Contrastive Code Representation Learning

Paras Jain* and **Ajay Jain*** and **Tianjun Zhang** and
Pieter Abbeel and **Joseph E. Gonzalez** and **Ion Stoica**
Department of EECS, UC Berkeley
{parasj, ajayj, tianjunz,
pabbeel, jegonzal, istoica}@berkeley.edu

Abstract

Recent work learns contextual representations of source code by reconstructing tokens from their context. For downstream semantic understanding tasks like code clone detection, these representations should ideally capture program functionality. However, we show that the popular reconstruction-based RoBERTa model is sensitive to source code edits, *even when the edits preserve semantics*. We propose ContraCode: a contrastive pre-training task that learns code functionality, not form. ContraCode pre-trains a neural network to identify functionally similar variants of a program among many non-equivalent distractors. We scalably generate these variants using an automated source-to-source compiler as a form of data augmentation. Contrastive pre-training outperforms RoBERTa on an adversarial code clone detection benchmark by 39% AUROC. Surprisingly, improved adversarial robustness translates to better accuracy over natural code; ContraCode improves summarization and TypeScript type inference accuracy by 2 to 13 percentage points over competitive baselines. All source is available at https://github.com/parasj/contracode.

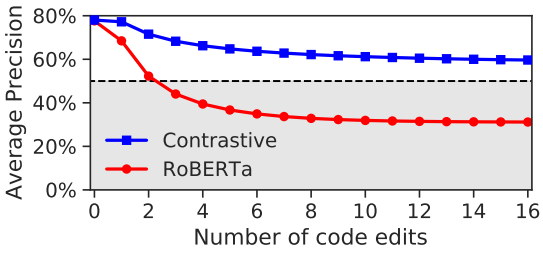


Figure 1: **Robust code clone detection:** On source code, *RoBERTa is not robust to simple label-preserving code edits* like renaming variables. Adversarially selecting between possible edits lowers performance below random guessing (dashed line). Contrastive pre-training with ContraCode learns a more robust representation of functionality, consistent across code edits.

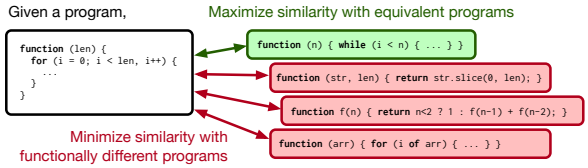


Figure 2: For many analyses, programs with the same functionality should have similar representations. ContraCode learns such representations by pre-training an encoder to retrieve equivalent, transformed programs among many distractors.

1 Introduction

Programmers increasingly rely on machine-aided programming tools that analyze or transform code automatically to aid software development (Kim et al., 2012). Traditionally, code analysis uses hand-written rules, though the wide diversity of programs encountered in practice can limit their generality. Recent work leverages machine learning for richer language understanding, such as learning to detect bugs (Pradel and Sen, 2018) and predict performance (Mendis et al., 2019).

Still, neural models of source code are susceptible to adversarial attacks. Yefet et al. (2020) and Schuster et al. (2021) find accuracy degrades significantly under adversarial perturbations for both discriminative and generative code models. In our work, we investigate adversarial attacks on code clone detection. Successful adversarial attacks could circumvent malware detectors.

While self-supervision can improve adversarial robustness (Hendrycks et al., 2019), we find that RoBERTa is sensitive to stylistic implementation choices of code inputs. Fig. 1 plots the performance of RoBERTa and ContraCode, our proposed method, on a code clone detection task as small label-preserving perturbations are applied to the input code syntax. With just three minor adversarial edits to code syntax, RoBERTa underperforms the random classifier (in gray). In Fig. 3,

* equal contribution

Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 5954–5971
November 7–11, 2021.

we show that RoBERTa's representations of code are sensitive to code edits in agreement with prior work (Wang and Christodorescu, 2019; Wang and Su, 2019; Rabin and Alipour, 2020).

To address this issue, we develop ContraCode: a self-supervised representation learning algorithm that captures program semantics. We hypothesize that *programs with the same functionality should have similar underlying representations* for downstream code understanding tasks.

