Type Batched Program Reduction

Anonymous Author(s)

ABSTRACT

Given a program with a property of interest, program reduction searches for a smaller program that preserves the property and is easier to understand. Domain agnostic program reducers can reduce programs of multiple languages without extra domain knowledge. Despite their reusability, they may still take hours to run, hindering productivity and scalability. This paper proposes *type batched program reduction*, which uses machine learning to suggest portions of a program, or *batches*, that are most likely to be advantageous to reduce at a particular point in the reduction. We also extend this to jointly reduce multiple portions of a program at once, improving the performance further. Suggesting an appropriate order for removing batches from a program along with their potential simultaneous removal enables our reducer to outperform the state of the art reducers in reduction time over a set of large programs from multiple domains. This work lays foundations for further improvements in ML guided program reduction.

KEYWORDS

Program Reduction, Machine Learning, Delta Debugging

ACM Reference Format:

Anonymous Author(s). 2023. Type Batched Program Reduction. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Given a program P, *program reduction* searches for a smaller program p that preserves some property of interest [38, 45]. For instance, P may cause a compiler like GCC to crash, and a smaller program that produces the same crash can be helpful for debugging. Smaller programs are easier to understand and can speed up this task [5, 45]. Program reduction has also been useful for minimizing the attack surface of programs [11, 13, 31, 35, 42], reducing resource consumption [3], and helping to understand neural models of code [32, 33].

In general, program reduction is an *anytime algorithm*. Reducers apply a set of transformations to a program and check an oracle ψ to see whether the desired properties still hold. If they do, then the search continues from this new and smaller program. Reduction can stop at any time and return the smallest variant so far. In practice, reducers stop when they (1) see no more reduction opportunities or (2) time out [8, 34, 38].

ISSTA 2023, 17-21 July, 2023, Seattle, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

There are two types of program reducers: Domain specific [4, 6, 23, 34, 37] and domain agnostic [8, 14, 22, 38]. Domain specific reducers are capable of effectively reducing programs of a particular domain. While they tend to produce the smallest programs, they require significant effort and expertise to implement. This limits their reusability and availability for reducing inputs in other languages. In contrast, domain agnostic reducers can reduce any program when given a parse tree for the program. Given the tree, they apply reduction operations on the nodes of the tree. The generality of these reducers makes their use feasible on multiple domains with little or no extra effort. However, since these reducers do not leverage any domain knowledge, they are not aware of the semantics of the program being reduced and may generate semantically invalid program candidates during reduction. For example, a domain agnostic program reducer such as Perses [38] or Pardis [8] may try to remove the definition of a function from a program while its call site is still present within the program. Breaking these dependencies violates the semantic validity of programs and makes reducers spend a significant amount of time on useless and invalid program candidates. Reduction processes that take hours instead of minutes can disrupt developer workflow and hinder productivity [28]. They can also limit the scalability of emerging uses for program reduction [3, 32].

In this paper, we propose *type batched program reduction* to improve the performance of domain agnostic program reduction. In particular, we note that traditional domain agnostic reducers traverse the parse tree in orders that can hinder successful reduction. These reducers traverse the tree from the top down to visit nodes with a larger number of descendants earlier. However, removing such nodes can often fail due to dependencies within the program. Our approach partitions the nodes of the tree into *batches* and uses machine learning to select the batches of nodes that are estimated to be the most effective at reducing the program. We leverage the grammar of the input to form these partitions based on the *rule types* of nodes. In practice, we find that our batches explore the tree in orders that are less likely to break dependencies. Unlike traditional techniques, the traversal orders suggested by our approach enable visiting nodes at different levels and locations within the tree. Additionally, selecting the most advantageous nodes increases the probability of successfully removing them, making it more fruitful to simultaneously reduce over multiple portions of the tree.

More specifically, we make the following contributions:

- We train time-varying machine learning models to predict the probability of removal success for different node types. We use these to assess the profitability of batches of nodes for reduction.
- We develop a program reduction algorithm that prioritizes the most profitable batches of nodes for reduction.
- We extend this technique to perform a *probabilistic joint reduction*, reducing simultaneously over multiple portions of the program.

Our type batched reducer improves the average reduction time of two state of the art domain agnostic program reducers, Perses [38] and Pardis [8] by 34%-57% and 29%-49% on large real world C and Rust programs and performs 6%-20% fewer tests on Go programs.

2 BACKGROUND AND MOTIVATION

State of the art techniques for domain agnostic program reduction work by traversing a parse tree or abstract syntax tree (AST) of the program being reduced [8, 14, 38]. The internal nodes of the parse tree are labeled with a *type* corresponding to the rules of a formal grammar for the parsed language [29]. They capture the syntactic structure of the program, while the leaves capture the individual tokens or elements of the original program [29]. When visiting a node during this traversal, a reduction technique decides to take an action to create smaller candidate programs. Actions on a node can include operations like (1) removing that node, (2) replacing the node with a compatible descendant, or (3) removing a subset of children from that node [8, 14, 38]. If the candidate tree still reproduces the property of interest, then the traversal continues over the parse tree of the modified candidate. In general, the search can be guided by attributes like the number of descendants to help improve the performance of reduction [8, 38].

Existing traversal based techniques provide a clear framework for program reduction, but they often explore parse trees in orders that prevent successfully finding desirable smaller programs. This can significantly harm either the running time or the quality of the reduction. For example, trying to remove the definition of a function or type within a program before removing their uses will typically result in a program that cannot compile.

To understand the drawbacks of traversal based techniques, consider the program in Listing 1 with its parse tree shown in Figure 1. The property of interest to preserve when reducing this program is printing `Hello World!` (line 13).

Listing 1: C program with a statement to preserve on line 13.

```
1  struct S {
2      int f1;
3      int f2;
4  };
5  void foo() {
6      struct S s1 = {1, 2};
7      int i = 0;
8      bool increment = true;
9      if (increment) {
10         i += 2*i + i + 1;
11     }
12     s1.f1 = 1/i;
13     printf("Hello World!\n");
14 }
15 int main() {
16     foo();
17     return 0;
18 }
```

At its core, the problem here is that there are semantic dependencies between elements within the program. The use of type `S` depends on its definition, for instance. Similarly, calling function `foo` requires the definition of `foo` to be present. Attempting to reduce a program while ignoring these dependencies will typically lead to many invalid candidate programs, like programs that use undefined types. However, manually constructing these dependencies and using them in the search process would require detailed knowledge about the semantics of the input format. Such a reducer would no longer be domain agnostic. It would require significant labor to be hand tailored to the specific language and domain.

The key insight of this work is that dependencies can be approximately inferred and modeled. For example, in Figure 1, the grammar rules for parsing statements differ from those used to parse function and type declarations. This means that the types of the nodes in the parse tree can help give us information about the possible dependencies. We can first try removing statement nodes and only afterward try removing declarations. This *preserves* the dependencies that a naive traversal may violate, but it only directly uses information from the grammar. The general pattern is that it can be advantageous to consider reduction on some set of nodes before others, and the types of internal nodes provide leverage in making that decision. We defer discussing how to choose an ordering over these types for now.

Using this insight, we construct *type batched program reduction*. Instead of attempting to perform expensive reduction operations at *every* node during the tree traversal, type batched reduction selects one type at a time and only operates on nodes of that selected type. Once all nodes of the selected type have been considered, it selects one of the remaining unselected types and traverses the tree again. After all types have been selected *or* it looks unprofitable to continue type batching, a traditional syntax guided approach is used to finish reduction [8]. By leveraging the relationship between node types and dependencies, this approach is able to remove large portions of a program significantly faster within fewer number of oracle queries (empirical results presented in section 4).

