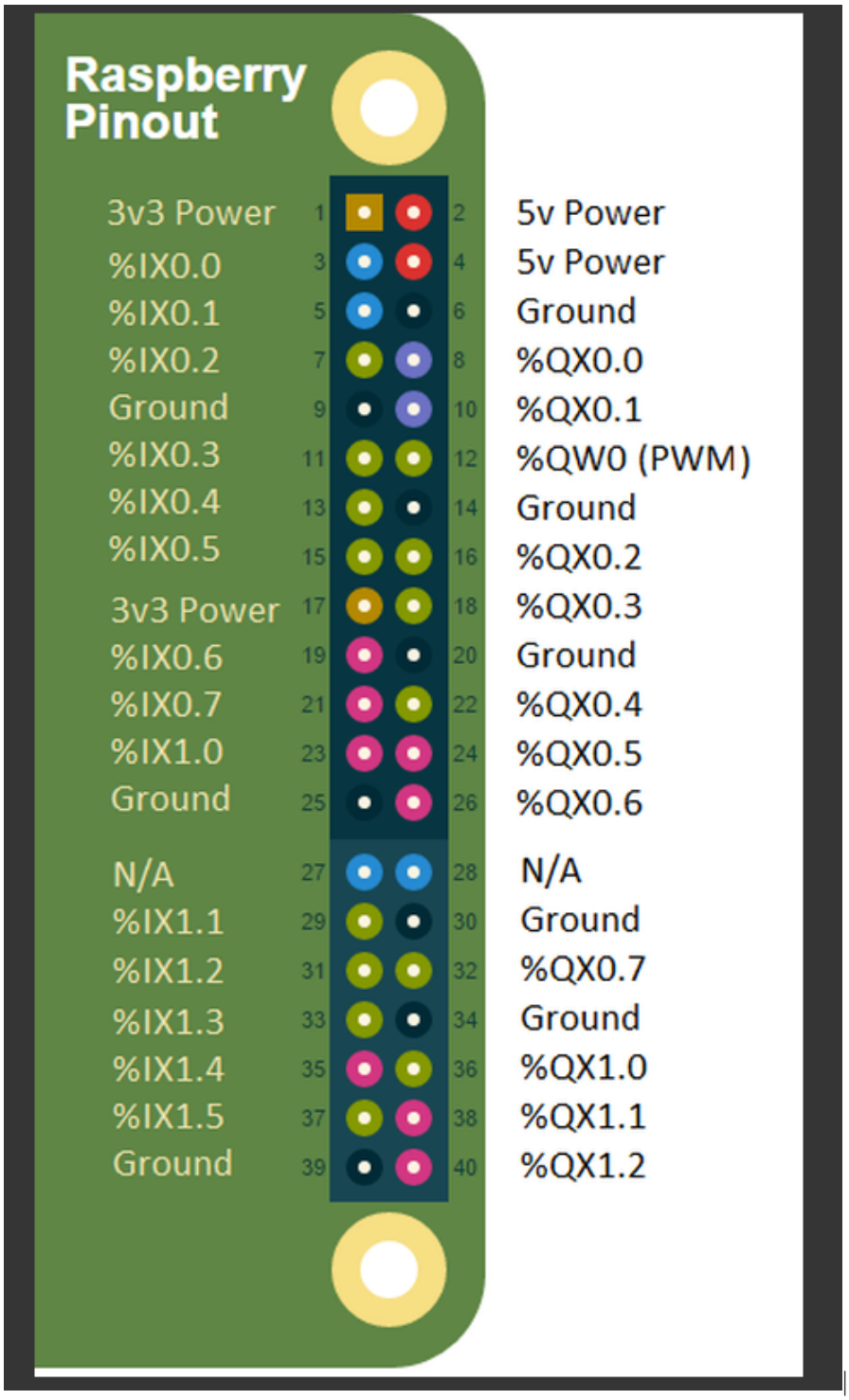
"Refuduino": A PLC Busybox

The "Refuduino" provides a physical platform to demonstrate the effects of the REFUSR System. It consists of a Raspberry Pi 4("RPi") single-board computer(SBC) running OpenPLC, an IEC 61131-3 Compliant Open Source Industrial Controller and an Arduino Nano microcontroller wired together at their respective I/O pins. This provides us with a physical platform to demonstrate that code effects generated by the GA-generated programs can be observed & recovered from a physical system and that these effects can be something as simple as a logical truth table. In other words, bits can be real switches, gates can open/close literal flood gates, etc.

Interconnects between RPi and Arduino

```
PROGRAM Boiler
  VAR
    Data  : ARRAY[1..10] OF BOOL;
    Ready : BOOL;
    CollectInput : F_CollectInput;
  END_VAR
  VAR
    TICK     AT %IX1.0 : BOOL;
    IN1      AT %IX0.3 : BOOL;
    IN2      AT %IX0.4 : BOOL;
    IN3      AT %IX0.5 : BOOL;
    IN4      AT %IX0.6 : BOOL;
    IN5      AT %IX0.7 : BOOL;
    OutReady AT %QX0.0 : BOOL := FALSE;
    FeedNext AT %QX0.1 : BOOL := FALSE;
    Out      AT %QX0.2 : BOOL;
  END_VAR
  CollectInput(TICK:=TICK, IN1:=IN1, IN2:=IN2, IN3:=IN3, IN4:=IN4, IN5:=IN5);
  Ready := CollectInput.Finished;
  FeedNext := 1;
  IF Ready THEN
```

Figure 1: ST code

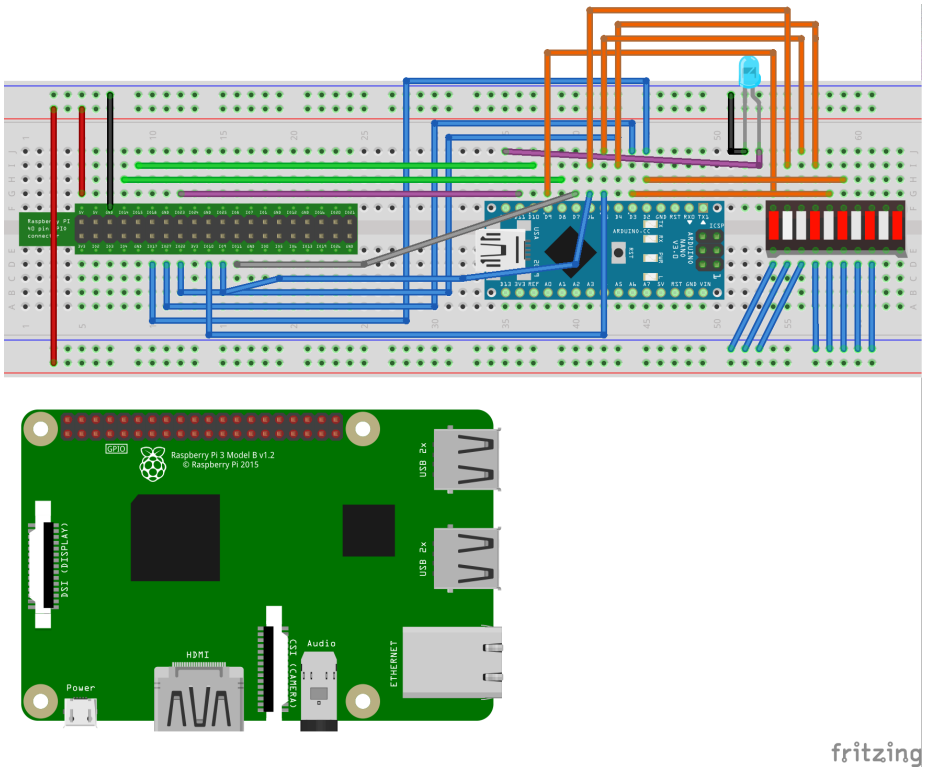

Raspberry Pinout
3v3 Power 1
2 5v Power
%IX0.0 3
4 5v Power
%IX0.1 5
6 Ground
%IX0.2 7
8 %QX0.0
Ground 9
10 %QX0.1
%IX0.3 11
12 %QW0 (PWM)
%IX0.4 13
14 Ground
%IX0.5 15
16 %QX0.2
3v3 Power 17
18 %QX0.3
%IX0.6 19
20 Ground
%IX0.7 21
22 %QX0.4
%IX1.0 23
24 %QX0.5
Ground 25
26 %QX0.6
N/A 27
28 N/A
%IX1.1 29
30 Ground
%IX1.2 31
32 %QX0.7
%IX1.3 33
34 Ground
%IX1.4 35
36 %QX1.0
%IX1.5 37
38 %QX1.1
Ground 39
40 %QX1.2

```
void loop() {

  tick = !tick;
  digitalWrite(PLC_CLK,tick);
  digitalWrite(CLK_LED,tick);
 // digitalWrite(13,tick);

  //Serial.println(tick);

  if (digitalRead(PLC_NEXT)== HIGH) {
    Serial.println("PLC Ready for bits!");
    Serial.print("Sending:");
        for (int c=0; c < 4;c++) {
           for (int i=0; i < 5;i++) {
                int rand_int=random(32);
                int shifted=rand_int >> i;
              //   Serial.print(shifted);
              //   Serial.print(" ");
                data[i]=shifted & 1;
                Serial.print(data[i]);
                Serial.print(" ");
                digitalWrite(ixo_pin[i],data[i]);
           }
        }
        Serial.println();
  }
  delay(1000);

  if (digitalRead(PLC_READY) == HIGH) {
    digitalWrite(12,HIGH);
    result_val=digitalRead(PLC_RESULT);
    Serial.print("Recovered value from PLC:");
    Serial.println(result_val);
  } else {
    digitalWrite(12,LOW);
  }
}
```

Figure 2: Arduino code

Note that the VAR declarations in the partial ST code listing on the left correspond with the labels of the GPIO diagram in the center. Input pins on the Pi correspond to output pin declarations on the Arduino code listing on the right. The Fritzing diagram below details the actual wiring between the two devices.

Fritzing Diagram of "Refuduino"



Demo Video

The link below leads to a video demonstrating the current state of the REFUSR hardware system. The LED activity shows bits being written to the %IX0.n registers, the generate symbolic expression being computed over these bits with respective LEDs indicating the completion of the evaluation and the result written to %QX0.2. An analogue would be the status-and-register-state front-panel from a 1960s-era minicomputer.

Video demonstrating the REFUSR "Refuduino" Hardware.

Foundations of the Property Testing Library: Junta Testing

Adaptive Junta Testing

Based on a preliminary literature review, we identified as state-of-the-art the adaptive junta tester described in "Distribution-Free Junta Testing," Zhengyang Liu, Xi Chen, Rocco A. Servedio, Ying Sheng, and Jinyu Xie. STOC'18, June 25–29, 2018, Los Angeles, CA, USA. The algorithm

"Adaptive" refers to not assuming but rather being adaptable to an input distribution of interest. This permits a better theoretical performance, with a polynomial rather than exponential query complexity, and also makes it possible to directly use empirical execution data to learn the probability distribution that the junta tester samples from, so that its results can be most a propos of the part of the input space that is most relevant in practice.

Novel Automatic Order Search

The algorithm as developed and described in the source paper could only test the junta property of a given order. That is, the junta predicate checking algorithm took the number of input bits k as a parameter and could only reply with an acceptance or probabilistic rejection of a junta of that order.

By identifying the state that should be shared across iterations as the partial list of input bits with a proven influence on the output at some point in the input space, we were able to modify the algorith to share this state across searches of increasing junta order and thus to automatically identify the junta order and also to explicitly indicate the set of inputs in the junta.

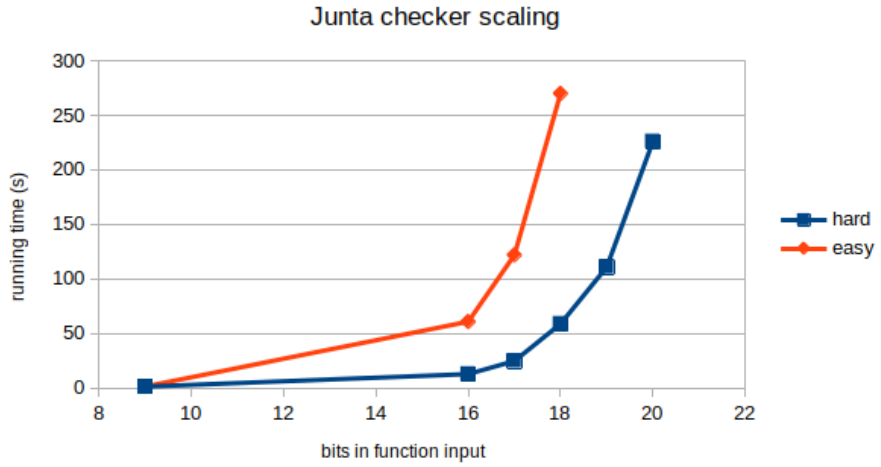
Preliminary Benchmarks

We evaluated the performance of the junta checker on artificial test cases designed to both stress the worst-case scaling behavior of the algorithm as input space increases and to evaluate a hypothesis about the characteristics of the search space that might prove challenging at a fixed junta order.

