Self-Supervised Learning to Prove Equivalence Between Programs via Semantics-Preserving Rewrite Rules

Steve Kommrusch, Martin Monperrus, and Louis-Noël Pouchet

Abstract—We target the problem of automatically synthesizing proofs of semantic equivalence between two programs made of sequences of statements. We represent programs using abstract syntax trees (AST), where a given set of semantics-preserving rewrite rules can be applied on a specific AST pattern to generate a transformed and semantically equivalent program. In our system, two programs are equivalent if there exists a sequence of application of these rewrite rules that leads to rewriting one program into the other. We propose a neural network architecture based on a transformer model to generate proofs of equivalence between program pairs. The system outputs a sequence of rewrites, and the validity of the sequence is simply checked by verifying it can be applied. If no valid sequence is produced by the neural network, the system reports the programs as non-equivalent, ensuring by design no programs may be incorrectly reported as equivalent. Our system is fully implemented for a given grammar. To efficiently train the system to generate such sequences, we develop an original incremental training technique, named self-supervised sample selection. We extensively study the effectiveness of this novel training approach on proofs of increasing complexity and length. Our system, S4Eq, achieves 97% proof success on a curated dataset of 10,000 pairs of equivalent programs.

Index Terms—program equivalence, symbolic reasoning, self-supervised learning, machine learning.

1 Introduction

Deep neural networks have excelled at a variety of classification, reinforcement learning, and sequence generation tasks [1]. However, their stochastic nature complicates the use of such networks in formal settings where one requires a *guarantee* that the result produced is provably correct, such as to assess semantic equivalence between programs.

In this work proving program equivalence means determining whether two (symbolic) programs always produce identical output if given the same input, for all possible inputs. Program equivalence is a central problem in computing [2]–[4]. The problem ranges from undecidable [5], to trivial in the case of testing the equivalence of a program with itself. Proving program equivalence is useful for e.g. verifying compiler correctness [6], replacing code fragments by more optimized ones [7], malicious software detection [8] or automated student feedback [9]. In this work, we propose a machine learning framework for proving program equivalence, named S4Eq.

S4Eq takes as input two programs and generates a sequence of rewrite rules under a well-defined system for equivalence using semantics-preserving rewrite rules [10]. Our work studies programs represented as a list of statements with straight-line control-flow, using multiple variable types and complex mathematical expressions to compute values. S4Eq outputs a sequence of rewrite rules which is then easily checked for validity. Therefore, only valid sequences are outputted and the system guarantees no false positives by design (no programs are stated equivalent if they are not).

We target a challenging instance of the program equivalence problem where the set of rewrite rules we consider may fundamentally change the number and order of operations in the program, such as factorization/distribution or common sub-expression elimination. We design a novel self-supervised learning technique, exploiting the ability of our system to automatically synthesize valid new programs and proofs. We initially train a model in a supervised manner with synthetic data which has a broad distribution on the use of rewrite rules. Then we propose a self-supervised technique based on comparing results between broad and narrow proof searches to incrementally train our model. Rewrite rule sequences demonstrating equivalence found by a quick, narrow search are not considered interesting for further training; while sequences found by a broad search indicate samples for which the model's rewrite rule selections could be improved. We name this procedure *self-supervised sample selection*. We fully implement our learning and inference models in the popular OpenNMT-py framework [11], based on the transformer model.

To demonstrate the applicability of S4Eq on human-written programs, we mined C functions on the popular code hosting platform GitHub [12]. We collected 13,215 unique programs that we analyze for equivalence by S4Eq. This shows that S4Eq can prove program equivalence using various compilation and optimization transformations, such as Common Sub-expression Elimination [13] that exist on AST samples in the field. To summarize, we make the

- *Steve Kommrusch and Louis-Noël Pouchet are with Colorado State University, USA.*
  E-mail: {steveko, pouchet}@cs.colostate.edu
- *Martin Monperrus is with KTH Royal Institute of Technology, Sweden.*
  E-mail: monperrus@kth.se

Manuscript submitted 2021-09-21.

Prog A (source code): (a)

```
y_diff = ( particles [ i ] . y_pos - particles [ j ] . y_pos ) ;
r = sqrt ( ( x_diff * x_diff ) + ( y_diff * y_diff ) ) ;
mass = particles [ j ] . mass ;
mass /= ( ( r + EPSILON ) * ( r + EPSILON )
                          * ( r + EPSILON ) ) ;
x_diff *= mass ;
y_diff *= mass ;
sumX += x_diff ;
sumY += y_diff ;
```

Prog A (abstracted): (b)

```
t1 = ( i1 - i2 ) ;
t2 = f1 ( ( i3 * i3 ) + ( t1 * t1 ) ) ;
t3 = i4 ;
t4 = t3 / ( ( t2 + i5 )
      * ( t2 + i5 ) * ( t2 + i5 ) ) ;
t5 = i3 * t4 ;
t6 = t1 * t4 ;
o1 = i6 + t5 ;
o2 = i7 + t6 ;
```

Prog A (prefix encoding of AST): (c)

```
s28 = ( -s s01 s02 ) ;
s27 = ( u1s ( +s ( *s s03 s03 ) ( *s s28 s28 ) ) ) ;
s26 = s04 ;
s25 = ( /s s26 ( *s ( +s s27 s05 )
            ( *s ( +s s27 s05 ) ( +s s27 s05 ) ) ) ) ;
s24 = ( *s s03 s25 ) ;
s23 = ( *s s28 s25 ) ;
s30 === ( +s s06 s24 ) ;
s29 === ( +s s07 s23 ) ;
```

Equivalence Rewrite Rule Sequence: (g)

```
stm4 MultOne Nr
stm4 Inline s26          stm3 Rename s26
stm3 DeleteStm           stm4 Rename s25
                         stm5 Rename s24
                         stm4 Commute N
                         stm3 NeutralOp Nr
                         stm3 DivOne Nr
                         stm3 FlipRight N
                         stm5 Commute N
```

Prog B (source code): (d)

```
distancey = y - src -> center_y ;
rij = sqrt ( distancex * distancex + distancey * distancey ) ;
cst_j = src -> center_mass * 1.0 / ( ( rij + E0 ) * ( rij + E0 )
                                   * ( rij + E0 ) ) ;
cord_x = cst_j * distancex ;
cord_y = cst_j * distancey ;
Fx += cord_x ;
Fy += cord_y ;
```

Prog B (abstracted): (e)

```
t1 = i1 - i2 ;
t2 = f1 ( i3 * i3 + t1 * t1 ) ;
t3 = i4 * 1s / ( ( t2 + i5 )
      * ( t2 + i5 ) * ( t2 + i5 ) ) ;
t4 = t3 * i3 ;
t5 = t3 * t1 ;
o1 = i6 + t4 ;
o2 = i7 + t5 ;
```

Prog B (prefix encoding of AST): (f)

```
s28 = ( -s s01 s02 ) ;
s27 = ( u1s ( +s ( *s s03 s03 ) ( *s s28 s28 ) ) ) ;
s26 = ( *s s04 ( /s 1s ( *s ( +s s27 s05 )
            ( *s ( +s s27 s05 ) ( +s s27 s05 ) ) ) ) ) ;
s25 = ( *s s26 s03 ) ;
s24 = ( *s s26 s28 ) ;
s30 === ( +s s06 s25 ) ;
s29 === ( +s s07 s24 ) ;
```

ProgInt after "stm3 Deletestm": (h)

```
s28 = ( -s s01 s02 ) ;
s27 = ( u1s ( +s ( *s s03 s03 ) ( *s s28 s28 ) ) ) ;
s25 = ( /s s04 ( *s 1s ( *s ( +s s27 s05 )
            ( *s ( +s s27 s05 ) ( +s s27 s05 ) ) ) ) ) ;
s24 = ( *s s03 s25 ) ;
s23 = ( *s s28 s25 ) ;
s30 === ( +s s06 s24 ) ;
s29 === ( +s s07 s23 ) ;
```

Fig. 1: Exemple of various program representations, and application of rewrite rules, in S4Eq.

following contributions:

- We present S4Eq, an end-to-end deep learning framework to find equivalence proofs between two complex program blocks. S4Eq produces a sequence of semantics-preserving rewrite rules to transform one program into the other, via successive rewrites. We consider rewrites which support complex program transformations such as Common Subexpression Elimination and computational strength reduction. S4Eq emits a *verified* sequence of rewrites, leading to no false positive by design.
- We design self-supervised sample selection, an original training technique tailored to our problem domain. This approach further improves the ability of the deep learning system to find more complex proofs.
- We present extensive experimental results to validate our approach, demonstrating our system can successfully prove equivalence on both synthetic programs and programs derived from GitHub with up to 97% success. This paves the way for unsupervised deployment to validate correctness of transformations of program blocks (typical in e.g. low-level compiler optimizations [14]), that are typically unsupported by verification tools [15].
- We provide all our datasets to the community including synthetic generation techniques for the problem of program equivalence via rewrite rules, as well as the programs built from the ASTs we mined from C functions in GitHub [16].

2 Problem Statement

We now give the basics of a rewrite-rule based system to prove equivalence between program blocks, the language and rewrites we support, and we finally present theoretical considerations and practical uses of our system.

2.1 Straight-line Programs

We target programs made of statements without explicit control-flow instructions (that is, straight-line code [8], [17]). Such programs may be the result of applying full unrolling or flattening on loops ([18], [19]), or may simply occur in human-written C code as exemplified in Fig. 1(a) and Fig. 1(d).

Our system takes as input programs represented as abstract syntax trees, or ASTs, using specific symbols for input and output values, e.g., `i1,i2,i3` and `o1,o2` in Fig. 1(b) and specific typed operator symbols, e.g. `-s` in Fig. 1(c) to represent subtraction of scalars. We require a program to behave as a pure function (no side effect) with a single-entry and a single-exit (SESE), where the set of input and output variable names is known, each distinct name representing a different variable (no aliasing). Under those assumptions, our system targets proving the equivalence of two programs, represented with their ASTs. The process of abstracting some source code to a safe AST-based representation for equivalence checking is outside the scope of our paper, we assume this process is done beforehand.

For example, Fig. 1(a) and (d) are two actual source code snippets mined from GitHub. The two straight-line programs we generated from them in Fig. 1(b) and (e) do *not* safely capture their exact semantics: live-in variables of different names, e.g. `xdiff` and `distancex`, are both translated to the same `i3` symbol in the AST we extracted. In a formal deployment setting, one shall ensure no aliasing between symbols that are non-local to the programs (e.g., `xdiff` vs. `distancex`), and compute correspondence between live-in/live-out symbols of each program [20]. This is simplified in cases where e.g. program B is an optimized version of program A, and meant to fully replace A in some larger programs: then by design A and B would always be called with the exact same context. It reduces the entire aliasing and matching problem to simply syntactic name matching.

Our input language covers programs made of multiple symbolic expressions using variables of different types, operators, pure function calls, and neutral or absorbing elements (*e.g.*, $0, 1$). We support both "vector" and "scalar" types, as well as operators and functions that mix these types. We support programs with single or multiple outputs of varying types.

2.2 Transforming Programs with Rewrite Rules

We now outline how program transformations are represented in our system. Figure 1(c) and (f) display our actual input program representations, which is strictly equivalent to the programs in Fig. 1(b) and (e). We use a prefix encoding of the AST to feed to our deep learning system presented in later Sec. 3. The rewrite rules we use operate on this representation. The parenthesis positioning in this encoding allows for direct recognition of subtrees, and nodes of this tree can be referenced to in the rewrite rules. For example,

TABLE 1: The 23 rewrite rule categories considered by S4Eq. Specializing to different data types supported, 86 rewrite rules are actually considered.

Rewrite Rule and Arguments	Example or Description	Rewrite Rule and Arguments	Example or Description
SwapPrev	Swap assign statements	Inline VarID	Replace VarID with its last assigned expression
UseVar VarID	Replace expression with VarID	NewTmp NodeID VarID	Assign VarID to expression at NodeID
DeleteStm	Delete assign stm	Rename VarID	Change assignment to VarID
AddZero NodeID	$\vec{v} \rightarrow (\vec{0} + \vec{v})$, b→0+b	SubZero NodeID	$\vec{v} \rightarrow (\vec{v} - \vec{0})$, b→b-0
MultOne NodeID	$\vec{v} \rightarrow (1 \times \vec{v})$, b→1×b	DivOne NodeID	a → a/1
Cancel NodeID	$(\vec{v} - \vec{v}) \rightarrow \vec{0}$,(b/b)→1	NeutralOp NodeID	$(\vec{v} - \vec{o}) \rightarrow \vec{v}$, 1×a→a
DoubleOp NodeID	$-(-\vec{v}) \rightarrow \vec{v}$, 1/1/x→x	AbsorbOp NodeID	$(\vec{v} \times 0) \rightarrow \vec{0}$, (b×0)→0
Commute NodeID	(a + b) → (b + a)	DistributeLeft NodeID	(a + b)c → ac + bc
DistributeRight NodeID	a(b + c) → ab + ac	FactorLeft NodeID	ab + ac → a(b+c)
FactorRight NodeID	ac + bc → (a+b)c	AssociativeLeft NodeID	a(bc) → (ab)c
AssociativeRight NodeID	(ab)c → a(bc), (ab)/c→a(b/c)	FlipLeft NodeID	$-(\vec{v} - \vec{w}) \rightarrow \vec{w} - \vec{v}$
FlipRight NodeID	a/(b/c) → a(c/b)		

ProgA in Fig. 1(c) is transformed into ProgB in (f) with the 11 step rewrite rule sequence shown in (g). Our general rewrite rule syntax in this paper is:

```
stm# RuleName [NodeID] [VarID]
```

where `stm#` is the statement number in ProgA which should have `RuleName` applied. `NodeID` optionally identifies the node within the right hand side of the assignment statement, and `VarID` is the optional name of the variable to use for applying the rule. Precisely, `NodeID` is a description of the path from the root of the statement to the node of interest, e.g. `Nr`, which is our syntax to model the right child (r) of the root expression node (N). Making the path explicit facilitates learning representations independent of the program size.