ContraCode generates syntactically diverse but functionally equivalent programs using source-to-source compiler transformation techniques (*e.g.*, dead code elimination, obfuscation and constant folding). It uses these programs in a challenging discriminative pretext task that requires the model to identify similar programs out of a large dataset of distractors (Fig. 2). To solve this task, the model must embed code semantics rather than syntax. ContraCode improves adversarial robustness in Fig. 1. Surprisingly, adversarial robustness transfers to better natural code understanding.

Our novel contributions include:

1. the novel use of compiler-based transformations as data augmentations for code,
2. the concept of program representation learning based on functional equivalence, and
3. a detailed analysis of architectures, code transforms and pre-training strategies, showing ContraCode improves type inference top-1 accuracy by 9%, learned inference by 2%–13%, summarization F1 score by up to 8% and clone detection AUROC by 2%–46%.

2 Related work

Self-supervised learning (SSL) is a learning strategy where some attributes of a datapoint are predicted from remaining parts. BERT (Devlin et al., 2018) is a SSL method for NLP that reconstructs masked tokens as a pretext task. RoBERTa (Liu et al., 2019) further tunes BERT. Contrastive approaches minimize distance between learned representations of similar examples (positives) and maximize distance between dissimilar negatives (Hadsell et al., 2006). CPC (Oord et al., 2018; Hénaff et al., 2019) encodes segments of sequential data to predict future segments. SimCLR (Chen et al., 2020a) and MoCo (He et al., 2019; Chen et al., 2020b) use many negatives for dense loss signal.

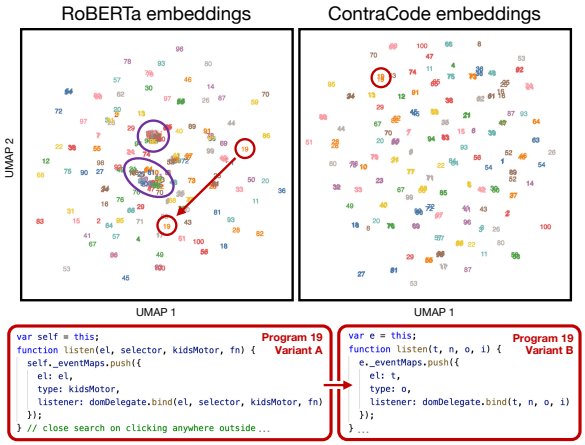


Figure 3: A UMAP visualization of JavaScript method representations learned by RoBERTa and ContraCode, in $\mathbb{R}^2$. Programs with the same functionality share color and number. RoBERTa's embeddings often do not cluster by functionality, suggesting that it is sensitive to implementation details. For example, many different programs **overlap**, and renaming the variables of Program 19 significantly **changes the embedding**. In contrast, variants of Program 19 cluster in ContraCode's embedding space.

Code representation learning We address clone detection (White et al., 2016), type inference (Hellendoorn et al., 2018), and summarization (Alon et al., 2019a). Others explored summarization (Movshovitz-Attias and Cohen, 2013; Allamanis et al., 2016; Iyer et al., 2016; Ahmad et al., 2020) and types (Pradel et al., 2019; Pandi et al., 2020; Wei et al., 2020; Allamanis et al., 2020; Bielik and Vechev, 2020; Allamanis et al., 2018) for various languages. Inst2vec (Ben-Nun et al., 2018) embeds statements in LLVM IR by processing a flow graph with a context prediction objective (Mikolov et al., 2013). Code2seq (Alon et al., 2019a) embeds AST paths with an attentional encoder for seq2seq tasks. Kanade et al. (2020) and Feng et al. (2020) pre-train a Transformer on code using the masked language modeling (MLM) objective (Devlin et al., 2018; Taylor, 1953).