We designed two artificial function families with a fixed junta order, and ran experiments with increasing total input space size, so that the junta-forming inputs would grow more and more sparse among all the nominal inputs, a behavior that is a worst-case instance of a pattern that is expected to be common in naturally occurring functions with significant control flow, where, because of clipping of the output behavior at boundaries established by conditionals, most of the input bits associated with such inputs will yield values at conditional expressions that do not influence branches taken, and thus do not influence the outputs. Within this class, we posited a "hard" function and an "easy" function, where the easy one was constructed as a XOR of a subset of 9 of the inputs, and hard one was the same function but also constructed to be constant on a

different 3 bit subplane of the inputs, with the intent of depriving the algorithm of variability information on a significant part of the input space by masking the variability in the XORed inputs. Nonetheless, the expectations reflected in the names did not obtain, and the so-called hard function took less time to solve than the so-called easy one, as if it had about 2 fewer bits in the input space. This suggests that the query complexity of the tester has a stricter dependency on the number of bits influencing the output than expected, regardless of dependencies among bits in the input space with respect to determining the output. This informed the development of distribution modeling techniques that can either represent or ignore such dependencies (see below).

In either case, the scaling appeared at least exponential in the size of the input space, reflecting the search over power sets of decreasing sets of the inputs taking place in the algorithm. We ran experiments from 9 to 20 bits in the input space, and obtained the results depicted below.



Distribution Modeling

The adaptive junta checker requires the input distribution that samples are drawn from for checking to be specified, and ensuring that this distribution approximates the empirical distribution of inputs to the function under consideration concentrates the tester's efforts in regions of the input space most likely to be encountered in practice and thus ensures the results of checking are biased as much as possible toward inputs of practical interest.

To that end, we developed three methods of specifying the input distribution. The first uses a fixed, symmetric beta distribution between zero and one for each bit, which plays the role of an uninformative prior that is slightly biased toward bit vectors of an equal number of one and zero bits.

Two other data-adaptive methods were also developed. The second is based on iteratively increasing clustering with k-means and builds a mixture model of the empirical distribution. It is capable of learning, representing, and yielding samples from a multimodal distribution, and inference is reasonably efficient since, like the junta tester itself, it was developed to share state between models when testing a model of different dimension. It was found to be able to exactly identify the canonical basis for the three dimensional binary space from much higher dimensional data.

The third method fits a vector of Bernoulli random variables to the data, and permits smoothing of the distribution. It thus cannot represent higher-order dependencies among variables or multiple modes, but is simple, fast, and by being less biased away from uniform can serve an intermediate role between the uninformative prior and the mixture model.

Towards General Property Testing

Briefly, finding a junta is a useful preprocessing step for focusing sampling for testing other properties only on inputs which are capable of influencing the output of a function under scrutiny. We are moving on to develop more general property testing according to the framework in Chapter 6 of Oded Goldreich, *Introduction to Property Testing,* 2021 that builds on the junta tester.

The Cockatrice GP Framework and the REFUSR System (In Progress)

`Cockatrice.jl` is a general genetic programming (GP) framework that are developing for use in the REFUSR project (and its sister project, ROPER, which was partially developed under the AIMEE contract, but which falls outside the scope of ReMath). What Cockatrice provides in this case is a multiprocessing system that takes care of the basic evolutionary search architecture: a collection of geographically structured "island" populations are maintained, abstracting away from the details of individual genotypes, their phenotypic expression, and fitness functions, and tournament selection is scheduled and dispatched.

Geographical Distribution

As a means of supporting population diversity and facilitating parallelism, we have divided our populations into a set of "islands", which evolve independently from one another, save for occasional migration. Each island has its own two-dimensional, toroidal geography.

When a tournament is arranged, Cockatrice begins by chosing a random point on the island, and then randomly selects competitors from the neighbourhood of that point. The closer an individual is to the point in question, the more likely their participation in that tournament becomes. The steepness of the

distribution curve around that point can be adjusted by tweaking the `locality` parameter in the Cockatrice's configuration file.

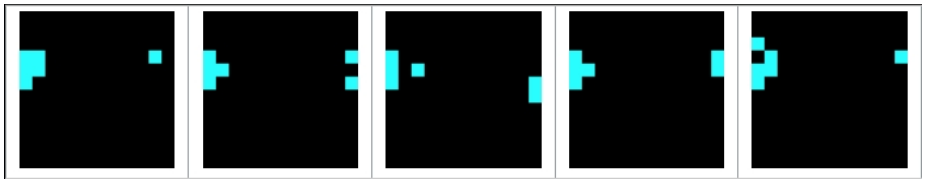


Figure 3: Five samples of tournament batches

Linear Genetic Programming in REFUSR

When designing the GP system used in REFUSR, itself, we first considered using a tree-based form of GP, in the tradition of John Koza, where each individual genotype is represented as an abstract syntax tree or symbolic expression. The idea here was that since we're ultimately looking to evolve program specifications (constrained by problem sets and inferred properties), it would be nice if our genotype representation was *already* "human readable", or expressible as a clear algebraic expression.

This brought us up against certain obstacles in memory and runtime efficiency, however, so we opted for Banzhaf-style linear GP instead, where programs are represented as a sequence of imperative instructions for a virtual register transfer machine – something resembling an assembly language for a simple architecture.

A **linear program** is a vector of **register transfer instructions**, which in the context of this report we will simply call "instructions". Each instruction is a structure with three degrees of freedom:

1. choice of source register
2. choice of destination register
3. choice of operator

When an instruction is executed on the VM, the operands are fetched from the registers – from the source register, if the operator takes just one argument, or from the source and destination registers, if the operator takes two. The result of applying the operator to these arguments is then stored in the destination register.

In the context of this report, the term **genotype**, unless otherwise specified, will refer to just these linear programs.

For the time being we're doing without jumps and conditionals. This both eases the translation between linear programs and symbolic expressions, and allows us to execute programs in a massively parallel fashion. Rather than evaluating a function of type $f : \mathbb{B}^n \to \mathbb{B}^1$ for each of m test cases, we can vectoralize our

instruction set and evaluate a function of type $f : \mathbb{B}^{mn} \to \mathbb{B}^m$, sweeping over every test case at once.

Crucially, every linear program of jump-free register transfer instructions will eventually halt – indeed, a program of length ℓ will halt after exactly ℓ steps.

It should be noted that *every concatenation of register transfer instructions* constitutes a valid linear program. The only restriction we impose is on length, for which we set an upper bound, configurable for each experiment, in order to guard against the exhaustion of computational resources.

The Probability Distribution of Linear Programs

The set of linear programs under length ℓ is easy to sample from with uniform probability. The distribution of instructions, $\mathbb{INST}$, is just the product of the three independent distributions of available source registers, destination registers, and operators.

$$\mathbb{INST} = \mathbb{SRC} \times \mathbb{DST} \times \mathbb{OP}$$

The distribution of programs of length ℓ, similarly, is just

$$\mathbb{INST}^\ell = \mathbb{INST}_1 \times \mathbb{INST}_2 \times \cdots \times \mathbb{INST}_\ell$$

To sample uniformly from the space of possible programs of length ℓ, all that needs to be done is to independently select ℓ instructions – much in the same way that to sample uniformly from the set of 64-bit integers it is sufficient to flip a fair coin 64 times.

An Argument for Immutable Input Registers and Mutable Scratch Registers

By **machine state**, here, we will mean only the register vector, as this is the only struture in the VM that our instructions are able to influence. The program counter increments inexorably with each instruction's execution, since there are no jumps, so we can ignore it in the following discussion.

Before each program is executed, the virtual machine is initialized by setting all registers to zero, except for a subset of **input registers**, D, which are loaded with the input values for a given test case. Between initialization and termination, nothing but the program itself is allowed to influence the machine state.

Linear programs, as understood in this context, are deterministic. The effect of an instruction is uniquely determined by the machine state.

Consider the quantity of information, or entropy, contained in the machine state after initialization. With the execution of any given instruction, one of

two things can happen: either the entropy level decreases or it remains constant. There is no way for entropy to increase.

Information Preserving	Information Destroying	Depends on Context
`R[1] := R[1] & true`	`R[1] := R[1] & false`	`R[1] := R[1] & R[2]`
`R[1] := R[1] | false`	`R[1] := R[1] | true`	`R[1] := R[1] | R[2]`
`R[1] := ~R[1]`	`R[1] := R[1] xor R[1]`	`R[1] := R[1] xor R[2]`

Under these conditions, as program length increases, so does the probability that that program computes a constant function. So long as there is *some* probability of encountering information-destroying instructions and instruction combinations, the loss of information will accumulate, bit by bit, until the output of the program in no way depends on input.

And since programs reproduce by division and concatenation, if a subsegment of a program is information-destroying, that loss will be inherited by its offspring.

Forcing the input registers to be *immutable* by removing them from the set of possible destination registers acts as a guard rail against information loss, and it ensures that any program whatsover can be rehabitated by an appropriate suffix. This is a simple way to protect the population's computational resources.

In our implementation, the input registers D are immutable – they can be chosen as an instruction's source register but never destination – while a set of mutable scratch registers, R, are allowed act as both source and destination.

Decompilation and Simplification

In order for our results to be of any use to a subject area expert, we translate the programs that constitute our genetic representations into concise symbolic expressions (the absence of jumps and conditionals greatly simplifies this process as well).

REFUSR's linear genotypes are composed of a series of primitive register transfer instructions. They resemble assembly code for a simple virtual machine.

The following snippet of code can be taken as a concrete example. This is register transfer code formed the genotype for a champion specimen in an experiment in which the goal was to evolve a 4-to-1 multiplexor gate.

```
R[01] ← ~ D[03]
R[04] ← mov D[01]
R[01] ← R[01] & R[04]
R[01] ← R[01] & D[04]
R[03] ← ~ D[02]
R[03] ← R[03] | D[03]
R[03] ← R[03] | D[01]
```

```
R[04] ← ~ R[03]
R[02] ← ~ R[04]
R[03] ← ~ R[02]
R[01] ← R[01] | R[03]
R[04] ← mov R[01]
R[02] ← mov D[03]
R[01] ← R[01] | R[02]
R[02] ← ~ R[04]
R[03] ← ~ R[02]
R[01] ← R[01] & D[05]
R[01] ← R[01] & D[01]
R[01] ← R[01] | R[03]
R[03] ← mov D[03]
R[04] ← ~ R[03]
R[02] ← ~ R[04]
R[03] ← ~ R[02]
R[03] ← R[03] | D[01]
R[04] ← ~ R[03]
R[02] ← ~ R[04]
R[04] ← mov R[01]
R[04] ← R[04] | D[06]
R[03] ← ~ R[02]
R[01] ← R[01] | R[03]
R[01] ← R[01] & R[04]
R[01] ← R[01] & R[04]
```

Internally, each instruction is defined as a Julia `struct`, with fields for the source and destination register, the operator (a function), and that operator's arity. (Though we're currently not using any, since they invariably destroy execution information and inhibit evolutionary search, constants can be defined as nullary functions in this fashion, by setting the arity field to zero, and the operator field to `() -> true` or `() -> false`.)

We can translate each instruction to a simple symbolic expression – indeed, a member of the Julia `Expr` type – that expresses an assignment.

```
function to_expr(inst::Inst)
    op = nameof(inst.op)
    dst = :(R[$(inst.dst)])
    src_t = inst.src < 0 ? :D : :R
    src_i = abs(inst.src)
    src = :($(src_t)[$(src_i)])
    if inst.arity == 2
        :($dst = $op($dst, $src))
    elseif inst.arity == 1
        :($dst = $op($src))
```

```
    else # inst.arity == 0
        :($dst = $(inst.op()))
    end
end
```

A sequence of instructions can then be composed into a single assignment expression by performing a series of subexpression replacements, while iterating backwards through the instruction sequence. Whenever we encounter an assignment of the form `lhs := rhs`, we simply replace all occurrences of the subexpression `lhs` with the expression `rhs` in our accumulated expression.