For S4Eq we have manually specified the rewrite rules summarized in Table 1. The 23 different rule groups displayed are specialized for each supported data type, leading to 86 distinct rules. Note we specifically focus on rewrites that go beyond rescheduling operations: we consider rewrites that alter the count and type of operations, such as factorization/distribution, as well as all rules needed to implement symbolic common sub-expression elimination (CSE) on the whole program via a composition of more basic rewrites.

The system of rewrites is how equivalence has been proven in prior work [21] and we use an identical formalism in this paper. A rewrite rule is made of a tuple <P, R> where P is a pattern, or subtree template, dictating when the rule can be matched on a given subtree of the original AST. R is a rewrite, another subtree template, with which the P subtree is replaced when a successful match occured. To test the validity of applying a rule to a node or subtree in the original AST, we simply need to check the pattern P exists at the expected position in the subtree, if so replacing it by R is valid.

We now illustrate with an example. Taking the fifth assignment in Fig. 1(c), `s24 = (*s s03 s25)`. We focus on the expression `(*s s03 s25)`. The rule `Commute NodeID` from Table 1 is specialized for all applicable operators and types, in particular the rule $a * b = b * a$ for scalars is defined using <P, R> with P =`(*s <subtree-1> <subtree-2>)` and R =`(*s <subtree-2> <subtree-1>). Stm5 Commute N` is a match: P =`(*s <subtree-1> <subtree-2>)` matches `(*s s03 s25)` with `<subtree-1> = s03` and `<subtree-2> = s25`, the rewritten statement would be `(*s s25 s03)`. `Stm5 Commute Nr` is not a match, and therefore an invalid rewrite: the path Nr indicates `s03`, which is not matched by P. Similarly, `Stm1 Commute N` is not a match, `-s`, the subtraction of scalars, differs from `*s`, the multiplication of scalars. `Stm2 Commute Nrr` is a match: P is found on `(*s s28 s28)`.

2.3 Checking the Validity of a Sequence of Rewrites

Our definition of rewrite rules as <P, R> makes trivial the process of checking the validity of a sequence S of rewrites on program $Prog$, and is as follows. For each rewrite S_i in S, in increasing order of i, do: (a) check P_{S_i} is matched at the node specified in S_i. If no, fail: the sequence is invalid. If yes, (b) apply the rewrite $P_{S_i} \rightarrow R_{S_i}$ at the node specified in $Prog$ to generate a new program $Prog'$. (c) Set $Prog = Prog'$. Note this algorithm has essentially a low polynomial worst-case time complexity (pattern size times program size).

It follows our simple procedure to determine equivalence between two programs $P1, P2$ and a candidate sequence S to rewrite $P1$ into $P2$. If the sequence is valid as per the procedure above, and after all rewrites applied on $P1$ the resulting program is *syntactically* identical to $P2$, then the two programs are equivalent. Additional proofs can be found in [21].

2.4 S4Eq in a Nutshell

Our system S4Eq operates as follows. For inference (testing whether two programs are equivalent), a pair of programs $P1, P2$ in prefix AST form is input to the system. A maximal length l for the rewrite sequence is set. For no more than l steps, the system (a) proposes a rewrite to be applied; (b) its validity is verified, and if valid, the rewrite is applied to $P1$, generating a new $P1'$ program, which becomes the $P1$ program tested for equivalence with $P2$ at the next step. If at any point $P1$ is syntactically identical to $P2$, the system outputs the full sequence and $P1, P2$ as equivalent. In all other cases, the system outputs $P1, P2$ as non-equivalent.

As extensively developed in Sec. 4.1 and later, for training, we take as input a grammar for the supported language, and the rewrite rules as defined in Table 1. We fully automatically generate new programs and proofs by simply iterating on the grammar productions to generate programs, and iterating on applicable rewrites on a program to generate a pair of equivalent programs and one proof of equivalence for them.

2.5 Problem Complexity

Given a pair of programs that fit our representation, determining the existence (or absence) of a sequence of valid rewrites to rewrite one program into another is NP-complete [22]. Indeed, while some rewrites could be easily handled with a kind of greedy application scheme, such as multOne or Cancel, others immediately create combinatorial explosion, such as Commute, and triggers the NP complexity. Intuitively, to solve equivalence between a sequence of commutative/distributive operators in polynomial time, some sort of canonical representation should exist, and it should not be more than polynomially bigger in size than the original expression. Then one would first transform to this canonical representation, and then perform a polynomial-time check between the canonical representations of both programs. While in very specific instances this representation may exist [23], our supported language goes beyond these restricted scenarios and supports arbitrary expressions made of scalars, vectors, etc. for which, to the best of our knowledge, no canonical representation has been proposed. By analogy to the Boolean Satisfiability problem, converting to a conjunctive normal form could already lead to an exponentially larger representation, leading to a worst-case NP complexity [22].

Intuitively, we can view the program equivalence solution space PES as a very large graph, where every possible syntactically different program in the language is represented by its own vertex v. Then, two vertices v_i and v_j are connected by a labeled edge iff applying one particular rewrite rule on a particular node of v_i is valid, and leads to producing the program v_j. The edge is labeled by the rewrite rule applied (as defined above). This graph is a multigraph, as multiple different rewrites may connect the same two programs. It also contains cycles, as a sequence of rewrites can "undo" changes. Therefore, any two programs connected by a path in this graph are semantically equivalent: the rewrite sequence is the set of labels along the edges forming the path. Building the rewrite rule sequence S for $ProgB \equiv S(ProgA)$ amounts to exposing one path (out of possibly many) from $ProgA$ to $ProgB$ in this graph when it exists, the path forming the proof of equivalence. In this work we build a deep learning system to learn a stochastic approximation of an iterative algorithm to construct such feasible path when possible. In a nutshell, we view program equivalence as a pathfinding problem in PES, and train a neural model to find efficiently paths in PES, only by sampling from PES at training time. Our approach avoids entirely the need to craft smart exploration heuristics for such large equivalence graph to make this path-finding problem practical (akin to building tactics in theorem provers): instead, the traversal heuristic is learned automatically, without any user input, by deep learning.

As detailed in later Sec. 3, one fundamental contribution of our work is to design a self-supervised incremental training approach to address this pathfinding problem. It is motivated by the fact that the *in-practice* complexity of proving equivalence may vastly differ between two problem instances. Our system identifies instances that were not solved well, builds a family of new problem instances derived from it, and incrementally trains on those to improve its abilities on complex cases. Implicitly, instances with high in-practice complexity that are not solved well will be isolated and used for refining the system's capabilities.

2.6 Applications of Equivalence Proof Search

Our system proves equivalence between symbolic straight-line programs. We have exemplified on a two-type language with a standard operator mix, up to token renaming, which covers instances of SIMDized functions, register-level code, etc. As such, S4Eq can be used to address the verification and certification of certain compiler optimizations [24]. While numerous approaches based on abstract interpretation address equivalence under *reordering* of operations, e.g. [4], [25], [26], they typically cannot handle transformations at the statement level. S4Eq can take blocks in two programs and test them for equivalence, (re)discovering which statement corresponds to which in the transformed program, helping to address the well-known statement matching problem. On the contrary, S4Eq can prove equivalent after a full-unrolling of a code region, as is typical in high-performance code generation [27], a fundamental problem in compiler certification.

Another use case is the equivalence between linear algebra formulas, as could typically be found in student exercises. S4Eq is able solve all of the matrix expression equivalence programs from 2 relevant Khan academy modules designed to test student's knowledge of matrix algebra [28].

3 S4Eq: Deep Learning to Find Rewrite Rule Sequences

We propose to use a deep learning model to find rewrite rule sequences which transform one program into a semantically equivalent target program. The idea is to first learn from correct rewrite sequences, and then learn to solve previously unseen program equivalence problems.

3.1 Overview of S4Eq

Prior work has shown that source code has patterns that are similar to human language [29], and thus techniques used in natural language processing can work on source code as well, including deep learning [30]. Deep learning has the ability to learn syntactic structure and expected outputs related to programs. For S4Eq we aim to create a deep learning model which, given 2 programs, will predict a sequence of rewrite rules which can formally prove equivalence between the 2 programs.

For S4Eq we use the state of the art sequence-to-sequence deep learning model known as the transformer model [31]. Because sequence-to-sequence models are stochastic, they can be used to produce multiple answers

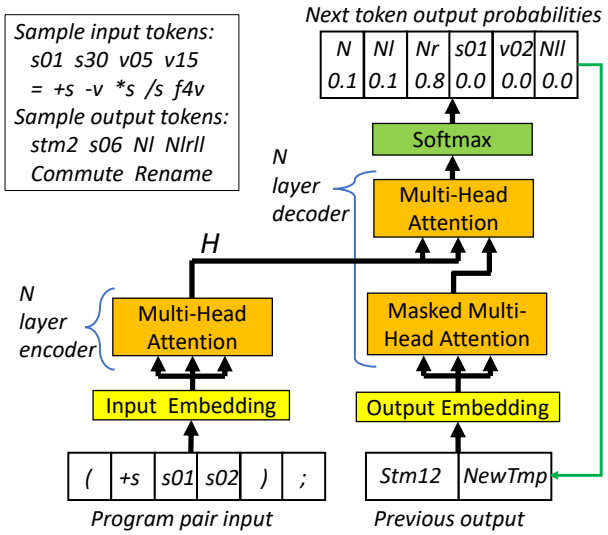

Fig. 2: Transformer model used to predict rewrite rules given 2 input programs.

for the same query; this is called *beam search*. By using beam search we can order rewrite rules proposed by the model. By composing beam search recursively, we can construct proofs which use heuristics learned by the model to guide proof search.

In S4Eq, we devise an original training process specifically for the task of proving equivalence. The S4Eq training process combines supervised and self-supervised learning and does not require human labeling.

3.2 Transformer Model

The transformer sequence-to-sequence model is meant to learn mappings between two sequences, typically of words [31]. It is widely used in automated translation [32] and text summarization [33]. Sequence-to-sequence models consists of two parts, an encoder and a decoder. The encoder maps the input sequence $X = (x_0, x_1, ..., x_n)$ to an intermediate continuous representation $H = (h_0, h_1, ..., h_n)$, also known as an embedding. Then, given H, the decoder generates the output sequence $Z = (z_0, z_1, ..., z_m)$. The size of the input and output sequences, n and m, can be different; and the languages for X and Z can also use different vocabularies. A sequence-to-sequence model is optimized on a training dataset to maximize the conditional probability of $p(Z \mid X)$, which is equivalent to:

$$\begin{aligned} p(Z \mid X) &= p(z_0, z_1, ..., z_m \mid x_0, x_1, ..., x_n) \\ &= \prod_{i=0}^{m} p(z_i \mid H, z_0, z_1, ..., z_{i-1}) \end{aligned}$$

For S4Eq, we introduce the input and output languages for X and Z in Section 2. Subfigures 1(c) and 1(f) are examples of such inputs. The output language for Z is illustrated in subfigure 1(g). For input to the transformer, we add a special token `Y` to separate the 2 programs, with ProgA being input before ProgB. The input and output languages are fully detailed in our GitHub repository [16].

In S4Eq, the transformer model we use is shown in Figure 2. The yellow boxes represent the model's learned interpretation for the tokens in the input and output. Tokens such as '*s' and '=' in the input language or 'stm3' and 'Commute' in the output language have learned embeddings used by the transformer model. The model accomplishes context-dependent interpretation with multiple attention layers which learn a complex representation for a program node by learning which other nodes should be "attended to" while creating the higher level representation.

In the transformer model, the encoder and decoder use layers which provide an attention mechanism which relates different positions of a single sequence in order to compute a representation of the entire sequence. For example, the attention layers provide a mechanism for information related to the assignment of a variable to affect the representation of other parts of the program where the variable is used.

Multi-head attention allows for the different heads to learn different aspects of the data. For our problem of program equivalence, the model is trained to produce correct rewrite rules and hence all of the learnable functions are learning representations useful for this task. To illustrate, one of the heads may tend to build a representation for addition and subtraction of vectors, while another head might build a representation for multiplication and division of scalars. What the heads learn is not constrained by the architecture, so the true meaning of the heads at each layer is not easily decipherable, but multi-head attention has been shown by others and by our own ablation studies to be valuable.

Figure 2 identifies 'N layers' for both the encoder on the left (which encodes the input programs to an intermediate representation) and the decoder on the right (which decodes the representation to produce the rewrite rule output). The 'N layer encoder' is using self-attention in which the information processed by a given layer is provided by the layer below. A similar situation holds for the 'N layer decoder', however the H connection from the encoder layers to the decoder layers allows the decoder to process information from the decoder and encoder. The transformer model we employ also includes residual and feed-forward sublayers and a full description of the interactions within the model can be read in the work by Vaswani *et al.* [34]. We used the Adam optimizer [35] to adjust the weights in the network based on the loss function between the target token expected in the training sample and the result produced by the current network.

