DEEP SYMBOLIC SUPEROPTIMIZATION WITHOUT HUMAN KNOWLEDGE

Anonymous authors
Paper under double-blind review

ABSTRACT

Deep symbolic superoptimization refers to the task of applying deep learning methods to simplify symbolic expressions. Existing approaches either perform supervised training on human-constructed datasets that defines equivalent expression pairs, or apply reinforcement learning with human-defined equivalent transformation actions. In short, almost all existing methods rely on human knowledge to define equivalence, which suffers from large labeling cost and learning bias, because it is almost impossible to define and comprehensive equivalent set. We thus propose HISS, a reinforcement learning framework for symbolic superoptimization that keeps human outside the loop. HISS introduces a tree-LSTM encoder-decoder network with attention to ensure tractable learning. Our experiments show that HISS can discover more simplification rules than existing human-dependent methods, and can learn meaningful embeddings for symbolic expressions, which are indicative of equivalence.

1 INTRODUCTION

Superoptimization refers to the task of simplifying and optimizing over a set of machine instructions, or code (Massalin, 1987; Schkufza et al., 2013), which is a fundamental problem in computer science. As an important direction in superoptimization, symbolic expression simplification, or symbolic superoptimization, aims at transforming symbolic expression to a simpler form in an effective way, so as to avoid unnecessary computations. Symbolic superoptimization is an important component in compilers, e.g. LLVM and Halide, and it also have a wide application in mathematical engines including Wolfram[1], Matlab, and Sympy.

Over the recent years, applying deep learning methods to address symbolic superoptimization has attracted great attention. Despite their variety, existing algorithms can be roughly divided into two categories. The first category is supervised learning, *i.e.* to learn a mapping between the input expressions and the output simplified expressions from a large number of human-constructed expression pairs (Arabshahi et al., 2018; Zaremba & Sutskever, 2014). Such methods rely heavily on a human-constructed dataset, which is time- and labor-consuming. What is worse, such systems are highly susceptible to bias, because it is generally very hard to define a minimum and comprehensive axiom set for training. It is highly possible that some forms equivalence are not covered in the training set, and fail to be recognized by the model. In order to remove the dependence on human annotations, the second category of methods leverages reinforcement learning to autonomously discover simplifying equivalence (Chen et al., 2018). However, to make the action space tractable, such systems still rely on a set of equivalent transformation actions defined by human beings, which again suffers from the labeling cost and learning bias.

In short, the existing neural symbolic superoptimization algorithms all require human input to define equivalence. It would have benefited from improved efficiency and better simplification if there were algorithms independent of human knowledge. In fact, symbolic superoptimization should have been a task that naturally keeps human outside the loop, because it directly operates on machine code, whose consumers and evaluators are machines, not humans.

Therefore, we propose Human-Independent Symbolic Superoptimization (HISS), a reinforcement learning framework for symbolic superoptimization that are completely independent of human

[1] `https://www.wolframalpha.com/`

knowledge. Instead of using human-defined equivalence, HISS adopts a set of unsupervised techniques to maintain the tractability of action space. First, HISS introduces a tree-LSTM encoder-decoder architecture with attention to ensure that its exploration is confined within the set syntactically correct expressions. Second, the process of generating a simplified expression is broken into two stages. The first stage selects a sub-expression that can be simplified and the second stage simplifies the sub-expression. We performed a set of evaluations on artificially generated instruction as well as publicly available code datasets, and show that HISS can achieve compeititive performance. We also find out that HISS can obtain meaningful embeddings for symbolic expressions, in the sense that equivalent expressions have closer embeddings than do non-equivalent expressions.

2 RELATED WORKS

Superoptimization origins from 1987 with the first design of Massalin (1987). With the probabilistic testing to reduce the testing cost, the brute force searching is aided with pruning strategy to avoid searching sub-spaces that contains pieces of code that have known shorter alternatives. Due to the explosive searching space for exhaustive searching, the capability of the first superoptimizer is limited to only very short programs. More than a decades later, Joshi et al. (2002) presented Denali, which splits the superoptimization problem to two phases to expand the capability to optimize longer programs. STOKE (Schkufza et al., 2013) follows the two phase, but sacrifices the completeness for efficiency at the second phase.