When the iteration is complete, we are left with a cumulative assignment instruction, which has the output register (`R[1]`) on the left-hand side, and the compound expression, generated through successive replacements, on the right.

```
function to_expr(code::Vector{Inst}; incremental_simplify=true)
    code = strip_introns(code, [1])
    if isempty(code)
        # If there's no code to execute, R[1] remains 0
        return false
    end
    expr = pop!(code) |> to_expr
    LHS, RHS = expr.args
    while !isempty(code)
        e = pop!(code) |> to_expr
        lhs, rhs = e.args
        RHS = Expressions.replace(RHS, lhs=>rhs)
    end
    # Since we initialize the R registers to `false`, any remaining R references
    # can be replaced with `false`.
    Expressions.replace(RHS, (e -> e isa Expr && e.args[1] == :R) => false)
end
```

When this function terminates, the only remaining variables in the expression will be those which correspond to the immutable input registers, and the program appears as a pure Boolean function over the inputs.

Simplifying Symbolic Expression Trees

The expressions generated from linear programs in this fashion tend to rapidly grow in complexity, becoming tarpits into which computational resources can be sunk, and unreadable thickets of parentheses that offer little insight to the subject area expert who might be hoping to use such formulas to assist in reverse engineering a black-boxed function.

Fortunately, expression rewriting and simplification tools have existed for decades, and there's no reason to reinvent this particular wheel from scratch. Like many of the other ReMath teams, the REFUSR project has converged

on the Python symbolic mathematics library, SymPy. SymPy's interoperation with our primarily Julia codebase isn't seamless, but we were greatly assisted by the PyCall API in this regard. All that remained was to write a little bit of translation code to allow Julia expressions to be translated into SymPy expressions (a proper subset of Python expressions) and back, and to implement the optimizations that would make this tool usable for us – culminating in an α-reduction invariant expression cache.

Intron Stripping

Before we even reach the decompilation stage, however, there's an inexpensive simplification that we can perform on the linear program itself. To do this, we use an algorithm which I believe goes back to Wolfgang Banzhaf.

```
@inline function semantic_intron(inst::Inst)::Bool
    inst.op in (&, |, mov) && (inst.src == inst.dst)
end


function get_effective_indices(code, out_regs)
    active_regs = copy(out_regs)
    active_indices = []
    for (i, inst) in reverse(enumerate(code) |> collect)
        if !(semantic_intron(inst)) && inst.dst in active_regs
            push!(active_indices, i)
            filter!(r -> r != inst.dst, active_regs)
            inst.arity == 2 && push!(active_regs, inst.dst)
            inst.arity >= 1 && push!(active_regs, inst.src)
        end
    end
    reverse(active_indices)
end


function strip_introns(code, out_regs)
    code[get_effective_indices(code, out_regs)]
end
```

This gives us what Banzhaf calls the "effective code" of the genotype, the code that actually influences the output. A great deal of execution time can be saved, in fact, by *only* executing these instructions, while ignoring the rest of the chromosome, and this is just what Cockatrice does. The example of linear code displayed above, in fact, shows only the effective code of the champion 4-to-1 MUX champion.

What remains, once we separate the effective code from a chromosome, is what Banzhaf calls the genotype's "introns", by analogy with non-coding DNA in

biological genotypes. The tendency for GP systems to accumulate intron bloat if left unchecked is one of the most interesting, general, and robust phenomena in this domain, but an adequate discussion of this topic is beyond of the scope of this report.

Incremental Simplification

The cost of expression simplification grows explosively with the size of the expression, and worst-case expression complexity tends to grow with the length of the instruction list. And, in general, as a population of programs evolves, in search of a target function, there will tend to be a complexification of implicit logical structure in those programs. For these reasons, it's often better to apply the simplification algorithm incrementally, after each replacement operation in the decompilation algorithm, than it is to wait for the entire list to be decompiled into a single complex expression before simplification begins.

On reflection, moreover, we can see that there's no need to attempt simplification after *every* assignment is decompiled into a subexpression substitution – there's only reason to do so when the `rhs` of the assignment shares variables with the accumulated expression, *minus* the assignment's `lhs`. If there are no shared variables, then there can be no *new* logical relations, unless the new `rhs` term evaluates to a constant `true` or `false`.

Since the only instruction in our instruction set that can result in a single instruction evaluating to a Boolean value is `xor`, we can easily faciliate that case by adding an *ad hoc* condition to the beginning of our `to_expr(inst::Inst)` method:

```
if inst.op == xor && inst.src == inst.dst return false end
```

The decompiler function now reads as follows:

```
function to_expr(code::Vector{Inst};
                 intron_free = true,
                 incremental_simplify = true,
                 alpha_cache=true)
    code = intron_free ? copy(code) : strip_introns(code, [1])
    if isempty(code)
        return false
    end
    expr = pop!(code) |> to_expr
    LHS, RHS = expr.args
    while !isempty(code)
        e = pop!(code) |> to_expr
        lhs, rhs = e.args
        RHS = Expressions.replace(RHS, lhs => rhs)

        if incremental_simplify
```

```
            # We only need to simplify again if rhs has common variables with RHS minus lhs
            RHS_minus_lhs = Exp.replace(RHS, lhs => :XYZZY)
            if rhs isa Bool || Exp.shares_variables(RHS_minus_lhs, rhs)
                RHS = Exp.simplify(RHS; alpha_cache)
            end
        end
    end
    # Since we initialize the R registers to `false`, any remaining R references
    # can be replaced with `false`.
    RHS = Expressions.replace(RHS, (e -> e isa Expr && e.args[1] == :R) => false)
    if incremental_simplify
        return Expressions.simplify(RHS; alpha_cache)
    else
        return RHS
    end
end
```

Caching

Let's look closely at particular, single-island population of 100 genotypes, each with a maximum code length of 100, using 6 immutable input registers, `D[1:6]`, and 6 mutable scratch registers, `R[1:6]`. `R[1]` is designated as the output register – whichever value is held by `R[1]` at the end of execution is taken as the program's return value.

What we are tracking here is growth in complexity of expressions obtained from our linear program population P by applying our "naïve" decompilation algorithm to those programs, $N(P)$, and the complexity of the expressions obtained from the latter by simplification, $S(N(P))$.

The exact numbers can vary wildly from run to run, even when, as here, we begin with identical initial populations. But the expression complexity of the naively decompiled programs is *consistently* orders of magnitude greater than what we find in the simplified program, and growing at a tremendously accelerated rate.

And it only gets worse from there, for the naive method of expression decompilation, while our simpification technique appears to consistently suppress expression bloat.

By utilizing a 512 mibibyte cache with the `simplify()` function, we're able to obtain an impressive, 100x speedup when decompiling a virgin, unevolved population.

```
julia> Expressions._use_cache(false); Expressions.flush_cache!()
LRU{Expr, Union{Bool, Expr, Symbol}}(; maxsize = 1048576)

julia> @btime s = LinearGenotype.decompile(rand(evoL.geo.deme), cache=false)
908.448 microseconds (3226 allocations: 281.33 KiB)
```

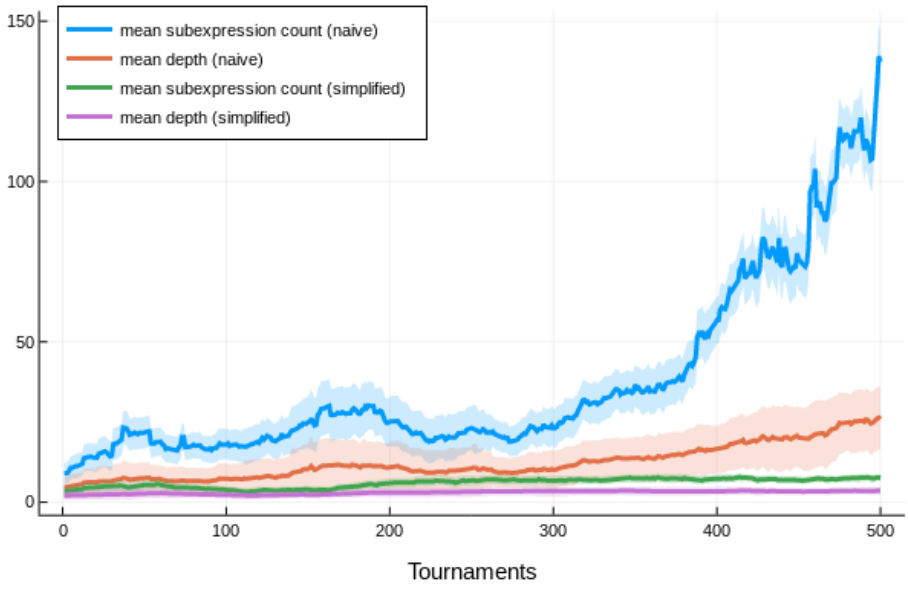


Figure 4: Mean Naive Expression Complexity Over Time

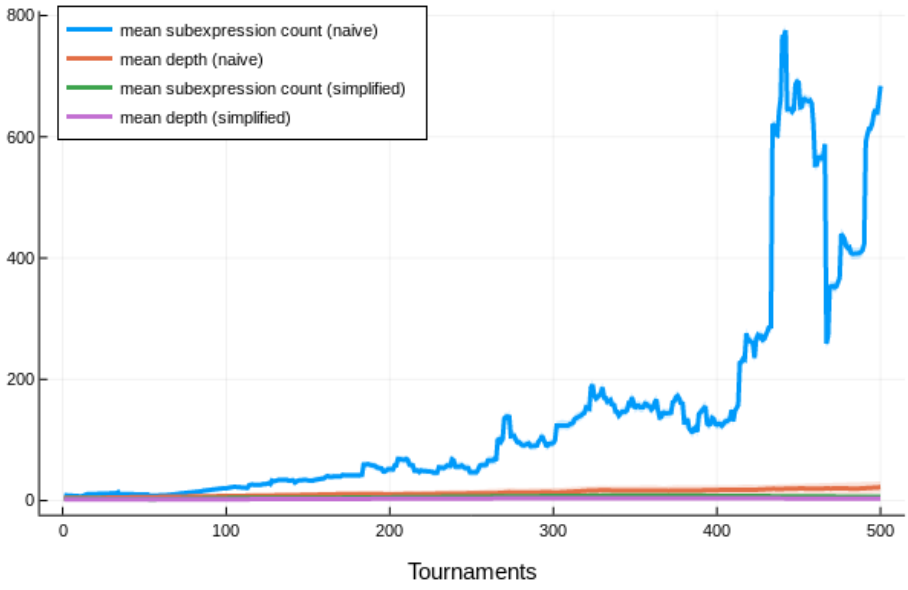


Figure 5: Mean naive expression complexity over time, a second trial

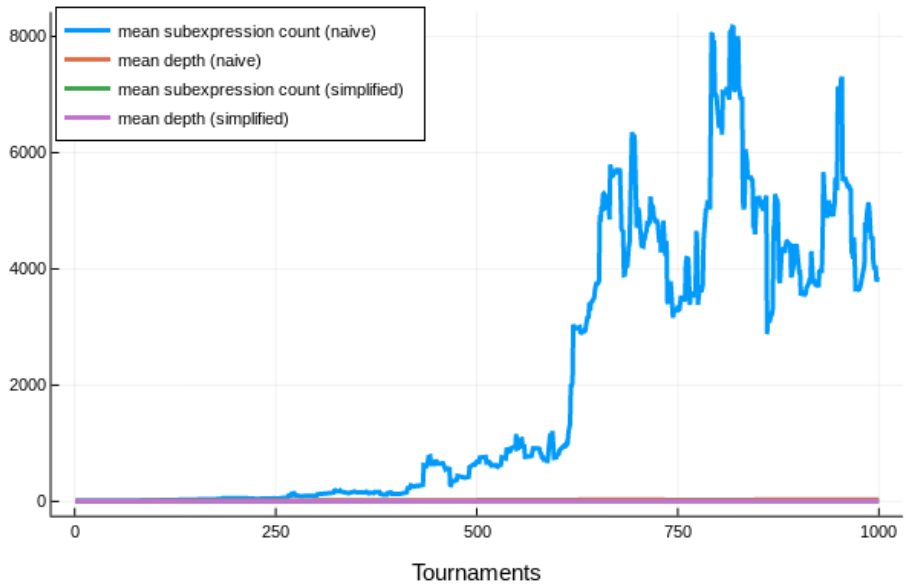


Figure 6: Mean naive expression complexity over time, after 1000 tournaments

```
true
```

```
julia> Expressions._use_cache(true); Expressions.flush_cache!()
LRU{Expr, Union{Bool, Expr, Symbol}}(; maxsize = 1048576)
```

```
julia> @btime s = LinearGenotype.decompile(rand(evoL.geo.deme), assign=false)
8.610 microseconds (57 allocations: 2.39 KiB)
:(D[1])
```

Now, naive (unsimplified) expression complexity tends to increase as the population evolves, as more or less coherent logical structure begins to crystalize in the soup of once merely random instructions. This makes the simplification algorithm increasingly costly to run. Indeed, before we implemented incremental simplification in the decompilation algorithm, simplifying genome at a late stage in the evolution would often take upwards of 30 minutes, if it didn't exhaust the memory of our workstation entirely.