The intermediate representation H encodes the complex representations for each token of the input, in our case a pair of programs. The decoder model will generate a first output token based on H and then generate subsequent tokens based on H and the previously output tokens. The Softmax function normalizes the output of the final decoder layer to a probability distribution:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \texttt{for} \quad \mathbf{z} = (z_1, ..., z_K) \in \mathbb{R}^K$$

The effect of the Softmax layer is to create an output that can be interpreted as representing the probability that a given token is correct, given the training the model has been exposed to. As the output is generated, when the tokens with the highest Softmax values are selected we create a rewrite rule which represents the most likely next edge in our path through the program space from ProgA to ProgB.

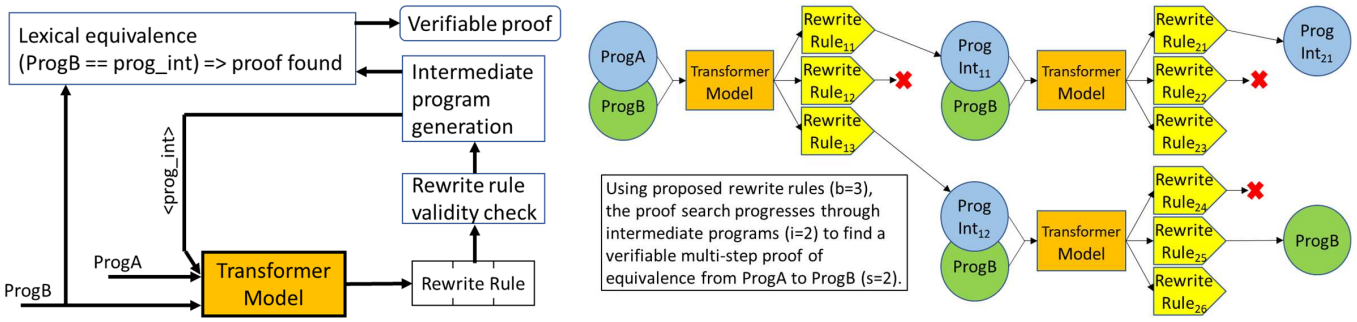

Fig. 3: Proof search: Transformer model produces multiple rewrite rules at each step. Invalid rules are discarded, lexical equivalence is checked, and up to i intermediate programs are used to produce new rules for the next search step.

3.3 Beam Search to Explore Multiple Possible Proofs

A typical approach when using sequence-to-sequence models is called "beam search", it is the process of asking the deep learning model to produce multiple answers for the same question. As the network produces likely token selections, the Softmax layer shown in Figure 2 is effectively producing probabilities for each token. Using these probabilities we can produce multi-token outputs sorted in order based on the total probability the model assigns to the sequences.

In S4Eq, we use this beam search to enumerate the possible next rewrite rule to apply. Each proposal is then checked for legality (whether the proposed rewrite rule can indeed be applied at the given program location) and novelty (the program resulting from the rule application does not match a previously seen program for this search). Figure 3 diagrams the system which receives ProgA and ProgB and generates a verifiable proof of equivalence. A rewrite rule includes the statement number, rule name, possible statement node identifiers, and possible variable identifiers. The validity of a rule is checked by first verifying the statement number exists in ProgA. Then, if the rule does not require a node ID (SwapPrev or DeleteStm) we check if the liveness rules are followed - for SwapPrev neither statement may make use of the variable assigned by the other statement, and for DeleteStm the variable must not be an output variable and must not be used in subsequent statements. For rules requiring a NodeID the operators or variables are checked for legality of the operation being applied along with the possible variable ID produced for use by the rule.

In addition to the rewrite rule beam, S4Eq uses a second type of beam during the proof search. The idea is to feed the network again based on the result of the application of the previously suggested rewrite rules. We denote the number of enumerated rewrite rules b. As the search advances, the b outputs from the neural network all lead to potential intermediate programs from which a search can continue. After having checked for legality, we limit this potential exponential growth in the search to at most configurable i intermediate programs which may be explored at a given proof step.

Consider a search where we set b to 3 and i to 2, as diagrammed in Figure 3. When a transformation search between 2 programs is attempted, at first there is only 1 sample (the original 2 programs to prove equivalent) fed into the transformer model which will propose 3 rewrite rules. Perhaps the first 2 are legal rewrite rules; both of these are checked for equivalence to the ProgB goal and assuming there is no match both will be fed into the transformer model on the next step. This will produce 6 proposed rewrite rules. If a rewrite rule would create an intermediate program that is already being searched (for example commuting the same node twice in a row) then the search process will not create the duplicate intermediate program. In the figure, rewrite rules which are illegal or create duplicate search programs are marked with a red X. The search routine will select the most likely proposed rule for each ProgInt/ProgB pair if it legally creates a novel intermediate program.

As diagrammed, the ProgInt_{11}/ProgB pair produces a legal novel program which is used for the next step, but the ProgInt_{12}/ProgB pair's 1st proposal is not usable. Since the 2nd proposal from the ProgInt_{11}/ProgB pair is also not usable, the legal 2nd proposal from the ProgInt_{12}/ProgB pair is used. Our search will first try to use the 1st proposed rule from each ProgInt/ProgB pair, then the 2nd, and so on until the next i intermediate programs are created. We will limit the intermediate programs that feed into the transformer to i as the search is continued up to the rewrite rule sequence step limit l (such as 25 steps). In rare cases, none of the proposed rewrite rules for any of the intermediate programs will produce a legal novel intermediate program and the search will terminate before the step limit. In our example, the 2nd rewrite rule proposed from the model when given ProgInt_{12}/ProgB to the transformer rewrites ProgInt_{12} into the lexical equivalent of ProgB, and hence, a 2 step proof has been found. ProgA is transformed into ProgB by applying Rewrite Rule_{13} and then Rewrite Rule_{21}.

3.4 Naming Conventions Used

Throughout this paper we make reference to various data sets and and configuration parameters. As an aid to understanding our system and experiments, we summarize them in Table 2 with a short description. Our *Optim* datasets are abstractions from GitHub source C code as described in Section 4.1.1.

3.5 Training Process

A key novelty of S4Eq lies is the way we train the neural network. We devise an original training process which is dedicated to the challenges of synthesizing equivalence proofs. This training process involves three kinds of training which we now present.

TABLE 2: Variable and dataset names

Variable or set description	Name
Beam search width produced by neural network	b
Number of intermediate programs during search	i
Easy intermediate program search count	i_{easy}
Hard intermediate program search count	i_{hard}
Rewrite rule step limit for equivalence search	l
Initial model trained only with supervised data	M_1
Models trained with unsupervised data	M_{2-7}
Baseline model fully trained with supervised data	Q
Supervised equivalent program pairs (includes known rewrite rules between pairs)	S
Unsupervised equivalent program pairs	U
Synthetically generated program pairs (includes known rewrite rules between pairs)	$Synth^{+rules}$
Synthetically generated program pairs for validating models (no rules included)	$Synth_{val}$
Synthetically generated program pairs for training models (no rules included)	$Synth_{train}$
Synthetically generated program pairs for final testing of models (no rules included)	$Synth_{test}$
Code optimization pairs from human-written code for validating models (no rules included)	$Optim_{val}$
Code optimization pairs from human-written code for training models (no rules included)	$Optim_{train}$
Code optimization pairs from human-written code for final testing of models (no rules included)	$Optim_{test}$
Program pair abstractions from human-written code	$Human$
Initial training set derived from $Synth^{+rules}$	T_1
Incremental training set derived from proof attempts on $Synth_{train}$ and $Optim_{train}$	T_{2-7}

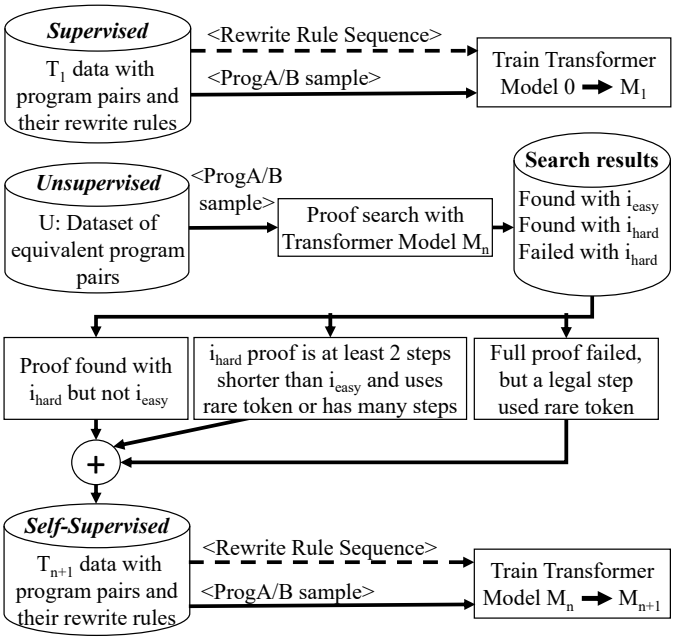

Fig. 4: Self-Supervised Sample Selection: proof attempts with $i = i_{easy}$ and i_{hard} intermediate programs are used to incrementally train the model.

3.5.1 Initial Supervised Training

The typical technique for training sequence-to-sequence models involves providing supervised training samples which provide the target output paired with sample input. S4Eq performs initial training using program pairs for which a rewrite rule sequence is known between the two programs ProgA and ProgB. The details of how to obtain such a supervised dataset are discussed in Section 4.1. We refer to this initial supervised dataset of program pairs and their rewrite rules as T_1, and the we refer to the initial model trained with it as M_1.

3.5.2 Incremental training with challenging proofs

After we have an initially trained model, we can use it to attempt new proofs. Because a proposed rewrite sequence between 2 programs can be checked automatically, we can automatically verify the generated outputs and use them to further optimize the model. At this point, to further optimize the model, we don't need to generate program pair inputs with rewrite rule outputs in a supervised manner, but only program pair inputs. In other words, this can be considered as self-supervision since the labeling is automated.

To be effective, the core challenge becomes to effectively select new samples. Hence, we term this technique *self-supervised sample selection*, and since our framework uses this technique on the problem of program equivalence, we name our framework S4Eq.

First, we need a dataset (or a data sample generator) of known equivalent programs which does not include the rewrite rule steps. We call this dataset U, for Unsupervised, as the pairs do not have target rewrite rules associated with them. Relative to the samples in T_1, the looser restrictions on the U dataset allows S4Eq to generalize to program distributions and proof distributions which may be different than those found in T_1.

Figure 4 gives a complete overview of our training process for the transformer models. As we show in Section 3.5.1, we start with supervised training and produce model M_1 using program pairs and their expected rules. Now we can use model M_1 to attempt proofs with known equivalent programs in dataset U.

Our key idea is to focus on challenging problems for the current model. Given an intermediate program limit, we can say that the model can 'easily' prove equivalence if a small number of intermediate programs are available at each search step (such as 1 or 2). In such a case, the most likely rewrite rules proposed by the model are checked for equivalence up to a given proof step limit, and if the proof is found, then the model is already well trained for the given problem. Our variable representing the number of intermediate programs searched at each proof step is i and the number of steps to search is l steps. To check for 'easy' proofs, we search for a proof with $i = i_{easy}$ where i_{easy} is a small number. To check for 'challenging' proofs, we set $i = i_{hard}$ where i_{hard} is a larger number (10 or more), allowing the proof search to explore more possible rewrite rule paths which are considered less likely to be correct by the current model. We attempt to prove each known-equivalent pair with $i = i_{easy}$ and also $i = i_{hard}$. We define challenging proofs as those found with i_{hard} but

not i_{easy}. When proofs are attempted with model M_n, the next training dataset (T_{n+1}) includes samples with proof steps found with i_{hard} but not i_{easy}; thus, the model is more likely to propose similar steps in the future.

In addition to including challenging proofs, if i_{hard} found a proof at least 2 steps shorter than the i_{easy} proof we include that proof given certain conditions. We set the probability of including such proofs based on length in order to bias the self-supervised samples to solve complex proofs. Also, based on distributions discussed in Section 4.1.2, we include the i_{hard} proofs when they include rare output tokens such as referring to higher statement numbers, or deeper expressions nodes, or rare rewrite rule names.

After creating our initial training samples in T_1, we train on various model hyperparameter options and use validation test sets to determine the model best suited for continued training. During incremental training, the model size parameters (number of layers, *etc.*) are constant but we train with variations on initial learning rate and decay rate. We then select the best model to continue training using the same validation sets. These validation sets help prevent catastrophic forgetting [36]. But additionally, if a model becomes weaker at solving certain problems then problems similar to those will get selected in the next iteration of the training, again reducing catastrophic forgetting.

Boosting methods in which models weak in one part of the problem distribution are boosted by other models have been shown to reduce risk of overfitting [37] and we anticipate that our methodology for incremental training based on challenging proofs will similarly resist overfitting.

3.5.3 *Incremental training with rare tokens*

If some input or output tokens are not yet well understood by a given model M_n, it is because the training datasets so far do not have sufficient samples demonstrating the use of those tokens. To overcome this problem, we propose another kind of incremental training based on rare tokens. The core idea is to oversample those proofs and rewrite rules that involve rare tokens. Even when a full proof search fails, when a rare output token is used legally by a single rewrite rule step in the proof, we keep it in the training dataset. This type of training improves the model representation for the rare token and the situations in which it should be applied and is based on the hindsight experience replay concept [38].