Consider our running example again. To reduce this program, a traversal based reduction technique performs a tree traversal guided by the number of token descendants below each node. This means that a node with a larger number of tokens is visited before a node with a smaller number of tokens. Starting with the root node that has the largest number of token descendants, the traversal based reducer queries an oracle to see whether removal of ① succeeds or fails. It fails since removing ① yields an empty file. The traversal based reducer then tries to remove the node with the next largest number of tokens, ③. Removing this node is also unsuccessful since it removes the definition of function `foo` while its use is still within the program (call site on line 16). The reducer then proceeds with node ⑤ but cannot remove it since the property of interest (i.e., printing `Hello World!` on line 13) will not be preserved by removing ⑤. Removing node ⑩ will also cause the oracle to fail because it is required to increment variable *i* to avoid run time issue on line 12. Next, nodes ④, ⑭, ② and ⑦ are visited in order and cannot get removed due to either generating an invalid C program or not preserving the property. The traversal based reducer is able to remove node ⑪ but it continues to fail in removing nodes ⑥, ⑫, ⑨, ⑧ and ⑬.

Now, let us see how a simple version of type batching might work on our running example. Given a selected type, this version will merely traverse the parse tree and try to remove each node of that type individually. We develop a more nuanced version of the technique called *probabilistic joint reduction* in section 3. For simplicity, we consider removing nodes of a given type one by one for now. Suppose that we select types in the order *expression statement* → *selection statement* → *declaration* → *compound statement*

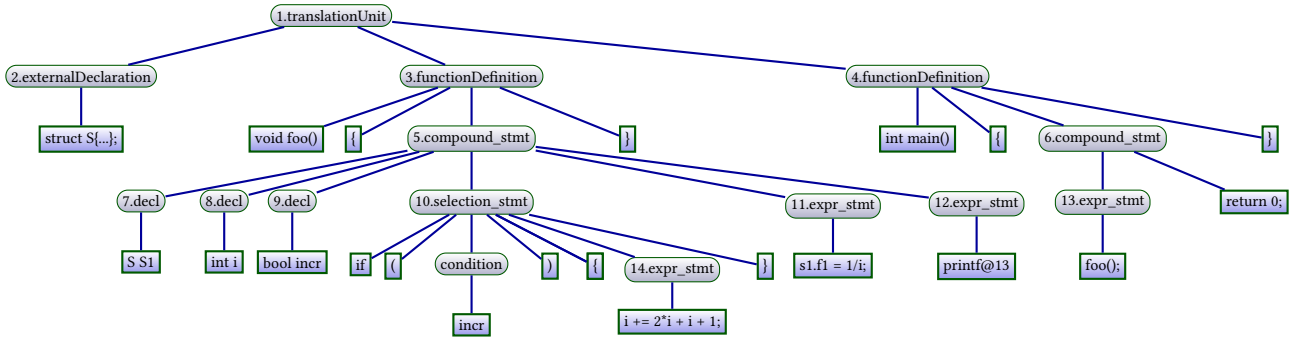


Figure 1: The parse tree of program in Listing 1. Each internal node is annotated with an ID and its grammar rule type.

Listing 2: Traversal based

```
struct S {
  int f1;
  int f2;
};
void foo() {
  struct S s1 = {1, 2};
  int i = 0;
  bool increment = true;
  if (increment) {
      i += 2*i + i + 1;
  }
  [redacted]
  printf("Hello World!\n");
}
int main() {
  foo();
  return 0;
}
```

Listing 3: Type batched

```
[redacted]
void foo() {
  [redacted]
  printf("Hello World!\n");
}
int main() {
  foo();
  return 0;
}
```

Figure 2: Reduced versions of program in Listing 1 generated by a traversal based reducer and our type batched reducer.

→ *function definition* → *external declaration* → *translation unit*, such that the type on the left is chosen before the type on the right.

Table 1: Successful (✓) and unsuccessful (×) removal trials by traversal based, type batched and type batched joint reducers.

reducer	order of removal trials (oracle queries) →	#queries		#removed tokens
		success	total	
Traversal based	(1, ×) (3, ×) (5, ×) (10, ×) (4, ×) (14, ×) (2, ×) (7, ×) **(11, ✓)** (6, ×) (12, ×) (9, ×) (8, ×) (13, ×)	1	14	8
Type batched	(14, ×) **(11, ✓)** (12, ×) (13, ×) **(10, ✓) (7, ✓) (9, ✓) (8, ✓)** (6, ×) (5, ×) (4, ×) (3, ×) **(2, ✓)** (1, ×)	6	14	55
Type batched joint	**({14, 11}, ✓)** (12, ×) (13, ×) **(10, ✓) ({7, 9, 8}, ✓)** (6, ×) (5, ×) (4, ×) (3, ×) **(2, ✓)** (1, ×)	4	11	55

Instead of starting with the root node, our type batched reducer starts by selecting the first type, *expression statement* from the list of our type ordering and adds only nodes of that type to its search space in a decreasing order of the number of their token descendants, such that an *expression statement* node with larger number of tokens as its descendants is visited before an *expression statement* node with fewer number of tokens. The first node to consider for removal is ⑭ since it has a grammar rule type that is the same as the selected type, *expression statement*, and it has the largest number of tokens among nodes with this type. Node ⑭ cannot get removed because it contains line 10 that is necessary to avoid run time issue on line 12. The next node to consider is ⑪ and is successfully removed. Note that it took 9 oracle queries for the traversal based approach to remove ⑪ while our type batched reducer could remove this node by performing only 2 queries. The next nodes with type *expression statement* are ⑫ and ⑬ but their removal is unsuccessful due to not preserving the property of interest. After all nodes of type *expression statement* are visited, the next type in our ordering, *selection statement* is selected. There is only one node of this type, node ⑩ and is successfully removed. The next type in our ordering is *declaration* with nodes ⑦, ⑧ and ⑨. The type batched reducer can remove all the three nodes one by one. The next types to consider are *compound statement* and *function definition*. Nodes with these types cannot get removed due to the same reasons explained above. The single node of type *external declaration* successfully gets removed. The last node to consider is the root node with an unsuccessful trial.

The first two rows of Table 1 contrast the traversal based technique with our type batched approach. Observe that the traversal based reducer uses 14 queries to remove 8 tokens, while our approach removes 55 tokens using the same number of queries. Their resulting outputs appear in Listing 2 and Listing 3, respectively.

Additional Challenges and Advantages. Type batching shows strong potential benefits, but using it effectively requires addressing some additional challenges. We identify and address two key challenges in making type batching work: (1) *type scheduling*, and (2) *stopping criteria*. We also observe that type batching can significantly increase the likelihood of individual nodes being successfully removed, which enables further new techniques like *type batched probabilistic joint reduction*.

Type scheduling refers to how we decide which types to run batches of at any point in time. It is the key mechanism that allows type batching to preserve dependencies. If, for instance, we chose to schedule *declarations* before *statements*, then attempting to remove declarations would fail if they are used by a statement. In contrast, removing statements first does not violate this dependency. The challenge lies in determining which orderings or schedules are better than others. In section 3, we show that it is possible to train simple time-varying models of program reduction that allow us to identify such orderings. These models can be trained using non-buggy code with a simulated property of interest and still yield appropriate orderings for reducing real world test data.

The stopping criteria determine when to switch from performing type batched operations to cleaning up the reduction with a traditional traversal based technique. Not all types of nodes are as likely to be removable as others. At some point it may be more beneficial to stop considering any further type batches over such unsuccessful types when they are deemed to be expensive and unlikely to be beneficial. We explore this further also in section 3.

Finally, type batched probabilistic joint reduction leverages the increased likelihood of successful removal for each node to attempt to remove multiple nodes at the same time where it is expected to be beneficial. In our example, a joint reduction enables a simultaneous removal of (14) and (11) and removes nodes (7), (9) and (8) together (the last row of Table 1). The technique builds upon the insights of recent works like Probabilistic Delta Debugging (PDD) [41], combining them with the benefits of type batching to accelerate reduction further for structured inputs such as programs. Unlike PDD, type batching enables the simultaneous removal of nodes across different levels and locations within the tree.