Adversarial attacks on code models Yefet et al. (2019) find code models are highly sensitive to adversarial code edits in a discriminative setting. Schuster and Paliwal (1997) discovers in-the-wild attacks on code autocompletion tools. Compared to language models, code models may be more vulnerable to adversarial attacks due to synthetic labels (Ferenc et al., 2018; Pradel and Sen, 2018; Benton et al., 2019) and duplication (Allamanis, 2019) that degrade generalization.

```
function x(maxLine) {
  const section = {
    text: '',
    data
  };

  for (; i < maxLine; i += 1) {
    section.text += `${lines[i]}\n`;
  }

  if (section) {
    parsingCtx.sections.push(section);
  }
}
```

Original JavaScript method

```
function x(t) {
  const n = {
    'text': '',
    'data': data
  };
  for (;i < t; i += 1) {
    n.text += lines[i] + '\n';
  }
  n && parsingCtx.sections.push(n);
}
```

Renamed variables, explicit object style, explicit concatenation, inline conditional

```
function x(t){const
n={'text':'','data':data};for(;i<t;i+=
1)n.text+=lines[i]
+'\n';n&&parsingCtx.sections.push(n)}
```

Mangled source with compressed whitespace

Figure 4: A JavaScript method from our unlabeled training set with two automatically generated semantically-equivalent programs. The method is from the StackEdit Markdown editor.

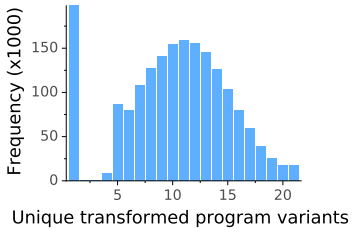


Figure 5: Histogram of the number of unique transformed variants per JavaScript method during pre-training.

3 Approach

Our core insight is to use compiler transforms as data augmentations, generating a dataset of equivalent functions (§3.1, 3.2). We then use a contrastive objective to learn a representation invariant to these transforms (§3.3).

3.1 Compilation as data augmentation

Modern programming languages afford great flexibility to software developers, allowing them to implement the same function in different ways. Yet, crowdsourcing equivalent programs from GitHub is difficult as verifying equivalence is undecidable (Joshi et al., 2002; Bansal and Aiken, 2006) and approximate verification is costly and runs untrusted code (Massalin, 1987).

Instead of searching for equivalences, we propose correct-by-construction data augmentation. We apply compiler transforms to unlabeled code to generates many variants with equivalent functionality, *i.e.* operational semantics. For example, dead-code elimination (DCE) is an optimization that removes operations that do not change function output. While DCE preserves functionality, Wang and Christodorescu (2019) find that up to 12.7% of the predictions of current supervised algorithm classification models change after DCE.

We parse a particular source code sequence, *e.g.* `W*x + b` into a tree-structured representation `(+ (* W x) b)` called an Abstract Syntax Tree (AST). We then transform the AST with automated traversal passes. A rich body of prior programming language work explores parsing and transforming ASTs to optimize a program. If source code is emitted by the compiler rather than machine code, this is called source-to-source transformation or transpilation. Transpilation is common for optimizing and obfuscating dynamic languages like JavaScript. Further, if each transform preserves code semantics, then any composition also preserves semantics.

	Code compression		Identifier modification
✓	Reformatting (R)	✓	Variable renaming (VR)
✓	Beautification (B)	✓	Identifier mangling (IM)
✓	Compression (C)		**Regularization**
✓	Dead-code elimination (DCE)	✓	Dead-code insertion (DCI)
✓	Type upconversion (T)	✓	Subword regularization (SW)
✓	Constant folding (CF)	✗	Line subsampling (LS)

✓ = semantics-preserving transformation ✗ = lossy transformation

Table 1: We augment programs with 11 automated source-to-source compiler transforms. 10 are correct-by-construction and preserve operational semantics.

We implement our transpiler with the Babel and Terser compiler infrastructures (McKenzie et al., 2020; Santos et al., 2020) for the JavaScript programming language. In future work, a language-agnostic compiler (Koppel et al., 2018) could be used to extend ContraCode to other languages. Each compiler transformation is a function $\tau : \mathcal{P} \to \mathcal{P}$, where the space of programs $\mathcal{P}$ is composed of the set of valid ASTs and the set of programs in tokenized source form. Fig. 4 shows variants of an example program. Table 1 and Appendix A list program transformations in detail, but we broadly group them into three categories:

- **Code compression** changes the syntactic structure of code and performs correct-by-construction transforms such as precomputing constant expressions.
- **Identifier modifications** substitute method and variable names with random tokens, masking some human-readable information in a program but preserving functionality.
- Finally, **Regularizing transforms** improve model generalization by reducing the number of trivial positive pairs with high text overlap. The line subsampling pass in this group

Algorithm 1 Transform dropout for stochastic program augmentation.