Consider again Figure 3 and a case where a required rewrite rule to prove ProgA equal to ProgB was, for example, `stm2 Commute Nlrll`. The node `Nlrll` is an example of a node ID which specifies the 5th level of the AST for statement 2. If there haven't been sufficient training samples with `Nlrll` then the model may not have a good internal representation for when the node should be produced by the final Softmax layer of the transformer model and the proof might fail. However, if, for example, $RewriteRule_{25}$ was `stm2 AssociativeLeft Nlrll` and this was a legal application of `AssociativeLeft` then the pair with $ProgInt_{12}$ and $ProgInt_{22}$ can be proven equal by applying $RewriteRule_{25}$. If `Nlrll` is a rare token, this sample can be included in the next training dataset and this will improve the model's representation for this rare token in order to improve use of this token after incremental training.

4 Experimentation

We now describe our experiments to assess S4Eq. We start by carefully devising two datasets for training and evaluating our system.

4.1 Dataset Generation

We devise two separate datasets with different properties. First, we wish to evaluate our algorithm broadly on code optimizations of human-written programs from open source C functions found on GitHub, we call this dataset $Optim$. Second, we develop a process to create synthetic program pairs based on applying rewrite rules, we call this dataset $Synth$.

4.1.1 *Equivalent program pairs from GitHub*

We want to have a dataset representative of developer code with straight-line programs matching our grammar. For this, we use an existing dataset of C programs mined from GitHub suitable for machine learning [39].

We process these C functions to find sequences of assign statements that correspond to our straight-line program grammar. We search for C snippets of mathematical computations, with at least 2 assignments, at least one multiply or divide statement, and require at least 1 temporary variable is used in an output variable. To create program abstractions (a process similar to function outlining [40]), we collapse complex data structure accesses into an intermediate variable. For example, C code of the form `delta = ca->tcp_cwnd - cwnd ; max_cnt = cwnd / delta ;` will be abstracted to `t1 = i1 - i2 ; o1 = i2 / t1 ;`. Two complete examples of abstracted programs are shown in subfigures 1(b) and 1(e).

Algorithm 1: GenerateKnownEqual

Input : Tokenized C Functions from GitHub: F
Output: Prefix encoded equivalent pairs created using code optimizations $Optim$

$Optim \leftarrow \emptyset$ {Compiler equivalence program pairs}
$S \leftarrow$FindSourcePrograms(F) {Possible programs}
foreach s *in* S **do**
  $\tilde{s} \leftarrow$Abstraction(s) {Abstractions of source code}
  $\tilde{s}_{cse} \leftarrow$CommonSubexpressionElimination($\tilde{s}$)
  $\tilde{s}_{str} \leftarrow$StrengthReduction($\tilde{s}_{cse}$)
  $\tilde{s}_{reuse} \leftarrow$VariableReuse($\tilde{s}_{str}$)
  $p \leftarrow$Encode($\tilde{s}$) {Encode into prefix format}
  $p_{cse} \leftarrow$Encode($\tilde{s}_{cse}$)
  $p_{str} \leftarrow$Encode($\tilde{s}_{str}$)
  $p_{reuse} \leftarrow$Encode($\tilde{s}_{reuse}$)
  $p_{rules} \leftarrow$Rules(p_{reuse}) {Probabilistic application of rewrite rules}
  if *CheckLimits*($p, p_{cse}, p_{str}, p_{reuse}, p_{rules}$) **then**
    $Optim \leftarrow$ $Optim$+MixPairs($p, p_{cse}, p_{str}, p_{reuse}, p_{rules}$)
  end
end

Algorithm 1 provides an overview of the process we use. After finding source GitHub programs and abstracting them, we perform 3 high-level compilation steps on

the abstracted C code: common subexpression elimination, strength reduction, and variable reuse. For training sample generation, Encode will transform abstracted C code into the prefix encoding of the AST. Encode will randomly reassign scalar variable IDs to temporary variables with each iteration of the `foreach` loop; so `t1` may be assigned to `s03` for one program and `s25` for another program. The goal of the random assignment by Encode is to help with generalization. The high-level compilation steps utilize many of the 23 rewrite rules shown in Table 1, but in order to ensure all rewrite rules are represented in $Optim$, we call the Rules function on p_{reuse} (the encoded program after all compilation steps) which may apply one or more rewrite rules to create p_{rules}. The Rules function is used heavily in Section 4.1.2 and is discussed further there. The CheckLimits function ensures that our samples meet the model limits[1].

Using these rules, the dataset $Optim$ derived from C functions from Github eventually contains 49,664 unique known equivalent program pairs for our experimentation. Note this approach does fully preserve the shape of the AST of the original GitHub C code snippet. It does not however preserve names, since we rename all distinct variables/array accesses to a unique and canonical name for that program. For example, `particles[i].y_pos` becomes `i1` in the generated program, while `y` is `i1` in Fig. 1.

4.1.2 *Synthetic equivalent program pairs*

As we discuss in Section 3.5, we need to create an initial training set with broad distribution on the input and output tokens necessary for our problem of proving programs equivalent. We create legal input programs by probabilistically applying production rules from a grammar which defines our target program space. This approach allows us to create arbitrarily large amounts of training data.

Algorithm 2: GenerateRewrites

Input : Probabilistic grammar: G, Number of samples desired: n
Output: Prefix encoded equivalent program pairs with rules $Synth^{+rules}$

$Synth^{+rules} \leftarrow \emptyset$ {Rewrite rule equivalence program pairs}
while *samples in* $R < n$ **do**
  $p_A \leftarrow$GenerateProgA(G) {Probabilistic production rule generation}
  $p_B \leftarrow$Rules(Rules(Rules(p_A))) {3 passes over p_A with probabilistic application of rewrite rules}
  if *CheckLimits*(p_A, p_B) **then**
    $Synth^{+rules} \leftarrow Synth^{+rules} + (p_A, p_B, rules)$
  end
end

Algorithm 2 shows the synthesis algorithm. Our program grammar defines a program as made up of a series of assign statements which assign scalar and vector variables to complex mathematical expressions. Our program generation process starts by creating assignment statements for the output variable(s). Then a subset of the variables used in the expression may have earlier assign statements added. This process continues adding assign statements randomly to the beginning of the program.

1. We limit each program to 20 statements, at most 100 AST nodes, at most 30 scalar variables, an expression depth of 5 levels of parenthesis (expression AST depth of 6), and at most 2 outputs.

Prog A (prefix encoding of AST): (a)
```
v26 = ( -v v27 ( h5v v27 v27 ) ) ;
v27 = ( +v ( -v 0v 0v ) ( -v 0v v26 ) ) ;
s21 = ( u4s ( is s01 ) ) ;
s01 = ( ns ( ns s21 ) ) ;
v08 === ( *v ( -s s21 s01 ) ( -v v26 v26 ) ) ;
s26 === ( h4s v26 ( nv v27 ) ) ;
```

Prog B (prefix encoding of AST): (b)
```
v26 = ( -v v27 ( h5v v27 v27 ) ) ;
v08 === 0v ;
v27 = ( -v 0v v26 ) ;
s26 === ( h4s v26 ( nv v27 ) ) ;
```

Inputs: v27, s01
Outputs: v08, s26

Rewrite Rule Sequence: (c)
```
stm2 AssociativeLeft N
stm5 Cancel Nr
stm5 AbsorbOp N
stm4 DeleteStm
stm2 NeutralOp Nll
stm3 SwapPrev
stm4 SwapPrev
stm2 DeleteStm
stm3 NeutralOp Nl
```

Fig. 5: Equivalence proven between 2 multi-statement programs generated synthetically. The equivalence proof is 9 steps long involving expression and statement rules.

Variables which are used but never assigned are considered program inputs. For example, in the program "`c = b + a; d = c * a + b;`"; a and b are inputs, d is an output, and c is a temporary variable. Subfigure 5(a) shows an example ProgA generated using our algorithm. It includes 6 statements and produces one vector output v08 and one scalar output s26 identified with === tokens.

Rules In order to create training samples with known paths between equivalent programs, after creating a program we randomly apply legal rewrite rules to the start program. For example, Figure 5(c) shows the rewrite rules randomly selected which transform ProgA in subfigure (a) into ProgB in subfigure (b).

Synthetic distribution Figure 6 diagrams the distribution of samples generated by Algorithm 2 with a plot of the number of AST nodes in ProgA and the number of rewrite rules used to generate ProgB in the sample. When GenerateProgA generates a program with more AST nodes, more rewrite rules are found by invoking of Rules function. We limit the number of AST nodes to 100, as shown in the distribution, but the number of rewrite rule steps is not strictly limited and we have some cases with over 40 steps between ProgA and ProgB (not shown in figure).

Certain rewrite rules, such as `Commute`, tend to be applicable to many nodes in a program AST, while others, such as `FactorLeft`, require rarer patterns in order to be legally applied. We adjust the likelihood of rewrite rules being applied to help balance the likelihood a proof will use any given rewrite rule. All 23 rules shown in Table 1 will occur in our synthetic dataset. Because the `Commute` rule can be applied to a large number of operators in our AST, we limit the likelihood it will be applied to about 9% per location with the result that 60.2% of the proofs use it for generating ProgB. Conversely, given our random program generation process, `FactorLeft` and `FactorRight` are not so likely to be applicable, so we bias the application so that about 65% of the AST nodes which would allow these rules have them applied which results in 7.1% and 7.2% of proofs using these rules.

For algorithm 2 the probabilistic grammar expansion

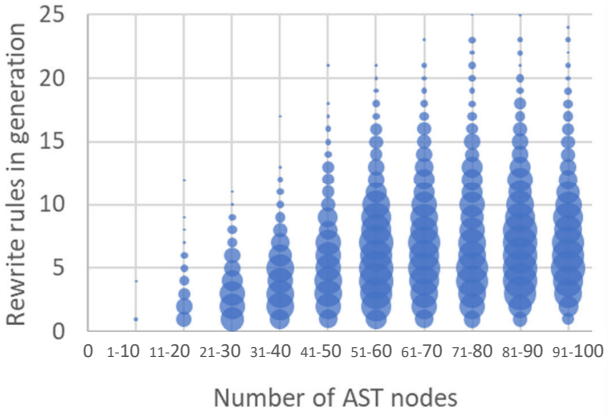

Fig. 6: Distribution of proof length for synthetic program equivalence dataset $Synth^{+rules}$.

used in GenerateProgA is tuned so as to create a range of program sizes. We skew the generation to prefer creation of programs with large AST node counts which have either many statements or deep expressions and rely on CheckLimits to insure programs outside our transformer model ranges are pruned out. For the T_1 initial training dataset, we create 150,000 equivalent program pairs; given that the average pair takes multiple rewrite rules to transform, we create 640,000 rewrite rule steps from these pairs for training the M_1 model.

Finally we note our set is made of about 90% of programs where at least one live-in symbol is used in computing some output, ensuring a high (but not exclusive) representation of "useful" programs where dead code elimination cannot trivially solve their equivalence. Dependencies between statements are produced during program generation, by forcing to reuse a variable previously written in a prior statement in the currently synthesized expression statement.

4.2 Experimental setup of incremental training

Recall from Section 3.5 that the model initially trained on synthetic programs with target rewrite rules is labeled M_1. We train our initial model M_1 for 100,000 steps, which is 5 epochs of our 640,000 sample T_1 dataset (after dividing by our effective batch size of 32). As per Figure 4, M_7 has gone through 6 iterations of incremental training beyond the initial M_1 model. In order to validate partially trained models during training, we create $Synth_{val}$ and $Optim_{val}$, which are each 1,000 equivalent program pairs randomly sampled from the $Synth$ and $Optim$ datasets. The success of the intermediate models on proving equivalences in $Synth_{val}$ and $Optim_{val}$ is used to select the model to use in the next step of incremental training.

Figure 7 diagrams how the algorithms introduced in Sections 4.1.1 and 4.1.2 are used along with self-supervised sample selection introduced in Section 3.5 to create training samples T_{1-7} to train our models.

For self-supervised sample selection, after training M_1 with T_1 we want to create training data which insures we continue improving performance on the synthetic dataset $Synth$ but also learn to solve equivalent programs in $Optim$. Referring to Figure 4, we create the unsupervised set of program pairs U at each iterative learning step by selecting

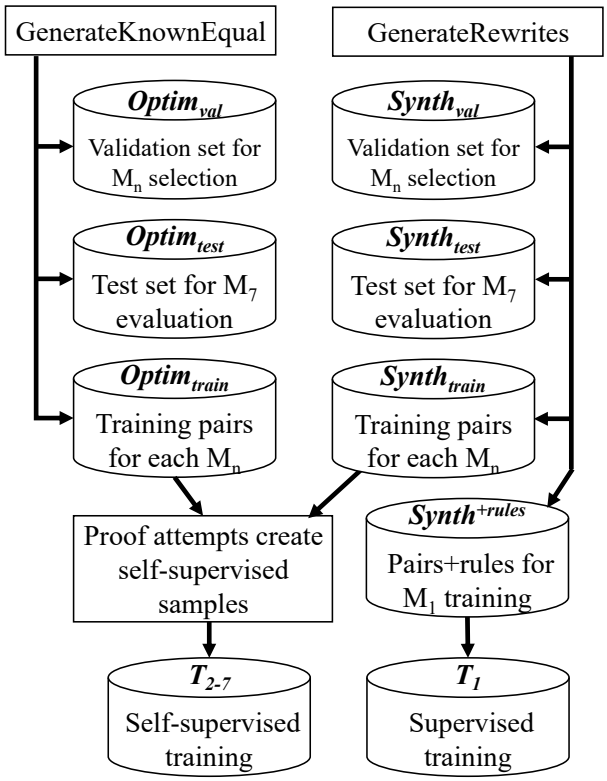

Fig. 7: Generation of program equivalence datasets used for model training and evaluation.

60,000 equivalent pairs from $Synth$ and 40,000 equivalent pairs from $Optim$.