The next section elaborates on the core design of type batched reduction while also presenting our solutions to the challenges and refinements based on the advantages above.

3 TYPE BATCHED REDUCTION

The fundamental idea underlying our approach is that reduction can be faster when some types of nodes in a parse tree are considered before others. We partition the nodes into *batches* and perform reduction on one batch of nodes at a time. Because our goal is to improve the overall speed of reduction, we want to order the batches to create a list that maximizes the expected rate of reduction:

$$\mathrm{ER}(\overline{batch}, p) = \mathbb{E}\left[\frac{\#\text{ tokens removed}}{\#\text{ oracle queries}}\middle| p, \overline{batch}\right] \quad (1)$$

For a program p, this rate of reduction expresses the number of tokens that we expect each oracle query to remove on average for a particular sequence of batches. If we can choose an ordering with a higher expected rate, then this can speed up the overall reduction process. We call this technique *type batched program reduction*.

As stated in section 2, we partition nodes by their *rule types* or the labels applied by the parser to nodes of the parse tree for a program. Each type τ captures both the structure of the program as well as aspects of meaning, like the differences between declarations and expressions. We use these types because they are ubiquitous and can provide a baseline for showing that the technique has promise. Extending this beyond rule types is left to future work in section 5.

Algorithm 1: Type batched program reduction.

```
   Input: P ∈ ℙ – The program to reduce as a parse tree
   Input: ψ : ℙ → 𝔹 – Oracle for the property to preserve with ψ(P) = true
   Result: A minimum program p ⊆ ℙ s.t. ψ(p) = true
1  Function type_batched_reduction(P, ψ):
2      types ← extract (P)
3      p ← P
4      while types ≠ ∅ do
5          best ← take_best(types, p)
6          types ← types - best
7          if stop_early(p, best) then
8              break
9          p ← typed_traversal_reduction(p, best, ψ)
10     return full_traversal_reduction(p, ψ)
```

The core of our technique is shown in algorithm 1. We build upon this design throughout the remainder of this section. Line 2 starts by extracting the observed rule types from the program being reduced. The loop starting on line 4 performs reduction on one batch of nodes at a time in the chosen order. Line 5 selects the best batch based on the current contents of the reduced program so far and the remaining partitions. Lines 7-8 allow the process to stop early if that looks more fruitful than processing further partitions. Line 9 performs program reduction while only considering nodes in the chosen partition. For simplicity, we assume that reduction in the loop only removes nodes from the parse tree (as opposed to hoisting them [40]). Finally, line 10 performs a clean-up pass using a normal traversal based program reducer. This clean-up allows the technique to gain better efficiency by ordering the partitions well in the loop, while still having the full reduction power of existing program reducers. Indeed, we expect that our technique can be seen as a higher efficiency first phase to complement existing reducers.

Algorithm 1 highlights some of the challenges that must be addressed in order for the technique to work and ways that the technique can be extended or evolved. The most critical of these challenges is choosing which batches to reduce. We call this selection process *scheduling*. As we shall explore later, scheduling well can *boost* efficiency, but scheduling poorly can adversarially *harm* efficiency. For this reason, having effective and automated techniques is critical. In the next subsection, we introduce a simple machine learning based approach to address this challenge. The next challenge arises because it can be advantageous to stop reduction early. The conditions for stopping early are called the *stopping criteria*. It can be possible, for instance, that all easy to remove batches have been processed, and the only batches left to consider are for nodes that are very unlikely to be removed. In such cases, early transitioning to the clean-up phase is desirable. Finally, even if we are able to find a good schedule, the efficiency is limited by the number of tokens that can be removed at once. We show how some of the effects of type batched reduction can overcome this burden with *probabilistic joint batched reduction*. We explore each one of these challenges in the following subsections.

3.1 Scheduling

The goal of scheduling is to produce an automated ordering for batches that maximizes $\mathrm{ER}(\overline{batch}, p)$ for the sequence of batches in the schedule. For illustration, consider a naive solution that uses some simplifying assumptions. Suppose that each individual type batch has a fixed rate $\mathrm{ER}_\tau(p)$ known *a priori*. This is like assuming that declaration nodes are always removable (1) at the same probability of success and with (2) the same number of tokens beneath them. This assumption leads to a straightforward greedy approach to scheduling: sort the batches by descending $\mathrm{ER}_\tau(p)$. This would allow algorithm 1 to reduce the most efficient batches earlier, leaving less remaining work for the following batches to perform. In spite of some impractical assumptions about $\mathrm{ER}_\tau(p)$, this greedy approach lays the foundation of our technique. Specifically, we propose to use machine learning to automate a *time-varying* approximation of $\mathrm{ER}_\tau(p)$ that we can use to guide scheduling.

However, as we shall see, this can be challenging for several reasons. First, it can be challenging to even compute $\mathrm{ER}_\tau(p)$ for

a *single* batch within a schedule. Oracle outcomes during reduction are not i.i.d. data. From the example in section 2, we expect $\text{ER}_{\text{declaration}}(p)$ in a program to be *lower* if declarations are ordered *before* expression statements compared to when they are ordered *after* expression statements. $\text{ER}_\tau(p)$ of a single batch can vary both over time and with the batches that run before it. Second, training such a model of expectation requires training data, but we do not want to require data from, e.g., real world bugs before building a model. Instead, we want to have the model and the technique trainable without real world bugs so that it can immediately be used on the next bug found. Finally, in this work we aim for our model to be simple. Our goal in this work is not to derive a model that will have the *best absolute* performance. Rather, we aim to derive a model that is easy to understand and that can convincingly show that the effect of type batching is real. We leave more complex models refining the approach to future work.

Simple approximations for $\text{ER}_\tau(p)$. We start by discussing how to approximate $\text{ER}_\tau(p)$. Suppose for now that training data exist from previous program reduction sessions using an existing reducer. As the reduction traverses a sequence of nodes $n_1, n_2, n_3, \ldots$, traversing each node n_i records a tuple $(p.\text{size}, n_i.\text{type}, \psi(c))$ for a smaller candidate program c produced by removing n_i. The simplest strategy we might consider uses a naive model where the probability of successful reduction for a type is constant. For each node type τ, we would compute P_τ, the probability that a candidate is successful when visiting a node of the given type:

$$\begin{aligned} &\text{Let } \mathit{of\_type}(c,\tau) = \exists n \text{ s.t. } c \in \text{candidates}(n) \wedge n.\text{type} = \tau. \\ &P_\tau = P(\psi(c) = \checkmark \mid \mathit{of\_type}(c,\tau)) \qquad (2) \\ &\quad = \frac{1}{|\{c \mid \mathit{of\_type}(c,\tau)\}|} \sum_{\{c \mid \mathit{of\_type}(c,\tau)\}} 1^{\{\psi(c)=\checkmark\}}{}_1 \end{aligned}$$

That is, P_τ is the number of *successful* candidates traversing a node of type τ divided by the total number of candidates when traversing a node of type τ. Given P_τ, we can compute $\mathbb{E}[\#\text{ tokens removed}|p,\tau]$ and $\mathbb{E}[\#\text{ queries}|p,\tau]$, which allow us to approximate $\text{ER}_\tau(p)$ for a particular input using the *estimated rate* for a type, $\widehat{\text{ER}}_\tau(p)$:

$$\begin{aligned} \text{ER}_\tau(p) &= \mathbb{E}\left[\frac{\#\text{ tokens removed}}{\#\text{ queries}}\middle| p,\tau\right] \qquad (3) \\ &\approx \frac{\mathbb{E}[\#\text{ tokens removed}|p,\tau]}{\mathbb{E}[\#\text{ queries}|p,\tau]} = \widehat{\text{ER}}_\tau(p) \end{aligned}$$