Input: Program source x, transformation functions $\tau_1, \ldots \tau_k$, transform probabilities $p_1, \ldots p_k$, count N
Returns: N variants of x
$\mathcal{V} \leftarrow \{x\}$, a set of augmented program variants
for SAMPLE $i \leftarrow 1 \ldots N-1$ **do**
  $x' \leftarrow x$
  for transform $t \leftarrow 1 \ldots k$ **do**
    Sample $y_t \sim$ Bernoulli(p_t)
    if $y_t = 1$ **then**
      if REQUIRESAST$(\tau_t(\cdot))$ and $\neg$ISAST(x') **then** $x' \leftarrow$ PARSETOAST(x')
      else if $\neg$REQUIRESAST$(\tau_t(\cdot))$ and ISAST(x') **then** $x' \leftarrow$ LOWERTOSOURCE(x')
      $x' \leftarrow \tau_t(x')$
    end if
  end for
  if ISAST(x') **then** $x' \leftarrow$ LOWERTOSOURCE(x')
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{x'\}$
end for
return $\mathcal{V}$

potentially modifies program semantics.

3.2 Diversity through transform dropout

Stochastic augmentations in other modalities like random crops generate diverse outputs, but most of our compiler-based transformations are deterministic. To produce a diverse set of transformed programs, we randomly apply a subset of available compiler passes in a pre-specified order, applying transform τ_i with probability p_i. Intermediate programs are converted between AST and source form as needed for the compiler. Algorithm 1 details our transform dropout procedure.

Figure 5 measures the resulting diversity in programs. We precompute up to 20 augmentations of 1.8M JavaScript methods from GitHub. Algorithm 1 deduplicates method variants before pre-training since some transforms will leave the program unchanged. 89% of the methods have more than one alternative after applying 20 random sequences of transformations. The remaining methods without syntactically distinct alternatives include one-line functions that are obfuscated. We apply subword regularization (Kudo, 2018) as a final transformation to derive different tokenizations every batch, so pairs derived from the same original method will still differ. All transformations are fast; our compiler transforms 300 functions per second on a single CPU core.

3.3 Contrastive pre-training

We extend the Momentum Contrast (MoCo) methodology (He et al., 2019) that was designed for contrastive image representation learning. In our case, we learn a program encoder f_q that maps a sequence of program tokens to a single, fixed dimensional embedding. We organize programs into *functionally similar positive pairs* and *dissimilar negative pairs*. Generating two augmentations of the same GitHub program yields a positive pair (x^q, x^{k^+}), and an augmentation of a different program yields a negative x^{k^-}. The program x^q is called a "query" used to retrieve the corresponding "key" x^{k^+} during contrastive pre-training. We use these to shape representation space, drawing positives together and pushing away from negatives. Negatives are important to prevent the encoder f_q from mapping all programs to the same, trivial representation (Arora et al., 2019).

Pre-training objective Like He et al. (2019), we use the InfoNCE loss (Oord et al., 2018), a tractable objective that frames contrastive learning as a classification task: can the positives be identified among negatives? InfoNCE computes the probability of selecting the positive by taking the softmax of projected embedding similarities across a batch and a queue of negatives. Eq. (1) shows the InfoNCE loss, a function whose value is low when q is similar to the positive key embedding k^+ and dissimilar to negative key embeddings k^-. t is a temperature hyperparameter proposed by Wu et al. (2018).

$$-\log \frac{\exp(q \cdot k^+/t)}{\exp(q \cdot k^+/t) + \sum_{k^-} \exp(q \cdot k^-/t)} \quad (1)$$

The query representation $q = f_q(x^q)$ is computed by the encoder network f_q, and x^q is a query program. Likewise, $k = f_k(x^k)$ using a separate key encoder f_k. The summation $\sum_{k^-}$ in the normalizing denominator is taken over the queue of precomputed negatives in the batch.