In order to attempt to prove equal the program pairs in U, we chose $i_{easy} = 2$ for the "easy" beam width as a beam width of only 1 can fail with a single misstep and we wanted to allow recovery from that case. Due to machine time constraints, we chose $i_{hard} = 20$ for the "hard" beam width and $s = 25$ for the maximum number of proof steps to search. From this data, we create T_n for use in training model M_n.

During incremental training, we train 4 model versions (we vary the learning rates) for 50,000 steps with the new T_n training dataset. Similar to early stopping techniques [41], we use our validation datasets to select the best performing model during incremental training. We save the model every 10,000 training steps and select M_n as the model with the highest total proofs found in the 2,000 test samples when $Synth_{val}$ and $Optim_{val}$ are combined. Our final iteration is selected when both $Synth_{val}$ and $Optim_{val}$ have improved performance by 1% or less when training the next model. For our experiments, this is M_7.

4.3 Research Questions

In this section, we describe our research questions for S4Eq and the protocol methodologies for evaluating them.

Our research questions are:

- RQ1: How effective is S4Eq on the synthetic test dataset?
- RQ2: How useful is incremental learning with self-supervised sample selection in improving S4Eq's model?

- RQ3: To what extent does our model generalize outside the training data?

For RQ1 we create a 10,000 sample test dataset drawn from the synthetic programs $Synth$ which we call $Synth_{test}$. For RQ2 we create a 10,000 sample test dataset drawn from $Optim$ which we call $Optim_{test}$. The samples in the test datasets do not overlap with any samples used for training nor with the validation datasets $Synth_{val}$ and $Optim_{val}$. To more broadly understand the system behavior, in both RQ1 and RQ2 we analyze subsets of $Synth_{test}$ and $Optim_{test}$ based on characteristics of the rewrite sequence which transforms the first program in the sample into the second. Of the 23 rewrite rules shown in Table 1, `Rename` is the one which occurs the least in our initial T_1 training data as it is least used when synthetically creating program pairs in $Synth$ (552 out of 10,000 samples in $Synth_{test}$ use it), so we report on the subset of proofs which use `Rename` to observe the system behavior on this group. `Newtmp` is the rule which improves most between M_1 and M_7 in the $Optim_{test}$ test set - improving from 545 proofs found by M_1 to 2,277 proofs found by M_7 so we also report based on this rule. The `DistributeLeft` rule is the one most improved on between M_1 and M_7 when attempting proofs in the $Synth_{test}$ test set - improving from 796 proofs found by M_1 and 1,771 proofs found by M_7. Proofs that use these 3 rewrite rules will tend to be longer proofs (longer proofs are more likely to use any given rewrite rule), so we also include a rule category which is subtractive. We report on proofs which do not include any statement rules (*i.e.,* they don't use `SwapPrev`, `Usevar`, `Inline`, `Newtmp`, `DeleteStm`, or `Rename`), which also allows for comparisons with our prior work [42]. Because our hindsight methodology focuses on NodeIDs at depth 5 of the AST (and these are individually our least common tokens), we report on proofs which use such an identifier. Our last 2 proof subsets are based on the length of the rewrite rule sequence used to transform ProgA to ProgB in the sample - we report on proofs of 1-10 steps as a group as well as proofs of 11 steps or more.

We perform all our experiments on systems with 12 Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz CPU cores and NVIDIA GeForce GTX 1060 6GB for GPU compute. Our model is based on the transformer model implemented in OpenNMT-py. Proof searches for incremental model training may use up to 30 systems in parallel.

4.3.1 Methodology for RQ1

To evaluate the effectiveness of S4Eq on finding rewrite rule sequences for the program pairs selected from the synthetic program dataset $Synth$, we analyze the distribution of programs and rewrite rules included in $Synth$ and present data on the success rate for proving various subsets of the test dataset $Synth_{test}$. Our goal is to understand if our system is unable to perform well on any of our selected subsets of $Synth_{test}$ (*i.e.,* solve at less than a 90% success rate).

4.3.2 Methodology for RQ2

Our incremental training approach is able to learn to solve proofs for which the supervised proof sequence is not provided during sample generation. To determine how well self-supervised sample selection improves the quality of the S4Eq model, we study the 10,000 sample $Optim_{test}$ dataset drawn from $Optim$ alongside our rewrite rule test dataset $Synth_{test}$.

In addition to our golden model M_7, which is trained using self-supervised sample selection, we create a model for comparison called Q which is trained in a more traditional way. Q is trained for the same number of training steps as M_7 but it continues training on the T_1 dataset from the M_1 model in a supervised manner. To align with our M_n training protocol, we train multiple models from M_1 with varying learning rates and select the strongest model using the $Synth_{val}$ and $Optim_{val}$ datasets. We will evaluate the ability of our models to prove the known equivalent program pairs in $Optim_{val}$ in order to determine if self-supervised sample selection adds value to our model.

4.3.3 Methodology for RQ3

We explore the ability of S4Eq to generalize using 2 different methods. The first method focuses on using the sample generation algorithm 2 for $Synth$ with different hyperparameters to create programs outside of the training distribution. The second method relates to actual use cases for program equivalence and focuses on finding equivalent programs from within different GitHub C functions. This search could be viewed as a demonstration of the system to find semantic equivalence for a variety of uses, such as identifying opportunities for library calls [43], or grouping student submissions into semantically equal groups for grading [9], *etc.*

Regarding the first method, we use algorithm 2 to create programs with exactly 3 outputs and 101 to 120 AST nodes. Recall that in training we limit $Synth$ and $Optim$ to include only 1 or 2 output programs with up to 100 AST nodes. We test our golden model on this dataset to determine if it has overfit the training data or if can solve problems from outside the training distribution.

Regarding the second method, the FindSourcePrograms routine in algorithm 1 finds 13,215 unique multi-statement program blocks from GitHub. No pair of these programs are lexically equivalent and hence at least one rewrite rule step will be required for S4Eq to prove equivalence. As we are interested in comparing the more complex programs in this set, we select only programs with at least 30 AST nodes resulting in a set of 4,600 programs. We then group the programs and test all pairs of programs which have the same number of inputs, outputs, and functions. This results in 152,874 unique pairs of programs to check for equivalence. Since we search in both directions for each pair, our full GitHub test set $Human$ has 305,748 program pairs to attempt equivalence proofs on.

4.4 Experimental Results

4.4.1 RQ1: Effectiveness on Synthetic Test Dataset

In this research question, we aim to show the breadth of program pairs and rewrite rule sequences in our synthetic dataset and to demonstrate the effectiveness of our M_7 golden model. Table 3 details the performance of a variety of subsets of our 10,000 pair synthetic test dataset $Synth_{test}$. Each cell in the table gives the passing rate and sample counts for each subset. Here the passing rate means the

TABLE 3: Success rate for subsets of test dataset $Synth_{test}$. Each entry includes tuned passing percentage and sample count within $Synth_{test}$.

Rewrite Rules	ALL	Functions 3 or more	Maximum Expression Depth 4-6	Nodes 30-100
Whole dataset	98%(10000)	98%(6390)	97%(3340)	98%(9470)
Rename	97%(552)	97%(354)	95%(74)	97%(546)
Newtmp	94%(844)	94%(591)	93%(477)	94%(828)
DistributeLeft	96%(2273)	95%(1650)	95%(1524)	96%(2214)
No statement rules	99%(4923)	98%(3208)	98%(2312)	99%(4524)
NodeID at depth 5	92%(533)	92%(429)	92%(506)	92%(522)
Rewrite steps 1-10	99%(8253)	99%(5066)	99%(2269)	99%(7725)
Rewrite steps 11+	94%(1747)	93%(1324)	92%(1071)	94%(1745)

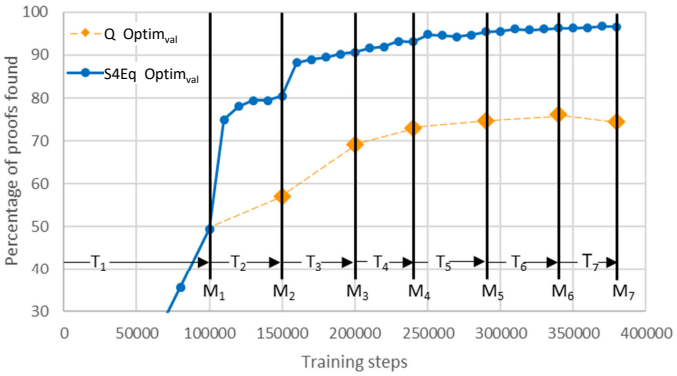

Fig. 8: Performance metrics on $Optim_{val}$ through training process. M_7 is the final golden model used for experiments, Q is purely supervised training on the T_1 training data.

percentage of program pairs proven equivalent with an appropriate sequence of rewrite rules (recall that all program pairs in $Synth$ are equivalent by construction). The first data row gives the effectiveness over the whole dataset. The other rows in the table provide data on subsets of the sample based on the rewrite rules used to generate the ProgB in the sample given the synthetically generated ProgB. A full description of the rows is given in Section 4.3. Orthogonal to the rewrite rules used to generate ProgB, the columns explore subsets of interest for the original ProgA in the sample. The first column provides data for all samples which conform to the rewrite rule subset given by the rows. The 2nd column shows results for the 6,390 samples that used at least 3 functions in ProgA. The 3rd column shows results for the 3,340 samples where ProgA starts with an expression of depth 4-6 (note that a NodeID at depth 5 is used in 533 samples given any ProgA but only 506 samples when ProgA started with a deep expression; this is due to some rewrite rules, such as `MultOne`, adding depth to the original ProgA). The final column provides data on the larger programs from the dataset (and the size corresponds to the program sizes considered in RQ3).

In the upper left data cell in Table 3, we find that of the 10,000 samples in $Synth_{test}$, M_7 was able to find a rewrite rule sequence from ProgA to ProgB for 98% of them, which is arguably very high. We see that some subsets of $Synth_{test}$ performed better than this overall result, such as cases where ProgA to ProgB proofs contain 1-10 rewrite rule steps (99% effectiveness). Other subsets performed worse, such as proving samples where a depth 5 NodeID was use to create ProgB. In general we see that program pairs generated with shorter rewrite rule sequences are more easily proven equal than longer ones, corresponding to the intuition that shorter proofs are easier to find. In the table, there are 5 subsets tied for the poorest result of 92% success: all 4 subsets with NodeID at depth 5, and cases where ProgA has a deep expression and generating ProgB used over 10 rewrite rule steps. These results show that S4Eq is challenged by deeply nested expressions and long proofs, however it still achieves over 90% success.

Table 3 also shows that S4Eq well handles rare tokens. The output token least represented in our 640,000 samples in T_1 is `Nrlrr`, used in only 278 samples. `Nrlrr` is one of 16 tokens used to indicate that a rule should be applied to a depth 5 node; all 15 of the other such tokens make up the 15 other least commonly used tokens in the T_1 output group. During self-supervised training, we compensate for this rarity to increase the number of such samples in T_{2-7}. As we show with the "NodeID at depth 5" row in Table 3, our fully trained model has learned to use these tokens effectively.

Our synthetic ProgA algorithm given in algorithm 2 is designed to insure a relatively balanced use of input tokens in T_1 for training. This works well, because the least used input token is one of the 5 tokens representing functions which receive 2 scalars and produce a vector output (`f4v`) with 43,458 of the 640,000 samples using it. Table 3 shows that those programs using at least 3 functions are proven well by S4Eq.

> Answer to RQ1: S4Eq is effective on the $Synth$ dataset, achieving over 90% successful proofs on all subsets analyzed. S4Eq handles well both rare input tokens used in the programs to be analyzed (such as function names) and rare output tokens (such as depth 5 NodeIDs).

4.4.2 RQ2: Incremental Training Benefit

In order to show how the an initial model improves with self-supervised sample selection, we compare S4Eq training up to model M_7 against a model trained only on the supervised training set T_1. For this comparison, we study the $Optim_{val}$ and $Optim_{test}$ datasets derived from GitHub program compilation steps since these include program pairs which can be proven equal using our rewrite rules. In Figure 8, the orange squares indicate the percentage of proofs found on the $Optim_{val}$ dataset as the model trains only in a supervised manner on T_1 (the initial supervised training set of synthetic programs with known rewrite rules). The blue circles represent training with self-supervised sample selection. With traditional training, we see the best performance of the Q model on $Optim_{val}$ occurs at 340,000 total training steps (S4Eq had trained to M_6 after that many steps). This result is significantly below the performance S4Eq achieves. The significant improvement of S4Eq on the $Optim_{val}$ dataset demonstrates the benefits on our incremental training procedure.

After training from 340,000 steps to 380,000 steps, Q decreased performance on the GitHub compiled test dataset.

This indicates that Q was starting to overfit on the distribution for $Synth$ and the latest learnings were not as applicable to the $Optim$ problems. On the contrary, since self-supervised sample selection generates new training samples, S4Eq is able to avoid overfitting and continues to improve the overall model's ability to find proofs on the $Optim_{val}$ dataset.

Also, we note that Q was able to slightly outperform M_7 on the $Synth_{val}$ dataset (98.96% pass versus 98.44%, not shown in the figure) which shows that the T_1 dataset had sufficient samples to train for the problem distribution in $Synth_{val}$.

TABLE 4: Performance of M_1, Q, and M_7 models on the 10,000 $Optim_{test}$ test pairs based on GitHub code. The self-selected samples in T_{2-7} are biased to areas where M_1 is weak, ultimately allowing M_7 to outperform Q in all categories.