Recent attempts to improve superoptimization are categorized to two fields: exploring transformation rules and accelerating trajectory searching. Searching the rules are similar to the problem of superoptimization on limited size program, but targeting more on comprehensiveness of the rules. Buchwald (2015) exhaustively enumerates all possible expressions given the syntax, and checks the equivalence of pairs of expressions by SMT solver. A similar method with adaption of the SMT solver to reuse the previous result is proposed by Jangda & Yorsh (2017). On the other hand, deep neural networks are trained to guide the trajectory searching (Cai et al., 2018; Chen & Tian, 2018).

Considering transformation rule discovery as a limited space superoptimization, the large action space and sparse reward are the main challenge for using neural network. Special neural generator structures are proposed for decoding valid symbolic programs, which leverage the syntax constrains to reduce the searching space as well as learn the reasoning of operations, and are gaining popularity in program synthesis (Parisotto et al., 2016; Zhong et al., 2017; Bunel et al., 2018), program translation (Chen et al., 2018; Drissi et al., 2018), and other code generation tasks (Ling et al., 2016; Alvarez-Melis & Jaakkola, 2016). Among the symbolic expression decoders, the family of tree structure RNNs (Parisotto et al., 2016; Drissi et al., 2018; Alvarez-Melis & Jaakkola, 2016; Chen et al., 2018) are more flexible than template-based predictors (Ling et al., 2016; Zhong et al., 2017).

3 THE HISS ARCHITECTURE

In this section, we will detail our proposed HISS architecture. We will first introduce a few notations. $\mathcal{T}$ denotes a tree; $\boldsymbol{a}$ denotes a vector, and $\boldsymbol{A}$ denotes a matrix. We introduce an LSTM$(\cdot)$ function that summarizes standard one-step LSTM operation as

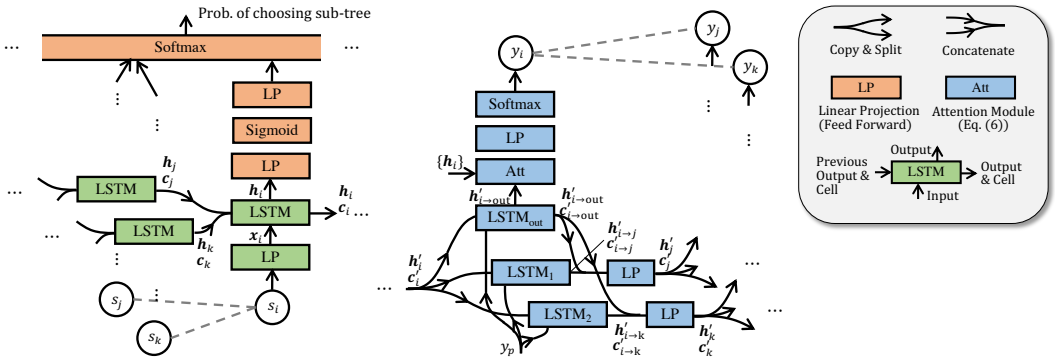
$$[\boldsymbol{h}_t, \boldsymbol{c}_t] = \text{LSTM}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{c}_{t-1}), \tag{1}$$

where $\boldsymbol{h}_t$, $\boldsymbol{c}_t$ and $\boldsymbol{x}_t$ denote the output, cell and input at time t of a standard LSTM respectively. This notation is very helpful for us to introduce our tree-LSTM structure in the subsequent subsections.

3.1 THE OVERALL FRAMEWORK

Our problem can be formulated as follows. Given a symbolic expression $\mathcal{T}_I$, represented in the *expression tree form*, our goal is to find a simplified expression $\mathcal{T}_O$, such that 1) the two expressions are equivalent; and 2) $\mathcal{T}_O$ contains smaller number of nodes than $\mathcal{T}_I$.

It is important to write the symbolic expressions in their expression tree form, rather than strings, because HISS will be operating on tree structures. An expression tree assigns a node for each operation or variable. Each non-terminal node represents an operation, and each terminal node, or leaf node, represents a variable or a constant. The arguments of an operation are represented as the



(a) The tree encoder (green) and sub-tree selector (orange).

(b) The tree decoder.