Per Jensen's inequality, this only approximates the expected rate [24]. Given a reducer that tries to remove syntactically elidable nodes while traversing them [8, 38], these expectations can be computed by traversing the parse tree as shown below. For instance, $\mathit{tokens}(n,\tau)$ considers removing all the tokens below a node n with probability P_τ. When it is unsuccessful with probability $(1-P_\tau)$, the expectations from the children of n are combined instead.

$$\mathbb{E}[\#\text{ tokens removed}|p,\tau] = \mathit{tokens}(p.\text{root},\tau)$$

$$\begin{aligned} \mathit{tokens}(n,\tau) &= P_\tau * 1^{\{n.\text{type}=\tau\}} * n.\text{tokens} \\ &\quad + (1 - P_\tau * 1^{\{n.\text{type}=\tau\}}) \sum_{c \in n.\text{children}} \text{tokens}(c,\tau) \end{aligned}$$

$$\mathbb{E}[\#\text{ queries}|p,\tau] = \mathit{queries}(p.\text{root},\tau)$$

$$\begin{aligned} \mathit{queries}(n,\tau) &= 1^{\{n.\text{type}=\tau \wedge \text{is\_removable}(\tau)^2\}} \\ &\quad + (1 - P_\tau * 1^{\{n.\text{type}=\tau\}}) \sum_{c \in n.\text{children}} \text{queries}(c,\tau) \end{aligned}$$

Thus, if we assume that P_τ is constant per type τ, we can use $\widehat{\text{ER}}_\tau(p)$ for greedy scheduling. However, as previously mentioned, these probabilities are not i.i.d. in practice, so we may ask how much that affects the results and whether a richer model is required. Figure 3 explores this question for three types of nodes in C programs.

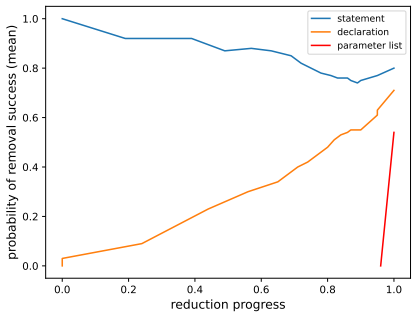


Figure 3: Dynamic probability of removal success for types.

The Y axis captures the probability of successful removal, while the X axis represents reduction progress measured by the fraction of nodes in the original parse tree that have been removed in the current parse tree. Observe that the probabilities per type change significantly during reduction. For instance, declarations are unlikely to be removable at the beginning of the reduction, but toward the end they are quite likely to be removable. Statements, in contrast, are consistently likely to be removable. Because modern reducers try to remove larger chunks before smaller ones, some types like parameter lists do not even appear until close to the end of the reduction. This tells us that desirable models for P_τ should at least account for the temporal effects of reduction.

Algorithm 2: Greedy scheduling with estimated rates

```
Function take_best(types, p):
    return argmax_{τ∈types} ÊR_τ(p)
Function stop_early(p, best)
    return ÊR_best(p) ≤ ÊR_clean-up(p)
```

To produce these time-varying models of P_τ, we train a separate logistic regression model [12] for each rule type. The only feature of the model is the normalized reduction progress: $\frac{p.\text{size}}{P.\text{size}}$. That is, the number of nodes in the parse tree of the partially reduced program p, before a node is removed, out of the total number of nodes in the

[1] We use $1^{\{B\}}$ to denote 1 when B is true and 0 when B is false.

[2] syntactically removable per the grammar [38].

original program. At the beginning of reduction, this has the value 1, and it approaches 0 as more of the tree is removed. The outcome labels are 1 for successful oracle queries and 0 for unsuccessful ones. With this, we estimate P_τ by extracting the estimated probabilities from the model using the current program size relative to the original. This is the version of P_τ used in computation of $\widehat{\text{ER}}_\tau(p)$ in algorithm 2. Note, this *does not* directly account for the impact of ordering between different types. Rather, it attempts to infer which orderings are more likely to be beneficial only independently considering the observed success rates as the combined reduction progresses. The idea is that the influence of one type on another is still captured in the individual success rates for each type alone. While we discuss further improvements in section 5, we show in section 4 that it is able to infer effective orderings in practice.

Using synthetic training data. Similar to a recent work on program reduction [9], we collect training data by (1) selecting a corpus of normally compiling source files, such as compiler test suites, and (2) randomly selecting a token in each file to preserve. We then run normal traversal based reduction on each file to minimize the source file while preserving the selected token and compiling. The outcome of each oracle query is recorded along with the type of the node being visited and the current size of the program. While the task producing the data is not the same as real world reduction, the relationships between node types are the same, yielding useful data. This is akin to a lightweight transfer learning process [43]. More information on our data sets is provided in section 4.

3.2 Stopping Criteria

The time-varying approach to $\widehat{\text{ER}}_\tau(p)$ allows us to choose the next type with the highest estimated rate at any moment. Eventually all types with high rates will have been chosen already, and all remaining rates will actually be slow. It is thus desirable to have some way of saying that all high-efficiency types have been chosen and that the algorithm should move on to the clean-up phase in algorithm 1 in which a traversal based reducer continues the reduction. The conditions for making this change are the *stopping criteria*.

Just as with take_best() from algorithm 2, we use the estimated rates to make this decision. Instead of asking which type has the best $\widehat{\text{ER}}_\tau(p)$, we want to know whether $\widehat{\text{ER}}_\tau(p)$ for the best type is greater than the estimated rate from the clean-up phase itself. If the rate from the batch is higher, it makes sense to continue using the batch. If the rate from the clean-up phase is at least equal to the best next batch, it is preferable to exit to the clean-up phase using the stop_early() of algorithm 2.

Note that computing the rate for the finishing phase is almost the same as for a single type. $tokens(n)$ and $queries(n)$ simply remove the type check on n ($1^{\{n.type=\tau\}}$) and use the probability $P_{n.\text{type}}$ for each individual node n in order to consider the probability of each different node type where appropriate.

3.3 Probabilistic Joint Batched Reduction

Using estimated reduction rates and stopping criteria allows us to select a list of node types online for improving the efficiency of reduction. One of the benefits of this approach is that it increases the probability that any one node is removable on its own. This also means that the probability of removing multiple nodes at the same time (jointly) increases, giving another opportunity to improve the reduction rate by changing typed_traversal_reduction() in algorithm 1. We leverage recent approaches from Probabilistic Delta Debugging (PDD) [41] to improve this process further.

The original idea of PDD is that every failed trial when removing a subset from a list of elements lowers the *belief* about whether each element in the subset can be removed. The reduction process leverages that belief in order to estimate the expected number of elements that can be removed (the *expected gain*, *EG*) and to prioritize which elements of the list to try removing next. This careful selection of how to remove multiple elements at the same time is what we would like to leverage within a type batch.

Unlike PDD that considers removal of elements in a *list*, we are interested in removing all nodes of a given type from across the *entire parse tree*. The gain that we are most interested in is also not +1 for each node considered, but rather the gain is the actual number of tokens removed from the entire tree. These differences lead to a different problem formulation as presented in Algorithm 3.