Sample or Proof Sequence Used	T_1 Samples	T_{2-7} Samples	M_1 Proved	Q Proved	M_7 Proved
Any rewrite rule	640,000	714,332	4,866	7,531	9,688
Rename	5,267	54,925	2,591	4,791	6,315
Newtmp	7,706	33,053	545	810	2,277
DistributeLeft	30,029	24,243	526	649	774
No statement rules	506,217	479,637	968	1,109	1,111
NodeID at depth 5	5,969	35,842	18	91	323
Rewrite steps 1-10	367,976	336,344	4,690	7,329	8,997
Rewrite steps 11+	272,024	377,988	176	202	691

Table 4 shows the benefit of training with samples that are sampled from challenging proofs. The first row of the table summarizes the total number of samples available in T_1 used to train M_1 and Q, the total number of samples in T_{2-7} used to train incrementally up to M_7, and the total proofs in the 10,000 sample GitHub test dataset $Optim_{test}$ found by the models M_1, Q and M_7. For example, in the upper right corner we show that there were 9,688 (out of 10,000) GitHub program pairs for which proofs were found by the best model M_7. Different subsets of these 9,688 proofs are shown in later rows regarding the rewrite rules needed to prove the GitHub program compilation cases. The 2nd-4th rows introduce the mechanism through which self-supervised sample selection most benefits the model. For the `Rename` rewrite rule, we see that, of the 9,688 samples proven equal by M_7, 6,315 of them used `Rename` for at least one step, but M_1 only used `Rename` in 2,591 of its proofs (less than half of the M_7 usage). We can see that from the T_1 and T_{2-7} columns that self-supervised selection recognized that M_1 was weak in this area (implying searches with i_{easy} rarely found the proof) but was able to augment the training data with more samples using the `Rename` rule (implying searches with i_{hard} were able to provide example successes). We can see that model Q, which continued training only with T_1 samples, did not learn this rule as well as M_7. We see a similar effect for the `Newtmp` rule; 2,277 proofs used this rule for M_7, but only 545 used it for M_1. We see that T_{2-7} included more of these samples to help improve the model. As a counterexample, we see that `DistributeLeft` is used in 774 successful proofs of the high-performing M_7 model, and in 526 of the proofs for the initial M_1 model. Given that M_7 was able to solve 97%

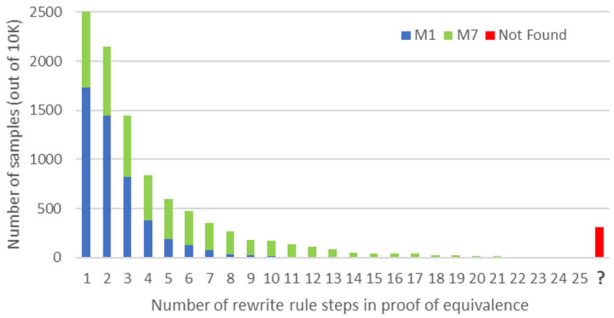

Fig. 9: Proofs found on test set $Optim_{test}$. The length charted is the proof length found with the tuned model M_7, with the unproven cases shown as unknown step count.

of the problems in $Optim_{test}$, M_1 had already solved over two thirds of these cases which implies T_{2-7} only needed to contain certain cases of `DistributeLeft` where the model was still challenged in order to improve the model. In this way, self-supervised sample selection provides new training samples to improve the areas in which the model is weak.

Table 4 also includes 'NodeID at depth 5', which indicates a rewrite rule was used on an AST node at depth 5 of an expression. Since we use NodeIDs to identify rule application positions, there are $2^{5-1} = 16$ different IDs needed to correctly locate a given instance. Fewer than 1% of T_1 samples (5969) include a depth 5 node, which correlates with poor base model performance (18 cases proven). However, part of our incremental training data are the hindsight steps which use a depth 5 node. Consequently, the performance increases to 323 found proofs, meaning that the self-supervised training greatly improves the results.

Self-supervised sample selection also improves performance on long proofs by increasing the number of samples from such proofs. The last 2 rows of Table 4 report on the number of rewrite rule steps required by the 3 different models to prove pairs equivalent. The last column shows us that M_7 successfully used long proofs for 691 samples, while M_1 only found 176 long proofs. Indeed, we also see that T_{2-7} contained more samples from long proofs than T_1 to help the model improve this category, explaining this difference.

In order to provide more insight into how S4Eq handles and reacts to challenging proofs, we now discuss the first proofs searched for with M_1 to create T_2 which was used to train M_2 and also the last proofs searched for by M_6 to create T_7 used to train M_7. Recall that our $Optim_{val}$ model evaluations are done with $i = 10$, and the proof searches done for self-supervised sample selection bracket this value (the searches use $i_{easy} = 2$ and $i_{hard} = 20$). M_1 was able to solve 49.5% of the $Optim_{val}$ samples; from the unsupervised program pairs M_1 attempted for T_2 generation, M_1 solved 45.0% of them with i_{easy} and 60.6% with i_{hard}. Additionally, for 8.7% of the pairs proven with i_{easy}, the proof found with i_{hard} was at least 2 steps shorter. After training with the 126,630 single-step samples generated for T_2, M_2 was able to solve 80.5% of the $Optim_{val}$ samples. Similarly, M_6 was able to solve 96.3% of the $Optim_{val}$ samples; it solved 94.2% of the unsupervised samples with i_{easy} and 97.4% with i_{hard}. After training with 83,045 samples in T_7, M_7 was able to solve 96.6% of the $Optim_{val}$ samples. Note that

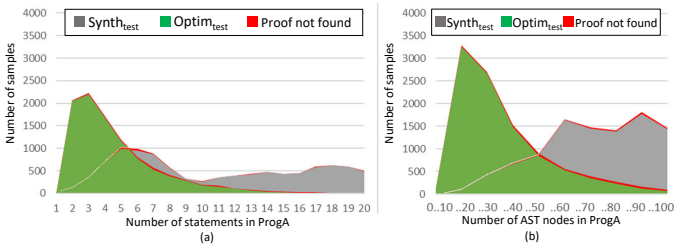

Fig. 10: Proofs found on test sets $Synth_{test}$ and $Optim_{test}$ plotted for varying statement and AST node counts. In all cases the performance of S4Eq closely tracks the dataset distribution.

while the absolute difference between i_{easy} and i_{hard} was over 15% for M_1 but just over 3% for M_6, the total number of single-step samples in T_1 was less than double the count in T_7; this is an effect of the challenging proofs increasing in step count as the models train, as indicated by Figure 9. While fewer proofs using i_{easy} fail with the later model, the lengths of the proofs which do fail tend to be longer.

The details on long proofs shown with Figure 9 demonstrate another benefit of self-supervised sample selection. Recall that M_1 can only prove 176 samples equivalent with a proof over 10 steps. Of the 691 samples that M_7 solves with over 10 steps, M_1 only solves 23 of them. The other 153 samples for which M_1 found a long proof are still proven by M_7, but in 10 steps or fewer. Figure 9 shows the benefit of self-supervised sample selection on the performance of proofs of increasing length in the $Optim_{test}$ dataset. Here we see that proofs which M_7 proved with only 1-3 steps were solved with over 50% success by M_1; yet proofs that M_7 found with over 10 steps were rarely solved by M_1.

We now assess the ability of self-supervised sample selection to avoid catastrophic forgetting by analyzing the samples which M_1, Q, and M_7 are able to prove. Of the 4,866 samples that M_1 is able to prove equivalent, *ALL* of them are included in the 9,688 samples that M_7 proves after training based on self-supervised sample selection. However, only 4,774 of them are included in the 7,531 which Q proves - Q 'forgot' how to prove 92 samples that the M_1 model it trained from had proven. *This shows clearly the benefit of self-supervised sample selection for avoiding catastrophic forgetting.*

> Answer to RQ2: Compared to supervised training, using self-supervised sample selection improved performance on our target dataset from 75% success (supervised training) to 97% success (S4Eq's novel self-supervised training). Self-supervised sample selection does focus on areas where the model needed the most improvement by selecting the most interesting new training samples.

4.4.3 RQ3: Generalization Ability

A concern with machine learning is that it may learn to perform well within the samples on which it is trained but not generalize well to unseen problems that humans would consider related. We asses this risk by presenting data on how well S4Eq has generalized beyond its training distribution.

Let us first discuss the differences in the distribution of programs and rewrite rules between $Synth$ and $Optim$.

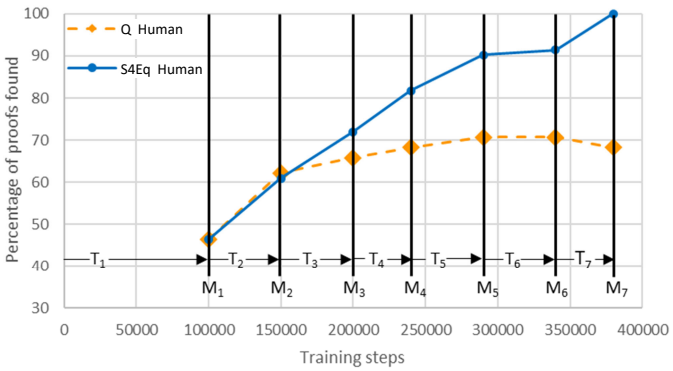

Fig. 11: M_7 searched 305,748 program pairs based on human-written GitHub functions and found 82 equivalent pairs. This chart shows the percentage of these 82 cases found as S4Eq trains and as Q trains (Q trains only on T_1 data).

Figure 10 shows the histograms of the number of assign stamements and tokens in ProgA for $Synth$ and $Optim$. We see the human-written code in $Optim$ shown as the green distribution tends to have many samples with fewer than 5 statements and fewer than 50 tokens, while the synthetic code shown as the grey distribution was designed to create more complex programs and hence has ProgAs with more statements and more AST nodes. This clearly shows a different distribution. The red areas show the proofs which are not found. The thinness of the red area showing that there is no obvious weakness on any area of the ProgA distribution. The rewrite rules needed to prove equivalence also varies between $Synth$ and $Optim$: 49% of the proofs for samples in $Synth_{test}$ are solved without rewrite rules related to statements while less than 12% of proofs on the $Optim_{test}$ dataset could be solved without these rules (compare the "No statement rules" rows from tables 3 and 4). Taken together, these data show that S4Eq has generalized well over 2 different datasets.

In order to show S4Eq can generalize outside its training domain, we used algorithm 2 with CheckLimits and GenerateProgA adjusted such that we created test samples with 101-120 AST nodes and 3 outputs, which is outside the initial training distribution. Recall that on $Synth$ (with up to 100 AST nodes and 1 or 2 outputs), S4Eq achieves 98% success on the $Synth_{test}$ test set. Now, on programs with 101-120 AST nodes and 3 outputs it is still able to prove 60% of the program pairs equivalent, showing that S4Eq can generalize to larger examples.

For our final evidence of generalization, we present equivalences found between human-written programs on GitHub. Of the 305,748 program pairs in $Human$, we found 82 provable cases of equivalence. Figure 1 shows an example of proven equivalence of GitHub samples. Figure 11 illustrates the generalization of S4Eq to this problem as training progresses by showing the percentage of the 82 proofs found by M_1 through M_7. The figure also shows the performance of Q (which only trains on T_1 sampled from $Synth$) on the same 82 proofs. We see that after 150,000 training steps M_2 and Q perform about equally well but ultimately Q plateaus with just over 70% of the proofs found at 340,000 steps. Like Figure 8 on $Optim_{test}$ data, Q seems to start overfitting

Model description	Percent proofs found $Synth_{val}$	$Optim_{val}$
M_7 (Golden model)	98%	97%
M_1 (Golden model before incremental training)	66%	50%
Faster learning rate (0.0002 vs 0.0001 in M_1)	2%	1%
Slower learning rate (0.00005 vs 0.0001 in M_1)	49%	35%
Fewer transformer layers (6 vs 8 in M_1)	56%	23%
Limit language to scalars (vs scalars+vectors in M_1)	63%*	40%
Linear algebra expressions only (and 50 AST node limit vs 100 in M_1)	99%*	9%

TABLE 5: S4Eq ablation study. With the exception of the 'Golden model', results are shown with only initial base training of the model. The synthetic datasets for the last 2 rows were regenerated based on the modified language grammar.

on the T_1 data after 340,000 steps and does not generalize well to different problem areas. However, we see that S4Eq continued to improve on the $Human$ test set from M_6 to M_7 showing it has generalized to the problem of finding equivalence between human-written programs well.

> Answer to RQ3: S4Eq is able to generalize to problem domains outside its initial supervised training dataset. S4Eq is able to solve 60% of synthetic programs that are outside of any training data it was presented with. Furthermore, it is able to progressively generalize to find up to 82 proofs of equivalence in the golden human-written samples from GitHub.

4.5 Ablation Study

Table 5 illustrates some of the model variations we explored. Except for the golden model, all of the results are reported only after initial training using the appropriate synthetic dataset. All of the models are also evaluated on the GitHub dataset.

The faster learning rate shown in Table 5 has poor results presumably due to known risks of divergence with large learning rates. We studied this result further by attempting to do an iteration with self-supervised sample selection using the poor proof success. As discussed in Section 4.4.2, self-supervised sample selection requires that sufficient examples of the input and output tokens are provided for successful incremental training. For example, using the M_1 that led to our golden model, the T_2 dataset had 126,630 samples for use during training, 11,737 of which used the Rename rule. For the high learning rate model, since even the i=20 search had poor success, its T_2 only had 1,482 samples in it and NONE included a successful use of Rename. Hence, unlike our golden model, incremental training on the high learning rate model did not significantly improve performance on $Synth_{val}$ nor $Optim_{val}$. We see also that a slower learning rate produced a slightly worse result than our M_1 result, hence we selected M_1 for our S4Eq training base.