Furthermore, since the population evolves through recombinatorial (and sometimes mutational) means – even though this is at the level of linear instructions and not symbolic expressions – we should expect common subexpressions to recur quite frequently, which makes a compelling case for caching.

Alpha-Reduction Invariant Caching

Since the possibilities for simplifying an expression do not depend on the particular choice of variable names, it would be a waste of time to perform the expensive computations involved in simplifying an expression e if we have already simplified e', so long as e can be produced by renaming the variables in e', in a one-to-one fashion. In the terminology of λ-calculus, e and e' are thus said to be α-equivalent, or equivalent modulo α-reduction.

Our caching algorithm captures this intuition by applying a canonical variable renaming on expressions and their simplifications before entering them into the cache, and by applying the same renaming scheme on an expression before consulting the cache for known simplifications.

```
function check_cache(e)
    try
        alpha, mapping = rename_variables(e)
        result_alpha = CACHE[alpha] # may throw a KeyError
        return restore_variables(result_alpha, mapping)
    catch er # KeyError
        if !(er isa KeyError) throw(er) end
        return nothing
    end
end

function cache(e, result)
    e_alpha, mapping = rename_variables(e)
    result_alpha, _ = rename_variables(result; mapping)
    CACHE[e_alpha] = result_alpha
end
```

The renaming scheme itself is simple: we perform a pre-order traversal on the expression (any fixed traversal scheme will do), and whenever we encounter a new variable, we choose a new name for it from a fixed list of variable names that we know do not occur in the expression.

```
function alpha_mapping(e; letter=:alpha)
    vars = variables_used(e)
    alpha = [:($(letter)[$(i)]) for i in 1:length(vars)]
    zip(vars, alpha)
end

function rename_variables(e::Expr; letter=:alpha, mapping=alpha_mapping(e;letter))
    e_alpha = subs(e, Dict(mapping))
    return (e_alpha, mapping)
end

function restore_variables(e::Expr, mapping)
```

```
    subs(e, Dict((v,a) for (a,v) in mapping))
end
```

For example, this gives us:

```
e = :((~(D[2] xor D[3]) | (D[4] & D[1] xor (D[1] | D[2]))) xor D[4])
D[2] --> alpha[1]
D[3] --> alpha[2]
D[4] --> alpha[3]
D[1] --> alpha[4]
alpha = :((~(alpha[1] xor alpha[2]) | (alpha[3] & alpha[4] xor (alpha[4] | alpha[1]))) xor a
```

Any expression that is α-equivalent to e will be mapped to the exact same expression by our `rename_variables()` function.

$$\forall e \forall x (x \equiv^{\alpha} e \implies rename(x) = rename(e))$$

It follows, incidentally, that the renaming operation is idempotent.

Example

With this "decompilation" system in place, the linear program presented above can be translated into the a concise expression, which defines the multiplexor as the following sum of products:

$$(D_1 \wedge D_3 \wedge D_5) \vee (D_1 \wedge D_4 \wedge \neg D_3) \vee (D_3 \wedge D_6 \wedge \neg D_1) \vee (D_2 \wedge \neg D_1 \wedge \neg D_3)$$

REFUSR also emits a graphical representation of these simplified expressions, which can be of great assistance in analysis. Here we can see not only that we're looking at a multiplexor, but that the two variables D_1 and D_3 serve as the control bits.

Performance Benchmarks

TODO: Say something about this. Alpha invariant caching only pays for itself once program complexity has reached a certain level. Try this again with mux 8-to-1?

The cost of "decompiling" a program can vary wildly, and so extensive benchmarking trials were necessary in order to obtain reliable observations regarding the impact of these three optimization strategies – to wit

1. caching expressions and their simplifications in a bounded least-recently-used (LRU) cache,
2. incrementally simplifying the expression at certain stages in the "decompilation" process rather than waiting for a finished expression, and

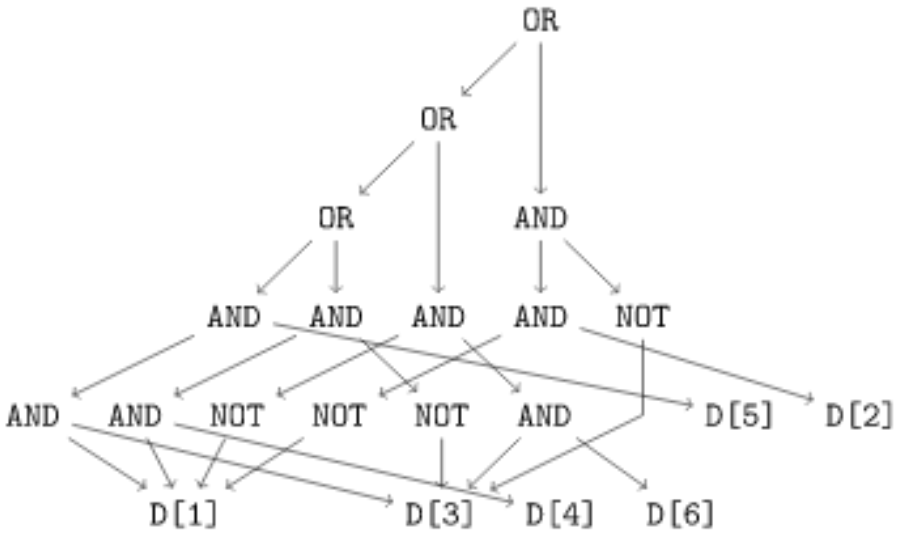


Figure 7: Syntax graph of a 4-to-1 multiplexor specimen

3. modifying the expression cache so that the caching and lookup operations would be invariant with respect to alpha-reduction

Various combinations of these strategies are indicated by the following keys in the violin plots below:

key	cache	alpha-reduction invariance	incremental simplification
no cache	no	no	no
inc 1 cache 0	no	no	yes
inc 1 α 0	yes	no	yes
inc 0 α 1	yes	yes	no
inc 1 α 1	yes	yes	yes

The implicit logical complexity of programs tends to increase as the population evolves, and so too does the cost of decompilation. We have therefore found it important to benchmark these techniques as applied to populations at various "snapshots" of their evolutionary development.

Two populations are experimented upon in the following: one instructed to evolve a solution to the 4-to-1 MUX problem, and one instructed to evolve a solution to the 8-to-1 MUX problem. The parameters supplied to each population differed in a few significant respects:

parameter	4-to-1	8-to-1
maximum program length	200	400
number of input registers	6	11
number of scratch registers	5	6
population size	100	144

Benchmarking the decompiler on a 4-to-1 MUX solving population

Benchmarking the decompiler on the 8-to-1 MUX solving population

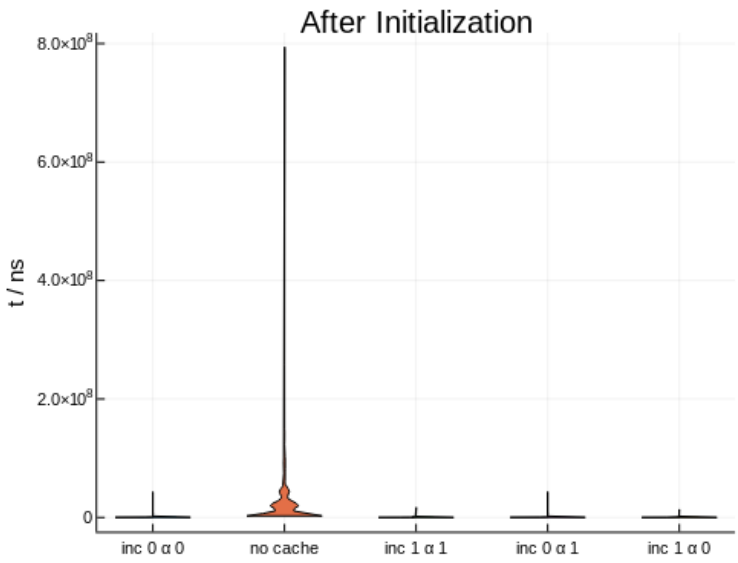


Figure 8: Benchmarking decompiler on 4-to-1 MUX population after initialization

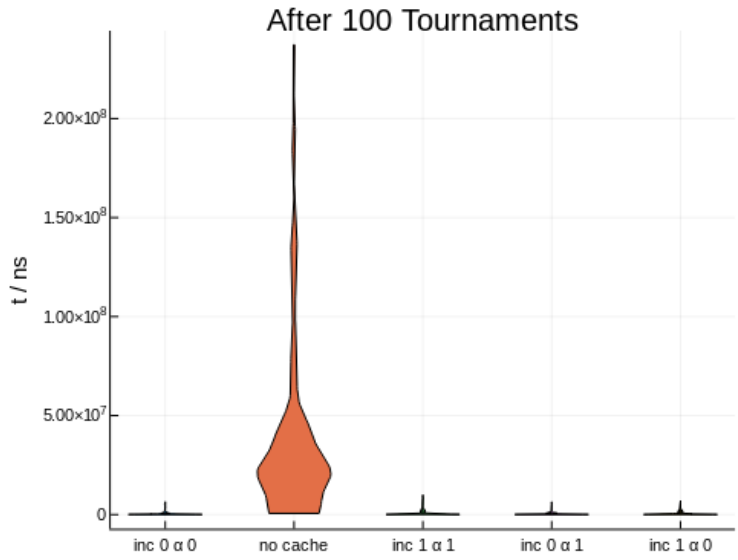


Figure 9: Benchmarking decompiler on 4-to-1 MUX population after 100 tournaments

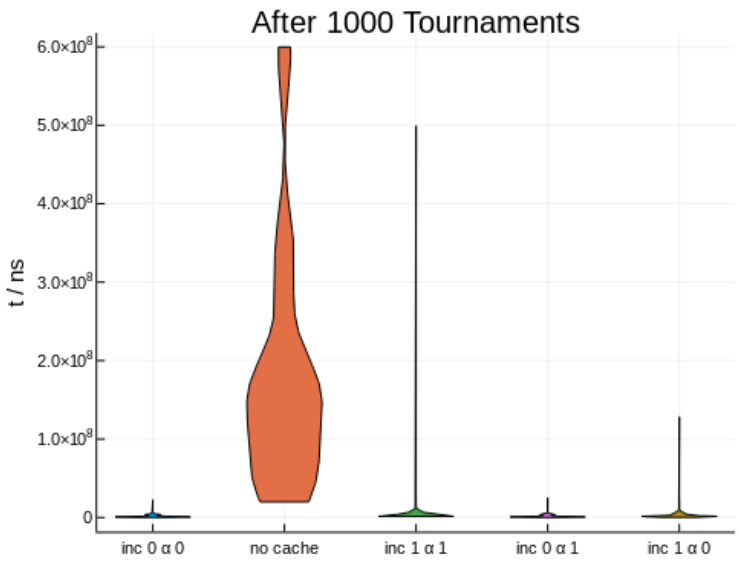


Figure 10: Benchmarking decompiler on 4-to-1 MUX population after 1000 tournaments