Figure 1: The HISS architecture, illustrated on a three-node binary sub-tree, where i is the parent of j, k, and p is the parent of i.

descendant sub-trees of the corresponding node. Compared to string representation, tree representation naturally ensures any randomly generated expression in its form is syntactically correct. It also makes working with sub-expressions easier – simply by working with sub-trees.

HISS approaches the problem using the reinforcement learning framework, where the action of generating simplified expressions are divided into two consecutive actions. The first action is to pick a sub-expression (or sub-tree) that can be simplified, and the second action generates the simplified expression for the selected sub-expression.

Accordingly, HISS contains three modules. The first module is a **tree encoder**, which computes an embedding for each sub-tree (including the entire tree) of the input expression. The embeddings are useful for picking sub-tree for simplification as well as simplifying a sub-tree. The second module is a **sub-tree selector**, which selects a sub-tree for simplification. The third module is a **tree decoder** with attention mechanism, which generates a simplified expression based on the input sub-tree embedding. The subsequent subsections will introduce each module respectively.

3.2 THE TREE ENCODER

The tree encoder generates embedding for every sub-tree of the input expression. We apply the N-ary Tree LSTM as proposed in Tai et al. (2015), where N represents the maximum number of arguments that an operation has. It is important to note that although different operations have different number of arguments, for structural uniformity, we assume that all operations have N arguments, with the excessive arguments being an NULL symbol.

The tree encoder consists of two layers. The first layer is called the embedding layer, which is a fully-connected layer that converts the one-hot representation of each input symbol to an embedding. The second layer is the tree LSTM layer, which is almost the same as the regular LSTM, except that the cell information now flows from the children nodes to their parent node. Formally, denote $\boldsymbol{c}_i$, $\boldsymbol{h}_i$ and $\boldsymbol{x}_i$ as the cell, output and input of node i respectively. Then the tree LSTM encoder performs the following information

$$[\boldsymbol{h}_i, \boldsymbol{c}_i] = \text{LSTM}\left(\boldsymbol{x}_i, \bigcup_{j \in \mathbb{D}(i)} \boldsymbol{h}_j, \bigcup_{j \in \mathbb{D}(i)} \boldsymbol{c}_j\right), \tag{2}$$

where $\mathbb{D}(i)$ denotes the set of children of node i. Fig. 1(a) plots the architecture of the tree LSTM encoder (in green). Since each node fuses the information from its children, which again fuse the information from their own children, it is easy to see that the output $\boldsymbol{h}_i$ summarizes the information of the entire sub-tree led by node i, and thus can be regarded as an embedding for this sub-tree.

3.3 THE SUB-TREE SELECTOR

The sub-tree selector preforms the first action to select a sub-tree for simplification. It takes the output of the tree encoder, $\{\boldsymbol{h}_i\}$, as its input, and produces the probability with which each sub-tree

is selected. It consists of two feed forward layers followed by a softmax layer across all the nodes in the input tree. Fig. 1(a) shows the architecture of the sub-tree selector (in orange).

3.4 THE TREE DECODER

Once the sub-tree selector has selected a sub-tree, and suppose the root node of the selected sub-tree is node i, the output of the encoder at node i, $\boldsymbol{h}_i$, is then fed into the tree decoder, which generates a simplified version of the sub-tree. The tree decoder can be regarded as the inverse process of the tree encoder – the latter fuses information from the children to the parents, whereas the former unrolls the information from parents all the way down to the entire N-ary tree.

The tree decoder adopts a novel LSTM architecture with attention, which, compared with the attention LSTM proposed by Chen et al. (2018), is more parameter- and computationally-efficient. It consists of two layers. The first layer is a tree LSTM layer, and the second layer is the symbol generation layer with attention. Fig. 1(b) illustrates the decoder structure.