Algorithm 3: Probabilistic joint reduction over types

```
   Input: τ – The type of AST node to reduce over
   Input: P ∈ ℙ – The partially reduced program so far
   Input: ψ : ℙ → 𝔹 – Oracle for the property to preserve with ψ(P) = true
   Result: A program p ⊆ P s.t. ψ(p) = true
1  Function probabilistic_joint_reduction(p, τ, ψ):
2      initialize_probabilities(p)
3      Q_{τ,prio} ← collect_type_frontier(p.root, τ)
4      while Q_{τ,prio} ≠ ∅ do
5          chunk ← pop_chunk(Q_{τ,prio})
6          if ψ(p - chunk) then
7              p ← p - chunk
8          else if let {n} = chunk then
9              frontier_τ ← collect_type_frontier(n, τ)
10             Q_{τ,prio}.insert(frontier_τ)
11         else
12             update_probabilities(chunk)
13             Q_{τ,prio}.insert(chunk)
14 Function collect_type_frontier(n, τ)
15     frontier ← ∅
16     for child of n do
17         if child.type = τ then
18             frontier ← frontier ∪ { child }
19         else
20             frontier ← frontier ∪ collect_type_frontier(child, τ)
21     return frontier
22 Function prio(n)
23     return n.probability * n.tokens
24 Function pop_chunk(Q)
25     chunk ← ∅
26     while Q ≠ ∅ ∧ EG(chunk) < EG(chunk + Q.top()) do
27         chunk ← chunk + Q.pop()
28     return chunk
```

The core function of the algorithm starts on line 1, providing a refined implementation for typed_traversal_reduction() in algorithm 1. As with other traversal based reducers, it maintains a priority queue of nodes to consider, $Q_{\tau,\text{prio}}$. However, this queue maintains the invariant that it only holds nodes of the correct type for the current batch. Each iteration of the loop on line 4 is similar to a trial from PDD. It (1) builds a chunk of nodes from the queue based on the expected gain, (2) tries removing those nodes from the tree, (3) updates the tree, the belief in removability, and/or the priority queue based on the outcome of the trial for the chunk. The process repeats until no nodes of the given type are left to explore.

The belief updates and convergence properties are the same as in PDD (lines 2, 4, 12). The main differences lie in our priority queue and how we measure expected gain. Note, unlike PDD, our priority queue holds nodes from across the entire parse tree rather than a single list. These nodes capture a τ *frontier* of the tree. They are the highest nodes of type τ that have not been ruled out by the PDD reduction. When a node is ruled out by PDD, its τ frontier is added to the exploration queue (lines 9-10).

Our changes to expected gain affect both how the queue is prioritized as well as how we construct chunks. In PDD, the benefit of removing an element from a list was simply 1. The expected gain for removing a chunk was then $(\prod_{n \in \text{chunk}} b_n)|\text{chunk}|$ where b_n measures the belief that a node is removable (a probability separate from those in the previous sections). In our case, however, the gain from removing each node may differ based on the number of tokens in the subtree below a node. Thus, we have:

$$EG(\text{chunk}) = (\prod_{n \in \text{chunk}} b_n) * \sum_{n \in \text{chunk}} |\text{tokens(n)}| \quad (4)$$

That is, the expected gain is the belief that all nodes in a chunk are removable multiplied by the total number of tokens removed by a chunk. Nodes in the $Q_{\tau,\text{prio}}$ are ordered by expected gain, which guides chunk construction by this new metric in pop_chunk.

4 EVALUATION

We empirically assess the main strategies we take in our approach and perform an overall evaluation of our type batched probabilistic joint reducer to compare it against the latest state of the art domain agnostic program reducers [8, 38]. To this end, we explore the following research questions:

- **RQ1.** How do different batch schedules impact the performance of reduction? In particular:
  - How do the schedules generated by our models compare against random schedules?
  - How do our time-varying schedules compare against time-invariant schedules?
- **RQ2.** How do type batching and probabilistic joint reduction interplay? In particular:
  - What is the impact of type batching on the performance of joint reduction?
  - What is the impact of joint reduction on the performance of type batched reduction?
- **RQ3.** How does a type batched probabilistic joint reducer compare to the state of the art reducers in terms of reduction time, number of oracle queries and final reduced size?

4.1 Experimental setup

Our experimental design uses several choices throughout. All benchmarks come from the Perses evaluation suite [38]. In total, we were able to replicate failures for 18 C, 8 Rust and 3 Go programs that comprise more than 75% of the original set of benchmarks. For smaller exploratory studies, we sampled 3 of the C benchmarks and used them consistently throughout the smaller scale studies. We run our experiments on Ubuntu 16.04 with Intel(R) Core(TM) i5-7300U CPU @ 2.6 GHz and 8 GB of RAM.

For training data across the languages, we used 40 non-buggy CSmith [44] generated programs for C and sampled files from Github trending projects for Rust and Go [10]. We collected logs from non-buggy reduction as described in section 3. C, Rust, and Go have 60, 188, and 48 elidable rule types, respectively [30]. For these, we collect 279,052 oracle queries as our training data for C, 465,488 for Rust, and 156,039 for Go.

4.2 RQ1: The impact of different schedules

To understand the effectiveness of our model-generated schedules, we compare them against two groups of baseline schedules on our smaller benchmark set. The first group, referred to as *random*, consists of schedules with the same types as our schedules but ordered in a random fashion. The second group called *time-invariant* consists of schedules that use constant probabilities for each type of node in the parse tree. Each group is constructed as follows:

Random schedules. Comparing against random schedules of types allows us to see how our schedules perform relative to the space of possible schedules. To create random schedules for comparison, we take the same first types selected by our technique (up to five types) and randomly sample 20 from their set of possible orderings (without replacement and excluding our schedule). We compare these against our schedules (also truncated to five types where applicable). We discuss later in section 5 that most schedules are short in practice.

Time-invariant schedules. These schedules use the greedy approach from algorithm 1 and algorithm 2, but the probabilities for each node type are time-invariant, or constant. These probabilities are computed *a priori* as in Equation 2. Comparing against these schedules allows us to understand the importance of accounting for how a type probability, P_τ, may change over time. These schedules have the same number of types as random and time-varying schedules (at most five) for comparison.

Results contrasting the different schedules are shown in Figure 4. The blue line depicts the average performance of random schedules, augmented by a shaded range for the standard deviation of random schedules. The green dashed line shows the performance of reduction using our time-varying schedules. The red line shows the performance of time-invariant schedules. The X axis is the reduction time in seconds, and the Y axis is the number of remaining tokens in the program as the reduction proceeds. As shown, our time-varying schedules yield a faster convergence towards a reduced program compared to either the average behavior of random schedules or the time-invariant schedules. We discuss the results of this study more in section 5, especially to explain the observation that our time-varying schedules fall within some standard deviation regions of random schedules. Overall, the results of this study suggest that using our time-varying models to generate schedules can provide a more effective framework to guide reduction.

Interestingly, using time-invariant schedules slows down the reduction process even more than random schedules in one case (`gcc-64990`). To explain the behavior of time-invariant schedules, we examined the orders of their scheduled types and observed that a large number of declarations (e.g., function definitions) are always scheduled before their uses (e.g., function call sites). This also explains the plateau without reduction progress at the beginning of

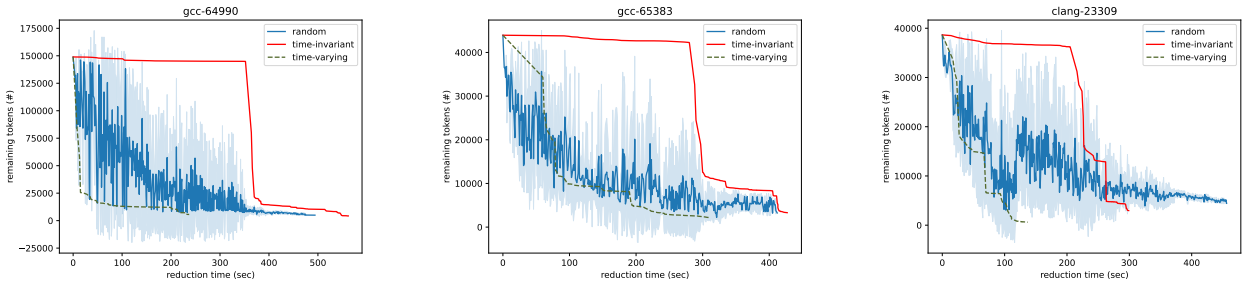


Figure 4: Performance of type batched probabilistic joint reducer using random, time-invariant and time-varying schedules.