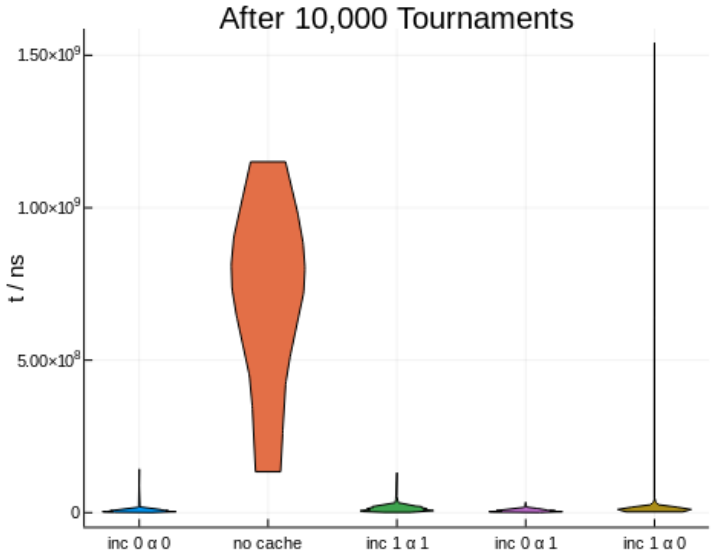


Figure 11: Benchmarking decompiler on 4-to-1 MUX population after 10,000 tournaments

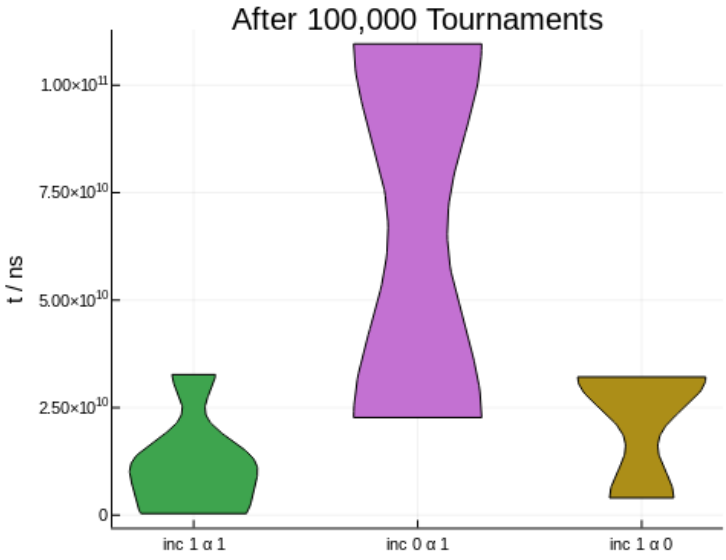
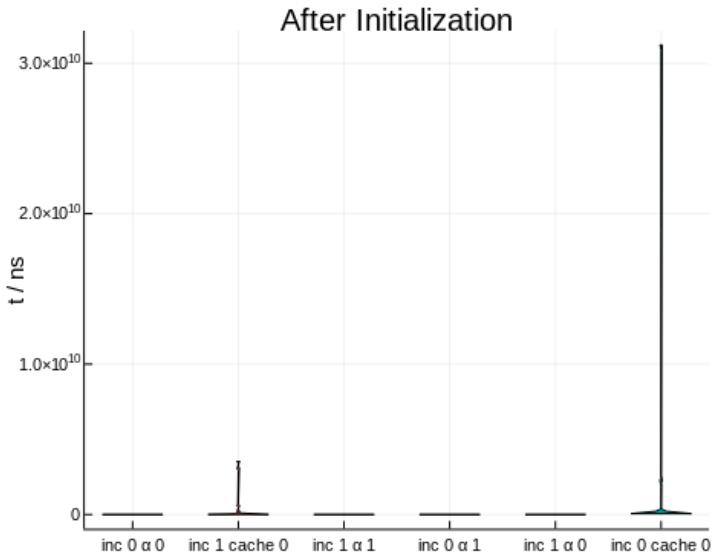
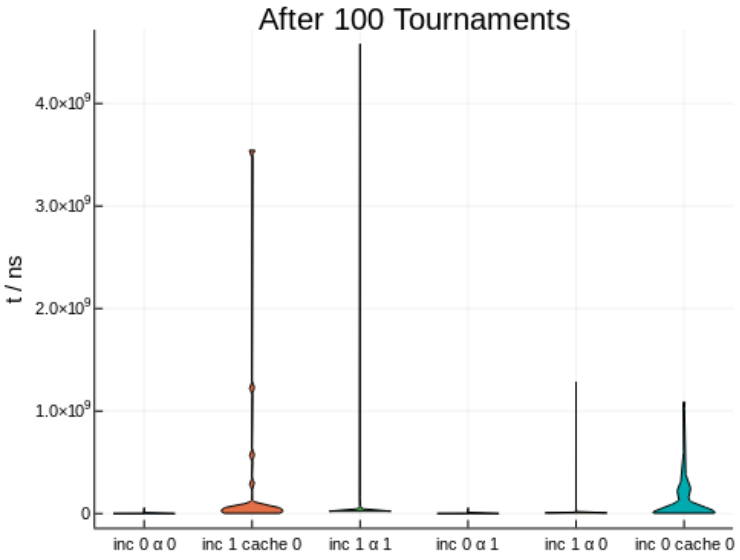
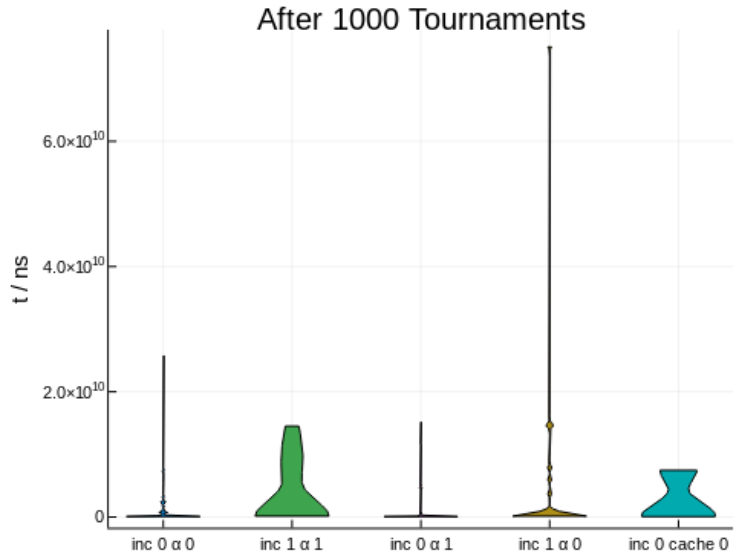


Figure 12: Benchmarking decompiler on 4-to-1 MUX population after 100,000 tournaments




After 100 Tournaments
t / ns
4.0×10^9
3.0×10^9
2.0×10^9
1.0×10^9
0
inc 0 α 0
inc 1 cache 0
inc 1 α 1
inc 0 α 1
inc 1 α 0
inc 0 cache 0


After 1000 Tournaments
t / ns
6.0×10^10
4.0×10^10
2.0×10^10
0
inc 0 α 0
inc 1 α 1
inc 0 α 1
inc 1 α 0
inc 0 cache 0

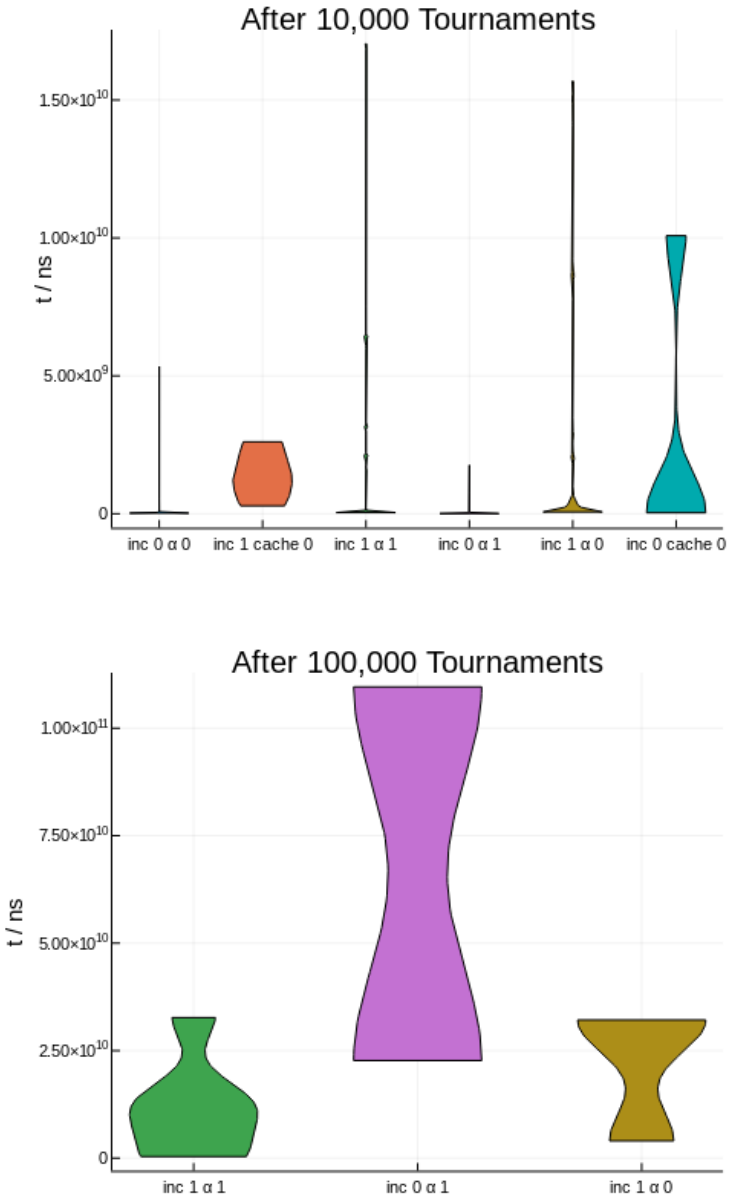


Figure 13: Benchmarking the decompiler on the 8-to-1 MUX population after 100,000 tournaments

Using a 300-second window

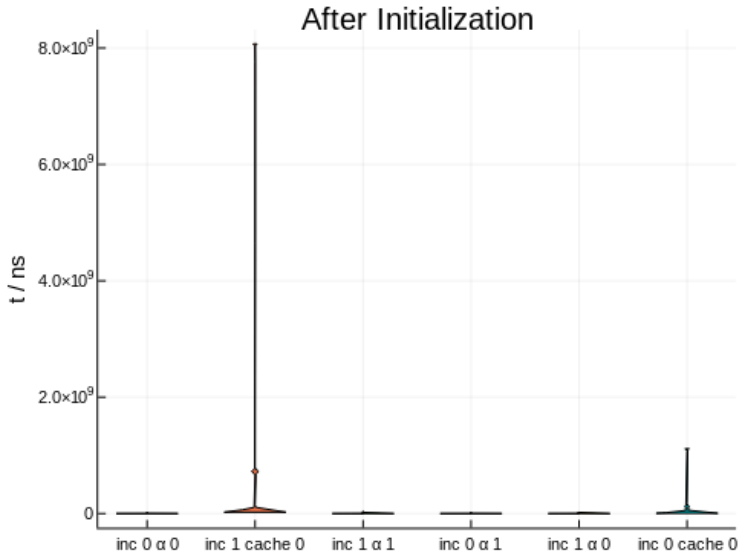


Figure 14: Benchmarking the decompiler on the 8-to-1 MUX population after initialization

![Benchmarking the decompiler on the 8-to-1 MUX population after 100 tournaments

Benchmarking Conclusions

The conclusions we have drawn are the following:

1. it is always better to cache than to not
2. alpha-reduction invariant caching pays for itself only once the load on the simplification engine is particularly high; for complex and long-evolved populations it is well worth the relatively minor cost it incurs, but it will not dramatically improve performance on relatively simple specimens
3. incremental simplification has been observed to make the intractable tractable on occasion (primarily by reducing the memory footprint of extremely complex simplification loads), but as currently implemented is somewhat costly in the average case; perhaps we can fine tune the threshold at which incremental simplification is brough into effect, rather than applying it at all stages where variable interaction is possible.