Tree LSTM Layer The tree LSTM in the decoder needs to accomplish two tasks. First, it needs to extract the information for generating the output for the current node. Second, it needs to split and pass on the information to its children. To better control the information flow, we introduce two tracks of LSTMs for the two different tasks. Formally, denote $[\boldsymbol{h}'_i, \boldsymbol{c}'_i]$ as the output and cell of node i, and assume $[j_1, \cdots, j_N]$ are children nodes of i. Also denote $\boldsymbol{y}_p$ as the decoder output for node p, which is the parent node of node i (If node i is already the root node of the selected sub-tree, then $\boldsymbol{y}_p$ becomes a special start token ¡s¿). Then the first LSTM track extracts the information that generates the current output:

$$[\boldsymbol{h}'_{i\to\text{out}}, \boldsymbol{c}'_{i\to\text{out}}] = \text{LSTM}_{\text{out}}(\boldsymbol{y}_p, \boldsymbol{h}'_i, \boldsymbol{c}'_i). \tag{3}$$

The second LSTM track splits and passes on the information to the children, *i.e.* $\forall n \in \{1, \cdots, N\}$

$$[\boldsymbol{h}'_{i\to j_n}, \boldsymbol{c}'_{i\to j_n}] = \text{LSTM}_n(\boldsymbol{y}_p, \boldsymbol{h}'_i, \boldsymbol{c}'_i). \tag{4}$$

Notice that we have appended a subscript to the LSTM$(\cdot)$ to emphasize that LSTM functions with different subscripts do not share parameters. Finally, the LSTM information for a specific children is derived by linearly projecting the output track and that specific children track:

$$\boldsymbol{h}'_{j_n} = \boldsymbol{W}_h[\boldsymbol{h}'_{j\to\text{out}}, \boldsymbol{h}'_{i\to j_n}] + \boldsymbol{b}_h, \quad \boldsymbol{c}'_{j_n} = \boldsymbol{W}_c[\boldsymbol{c}'_{j\to\text{out}}, \boldsymbol{c}'_{i\to j_n}] + \boldsymbol{b}_c. \tag{5}$$

We find that this linear projection is useful for adding additional dependencies between the parent output and the descendants, so that the generated expression is more coherent.

Symbol Generation Layer with Attention The symbol generation layer takes the output track produced by the previous tree LSTM layer, $\boldsymbol{h}'_{i\to\text{out}}$, as input, and outputs the probability distribution of generating different output symbols for the current node. It adopts an attention mechanism (Bahdanau et al., 2014) to attend to the relevant part in the encoder, so that the input and output expressions have better correspondence. Formally, when generating the output for decoder node i, the attention weight on encoder node j is computed from $\boldsymbol{h}'_{i\to\text{out}}$ and $\boldsymbol{h}_j$ as follows:

$$\begin{aligned} &e_i(j) = \boldsymbol{v}^T \tanh(\boldsymbol{W}_d \boldsymbol{h}'_{i\to\text{out}} + \boldsymbol{W}_e \boldsymbol{h}_j + \boldsymbol{b}_a), \\ &[a_i(1), \cdots, a_i(J)] = \text{softmax}([e_i(1), \cdots, e_i(J)]), \end{aligned} \tag{6}$$

where J is the total number of input nodes at the encoder. Finally, the probability of symbol generation at node i, denoted as $\boldsymbol{p}_i$, is computed by passing into a linear projection layer $\boldsymbol{h}'_{i\to\text{out}}$ and an attention context vector $\boldsymbol{c}_i$, which is a linear combination of the encoder embeddings with the attention weights, *i.e.*

$$\boldsymbol{p}_i = \boldsymbol{W}_o[\boldsymbol{h}'_{i\to\text{out}}; \boldsymbol{c}_i] + \boldsymbol{b}_o, \quad \text{where } \boldsymbol{c}_i = \sum_{j=1}^{J} a_i(j)\boldsymbol{h}_j. \tag{7}$$

4 LEARNING WITH HISS

In this section, we will elaborate the training and inference schemes of HISS. In particular, we will introduce several mechanisms to improve the exploration efficiency of HISS.