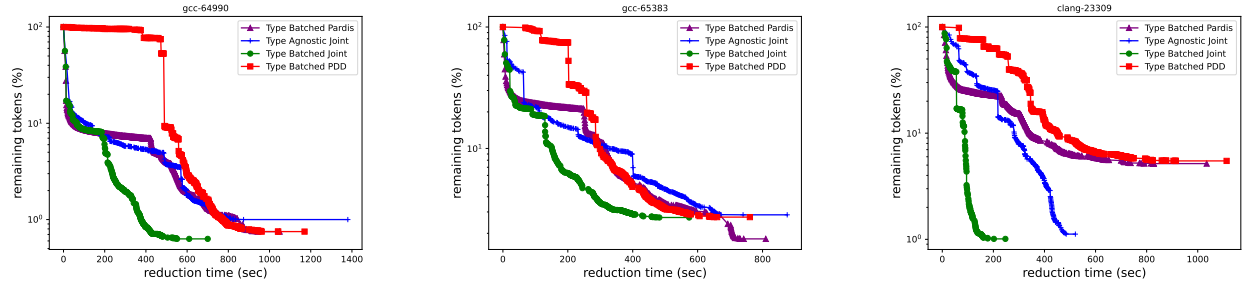


Figure 5: Reduction progress by type batched joint, type agnostic joint, type batched Pardis and type batched PDD.

time-invariant schedules in Figure 4. We find two reasons to explain why time-invariant schedules prioritize declarations over uses: (1) These declarations contain many tokens, so removing them can yield a large gain for the reducer. (2) The *overall* probability of removal success for these declarations is high. It means that although declarations are not removable at the beginning of reduction, most of them can successfully be removed as reduction proceeds and removes their uses. This increases their *overall* probability of removal, which is used in the time-invariant schedules. In contrast, time-varying schedules account for changes in P_τ. They prioritize uses over declarations by assigning a low probability to declarations at the beginning and a higher probability later as reduction proceeds. Uses and declarations are a specific example of one program element depending on another. Our time-varying scheduling tries to learn these relationships automatically and prioritize accordingly.

4.3 RQ2: Batching and joint reduction interplay

This research question studies the interplay between type batching and probabilistic joint reduction by examining whether each of them can improve the other with respect to the performance of reduction. In particular, we answer the following questions:

- Does type batching improve the performance of joint reduction? To answer, we measure the performance of joint reduction once with and once without type schedules.
- Does joint reduction improve the performance of type batching? To answer, we perform type batching once using joint reduction and once along with a traditional reducer.

To measure the impact of type batching on joint reduction, we compare two different versions of algorithm 1. In the *type agnostic* version, we remove the schedule loop of line 4 and make the clean-up phase use joint reduction over all types simultaneously on line 10. In the *type batched* version, we use the algorithm as written with joint reduction inside the schedule loop at line 9.

To measure the impact of joint reduction on type batching, we also consider two variants of algorithm 1. In the *joint* reduction version, we again use joint reduction inside the schedule loop at line 9. In the *traditional* reduction version, we replace line 9 once with Pardis [8], a traversal based reducer that visits nodes of the parse tree one at a time and once with PDD [41], the probabilistic form of Delta Debugging introduced in section 3.

We perform removal trials using the above techniques on our smaller benchmark set. Results are presented in Figure 5. As shown, type batching enhanced by joint reduction (type batched joint) has the fastest convergence among all the variants. These results strongly support that type batched reduction and joint reduction are synergistic and leveraging both of them together can help us to achieve a better overall performance of reduction.

4.4 RQ3: Comparison with the state of the art

Finally, we compare the overall performance of our type batched probabilistic joint reducer against state of the art domain agnostic program reducers, Pardis [8] and Perses [38]. This experiment runs a full set of reduction operations on our full benchmark set.

Our benchmark programs and their original sizes are shown in the first two columns of Table 2. The other columns show the metrics used in the evaluation of the techniques. In particular, the

Table 2: A comparison between the performance of Perses, Pardis and type batched probabilistic joint reducer.

Bug	O(#)	Perses				Pardis				Type batched probabilistic joint reducer			
		R(#)	Q(#)	$T(s)$	E(#/s)	R(#)	Q(#)	$T(s)$	E(#/s)	R(#)	Q(#)	$T(s)$	E(#/s)
clang-22382	21,069	334	5,030	3,535	6	343	2,847	4,273	5	331	1,882	2,261	9
clang-22704	184,445	266	4,289	1,788	103	173	4,506	3,296	56	233	4,755	3,542	52
clang-23309	38,647	279	5,896	1,690	23	802	8,783	2,915	13	154	780	280	137
clang-25900	78,960	306	4,326	1,397	56	307	2,383	848	93	315	1,889	629	125
clang-27747	173,840	220	3,948	1,301	133	230	1,992	901	193	401	1,730	660	263
clang-31259	48,800	374	4,963	2,013	24	376	2,847	1,397	35	362	1,623	726	68
gcc-59903	57,582	465	9,908	4,572	12	392	5,479	8,947	6	623	2,699	1,821	31
gcc-60116	75,225	480	7,514	9,088	8	528	5,397	3,107	24	608	4,032	1,830	41
gcc-61383	32,450	321	5,649	1,805	18	288	2,532	1,284	25	315	1,642	609	53
gcc-61452	26,733	370	5,552	3,993	7	342	2,539	2,884	9	360	1,867	2,067	13
gcc-61917	85,360	327	5,869	1,829	46	329	3,580	1,265	67	366	2,369	686	124
gcc-64990	148,931	354	5,183	2,103	71	354	2,696	1,232	121	332	1,834	710	209
gcc-65383	43,942	228	4,981	1,682	26	279	2,130	977	45	271	1,600	611	71
gcc-66186	47,481	441	4,346	1,396	34	391	2,085	890	53	394	1,720	625	75
gcc-66375	65,489	463	6,190	2,302	28	447	4,323	1,623	40	465	2,238	825	79
gcc-71626	6,134	60	535	36	169	60	299	32	190	60	164	12	506
gcc-71632	141	75	217	57	1.16	75	173	54	1.22	75	134	49	1.35
gcc-77624	1,306	22	196	10	128	22	205	15	86	22	89	5	257
geomean	31,102	243	3,315	1,093	27	250	2,088	927	32	249	1,261	471	63
median	48,141	324	5,006	1,797	27	334	2,618	1,275	43	332	1,782	673	73
rust-44800	802	464	3,967	7,307	0.05	464	3,525	9,650	0.04	464	1,798	3,400	0.10
rust-63791	8,144	5,875	11,492	T/O	0.16	5,782	10,555	T/O	0.16	5,568	10,729	T/O	0.18
rust-65934	107	100	198	468	0.01	100	166	438	0.02	100	120	249	0.03
rust-69039	191	120	826	4,372	0.02	120	689	3,528	0.02	123	571	3,068	0.02
rust-77002	348	286	3,862	2,290	0.03	286	3,646	1,966	0.03	297	1,536	1,280	0.04
rust-77993	4,989	16	175	705	7.05	16	169	1,079	4.61	16	139	501	9.93
rust-78336	980	15	106	438	2.20	15	73	352	2.74	15	89	329	2.93
rust-78622	157	29	77	319	0.40	29	40	178	0.72	29	57	235	0.54
geomean	659	126	671	1,625	0.15	126	544	1,520	0.17	127	441	1,080	0.23
median	575	110	512	1,498	0.11	110	429	1,523	0.10	112	355	891	0.14
go-28390	146	84	197	28	2.2	84	127	18	3.4	84	151	26	2.4
go-29220	127	60	160	16	4.2	60	158	16	4.2	65	147	18	3.4
go-30606	449	213	812	197	1.2	213	794	185	1.3	213	597	150	1.6
geomean	203	102	295	45	2.2	102	252	38	2.6	105	237	41	2.4
median	146	84	197	28	2.2	84	158	18	3.4	84	151	26	2.4

O, R and Q denote number of tokens in the original program, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of number of removed tokens per second. Timeout (T/O) is set to 4 hours. For each metric, the best value among the three techniques is highlighted.

reduced size (R), the number of oracle queries (Q), the reduction time (T) and the efficiency or rate of reduction (E) that is the number of tokens removed per second are presented.