We tested our transformer model with different numbers of attention layers, as well as different numbers of attention heads and hidden feature sizes. We show a typical result of these searches with only 6 attention layers. This parameter had a small loss for the $Synth_{val}$ success, but did not generalize as well to the $Optim_{val}$ dataset and hence it was not pursued further.

TABLE 6: Results for 10,000 pairs in $Optim_{test}$ as proof search parameters are varied. Parameters for M_7 are shown in **bold**.

l	i	b	Proofs Found	Time (sec)	l	i	b	Proofs Found	Time (sec)
50	1	1	87.7%	831	50	10	2	96.1%	4177
50	**10**	**5**	**96.9%**	**8964**	50	10	10	96.9%	13723
50	2	5	95.1%	2428	50	20	5	97.2%	15823
40	10	5	96.8%	8288	60	10	5	96.9%	9720

The last 2 rows of Table 5 explore training models on alternate language grammars. For 2 these cases only the synthetic proofs found column indicates the number of proofs found for the synthetic dataset which aligns with the model description. Perhaps surprisingly we see that a language with only scalar variables and operators performs worse than M_1 (which has both scalars and vectors) on both its own synthetic validation set as well as on $Optim_{val}$. One possible explanation for the weakness on both the synthetic validation set with only scalars and the compiled GitHub programs in $Optim_{val}$, which only use scalars, is that the existence of vectors in the training set helps the attention layers in the model generalize better to complex uses of the rewrite rules. The final row of the ablation study shows results for a language which has only a single statement with up to 50 AST nodes, but matrixes, vectors, and scalars are supported. We see strong success on a synthetic dataset with the same features, but this model does poorly on the GitHub compiled equivalence checks. When compared with prior work on a similar dataset [42] this row demonstrates that *for our problem of program equivalence the transformer model outperforms a well tuned graph-to-sequence model which itself was found to outperform a bidirectional RNN sequence-to-sequence model*.

In addition to the hyperparameters related to model selection and model training, S4Eq usage relies on parameters related to equivalence proof search. The three main parameters are b (neural beam width), i (intermediate programs), and l (search step limit), they all affect both search success and search time as shown in Table 6. The entry with $i = 1$ and $b = 1$ gives a sense for the quality of our final model: taking a single proposed rewrite rule from the neural network with only a single intermediate program

still finds 87.7% of the proofs for the pairs from $Optim_{test}$ when searching for $s = 50$ steps. This result shows that a significant majority of rules produced by the model are useful for the equivalence proof. Also of note, we show the proof success rate for M_7 when $s = 50$, $b = 5$, but i is set to 2, 10, and 20. This shows the performance and time results for $i_{easy} = 2$ and $i_{hard} = 20$ as well as our search i setting of $i = 10$. In particular, if we continued with self-supervised sample selection, 2.1% of proofs are found with i_{hard} but not i_{easy}, allowing for continued proofs on which to improve the model. The other entries in the table show that we selected our proof search parameters near the point where further time did not significantly improve the percentage of proofs found.

This ablation study does not include the full breadth of models and language representations we explored. For our transformer interface, we tested input variations (such as using infix instead of prefix or including statement numbers in the input) and output variations (such as different rewrite rule syntax including left/right path listing to identify expression nodes and also outputting the full rewrite sequence as a single long output). We also tested sequence-to-sequence RNN and graph-to-sequence models on early versions of our language [42], and we explored transformer model parameters guided by OpenAI's work on neural language model scaling [44]. In the end the parameters for our golden model performed best overall in these studies.

4.6 Execution time

Table 7 shows the machine hours needed for the key steps related to M_7 training and usage. For some steps we use up to 30 machines in parallel as indicated in Section 4.3. This table shows that while our proof search for self-supervised sample selection takes time, it does not double the model creation time for M_7 relative to a model trained with traditional hyperparameter searches such as Q. Also of note is that almost all of the $Human$ pairs are not equal, and the average search time per pair with a 50-step limit takes about 9 seconds. Meanwhile, as seen in Table 6, the equivalent pairs in $Optim_{test}$ only take about 0.9 seconds per pair to search on average because once the proof is found the search terminates.

TABLE 7: Execution time statistics on S4Eq tasks. Machine hours are approximate as the systems are pre-emptable by students.

Task Description	Approx Machine Hours
Train 16 M_1 candidates with varying hyperparameters	375
Search for easy and hard proofs on total of 600,000 equivalent pairs with models M_1 through M_6	540
Train 4 candidates each for models M_2 through M_7 with varying learning rates and learning rate decays	270
Total to create M_7 with self-supervised sample selection	1185
Total to create Q with only supervised samples	645
Search 305,748 mostly unequal abstractions of human-written code on M_7 with $b = 5, i = 10, s = 50$	775

5 Discussion

5.1 Impact of Supervised Training

We now discuss qualitative insights related to the datasets and models. We first note how well the Q model tracked M_2 on the problem of finding human-written equivalences in GitHub. This is a signal that further training with the supervised data may be an efficient path to a quality model. Indeed, Figure 8 shows the Q model plateauing only after 340,000 iterations. We suspect that the optimal time to pause supervised training and generate self-supervised samples is before the supervised model plateaus. That would allow for model weights which have not yet been committed to the fully trained result to be available for adjustment with the self-supervised samples. However, for our problem, it may well have been more efficient to train M_1 for 200,000 steps instead of only 100,000.

5.2 Threats to validity

We now discuss the threats to validity for our work. A first concern is the general time required to generate iterative training samples. While we did not seek to optimize this procedure, it is bound to consume compute resources which could potentially be used for further model training. If the full training distribution is available with supervised samples, then it may be that self-supervised sample selection would have minimal benefit. However, the ability of self-supervised sample selection to find proofs even shorter than the supervised proof provided may yield benefit if explored directly. Our work did not directly study the benefit of self-supervised sample selection on a single dataset; it studied the ability to extend a model into a problem domain which did not provide a ground truth for supervised learning. This leads to a second threat to validity: with moderate effort our code optimizations done to produce training samples for $Optim_{train}$ could have generated rewrite rules for supervised learning. In cases where such a process could be done, then even extending to a new problem domain is reduced to the problem of training a model with supervised samples. A third area for concern with our work would be how well it will generalize to more complex languages. Indeed, our prior work has shown that increasing program size and language complexity reduces accuracy of the model [42]. Such explorations are of keen interest for future work.

The language we studied was a subset of the full C language syntax. A more complete equivalence proof on the C language would have to account for recursively verifying functions called (`AddMatrix` might be called by 2 code sequences from different projects, but the code would not be equivalent unless the function was too), loop transformations (loop unrolling or nested loop reordering), as well as pointer and array referencing.

Within the prefix encoding of AST language, we define a set of rewrite rules and claim that our system guarantees no false positives by design. A threat to this claim would be an incorrect implementation of our semantics-preserving rewrite rules and the validity checks done before they are applied (for example, if we had a bug in which Commute was allowed on scalar subtraction). Indeed, during development we found and fixed bugs of this type as we

investigated why certain proofs were generated. While there is a risk here, our experience is that the risk is low now that we have reached high levels of accuracy and reviewed many proofs. The risk that non-equivalent programs would be identified as equal is hence zero conceptually, but not necessarily zero in practice.

6 Related Work

6.1 Static Program Equivalence

Algorithms for proving program equivalence restricted to specific classes of programs have been developed [45]–[48]. These approaches are typically restricted to proving the equivalence of different schedules of operations, possibly via abstract interpretation [49], [50] or even dynamically [26]. Popular techniques also involve symbolic execution to compare program behavior [43], [51]. The problem of program equivalence we target may be solved by other brute-force (or heuristical) approaches, where a problem is solved by pathfinding. This includes theorem provers [52], [53], which handle inference of axiomatic proofs. Program rewrite systems have been heavily investigated, [10], [54]–[59]. While semantics-preserving rewrite systems for program equivalence have been studied [60]–[63], our contribution recognizes this compositional formalism is well suited to deep learning sequence generator systems. The merits of stochastic search to accelerate such systems has been demonstrated [64]–[66]. The novelty of our work is to develop carefully crafted sequence generator neural networks to automatically learn an efficient pathfinding heuristic for this problem.

EqBench [67] proposes a test suite of 147 pairs of equivalent C/Java programs. As they include if-conditionals, it is not immediately usable with S4Eq. In contrast, we mine and build tens of thousands of equivalent program pairs, using a richer set of rewrite rules for expressions and functions. Our complete dataset, including the samples extracted from GitHub, is publicly available as a program equivalence test suite [16], providing a rich suite complementing EqBench's.

6.2 Incremental and Transfer Learning

For incremental learning in S4Eq, we use an "instance incremental scenario" in that for our problem we keep the output vocabulary constant while creating new data for incremental learning model updates [68]. Ye *et al.* discuss using an output verification process (in their case compilation and test for program repair) to adjust the loss function in later training iterations [69]; our approach is related in that we test outputs but instead of adjusting the training loss we create new training samples which helps to generalize the model to a different problem domain.

To the best of our knowledge, there are only a few works that use transfer learning in the software engineering domain, and none of them use it for generating equivalence proofs. Recently, Ding has done a comprehensive study on applying transfer learning on different software engineering problems [70], such as code documentation generation and source code summarization. He found that transfer learning improves performance on all problems, especially when the dataset is tiny and could be easily overfitted. In our work, we deploy transfer learning on program equivalence proofs and show that it also improves generalization.

Huang, Zhou, and Chin used transfer learning to avoid the problem of having a small dataset for the error type classification task [71]. They trained a Transformer model on the small dataset and achieved 7.1% accuracy. When training first on a bigger source dataset and tuning afterward on the small dataset, they reached 69.1% accuracy. However, they do not develop self-supervised sample selection, and implement a limited analysis of transfer learning. In our work, we develop a form of transfer learning for program equivalence, and carefully analyze its merits and limitations, reaching 97% accuracy on our dataset.

6.3 Symbolic Mathematics Using Machine Learning

Hussein *et al.* [72] develop a system which learns to apply a set of inequality axioms and derived lemmas using a reinforcement learning model with a feed-forward neural network. A challenge of using reinforcement learning is the determination of a feasible reward function from the environment and the resulting training time. HOList [73], [74] is a system that can interact with a large rule set to score tactics in the search for a proof but faces compute time limitations when training a reinforcement learning network for theorem proving. Unlike their model, we output both the tactic (rewrite rule) as well as the location to apply it (eliminating the need to score various premises individually). Similar to recent work on using transformers to propose actions [75], our network has learned the rewrite rules (actions on a program) which are most likely to transform the program into the target program. Our work aims at laying the foundation for program equivalence proofs by studying a language subset that includes multiple computation statements and maintains a high accuracy. Our approach to synthetic data modeling is similar to Lample *et al.* [76], who randomly create representative symbolic equations to develop a deep learning sequence-to-sequence transformer model which can perform symbolic integration. They note that their system requires an external framework to guarantee validity. Similarly, work by Mali *et al.* [77] reasons on mathematical problems but produces outputs which are not guaranteed correct. Our system outputs a verifiable reasoning that is straightforward to check for correctness, guaranteeing no false positive and the correct handling of all true negatives.

A Deep Reinforcement Learning Approach to First-Order Logic Theorem Proving [78] provides the theorem prover the allowed axioms as input to the model. In contrast, we produce the rewrite rules as a sequence. As it is a reinforcement learning model, they create training iterations as proof rewards improve with better models, while our approach creates incremental samples specifically chosen through beam search to improve output by providing a correct output for a proof the model currently is challenged by.

6.4 Program Analysis using Machine Learning

Numerous prior works have employed (deep) machine learning for program analysis [79]–[84]. PHOG [85] presents a probabilistic grammar to predict node types in an AST.

Program repair approaches, [30], [81] are deployed to automatically repair bugs in a program. Wang *et al.* [86] learn to extract the rules for Tomita grammars [87] with recurrent neural networks. The learned network weights are processed to create a verifiable deterministic finite automata representation of the learned grammar. This work demonstrates that deterministic grammars can be learned with neural networks, which we rely on. Recent work by Rabin *et al.* [88] shows that neural networks (in their case GGNNs) learn more general representations of semantically equivalent programs than code2vec [80], which creates code representations using AST paths. Bui *et al.* [89] show that using semantics preserving transformation can improve machine learning on code, and continue the work with a study on using self-supervised learning to create similar embeddings for semantically equivalent code [90]. Allamanis *et al.* [91] trains a bug repair neural network by using self-supervision. The model learns to repair artificially created bugs using rewrite rules on non-buggy code. We use beam search to identify model weaknesses and target learning to generate the transformations that prove semantic equivalence.

7 Conclusion

We introduced S4Eq, an end-to-end deep learning framework to find equivalence proofs between two complex program blocks. S4Eq emits a *verified* sequence of rewrites from one program to another when successful. We have designed self-supervised sample selection, an original training technique tailored to our problem domain. This approach further improved the ability of the deep learning system to find more complex proofs, with up to 97% success on our synthetic and GitHub-constructed evaluation sets. All our datasets are made available to the community, including synthetic generation techniques for the problem of program equivalence via rewrite rules, as well as the programs built from the ASTs we mined in C functions from GitHub.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation award CCF-1750399. This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, and by the Swedish Foundation for Strategic Research (SSF).

References

[1] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[2] D. M. Kaplan, "Regular expressions and the equivalence of programs," *Journal of Computer and System Sciences*, vol. 3, no. 4, pp. 361–386, 1969.

[3] B. Godlin and O. Strichman, "Inference rules for proving the equivalence of recursive procedures," *Acta Informatica*, vol. 45, no. 6, pp. 403–439, 2008.