The Use of Execution Trace Information

One plausible metric for efficiently approximating how "near" a candidate program is to the target is to measure the mutual information between its output

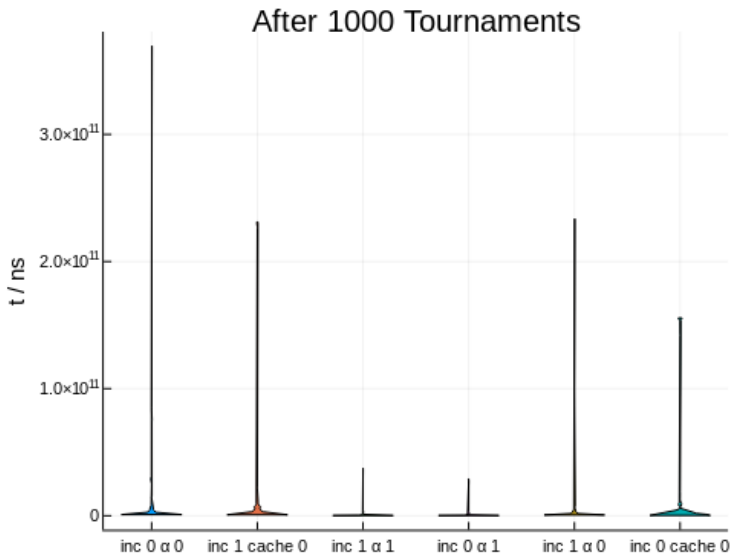


Figure 15: Benchmarking the decompiler on the 8-to-1 MUX population after 1000 tournaments

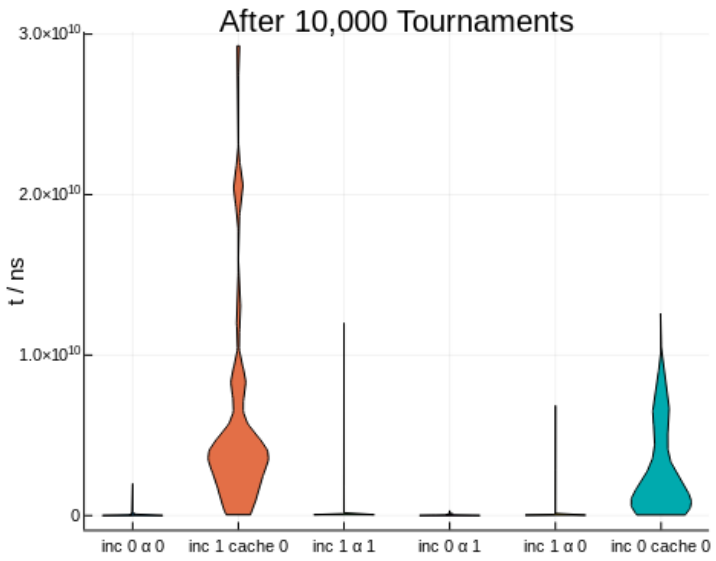


Figure 16: Benchmarking the decompiler on the 8-to-1 MUX population after 10,000 tournaments

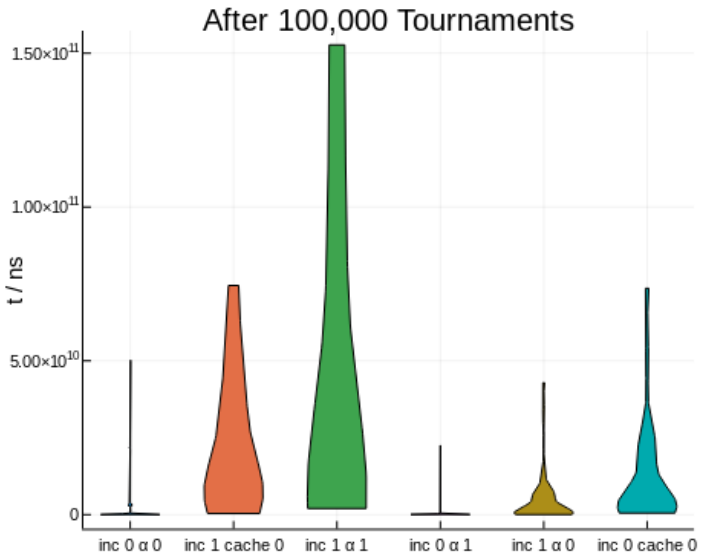


Figure 17: Benchmarking the decompiler on the 8-to-1 MUX population after 100,000 tournaments

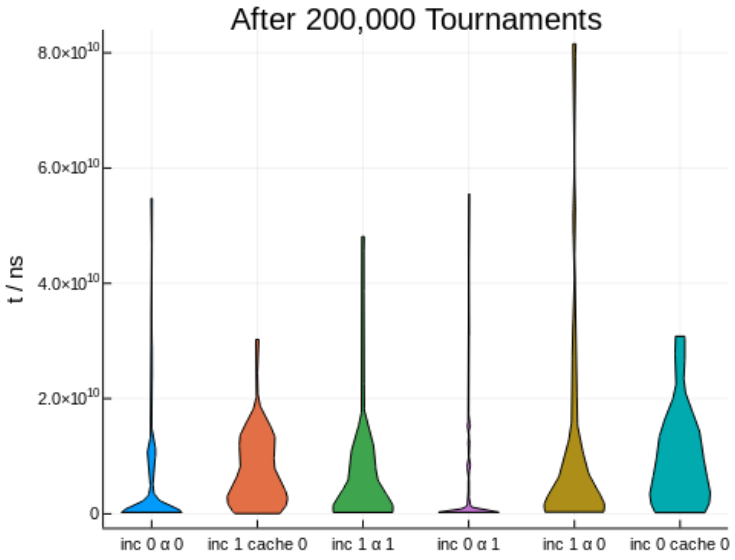


Figure 18: Benchmarking the decompiler on the 8-to-1 MUX population after 100,000 tournaments

vector and the target output vector (normalized by dividing the result by the target vector's entropy). For instance, these two programs will have amount of mutual information from any target. This is a desirable property, since, though they differ greatly in hamming distance, they are just one mutational step apart: the second is just the first, negated.

```
D[1] & D[2]   = 0001
~D[1] | ~D[2] = 1110
```

Programs reproduce not only through mutation but recombination as well. This motivates us to assign fitness scores not only on the basis of a program's final output but its intermediate states as well – a useless final output does not mean that a program doesn't contain computational resources that crossover might extract.

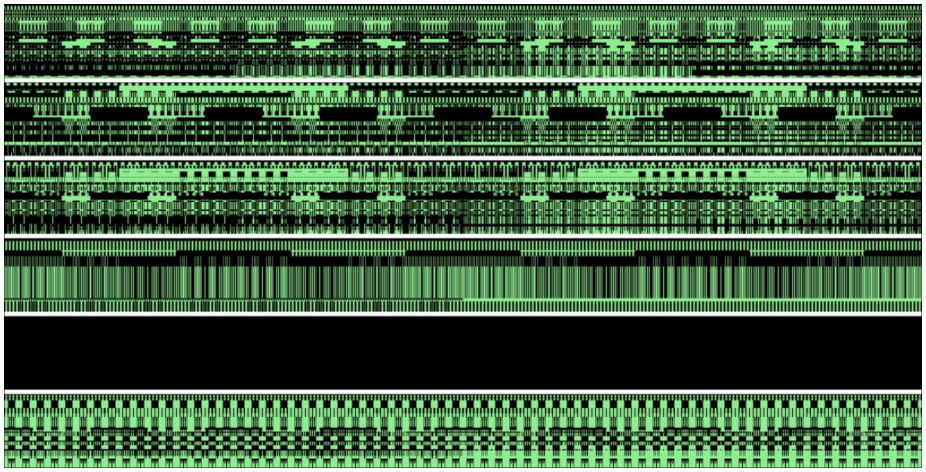


Figure 19: Execution trace of an 8-to-1 multiplexor champion

Cockatrice supplies this information by recording a **trace** of machine states after each instruction in a program is executed. This gives us a three-dimensional array T whose axes correspond to the (mutable) register index r, the choice of input vector (i.e., the truth table row from which the input is drawn) i, and the instruction index pc.

We then compute an information record by iterating over the trace's pc axis, and attending to the most recently modified register index on r. This gives us a vector $T_{pc,r}$. We calculate the mutual information of this state vector with the target, and divide the result by the target's Shannon entropy to normalize the score.

$$m = \left\langle \frac{I(T_{pc,r}; \text{Target})}{H(\text{Target})} \mid pc \in 1 \ldots \ell(\text{program}) \right\rangle$$

The high point of this vector, $\max(m)$, is inserted into the fitness vector as one of three attributes that will be used to rank tournament competitors and decide who will go on to reproduce, and who will be culled from the population.

Weighting the Distribution of Crossover Points with Trace Information

We then retain the trace information vector, and store it in one of the genotype's metadata fields. If that genotype should happen to reproduce, the trace information vector m will be used to influence the choice of crossover point.

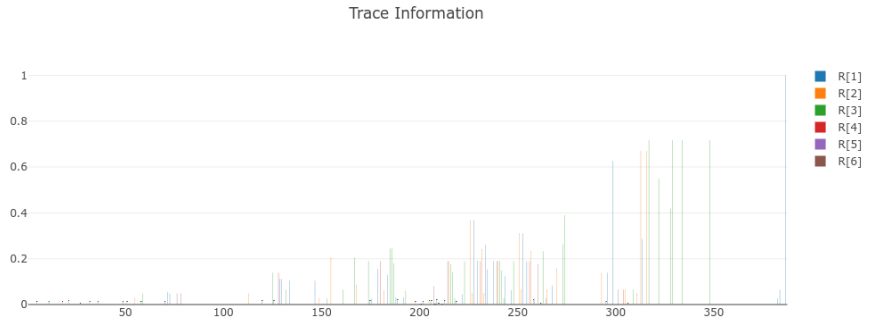


Figure 20: A plot of trace information by active register, taken from a champion of the 8-to-1 multiplexor experiment.

The basic crossover algorithm we use in Cockatrice is one-point crossover. Ordinarily, this involves selecting two parent genotypes, m and f, and then choosing, with uniform probability, an index for each, m_i and f_j. We then construct two child genotypes, b and s, like so:

$$b = \langle m_1, m_2, \ldots m_i; f_{j+1}, f_{j+2}, \ldots f_{\text{end}} \rangle \; s = \langle f_1, f_2, \ldots f_j; m_{i+1}, m_{i+2}, \ldots m_{\text{end}} \rangle$$

Our algorithm differs only in the choice of the crossover points i and j. Rather than drawing these from the set of m and f's indices with uniform probability, we define a probability distribution for each parent on the basis of the the information trace m.

The idea here is that we thereby increase the odds of chosing a particularly "fruitful" site for recombination, breaking each genotype at a site where it was potentially "on the right track", and cutting away from the genome sites where information is unduly destroyed.

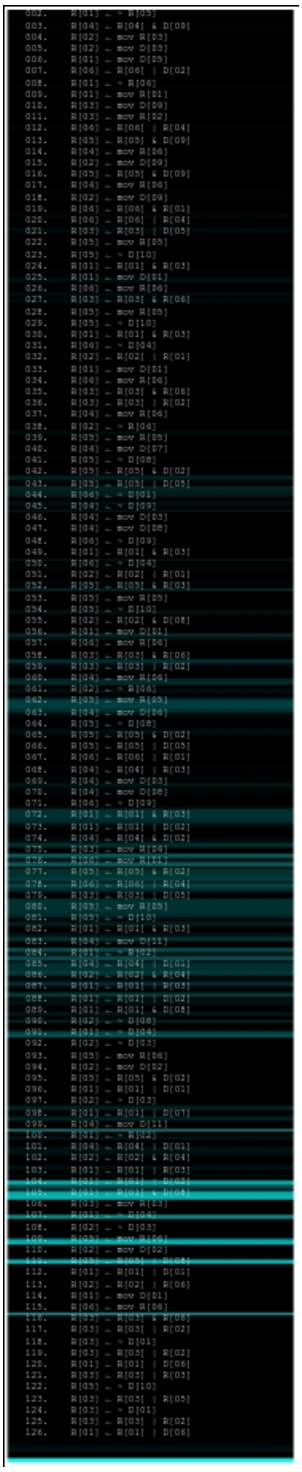


Figure 21: Weighting the distribution of crossover points on the chromosome with trace information

It is currently unclear if this strategy will have any noteworthy effects on performance. The most recent batch of experiments, where the two crossover strategies were each employed in 8 trials of the 4-to-1 multiplexor experiment (with fitness sharing), returned inconclusive results.