In 17 out of 18 C programs, our reducer performs a faster reduction with an average improvement of 57% and 49% in reduction time against Perses and Pardis, respectively. Moreover, with an average number of 249 tokens, our reducer generates outputs of similar size by higher efficiency where on average 63 tokens are removed per second. This number is 27 and 32 tokens for Perses and Pardis, respectively. In 7 out of 8 Rust programs, our reducer outperforms Perses by 25%-53% while generating outputs of similar size. Similarly, type batched reducer improves the reduction time of Pardis by 7%-65% in 6 out of 8 test cases. For the one rust test case that none of the three techniques finishes within the timeout, our reducer generates the smallest output among all techniques. Finally, for the largest Go program in our benchmark, we observe an improvement of 23% and 19% in reduction time against Perses and Pardis, respectively while generating reduced outputs of the same size. For the other two Go programs, our reducer performs reduction within a short reduction time similar to the other techniques.

To assess the significance of the results, we use a Wilcoxon signed-rank test [36] to compare results pairwise between batched reduction and the two other reducers. For the number of oracle queries, time, and rates, we use one-tailed tests to check that batched reduction uses fewer queries, uses less time, and has a higher rate of reduction. For the size, we use a two-tailed test to assess whether the sizes from batched reduction are significantly different than other reducers. Results for oracle queries, time, and rate, show that batched reduction *is* significantly better in each metric, with p-values less than $7.8 \times e^{-05}$ in all cases against both Perses and Pardis. The sizes from batched reduction are not proven to differ significantly from those of other reducers.

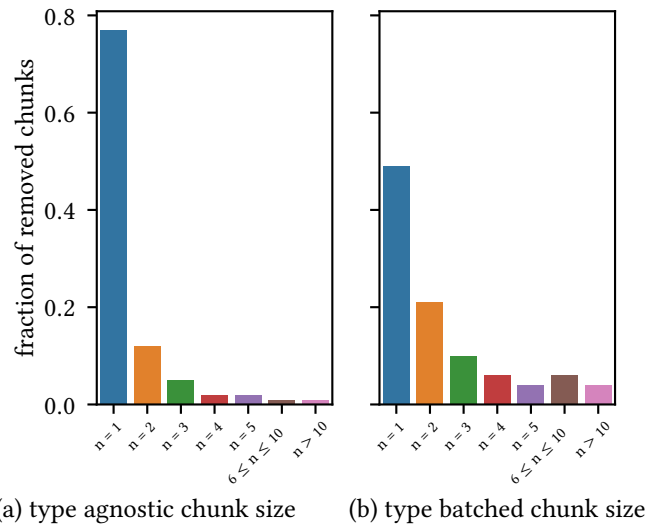
5 DISCUSSION AND FUTURE WORK

Here, we discuss few aspects of our type batched joint reducer.

To understand how type batching improves the overall performance of joint reduction, we compute the distributions of sizes of chunks that were successfully removed using our type batched joint reducer and its type agnostic version for the C benchmarks.

As can be seen in Figure 6, the number of larger chunks that were successfully removed increases using type batching. In more detail, chunks of size one decrease from 77% in type agnostic approach to 49% of total successful oracle queries when performing type batching while chunks of size two, three and four increase from 12%, 5% and 2% to 21%, 10% and 6% of total successfully removed chunks. Chunks of size larger than 10 comprise 4% of successful queries in the type batched approach while this number is only 1% for the type agnostic version. In particular, we observe chunks of size up to 58 when using our type batched reducer which enables successful removal of 58 nodes together. Having larger chunks of nodes can explain the faster convergence of joint reduction when using type batching compared to not using it.

The number of types included in a schedule that we refer to as the length of the schedule is another interesting metric that keeps



(a) type agnostic chunk size (b) type batched chunk size

Figure 6: Distribution of number of nodes (n) removed together using type batched vs. type agnostic joint reduction.

increasing until we reach a stopping point where the benefit of type batching becomes smaller than using a traversal based technique. On average, our schedules of types for our C programs have five types, meaning that it takes only five types on average until our reducer finds it less beneficial to continue the type batching process. This is interesting because we can see that by selecting a small set of types and defining an appropriate ordering among them, we are able to prune most portions of the program. The average length of schedules for our Rust and Go programs is two.

This work opens the door to several future improvements for using machine learning to directly guide program reduction. We currently use the rule types of the grammar to partition nodes into batches. While effective, other information or even neural embeddings of nodes [1] could help to further guide the batch partitioning. Moreover, our work presently focuses on *removing* nodes inside the batches. Other actions like hoisting subtrees [40] may be beneficial. Action selection for long term reward optimization using reinforcement learning [39] is likely to optimize long term reduction rates that can further improve program reduction, and yield speeds that minimize developer interruption. We expect this to improve cases in Figure 4, where our schedules still fall in the range of the standard deviation of random schedules.

6 RELATED WORK

Program reduction is a specific kind of *test case reduction* (TCR). TCR was popularized by Delta Debugging (DD) for helping with debugging and bug deduplication [45]. Despite its generality, DD performs poorly on structured data such as programs. This led to techniques like HDD [25], which applied DD level by level, and FlexMin, which used grammars to hoist subtrees in a test case [2, 27]. Much work over the last decade has focused on improving HDD and how it handles tree structured data [14, 18, 21, 22, 40]. Additional work has focused on enhancing the empirical performance of general DD [16] and improving its theoretical bounds to $O(n)$ [7, 41].

As a special case of TCR, program reduction focuses on reducing source code as a particularly complex form of data. The most well-known program reducer is C-Reduce for C programs [34], but it has inspired similar reducers for several languages [4, 6, 19, 20, 23, 35, 37, 42]. These reducers are domain specific, leveraging expert knowledge to guide the reduction process. In contrast, domain agnostic reducers are meant to apply without domain knowledge. They can operate with just a parse tree or a grammar for the input [8, 15, 17, 26, 38, 40]. While they generally produce larger results, they can also be used when expert knowledge or labor is not available. In fact, HDD, FlexMin, and their descendants are domain agnostic reducers. More recently, Perses proposed transforming input grammars to make syntactically removable input portions more explicit and avoid syntactically invalid candidates [38], and Pardis proposed more precise guidance for the traversal toward large program regions [8].

Several techniques have integrated learning into the reduction process. Among traversal based reducers, GTR learns from a corpus which tree transformations to consider at a node [14], and Pardis was modified to learn to skip unlikely oracle queries [9]. CHISEL applies reinforcement learning to select which DD action to apply on a list [13]. PDD updates its belief in each element of a list being removable based on failed oracle queries [41], attempting to remove subsets of a list that look most promising. In contrast to all of these, type batched reduction is the only one that uses machine learning to select which portions of a tree to consider reducing. While PDD and Chisel select portions of a *list* in practice, they do not leverage prior knowledge about the probability of success, relying on updating beliefs based on oracle failure. Thus, they do not directly attack the expected rate of reduction. Similar to the model guided Pardis [9], our aim is to use learning to reduce the probability of producing invalid programs, but model guided Pardis does not suggest which portions of the program to reduce. It also does not reduce multiple portions of a program jointly.