[4] S. Verdoolaege, G. Janssens, and M. Bruynooghe, "Equivalence checking of static affine programs using widening to handle recurrences," in *Computer aided verification*, Springer, 2009, pp. 599–613.

[5] R. Goldblatt and M. Jackson, "Well-structured program equivalence is highly undecidable," *ACM Transactions on Computational Logic (TOCL)*, vol. 13, no. 3, p. 26, 2012.

[6] G. C. Necula, "Translation validation for an optimizing compiler," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 83–94, 2000.

[7] P. Ginsbach, B. Collie, and M. F. O'Boyle, "Automatically harnessing sparse acceleration," in *Proceedings of the 29th International Conference on Compiler Construction*, 2020, pp. 179–190.

[8] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, 2017.

[9] J. Clune, V. Ramamurthy, R. Martins, and U. A. Acar, "Program equivalence for assisted grading of functional programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[10] N. Dershowitz, "Computing with rewrite systems," *Information and Control*, vol. 65, no. 2-3, pp. 122–157, 1985.

[11] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proc. ACL*, 2017.

[12] GitHub. (2021). "The 2020 state of the octoverse," [Online]. Available: https://octoverse.github.com/ (visited on 03/15/2021).

[13] J. Cocke, "Global common subexpression elimination," in *Proceedings of a symposium on Compiler optimization*, 1970, pp. 20–24.

[14] T. Kasampalis, D. Park, Z. Lin, V. S. Adve, and G. Roşu, "Language-parametric compiler validation with application to llvm," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 1004–1019.

[15] K. Banerjee, C. Mandal, and D. Sarkar, "Extending the scope of translation validation by augmenting path based equivalence checkers with smt solvers," in *18th International Symposium on VLSI Design and Test*, IEEE, 2014, pp. 1–6.

[16] S. Kommrusch, *S4Eq Software*, https://github.com/SteveKommrusch/PrgEq, 2021.

[17] O. Ibarra and S. Moran, "Probabilistic algorithms for deciding equivalence of straight-line programs," *J. ACM*, vol. 30, pp. 217–228, Jan. 1983.

[18] V. S. Pai and S. Adve, "Code transformations to improve memory parallelism," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, IEEE, 1999, pp. 147–155.

[19] J. Cong, M. Huang, P. Pan, Y. Wang, and P. Zhang, "Source-to-source optimization for hls," in *FPGAs for Software Programmers*, Springer, 2016, pp. 137–163.

[20] A. Zaks and A. Pnueli, "Program analysis for compiler validation," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2008, pp. 1–7.

[21] S. Kommrusch, T. Barollet, and L.-N. Pouchet, "Equivalence of dataflow graphs via rewrite rules using

a graph-to-sequence neural model," *arXiv preprint arXiv:2002.06799*, 2020.

[22] S. Kommrusch, "Machine learning for computer aided programming: From stochastic program repair to verifiable program equivalence," Ph.D. dissertation, Colorado State University, 2021.

[23] M. J. Ciesielski, P. Kalla, Z. Zheng, and B. Rouzeyre, "Taylor expansion diagrams: A compact, canonical representation with applications to symbolic verification," in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, IEEE, 2002, pp. 285–289.

[24] G. C. Necula, "Translation validation for an optimizing compiler," *SIGPLAN Not.*, vol. 35, no. 5, pp. 83–94, May 2000.

[25] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal, "Verification of loop and arithmetic transformations of array-intensive behaviors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 11, pp. 1787–1800, 2013.

[26] W. Bao, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan, "Polycheck: Dynamic verification of iteration space transformations on affine programs," in *ACM SIGPLAN Notices*, ACM, vol. 51, 2016, pp. 539–554.

[27] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 127–138.

[28] S. Khan, "Properties of matrix multiplication," *Khan Academy (accessed May 20, 2020)*, May 2020.

[29] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 837–847.

[30] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[31] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to sequence learning with neural networks," *Advances in NIPS*, 2014.

[32] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[33] R. Nallapati, B. Zhou, C. Gulcehre, B. Xiang, *et al.*, "Abstractive text summarization using sequence-to-sequence rnns and beyond," *arXiv preprint arXiv:1602.06023*, 2016.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[35] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015.

[36] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," *arXiv preprint arXiv:1312.6211*, 2013.

[37] T. G. Dietterich, "Ensemble methods in machine learning," in *Proceedings of the First International Workshop on Multiple Classifier Systems*, ser. MCS '00, Berlin, Heidelberg: Springer-Verlag, 2000, pp. 1–15.

[38] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba, "Hindsight experience replay," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 5048–5058.

[39] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[40] P. Zhao and J. N. Amaral, "Ablego: A function outlining and partial inlining framework: Research articles," *Softw. Pract. Exper.*, vol. 37, no. 5, pp. 465–491, Apr. 2007.

[41] L. Prechelt, "Early stopping-but when?" In *Neural Networks: Tricks of the trade*, Springer, 1998, pp. 55–69.

[42] S. Kommrusch, T. Barollet, and L.-N. Pouchet, "Proving equivalence between complex expressions using graph-to-sequence neural models," *CoRR*, vol. abs/2106.02452, 2021.

[43] F. Mora, Y. Li, J. Rubin, and M. Chechik, "Client-specific equivalence checking," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, Montpellier, France: Association for Computing Machinery, 2018, pp. 441–451.

[44] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *ArXiv*, vol. abs/2001.08361, 2020.

[45] S. Verdoolaege, G. Janssens, and M. Bruynooghe, "Equivalence checking of static affine programs using widening to handle recurrences," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 34, no. 3, p. 11, 2012.

[46] C. Alias and D. Barthou, "On the recognition of algorithm templates," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 2, pp. 395–409, 2004.

[47] D. Barthou, P. Feautrier, and X. Redon, "On the equivalence of two systems of affine recurrence equations," in *Euro-Par 2002 Parallel Processing*, 2002.

[48] G. Iooss, C. Alias, and S. Rajopadhye, "On program equivalence with reductions," in *International Static Analysis Symposium*, Springer, 2014, pp. 168–183.

[49] M. Schordan, P.-H. Lin, D. Quinlan, and L.-N. Pouchet, "Verification of polyhedral optimizations with constant loop bounds in finite state space computations," in *Proc. of the 6th International Symposium On*

Leveraging Applications of Formal Methods, Verification and Validation, Springer, 2014.

[50] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1027–1040.

[51] S. Badihi, F. Akinotcho, Y. Li, and J. Rubin, "Ardiff: Scaling program equivalence checking via iterative abstraction and refinement of common code," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 13–24.

[52] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[53] L. C. Paulson, *Isabelle Page*, https://www.cl.cam.ac.uk/research/hvg/Isabelle.

[54] B. Steffen, "Data flow analysis as model checking," in *International Symposium on Theoretical Aspects of Computer Software*, Springer, 1991, pp. 346–364.

[55] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of c and verilog programs using bounded model checking," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*, IEEE, 2003, pp. 368–371.

[56] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated software engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[57] K. S. Namjoshi and R. P. Kurshan, "Syntactic program transformations for automatic abstraction," in *International Conference on Computer Aided Verification*, Springer, 2000, pp. 435–449.

[58] S. Kalvala, R. Warburton, and D. Lacey, "Program transformations using temporal logic side conditions," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 4, p. 14, 2009.

[59] W. Mansky and E. Gunter, "A framework for formal verification of compiler optimizations," in *Interactive Theorem Proving*, Springer, 2010.

[60] E. Visser, "Program transformation with stratego/xt," in *Domain-specific program generation*, Springer, 2004, pp. 216–238.

[61] D. Lucanu and V. Rusu, "Program equivalence by circular reasoning," *Formal Aspects of Computing*, vol. 27, no. 4, pp. 701–726, 2015.

[62] U. S. Reddy, "Rewriting techniques for program synthesis," in *International Conference on Rewriting Techniques and Applications*, Springer, 1989, pp. 388–403.

[63] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021.

[64] A. S. Murawski and J. Ouaknine, "On probabilistic program equivalence and refinement," in *International Conference on Concurrency Theory*, Springer, 2005, pp. 156–170.

[65] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet, "Approximate probabilistic model checking," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, 2004, pp. 73–84.

[66] V. Gogate and P. Domingos, "Probabilistic theorem proving," *arXiv preprint arXiv:1202.3724*, 2012.

[67] S. Badihi, Y. Li, and J. Rubin, "Eqbench: A dataset of equivalent and non-equivalent program pairs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 610–614.

[68] Y. Luo, L. Yin, W. Bai, and K. Mao, "An appraisal of incremental learning methods," *Entropy*, vol. 22, no. 11, 2020.

[69] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," *CoRR*, vol. abs/2105.04123, 2021.

[70] W. Ding, "Exploring the possibilities of applying transfer learning methods for natural language processing in software development," M.S. thesis, Technische Universität München, 2021.

[71] S. Huang, X. Zhou, and S. Chin, "Application of seq2seq models on code correction," *Frontiers in artificial intelligence*, vol. 4, p. 590 215, 2021.

[72] A. Fawzi, M. Malinowski, H. Fawzi, and O. Fawzi, "Learning dynamic polynomial proofs," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 4179–4188.

[73] K. Bansal, S. Loos, M. Rabe, C. Szegedy, and S. Wilcox, "HOList: An environment for machine learning of higher order logic theorem proving," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, Long Beach, California, USA: PMLR, 2019, pp. 454–463.

[74] A. Paliwal, S. Loos, M. Rabe, K. Bansal, and C. Szegedy, "Graph Representations for Higher-Order Logic and Theorem Proving," *arXiv e-prints*, arXiv:1905.10006, arXiv:1905.10006, May 2019.

[75] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, "Decision transformer: Reinforcement learning via sequence modeling," *CoRR*, vol. abs/2106.01345, 2021.

[76] G. Lample and F. Charton, "Deep learning for symbolic mathematics," in *International Conference on Learning Representations*, 2020.

[77] A. A. Mali, A. G. O. II, D. Kifer, and C. L. Giles, "Recognizing and verifying mathematical equations using multiplicative differential neural units," in *35th AAAI Conference on Artificial Intelligence*, AAAI Press, 2021, pp. 5006–5015.

[78] M. Crouse, I. Abdelaziz, B. Makni, S. Whitehead, C. Cornelio, P. Kapanipathi, K. Srinivas, V. Thost, M. Witbrock, and A. Fokoue, "A deep reinforcement learning approach to first-order logic theorem proving," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 7, pp. 6279–6287, 2021.

[79] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and natu-

ralness," *ACM Comput. Surv.*, vol. 51, no. 4, 81:1–81:37, Jul. 2018.

[80] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 40:1–40:29, Jan. 2019.

[81] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, 19:1–19:29, Sep. 2019.

[82] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "DIRE: A neural approach to decompiled identifier naming," in *International Conference on Automated Software Engineering*, ser. ASE '19, 2019.

[83] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, Mumbai, India: ACM, 2015, pp. 111–124.

[84] R. Bavishi, M. Pradel, and K. Sen, *Context2name: A deep learning-based approach to infer natural variable names from usage contexts*, 2017.

[85] P. Bielik, V. Raychev, and M. Vechev, "Phog: Probabilistic model for code," in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, Jun. 2016, pp. 2933–2942.

[86] Q. Wang, K. Zhang, A. G. Ororbia II, X. Xing, X. Liu, and C. L. Giles, "An empirical evaluation of rule extraction from recurrent neural networks," *Neural Comput.*, vol. 30, no. 9, pp. 2568–2591, Sep. 2018.

[87] M. Tomita, "Dynamic construction of finite automata from examples using hill-climbing," in *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, Ann Arbor, Michigan, 1982, pp. 105–108.

[88] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of neural program models with respect to semantic-preserving program transformations," *Information and Software Technology*, vol. 135, p. 106 552, 2021.

[89] N. D. Q. Bui, "Efficient Framework for Learning Code Representations through Semantic-Preserving Program Transformations," *arXiv e-prints*, arXiv:2009.02731, arXiv:2009.02731, Sep. 2020.

[90] N. D. Q. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 511–521.

[91] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," in *NeurIPS*, 2021.

Steve Kommrusch is currently a PhD candidate focused on machine learning at Colorado State University. He received his BS in computer engineering from University of Illinois in 1987 and his MS in EECS from MIT in 1989. From 1989 through 2017, he worked in industry at Hewlett-Packard, National Semiconductor, and Advanced Micro Devices. Steve holds over 30 patents in the fields of computer graphics algorithms, silicon simulation and debug techniques, and silicon performance and power management. His research interests include Program Equivalence, Program Repair, and Constructivist AI using machine learning. More information available at: https://www.cs.colostate.edu/ steveko/.

Martin Monperrus is Professor of Software Technology at KTH Royal Institute of Technology. He was previously associate professor at the University of Lille and adjunct researcher at Inria. He received a Ph.D. from the University of Rennes, and a Master's degree from the Compiegne University of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, program hardening and chaos engineering.

Louis-Noël Pouchet is an Associate Professor of Computer Science at Colorado State University, with a joint appointment in the Electrical and Computer Engineering department. He is working on pattern-specific languages and compilers for scientific computing, and has designed numerous approaches using optimizing compilation to effectively map applications to CPUs, GPUs, FPGAs and System-on-Chips. His work spans a variety of domains including compiler optimization design especially in the polyhedral compilation framework, high-level synthesis for FPGAs and SoCs, and distributed computing. He is the author of the PolyOpt and PoCC polyhedral compilers, and of the PolyBench benchmarking suite.