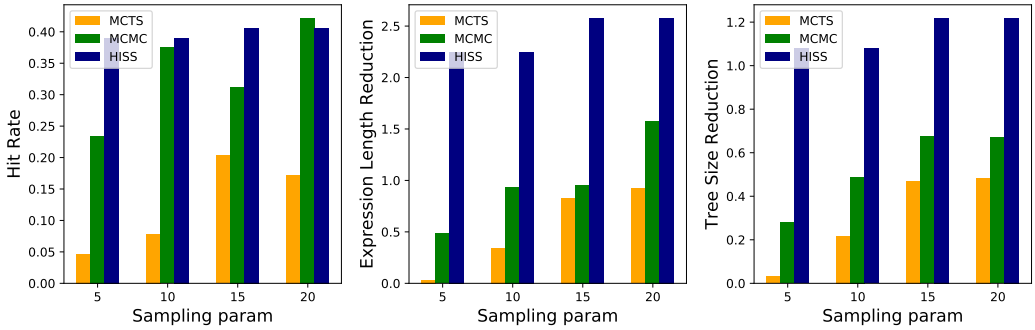


Figure 2: Comparison with human-independent methods in terms of hit rate (left), expression length reduction (middle), tree size reduction (right), on randomly generated short expressions.

4.1 TRAINING

We apply the standard REINFORCE framework (Williams, 1992) for training, where the reward function function is given by

$$R(\mathcal{T}_I, \mathcal{T}_O) = \gamma^{\text{card}(\mathcal{T}_O)} \text{ if } \mathcal{T}_I \equiv \mathcal{T}_O, \quad -0.1\gamma^{\text{card}(\mathcal{T}_O)} \text{ otherwise,} \tag{8}$$

where '$\equiv$' denotes that the two expressions are equivalent; $\text{card}(\cdot)$ denotes the number of nodes in the tree expression, or equivalently the length of the expression. This reward prioritizes equivalence, and given equivalence, favors shorter expressions. We applied a probabilistic testing scheme to determine equivalence as proposed in Massalin (1987). We introduce the following modifications to the standard REINFORCE algorithm to maintain the efficiency and stability of training.

Curriculum Learning Since generating the simplified expression is divided into two actions, sub-tree selection and sub-tree simplification, directly learning both can lead to very inefficient exploration. Instead, we introduce a two-stage curriculum. The first stage trains only the encoder and decoder on very short expressions (maximum depth less than four). The sub-tree selector is not trained. Instead we always feed the entire tree to the decoder for simplification. The second stage trains all the modules on longer expressions.

Sub-tree Embedding Similarity In order to guide the encoder to learning meaning embeddings, we introduce an additional ℓ_2 loss to enforce that the equivalent expressions have similar encoder embeddings, *i.e.* similar $\boldsymbol{h}_i$s . Specifically, for each input expressions $\mathcal{T}_I$, we decode a set of generated expressions $\mathbb{S} = \{\mathcal{T}_O\}$ with beam search, and obtain their embeddings $\{\boldsymbol{h}(\mathcal{T}_O)\}$ by feeding them back into the encoder (here we add an argument to $\boldsymbol{h}$ to emphasize that the embedding is a function of input expression). Then the ℓ_2 loss is expressed as follows:

$$L = \frac{1}{|\mathbb{S}|} \sum_{\mathcal{T}_O \in \mathbb{S}} \|\boldsymbol{h}(\mathcal{T}_O) - \boldsymbol{h}(\mathcal{T}_I)\|_2^2 \cdot (-1)^{\mathbb{1}[\mathcal{T}_I \neq \mathcal{T}_O]}, \tag{9}$$

where $\mathbb{1}[\cdot]$ denotes the indicator function, which equals one if the statement in its argument is true, and zero otherwise. Note that this ℓ_2 applies to the encoder only, and can be optimized by regular gradient descent methods. REINFORCE is not needed.

4.2 INFERENCE

Similar to training, the inference is performed by feeding the input expression to HISS and find the best results among the multiple outputs. In order to accelerate the inference process, we introduce an offline procedure. During the first stage of the curriculum training, *i.e.* training on very short expressions, all the simplified equivalence discovered are logged. During inference, if the sub-tree to be fed into the decoder has an exact match in the log, we will apply the logged simplified equivalence directly, rather than redoing the entire decoding process.

5 EXPERIMENT

We have performed two experiments. The first experiment compares HISS with human-independent naive search algorithms. The second experiment compares HISS with existing human-dependent state-of-the-art algorithms on benchmark datasets.