Following He et al. (2019), to reduce memory consumption during pre-training, we cache embedded programs from past batches in a queue containing negative samples, as shown in Fig. 6. The query encoder f_q is trained via gradient descent while the key encoder f_k is trained slowly via an exponential moving average (EMA) of the query encoder parameters. The EMA update stabilizes the pre-computed key embeddings across training iterations. Since keys are only embedded once per epoch, we use a very large set of negatives, over 100K, with minimal additional computational cost and no explicit hard negative mining.

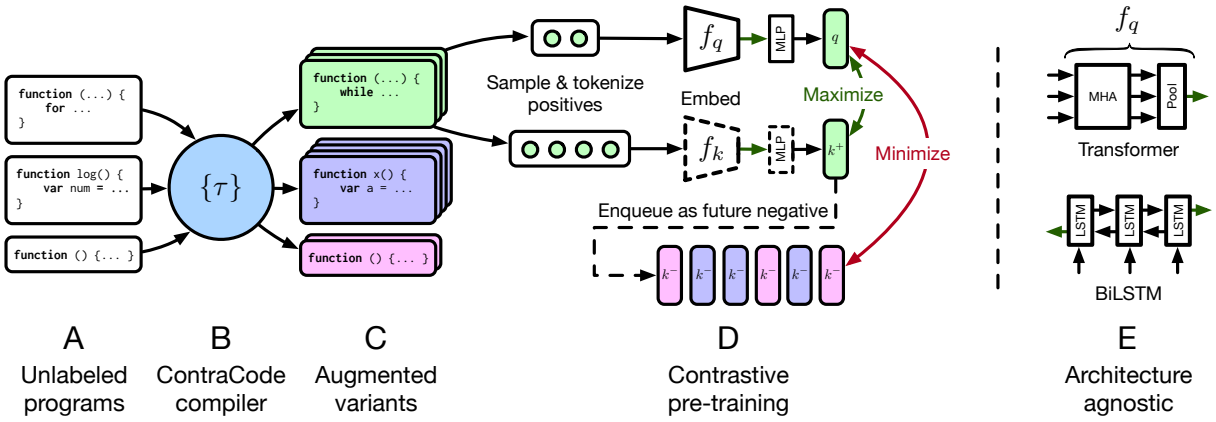


Figure 6: ContraCode pre-trains a neural program encoder f_q and transfers it to downstream tasks. **A-B.** Unlabeled programs are transformed **C.** into augmented variants. **D.** We pre-train f_q by maximizing similarity of projected embeddings of *positive* program pairs–variants of the same program–and minimizing similarity with a queue of cached negatives. **E.** ContraCode supports any architecture for f_q that produces a global program embedding such as Transformers and LSTMs. f_q is then fine-tuned on smaller labeled datasets.

ContraCode is agnostic to the architecture of the program encoder f_q. We evaluate contrastive pre-training of 6-layer Transformer (Vaswani et al., 2017) and 2-layer BiLSTM (Schuster and Paliwal, 1997; Huang et al., 2015) architectures (§4).

Transfer learning After pre-training converges, the encoder f_q is transferred to downstream tasks. For code clone detection, we use $f_q(x)$ without fine-tuning. For tasks where the output space differs from the encoder, we add a task-specific MLP or Transformer decoder after f_q, then fine-tune the resulting network end-to-end on labeled task data.

4 Evaluation

In order to evaluate whether ContraCode defend against adversarial code inputs, we benchmark adversarial code clone detection accuracy (Baker, 1992). We evaluate results over natural and adversarial edits. We then evaluate how improvements to adversarial robustness translate to improvements on established in-the-wild code benchmarks. While improvements on adversarial benchmarks would not be expected to translate to real code, we find significant improvements in extreme code summarization (Allamanis et al., 2016) and type inference (Hellendoorn et al., 2018) tasks.