Program reduction has also demonstrated recent utility across several other domains. With respect to security, program reduction has been used for *program debloating*, removing unnecessary features from software to shrink its attack surface [11, 13, 31, 35, 42]. Similarly, eliminating features can shrink programs and increase their efficiency in IoT settings [3] which facilitates resource efficiency. Program reduction can also benefit machine learning by helping to understand models of code [32, 33].

7 CONCLUSIONS

We proposed a type batched program reducer that is domain agnostic and capable of reducing programs within a shorter time compared to state of the art program reducers. Our reducer estimates the expected rate of reduction to select the most advantageous groups of nodes from the parse tree to explore at a given point in time. The rate estimates leverage time-varying models that predict the probability of removal success for different node types. Our joint reducer further enables simultaneous reduction over multiple nodes in the tree. Together, these designs significantly improve reducing programs of multiple domains.

REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (jan 2019), 29 pages. https://doi.org/10.1145/3290353

[2] Nicolas Bruno. 2010. Minimizing database repros using language grammars. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings (ACM International Conference Proceeding Series, Vol. 426)*, Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan (Eds.). ACM, 382–393. https://doi.org/10.1145/1739041.1739088

[3] Arpit Christi, Alex Groce, and Austin Wellman. 2019. Building Resource Adaptations via Test-Based Software Minimization: Application, Challenges, and Opportunities. In *IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019*, Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ramezani Ivaki, and Nuno Laranjeiro (Eds.). IEEE, 73–78. https://doi.org/10.1109/ISSREW.2019.00046

[4] JS Delta. 2014. *A delta debugger for JavaScript.* https://github.com/wala/jsdelta

[5] GCC Documentation. 2022. How to Minimize Test Cases for Bugs. https://gcc.gnu.org/bugs/minimize.html

[6] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1017–1032. https://doi.org/10.1145/3453483.3454092

[7] Golnaz Gharachorlu and Nick Sumner. 2018. Avoiding the Familiar to Speed Up Test Case Reduction. In *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*. IEEE, 426–437. https://doi.org/10.1109/QRS.2018.00056

[8] Golnaz Gharachorlu and Nick Sumner. 2019. PARDIS: Priority Aware Test Case Reduction. In *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11424)*, Reiner Hähnle and Wil M. P. van der Aalst (Eds.). Springer, 409–426. https://doi.org/10.1007/978-3-030-16722-6_24

[9] Golnaz Gharachorlu and Nick Sumner. 2021. Leveraging Models to Reduce Test Cases in Software Repositories. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 230–241. https://doi.org/10.1109/MSR52588.2021.00035

[10] Github. 2022. Trending: See what the GitHub community is most excited about today. https://github.com/trending

[11] Zhongshu Gu, William N. Sumner, Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. DRIP: A framework for purifying trojaned kernel drivers. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*. IEEE Computer Society, 1–12. https://doi.org/10.1109/DSN.2013.6575342

[12] Jr Harrell, Frank E. 2015. *Regression Modeling Strategies.* Springer International Publishing AG.

[13] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 380–394. https://doi.org/10.1145/3243734.3243838

[14] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 861–871. https://doi.org/10.1109/ASE.2017.8115697

[15] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2016, Seattle, WA, USA, November 18, 2016*, Tanja E. J. Vos, Sigrid Eldh, and Wishnu Prasetya (Eds.). ACM, 31–37. http://dl.acm.org/citation.cfm?id=2994296

[16] Renáta Hodován and Ákos Kiss. 2016. Practical Improvements to the Minimizing Delta Debugging Algorithm. In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 2016*, Leszek A. Maciaszek, Jorge Cardoso, André Ludwig, Marten van Sinderen, and Enrique Cabello (Eds.). SciTePress, 241–248. https://doi.org/10.5220/0005988602410248

[17] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 194–203. https://doi.org/10.1109/ICSME.2017.26

[18] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging. In *12th IEEE/ACM International Workshop on Automation of Software Testing, AST@ICSE 2017, Buenos Aires, Argentina, May 20-21, 2017*. IEEE Computer Society, 23–29. https://doi.org/10.1109/AST.2017.4

[19] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 556–566. https://doi.org/10.1145/3338906.3338956

[20] Christian Gram Kalhauge and Jens Palsberg. 2021. Logical bytecode reduction. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1003–1016. https://doi.org/10.1145/3453483.3454091

[21] Ákos Kiss. 2020. Generalizing the Split Factor of the Minimizing Delta Debugging Algorithm. *IEEE Access* 8 (2020), 219837–219846. https://doi.org/10.1109/ACCESS.2020.3043027

[22] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: A Recursive Variant of the Hierarchical Delta Debugging Algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST 2018*. 16–22.

[23] Philipp Krüger. 2019. *Elm-Reduce: Delta Debugging Functional Programs.* Bachelor's thesis. Computer Science, Karlsruhe Institute of Technology.

[24] Marek Kuczma. 2009. *An Introduction to the Theory of Functional Equations and Inequalities.* Birkhäuser Verlag AG.

[25] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical Delta Debugging. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 142–151. https://doi.org/10.1145/1134285.1134307

[26] Ghassan Shakib Misherghi. 2007. *Hierarchical Delta Debugging.* Master's thesis. Computer Science, University of California, Davis.

[27] Kristi Morton and Nicolas Bruno. 2011. FlexMin: a flexible tool for automatic bug isolation in DBMS software. In *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*, Goetz Graefe and Kenneth Salem (Eds.). ACM, 1. https://doi.org/10.1145/1988842.1988843

[28] Chris Parnin and Spencer Rugaber. 2011. Resumption strategies for interrupted programming tasks. *Softw. Qual. J.* 19, 1 (2011), 5–34. https://doi.org/10.1007/s11219-010-9104-9

[29] Terence Parr. 2013. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf.

[30] Perses. 2020. Language-agnostic program reducer. https://github.com/uw-pluverse/perses

[31] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[32] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding Neural Code Intelligence through Program Simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 441–452. https://doi.org/10.1145/3468264.3468539

[33] Md Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. 2022. Syntax-Guided Program Reduction for Understanding Neural Code Intelligence Models. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 70–79. https://doi.org/10.1145/3520312.3534869

[34] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346. https://doi.org/10.1145/2254064.2254104

[35] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/3238147.3238160

[36] David J. Sheskin. 2011. *Handbook of Parametric and Nonparametric Statistical Procedures.* Chapman and Hall/CRC.

[37] Daniil Stepanov, Marat Akhin, and Mikhail A. Belyaev. 2019. ReduKtor: How We Stopped Worrying About Bugs in Kotlin Compiler. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 317–326. https://doi.org/10.1109/ASE.2019.00038

[38] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 361–371. https://doi.org/10.1145/3180155.3180236

[39] Richard S. Sutton and Andrew G. Barto. 2015. *Reinforcement Learning: An Introduction*. The MIT Press.

[40] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. 2021. Extending Hierarchical Delta Debugging with Hoisting. In *2nd IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 60–69. https://doi.org/10.1109/AST52587.2021.00015

[41] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 881–892. https://doi.org/10.1145/3468264.3468625

[42] Ryan Williams, Tongwei Ren, Lorenzo De Carli, Long Lu, and Gillian Smith. 2021. Guided Feature Identification and Removal for Resource-Constrained Firmware. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 28 (dec 2021), 25 pages. https://doi.org/10.1145/3487568

[43] Qiang Yang, Yu Zhang, Wenyuan Dai, and Sinno Jialin Pan. 2020. *Transfer Learning*. Cambridge University Press.

[44] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[45] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.