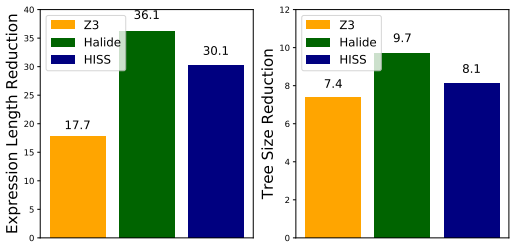


Figure 3: Comparison with human-dependent methods in terms of expression length reduction (left), and tree size reduction (right), on random expressions generated by Halide's generation tool

5.1 Comparing with Human-Independent Methods

Since there are no existing human-independent methods specifically for symbolic superoptimization, we compare several search algorithms. Due to the search complexity, the evaluation can only be performed on very short expressions (maximum depth is three). Therefore, our stage-one model in the curriculum learning (the one with no sub-tree selection) suffices to perform the task.

Baselines Two baseline searching methods are compared: Monte Carlo Tree Search (MCTS) (Bertsekas, 1995) and Markov Chain Monte Carlo (MCMC) (Schkufza et al., 2013). MCTS decides the expression tree from root to leaves, and adopts Upper Confidence Bound (Kocsis & Szepesvári, 2006) for balancing exploration and exploitation. Similar to Schkufza et al. (2013), MCMC takes one of four transformations: 1) replace a operator by another random operator, and generate or discard operands if two operator takes different number of operands. 2) replace a variable/constant with another random variable/constant. 3) replace a sub-expression with a random single variable/constant. 4) replace a variable/constant with a random expression. The probability distribution of taking the transformation is defined as same as in Schkufza et al. (2013).

Dataset The random expression is generated from Halide Intermediate Representation (IR) syntax, which contains 16 operators, taking one to three operands. To limit the searching space, the operands are limited to five common constants (0, 1, 2, true, false), and 39 variables. We applied brute force method to select only the expression that matches to a simpler equivalent within some computation budget. The dataset contains short expressions of depth two to three, length two to 29. Notice that even the dataset is drafted using brute force matching method, the brute force matching is extremely time consuming, and definitely fails to find the equivalent ones in most of time due to vast searching space. Also, the matched equivalent is not known to the models in any way.

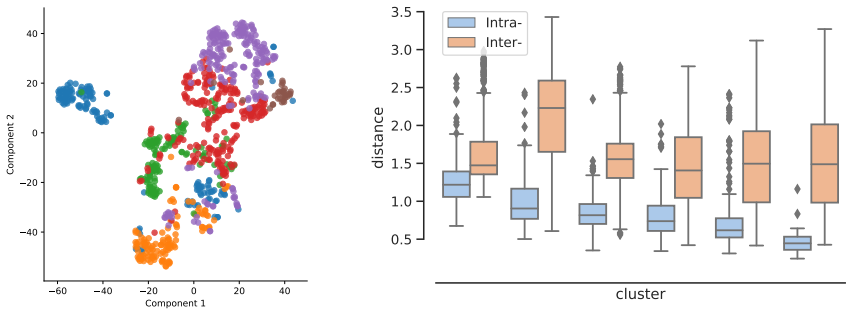
Metrics Three metric are introduced: 1) *hit rate*, defined as percentage of expressions that the model successfully found an equivalence given the computation budget parameter; 2) *expression length reduction*, defined as reduction in the total number of tokens; and 3) *tree size reduction*, defined as reduction in the number of nodes in the expression tree.

Results The performance comparison of three models is shown in Fig. 2. The sampling parameter in the horizontal axis refers to the beam size for HISS , the max trials budget for MCTS for each token decoded, and the sampling budget for MCMC. These quantities equivalently define the number of search attempts per token. As can be seen, HISS is significantly more powerful in finding the simpler equivalent than MCTS and MCMC. MCMC performs almost equally well as HISS in terms of Hit Rate, and both of them far outperform MCTS. However, both MCTS and HISS adopt top-down decoding in the huge decoding space, while MCMC starts with the input expression and applies local transformation, which makes it much easier to find an equivalence. Also, MCMC achieves much worse average length reduction and average tree size reduction than HISS.

5.2 Comparing with Human-Dependent Methods

In this section, we compare HISS with existing human-dependent state-of-the-art methods. We use the full model of HISS, *i.e.* the stage-two model with sub-tree selection, for the comparison.

Baselines The two baselines are included: 1) *Halide* (Ragan-Kelley et al., 2013), which applies manually defined rules; 2) *Z3*, the simplification function in Z3, a high performance theorem prover developed by De Moura & Bjørner (2008), to perform transformations using its pre-defined rules.