Clone detection experiments show that contrastive and hybrid representations with our compiler-based augmentations are predictive of program functionality in-the-wild, and that contrastive representations are the most robust to adversarial edits (§4.1). Contrastive pre-training outperforms baseline supervised and self-supervised methods on all three tasks (§4.1-4.3). Finally, ablations suggest it is better to augment unlabeled programs during pre-training rather than augmenting smaller supervised datasets (§4.4).

Experimental setup Models are pre-trained on CodeSearchNet, a large corpus of methods extracted from popular GitHub repositories (Husain et al., 2019). CodeSearchNet contains 1,843,099 JavaScript programs. Only 81,487 methods have both a documentation string and a method name. The asymmetry between labeled and unlabeled programs stems from JavaScript coding practices where anonymous functions are widespread. The pre-training dataset described in Section 3.1 is the result of augmenting all 1.8M programs.

As our approach supports any encoder, we evaluate two architectures: a 2-layer Bidirectional LSTM with 18M parameters, similar to the supervised model used by Hellendoorn et al. (2018), and a 6-layer Transformer with 23M parameters. For a baseline self-supervised approach, we pre-train both architectures with the RoBERTa MLM objective, then transfer it to downstream tasks.

4.1 Robust Zero-shot Code Clone Detection

ContraCode learns to match variants of programs with similar functionality. While transformations produce highly diverse token sequences (quantified in the supplement), they are artificial and do not change the underlying algorithm. In contrast, human programmers can solve a problem with many data structures, algorithms and programming models. To determine whether pre-

	Natural code		Adversarial (N=4)		Adversarial (N=16)	
	AUROC	**AP**	**AUROC**	**AP**	**AUROC**	**AP**
Edit distance heuristic	69.55$\pm$0.81	73.75	31.63$\pm$0.82	42.85	12.11$\pm$0.54	32.46
Randomly initialized Transformer	72.31$\pm$0.79	75.82	22.72$\pm$0.20	37.73	3.09$\pm$0.28	30.95
+ RoBERTa MLM pre-train	74.04$\pm$0.77	77.65	25.83$\pm$0.21	39.46	4.51$\pm$0.33	31.17
+ ContraCode pre-train	75.73$\pm$0.75	78.02	**64.97**$\pm$0.24	**66.23**	**58.32**$\pm$0.88	**59.66**
+ ContraCode + RoBERTa MLM	**79.39**$\pm$0.70	**81.47**	37.81$\pm$0.24	51.42	10.09$\pm$0.50	32.52

Table 2: **Zero-shot code clone detection** with cosine similarity probe. Contrastive and hybrid representations improve clone detection AUROC on unmodified (natural) HackerRank programs by +8% and +10% AUROC over a heuristic textual similarity probe, respectively, suggesting they are predictive of functionality. Contrastive representations are also the most robust to adversarial code transformations.

```
function processData(input) {
    var parse_fun = function (s) { return parseInt(s, 10); };

    var lines = input.split('\n');
    var A = parse_fun(lines[0]);
    var B = parse_fun(lines[1])

    console.log(A + B);
}


process.stdin.resume();
process.stdin.setEncoding("ascii");
var _input = "";
process.stdin.on("data", function (input) { _input += input; });
process.stdin.on("end", function () { processData(_input); });
```

```
(function() {
  var input;

  process.stdin.setEncoding('ascii');

  input = "";

  var sum = function(a,b){return a+b}

  process.stdin.on('data', function(data) {
    if (data === "\n")
        process.stdin.emit("end");
    input += data;
  });

  process.stdin.on('end', function() {
    var sum = input.split("\n").reduce(function(a,b){return (+a)+(+b)});
    process.stdout.write(sum);
    process.exit(0);
  });
}).call(global);
```

Figure 7: Code clone detection example. These programs solve the same HackerRank coding challenge (reading and summing two integers), but use different coding conventions. The neural code clone detector should classify this pair as a positive, *i.e.* a clone.

trained representations are consistent across programs written by different people, we benchmark *code clone detection*, a binary classification task to detect whether two programs solve the same problem or different ones (Fig. 7). This is useful for deduplicating, refactoring and retrieving code, as well as checking approximate code correctness.