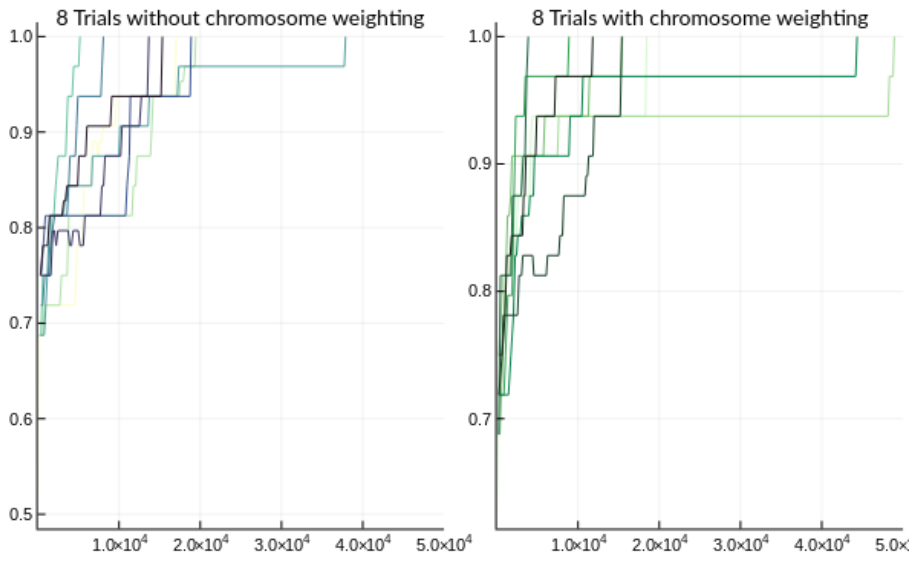


Figure 22: Chromosome weighting trials: inconclusive results

Implicit Fitness Sharing and Interaction Matrices

One of the most serious hazards that an evolutionary process can encounter is a premature collapse of population diversity. This can happen when a particular genetic line G acquires a decisive competitive advantage early in the process by performing very well on a subset $S \subset T$ of the test cases – a significantly larger subset, let's say, than those handled by competing gene lines. It may be the case that G lacks the computational resources to handle cases outside of S, but so long as the rewards meted out by the fitness function are proportionate to the number of cases correctly solved, without regard for the rarity or difficulty of solutions, we may soon see a population dominated almost entirely by recombinations of G, each exhibiting strikingly similar behaviour.

One way to offset such a premature convergence and loss of genetic information is to adjust the reward for a test case according to the frequency with which that test case has been successfully solved. This can easily be done by maintaining a data structure called an *interaction matrix*, a 2-dimensional array whose rows represent test cases and whose columns represent individuals in the population. `I[i,j]` is set to `1` if individual `j` solves test case `i`, and `0` otherwise. Each test

case can then be assigned a "difficulty" score simply by taking the mean of its corresponding row, negated. Each individual then receives an award equal to the mean of the difficulty scores for the problems it's solved.

```
difficulty_scores = [(~).(row .xor answer_vector) |> mean for row in eachrow(IM)]
correct_results   = (~).(answer_vector .xor result_vector) #
adjusted_rewards  = correct_results .* difficulty_scores
aggregate_reward  = mean(adjusted_rewards)
```

Interaction matrices are used to calculate the relative selective pressures of each test case – each set of inputs for a Boolean function, or the input row of its truth table. Each case is assigned a difficulty score, equal to 1 minus the frequency with which its solution appears in the existing population (i.e., (~).(row .xor answer_vector) |> mean, in Julia). An individual is assigned a score equal to the mean difficulty of the cases they solved.

Each subpopulation, or "island", maintains its own interaction matrix. In the visualizations to the left, each row represents a test case, and each column represents an individual program. Test cases are sorted by Gray code, to preserve locality on the Boolean hypercube (two adjacent test cases differ by exactly one bit flip), and individuals are sorted according to Hilbert curve through the 2-dimensional island population, to preserve geographical locality.

This provides us with a succinct impression of each population's phenotypic diversity. A lack of diversity in a population shows up in the visualized interaction matrix in the form of solid horizontal stripes, punctuated only by thin and ephemeral columns of noise. A phenotypically diverse population manifests a diversity of column patterns – not exactly noise, but a greater variety and complexity of pattern.

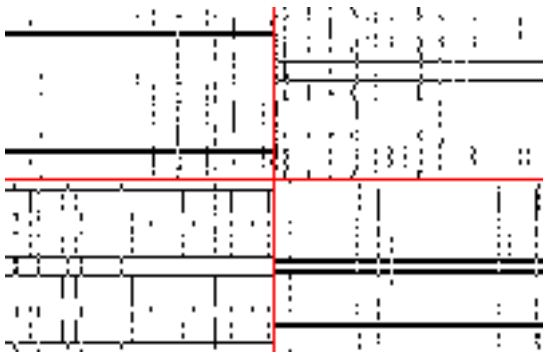


Figure 23: Interaction matrix of 4-island population *without* fitness sharing, after 100k tournaments. Problem unsolved.

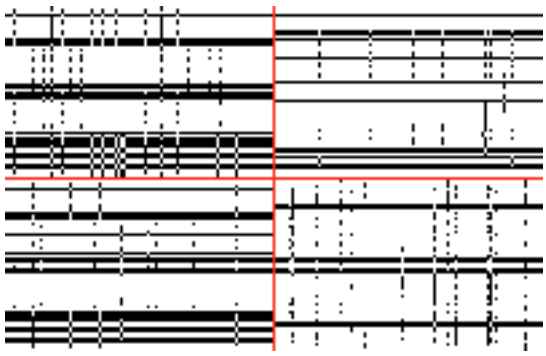


Figure 24: Interaction matrix of the same 4-island population *without* fitness sharing, after 20k tournaments. Problem unsolved.

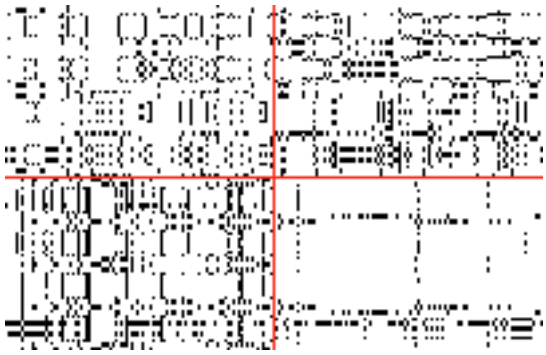


Figure 25: Interaction matrix of a 4-island population *with* fitness sharing, after 20k tournaments. Problem solved.

Fitness Sharing Provides a Decisive Advantage

The following plots show the best objective performance found in two batches of 18 trials of the 4-to-1 MUX problem: the first batch without fitness sharing, the second batch with.

Only 61% of the trials without fitness sharing discovered the target function in 50,000 tournaments or fewer, as compared to 100% of the fitness sharing trials.

And the trials with fitness sharing discovered the target in 42% of the time taken by even the successful non-sharing trials, on average.

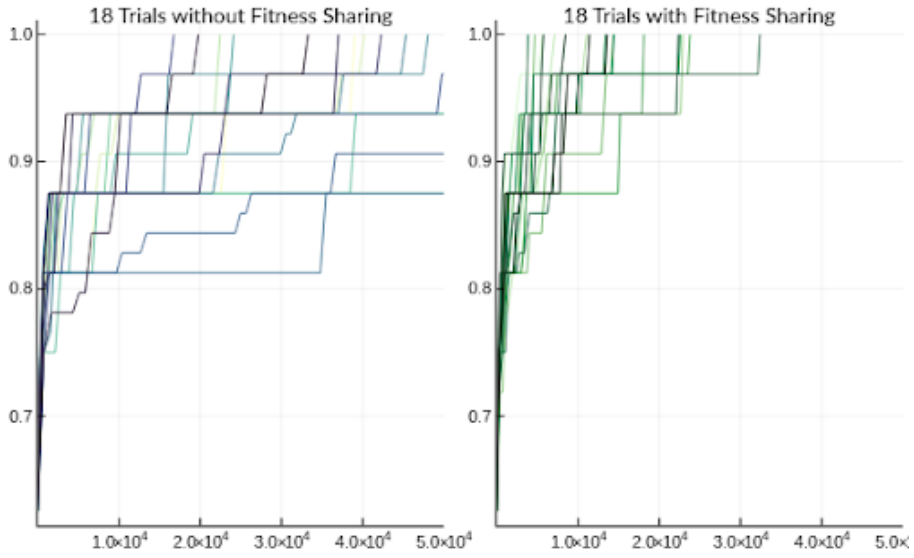


Figure 26: 18 trials with and without fitness sharing on the 4-to-1 multiplexor problem

It has only been with fitness sharing, moreover, that we have been able to solve the substantially more complex problem of reversing an 8-to-1 multiplexor, a Boolean function of eleven variables. With fitness sharing in effect, Cockatrice is able to consistently find solutions for this problem, typically within a few hours of wall clock time.

Applying Informational Crossover Weighting to the 8-to-1 MUX Problem

Though we have not yet run a statistically significant number of trials, we have seen some indications the informational weighting of crossover point distributions *may* accelerate the search for targets as complex as the 8-to-1 MUX. In

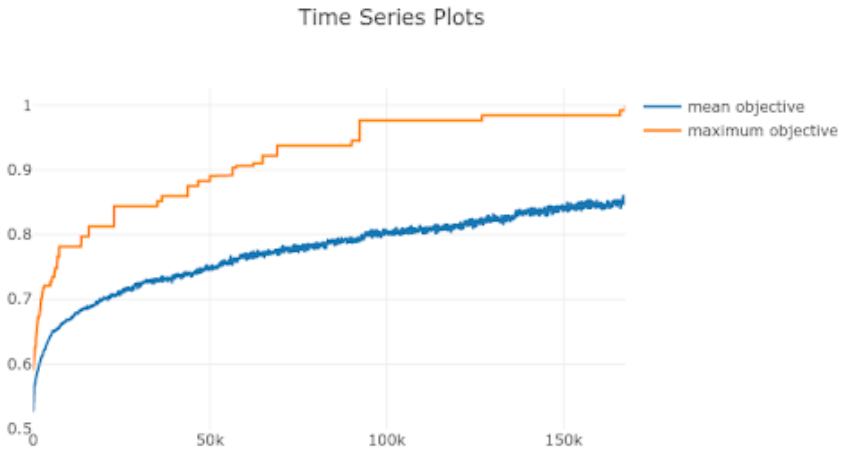


Figure 27: Time series plot of mean and maximum performance in an 8-to-1 multiplexor search.