(a) (b)

Figure 4: Evaluation of similarity of the embeddings of equivalent expressions. (a) Scatter plots of embeddings projected onto two-dimensional space using t-SNE. Points corresponding to equivalent expressions are shown in the same color. (b) Box plots of intra- (blue) and inter-subset (orange) distances of the embeddings. The bars in the box represent 25%, 50% and 75% quantile values. The line intervals detnote the 1.5 inter-quaterline range (IQR) beyond the quartertile values. The dots represent the extreme values.

Dataset The dataset contains reducible expressions generated by Halide's random expression generation tool (Ragan-Kelley et al., 2013), which is for testing the simplification function in Halide. The dataset contains long sequences The dataset is biased toward Halide simplification, as the random patterns are very likely to match to some pre-defined rules in Halide.

Constant Folding Unlike the dataset in the previous subsection, which contains only five constant values, the expressions in this dataset contains a large number of constant values. We thus apply a constant folding technique to HISS as it is also performed in all other baselines. Specifically, once the expression is rewritten by the neural network in symbolic domain, it will be checked if all the leaf nodes in the sub-tree are constant. If so, the expression is executed and replaced by a new single node with the execution result. Constant folding is applied in both training and inference.

Metrics Expression length reduction and tree size reduction are applied as the metrics.

Results HISS outperformed Z3 in both metrics, but slightly worse than Halide. The reason is twofold. First, the data are generated using only Halide pre-defined rules, and thus algorithms that have access to these oracle rules have an advantage. HISS does not benefit from going beyond these predefined rules. Second, the HISS operates in symbolic domain, but the most common pattern that is simplified in the dataset is constant based. For example, in Halide rule, $((x - 64) + z) < 32$ should be rewritten to $(x + z) < (32 + 64)$. However, the transformation rule can hardly be preferred by HISS in symbolic domain, since it reduces neither the tree size or expression length in symbolic domain. Nevertheless, HISS still approaches the performance with the algorithm with oracle knowledge.

5.3 SUB-TREE EMBEDDING ANALYSIS

We would like to see if HISS can learn similar embeddings for different expressions that produce the same result. To evaluate this, in experiment 5.1, we select the six most-populated subsets of equivalent expressions in the test set, and evaluate the similarity of their embedding in two ways. First, the embeddings are further projected to two-dimensional space using t-SNE (Maaten & Hinton, 2008), which form a scatter plot as in Fig. 4(a). The points corresponding to equivalent expressions are shown in the same color. As can be seen, the embeddings equivalent expressions are highly clustered. Notice that this result is on low-dimensional projection of the embedding. To better evaluate their similarity in the original space, we compare their inter- and intra-subset distances. The inter-/intra-subset distance of a subset is defined as the Euclidean distance between the centroid of the subset and the samples outside/within the subset. Fig. 4(b) illustrated the box plot of these distances. As shown, there is a significant difference between intra- and inter-subset distances – except for the first subset, the quatertile intervals are well separated. We can therefore conclude that HISS is able to learn meaningful embeddings that are indicative of expression equivalence.

Table 1: Intuitive (left) and unintuitive (right) rewrite rules discovered by HISS.

Input	Output	Input	Output
$x\&\&\text{false}$	false	$(!\text{true}) < (y - 1)$	$!(\text{true} >= y)$
$(x + y) - x$	y	$(y - \text{true})\|\|(y >= \text{true})$	true
$0/x$	0	$\max(z, 1) >= (\text{false}\|\|x)$	true
$x + x$	$2 * x$	$(1 - z) >= (z - 1)$	$1 >= z$
y/y	1	$\min(x, \text{true})\&\&(z/2)$	$x\&\&z$
$\max(x, x)$	x	$1 - (1 < x)$	$1 >= x$
$!(0 < y)$	$0 >= y$	$\text{select}(z, z, \text{true}) == \text{select}(z, \text{false}, \text{true})$	$!z$
$!(2 - y)$	$2 == y$	$(x * y)\&\&\text{true}$	$y\&\&x$
$(-y) == y$	$y == 0$	$(y + 1) > (y\&\&2)$	$y >= 0$