Benchmarks exist like BigCloneBench (Svajlenko et al., 2014), but to the best of our knowledge, there is no benchmark for the JavaScript. We collected 274 in-the-wild JavaScript programs that correctly solve 33 problems from the HackerRank interview preparation website. There are 2065 pairs solving the same problem and 70K pairs solving different problems, which we randomly subsample to 2065 to balance the classes.

Since we probe zero-shot performance based on pre-trained representations, there is no training set. Instead, we threshold cosine similarity of pooled representations of the programs u and v: $u^T v / \|u\| \|v\|$. Many code analysis methods for clone detection measure textual similarity (Baker, 1992). As a baseline, we threshold the dissimilarity score, a scaled Levenshtein edit distance between normalized and tokenized programs.

Table 2 reports the area under the ROC curve (AUROC) and average precision (AP, area under Precision-Recall). All learned representations improve over the heuristic on natural code. Self-supervision through RoBERTa MLM pre-training improves over a randomly initialized network by +1.7% AUROC. Contrastive pre-training achieves +3.4% AUROC over the same baseline. A hybrid objective combining both the contrastive loss and MLM has the best performance with +7.0% AUROC (+5.4% over MLM alone). Although MLM is still useful over natural code, ContraCode learns overall stronger representations of functionality.

However, are these representations robust to code edits? We adversarially edit one program in each pair by applying the loss-maximizing code compression and identifier modification transformation among N samples from Algorithm 1. These transformations preserve program function-

ality, so ground-truth labels are unchanged. With only 4 edits, RoBERTa performs worse than the heuristic (-5.8% AUROC) and worse than random guessing (50% AUROC), indicating it is highly sensitive to these kinds of implementation details. ContraCode retains much of its performance (+39% AUROC over RoBERTa) as it explicitly optimizes for invariance to code edits. Surprisingly, the hybrid model is less robust than ContraCode alone, perhaps indicating that MLM learns non-robust features (Ilyas et al., 2019).

4.2 Fine-tuning for Type Inference

JavaScript is a dynamically typed language, where variable types are determined at runtime based on the values they represent. Manually annotating code with types helps tools flag bugs by detecting incompatible types. Annotations also document code, but are tedious to maintain. Type inference tools automatically predict types from context.

To *learn* to infer types, we use the annotated dataset of TypeScript programs from DeepTyper (Hellendoorn et al., 2018), excluding GitHub repositories that were made private or deleted since publication. The training set contains 15,570 TypeScript files from 187 repositories with 6,902,642 total tokens. Validation and test sets are from held-out repositories. For additional supervision, missing types are inferred by static analysis to augment user-defined types as targets. A 2-layer MLP head predicts types from token embeddings output by the DeepTyper LSTM. We early stop based on validation set top-1 accuracy.

For the rest of our experiments, baseline RoBERTa models are pre-trained on the same *augmented* data as ContraCode for fair comparison. Learning representations that transfer from unlabeled JavaScript programs is challenging because TypeScript supports a superset of JavaScript's grammar, with types annotations and other syntactic sugar that need to be learned during fine-tuning. Further, the pre-training data only has methods while DeepTyper's dataset uses entire files (modules). The model is only given source code for a single file, not dependencies.

In Table 3, contrastive pre-training outperforms all baseline learned methods. ContraCode is applied in a drop-in fashion to each of the baselines. Pre-training with our contrastive objective and data augmentations yields absolute accuracy improvements of +1.2%, +6.3%, +2.3% top-1 and +1.8%, +5.7%, +2.8% top-5 over the Transformer, RoBERTa, and DeepTyper, respectively.