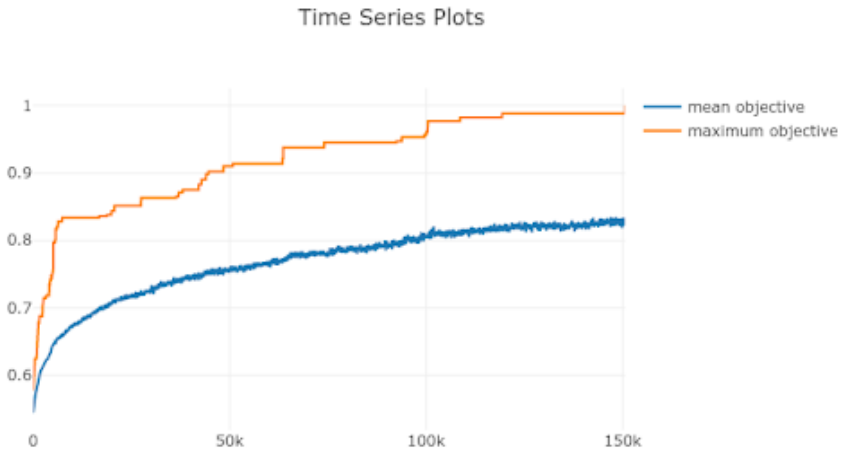


Figure 28: Time series plot of mean and maximum performance in an 8-to-1 multiplexor search.

```
003.  R[04] ← R[04] & D[09]
005.  R[02] ← mov D[03]
007.  R[06] ← R[06] | D[02]
008.  R[01] ← ~ R[06]
011.  R[03] ← mov R[02]
012.  R[06] ← R[06] | R[04]
017.  R[04] ← mov R[06]
018.  R[02] ← mov D[09]
019.  R[06] ← R[06] & R[01]
020.  R[06] ← R[06] | R[04]
021.  R[03] ← R[03] | D[05]
025.  R[01] ← mov D[01]
027.  R[03] ← R[03] & R[06]
030.  R[01] ← R[01] & R[03]
031.  R[06] ← ~ D[04]
032.  R[02] ← R[02] | R[01]
033.  R[01] ← mov D[01]
035.  R[03] ← R[03] & R[06]
036.  R[03] ← R[03] | R[02]
038.  R[02] ← ~ R[06]
049.  R[01] ← R[01] & R[03]
050.  R[06] ← ~ D[04]
051.  R[02] ← R[02] | R[01]
055.  R[02] ← R[02] & D[08]
056.  R[01] ← mov D[01]
058.  R[03] ← R[03] & R[06]
059.  R[03] ← R[03] | R[02]
070.  R[04] ← mov D[08]
072.  R[01] ← R[01] & R[03]
073.  R[01] ← R[01] | D[02]
074.  R[04] ← R[04] & D[02]
076.  R[06] ← mov R[01]
078.  R[06] ← R[06] | R[04]
110.  R[02] ← mov D[02]
113.  R[02] ← R[02] | R[06]
114.  R[01] ← mov D[01]
120.  R[01] ← R[01] | D[06]
124.  R[03] ← ~ D[01]
125.  R[03] ← R[03] | R[02]
126.  R[01] ← R[01] | D[06]
128.  R[04] ← ~ R[03]
129.  R[01] ← R[01] & R[03]
130.  R[01] ← R[01] & R[03]
131.  R[06] ← R[06] | R[04]
132.  R[03] ← R[03] | D[05]
134.  R[01] ← R[01] & R[02]
136.  R[02] ← ~ R[06]
147.  R[01] ← R[01] & R[03]
149.  R[02] ← R[02] & D[03]
153.  R[03] ← mov R[02]
155.  R[02] ← R[02] | R[01]
157.  R[06] ← ~ D[04]
158.  R[06] ← R[06] | D[01]
161.  R[03] ← R[03] & R[06]
167.  R[03] ← R[03] | R[02]
168.  R[02] ← R[02] & D[07]
170.  R[06] ← ~ D[04]
171.  R[06] ← R[06] | D[01]
172.  R[01] ← mov D[01]
174.  R[03] ← R[03] & R[06]
175.  R[01] ← R[01] & D[11]
176.  R[01] ← R[01] | D[06]
179.  R[01] ← R[01] & R[03]
180.  R[04] ← ~ R[03]
182.  R[02] ← R[02] & D[03]
184.  R[01] ← R[01] & D[11]
185.  R[03] ← R[03] | R[02]
186.  R[03] ← R[03] | R[02]
187.  R[03] ← R[03] | D[05]
188.  R[03] ← R[03] & D[04]
189.  R[01] ← R[01] | D[02]
192.  R[01] ← R[01] & R[03]
193.  R[03] ← mov R[02]
195.  R[06] ← ~ D[04]
198.  R[02] ← mov D[07]
199.  R[06] ← R[06] & R[02]
201.  R[01] ← R[01] & R[03]
202.  R[01] ← R[01] | D[06]
205.  R[02] ← R[02] & D[08]
206.  R[03] ← mov R[02]
207.  R[06] ← R[06] | R[04]
208.  R[02] ← R[02] & R[04]
209.  R[01] ← R[01] | R[03]
210.  R[01] ← R[01] | D[02]
211.  R[06] ← R[06] & R[02]
213.  R[01] ← R[01] & R[03]
214.  R[06] ← R[06] | R[04]
215.  R[02] ← ~ R[06]
216.  R[03] ← R[03] | R[02]
217.  R[03] ← R[03] | D[05]
218.  R[03] ← R[03] & D[04]
219.  R[01] ← R[01] | D[02]
222.  R[01] ← R[01] & R[03]
223.  R[03] ← mov R[02]
225.  R[06] ← ~ D[04]
226.  R[02] ← R[02] | R[01]
227.  R[02] ← R[02] & R[04]
228.  R[01] ← R[01] | R[03]
230.  R[01] ← R[01] & D[11]
231.  R[04] ← ~ R[03]
232.  R[02] ← R[02] | R[01]
233.  R[02] ← R[02] & R[04]
234.  R[01] ← R[01] | R[03]
235.  R[01] ← R[01] | D[02]
236.  R[06] ← R[06] & R[02]
238.  R[01] ← R[01] & R[03]
239.  R[06] ← R[06] | R[04]
240.  R[02] ← ~ R[06]
241.  R[03] ← R[03] | R[02]
242.  R[03] ← R[03] | D[05]
243.  R[03] ← R[03] & D[04]
244.  R[01] ← R[01] | D[02]
247.  R[01] ← R[01] & R[03]
248.  R[03] ← mov R[02]
250.  R[06] ← ~ D[04]
251.  R[02] ← R[02] | R[01]
252.  R[02] ← R[02] & R[04]
253.  R[01] ← R[01] | R[03]
255.  R[01] ← R[01] & D[11]
256.  R[04] ← ~ R[03]
257.  R[02] ← R[02] | R[01]
258.  R[03] ← R[03] & D[04]
259.  R[01] ← R[01] | D[02]
260.  R[06] ← R[06] & R[02]
262.  R[01] ← R[01] & R[03]
263.  R[03] ← mov R[02]
264.  R[06] ← R[06] | R[04]
265.  R[02] ← R[02] & R[04]
268.  R[01] ← R[01] & D[10]
270.  R[02] ← R[02] | R[01]
273.  R[03] ← R[03] & R[06]
274.  R[03] ← R[03] | R[02]
290.  R[02] ← mov D[02]
293.  R[02] ← R[02] | R[06]
295.  R[04] ← mov D[11]
296.  R[01] ← ~ R[02]
297.  R[04] ← R[04] | D[01]
299.  R[01] ← R[01] | R[03]
301.  R[06] ← R[06] | R[04]
304.  R[04] ← mov R[06]
305.  R[02] ← ~ R[06]
306.  R[04] ← R[04] & D[02]
309.  R[03] ← mov R[02]
311.  R[02] ← R[02] & D[03]
312.  R[06] ← ~ D[04]
313.  R[02] ← R[02] | R[01]
314.  R[01] ← R[01] & D[10]
316.  R[02] ← R[02] | R[01]
317.  R[03] ← R[03] | R[02]
318.  R[06] ← R[06] | R[04]
322.  R[03] ← R[03] & R[06]
324.  R[06] ← ~ D[04]
325.  R[06] ← R[06] | D[01]
328.  R[03] ← R[03] & R[06]
329.  R[03] ← R[03] | R[02]
330.  R[06] ← R[06] | R[04]
334.  R[03] ← R[03] & R[06]
336.  R[02] ← ~ R[06]
348.  R[03] ← R[03] | R[02]
376.  R[01] ← ~ D[04]
381.  R[01] ← R[01] | D[01]
383.  R[01] ← R[01] | D[07]
384.  R[01] ← R[01] | D[02]
387.  R[01] ← R[01] & R[03]
```

A Representative Champion of the 8-to-1 Multiplexor Experiment

Figure 29: Disassembly of 8-to-1 multiplexor champion

$$(((((((D_1 \vee D_2) \vee D_7) \vee \neg D_4) \wedge (((D_1 \vee D_5) \vee \neg D_2) \vee \neg D_4)) \wedge (((D_{11} \vee D_4) \vee \neg D_1) \vee \neg D_2)) \wedge (((D_{10} \vee \neg D_1) \vee \neg D_2) \vee \neg D_4)) \wedge (((D_2 \vee \neg D_1) \vee D_4 \wedge D_8) \vee D_9 \wedge \neg D_4)) \wedge ((((D_2 \vee D_5) \vee \neg D_1) \vee D_4 \wedge D_8) \vee D_9 \wedge \neg D_4)) \wedge ((((D_4 \vee D_1 \wedge D_2) \vee D_1 \wedge D_9) \vee D_2 \wedge D_6) \vee (D_3 \wedge \neg D_1) \wedge \neg D_2)$$

Figure 30: 8-to-1 multiplexor champion as a symbolic expression

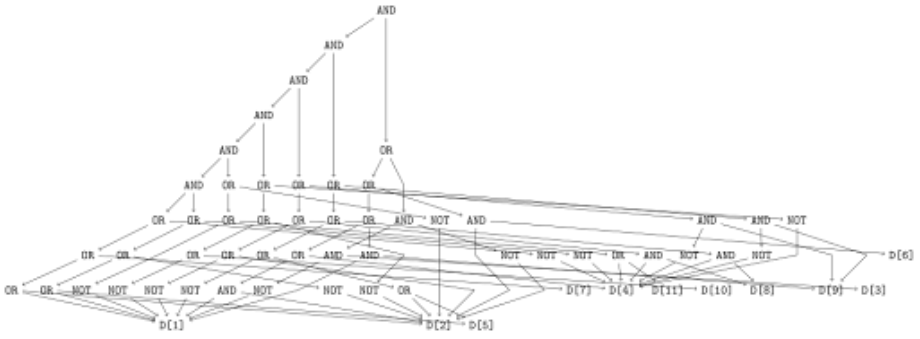


Figure 31: Syntax graph of a 8-to-1 multiplexor champion

recent experiments we have been able to discover the target function in approximately half the number of tournaments that were generally necessary in populations unaided by informational crossover weighting.

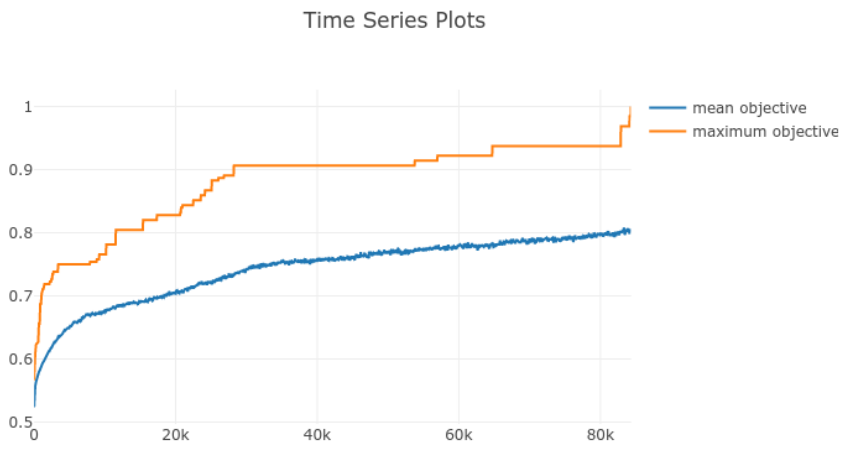


Figure 32: 8-to-1 MUX experiment with informational crossover point weighting

Next Steps: System Integration

The primary goal of Phase 2 will be to integrate these three modules into a cohesive system. The Refuduino PLC whisperer will allow us to rapidly probe a black boxed PLC device with crafted inputs and record its behaviour. The probabilistic property tester (PPT) will determine *which* inputs can provide us with the best evidence for various semantic properties. Those properties will then be translated into constraints that will be used to guide the Cockatrice genetic search, while the PPT aids in the selection of input samples for functions too complex for their truth tables to be exhaustively enumerated.

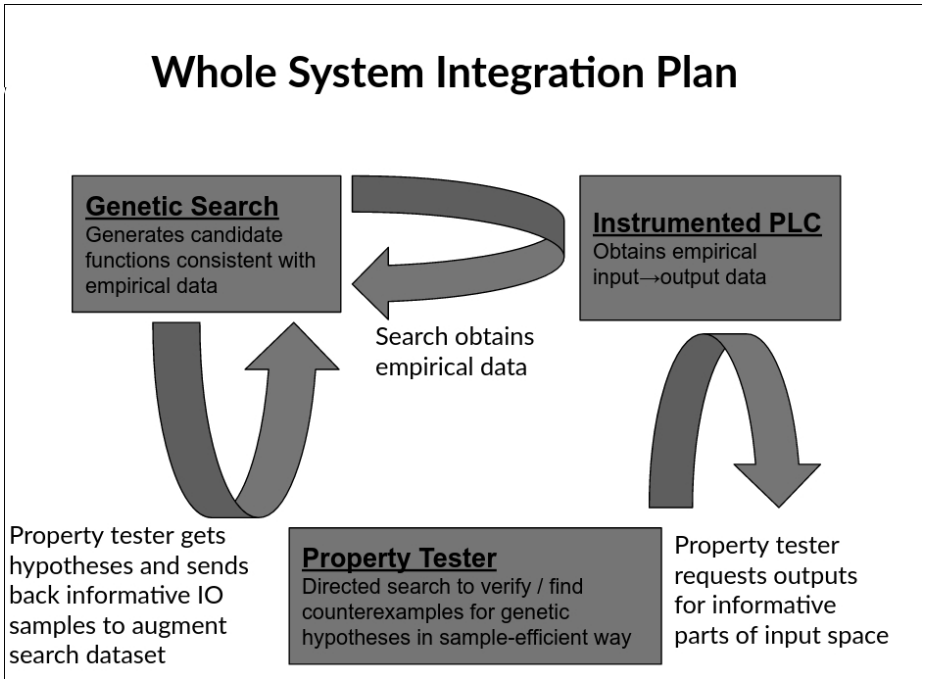


Figure 33: Integration plan