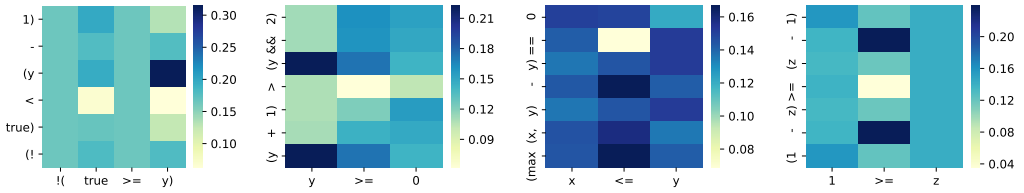


Figure 5: Attention weights on input sequence for each token decoded. The x-axis shows the output sequence, and the y-axis shows the input sequence. Input sequence is encoded from leaves to root and output sequence is decoded from root to leaves. Tokens are re-arranged in natural order for better visualization.

5.4 ATTENTION VISUALIZATION

To understand the attention mechanism, we visualize the attention to the input sequence shown in Fig. 5. We find that when decoding an operation, the attention tends to be flat (with a few exceptions in the right two figures), because it needs to understand the overall logic. This is different from machine translation or summarization, where output attention of a single word is usually focus on several input tokens. On the other hand, when decoding a variable, the model attends sharply to the corresponding variable in the input.

5.5 EXAMPLE SIMPLIFICATION RULES

In order to appreciate the ability of HISS in finding equivalence, we list some simplification rules discovered by HISS on randomly generated expressions in Table 1. There are two column groups. In the left column group, we list some intuitive rewrite rules, where we can find many axiomatic identities. Notice that most human-defined equivalence in the existing algorithms are such fundamental identities. Therefore, the left column group shows that these identities can be autonomously learnt even without human knowledge. More interestingly, the right column group shows some rewrite rules that are unintuitive to humans. It takes the authors quite a while to figure out the equivalence. These rules are hardly useful in practice, because no humans will code in this way, but it is a vivid illustration of the advantage of HISS in finding powerful simplifications beyond human knowledge.

6 CONCLUSIONS

We have presented HISS as a symbolic expression simplication algorithm that is independent of human knowledge. We demonstrated that removing the dependence on humans is advantageous for this task, because machines can autonomously figure out rewrite rules that humans fail to discover, and thus achieve comparably well simplification results. We also showed that we are one step closer to finding an equivalence-preserving embedding for symbolic expressions. Although HISS has achieved promising results, there is still much room for improvement. Although HISS has adopted several techniques to reduce the complexity of the search space, learning simplification rules on very long expressions is still challenging, which calls for the exploration on more efficient reinforcement learning algorithms as a future research direction.

REFERENCES

David Alvarez-Melis and Tommi S Jaakkola. Tree-structured decoding with doubly-recurrent neural networks. 2016.

Forough Arabshahi, Sameer Singh, and Animashree Anandkumar. Combining symbolic expressions and black-box function evaluations in neural programs. *arXiv preprint arXiv:1801.04342*, 2018.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA, 1995.

Sebastian Buchwald. Optgen: A generator for local optimizations. In *International Conference on Compiler Construction*, pp. 171–189. Springer, 2015.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Cheng-Hao Cai, Yanyan Xu, Dengfeng Ke, and Kaile Su. Learning of human-like algebraic reasoning using deep feedforward neural networks. *Biologically inspired cognitive architectures*, 25: 43–50, 2018.

Xinyun Chen and Yuandong Tian. Learning to progressively plan. *arXiv preprint arXiv:1810.00337*, 2018.

Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*, pp. 2547–2557, 2018.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, and Robert Keller. Program language translation using a grammar-driven tree-to-tree model. *arXiv preprint arXiv:1807.01784*, 2018.

Abhinav Jangda and Greta Yorsh. Unbounded superoptimization. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 78–88. ACM, 2017.

Rajeev Joshi, Greg Nelson, and Keith Randall. *Denali: a goal-directed superoptimizer*, volume 37. ACM, 2002.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

Henry Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGARCH Computer Architecture News*, volume 15, pp. 122–126. IEEE Computer Society Press, 1987.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGPLAN Notices*, volume 48, pp. 305–316. ACM, 2013.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.