Method	Acc@1	Acc@5
TypeScript CheckJS	45.11%	—
DeepTyper, variable name only	28.94%	70.07%
GPT-3 Codex (zero-shot, 175B)	26.62%	—
GPT-3 Codex (few-shot, 175B)	30.55%	—
Transformer	45.66%	80.08%
+ RoBERTa MLM pre-train	40.85%	75.76%
+ ContraCode pre-train	46.86%	81.85%
+ ContraCode + MLM (hybrid)	**47.16%**	**81.44%**
DeepTyper BiLSTM	51.73%	82.71%
+ RoBERTa MLM pre-train	50.24%	82.85%
+ ContraCode pre-train	**54.01%**	**85.55%**

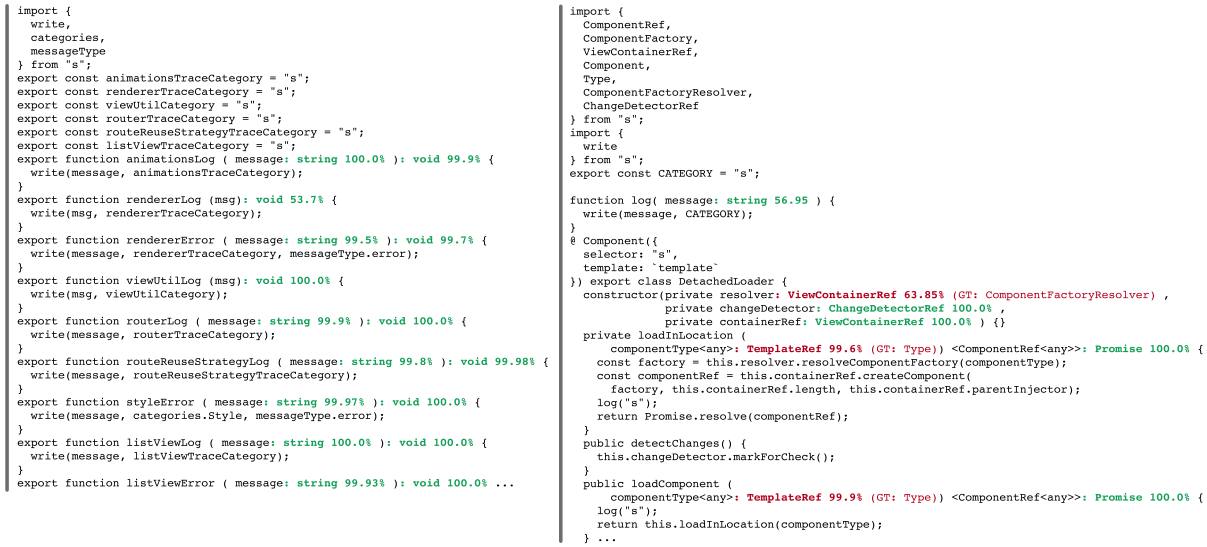
Table 3: **Type inference accuracy on TypeScript programs**. As ContraCode does not modify model architecture, contrastive pre-training improves both BiLSTM and Transformer accuracy (1.5% to 2.28%). Compared with TypeScript's built-in type inference, we improve accuracy by 8.9%.

The RoBERTa baseline may perform poorly since the MLM objective focuses on token reconstruction that is overly sensitive to local syntactic structure, or because sufficient fine-tuning data is available, described as weight "ossification" by Hernandez et al. (2021). To combine the approaches, we minimized our loss in addition to MLM as a hybrid local-global objective to pre-training a Transformer, improving accuracy by +6.31% over the RoBERTa Transformer.

We also evaluate the recent GPT-3 Codex model by OpenAI (Chen et al., 2021) using their API. We benchmark the 175B parameter DaVinci model in both a zero-shot as well as a few-shot prompting setup. Although the Codex model was trained over TypeScript programs, it performs poorly as it achieves an accuracy of 26.6% in the zero-shot setup and 30.6% in the few-shot setup. We only evaluate Top-1 accuracy for GPT-3 models as GPT-3 does not reliably output confidence scores.

Learning outperforms static analysis by a large margin. Overall, our best model has +8.9% higher top-1 accuracy than the built-in TypeScript CheckJS type inference system, showing the promise of learned code analysis. Surfacing multiple candidate types can also be useful to users, while CheckJS only has a single prediction.

Fig. 8 shows two files from held-out repositories. For the first, our model consistently predicts the correct return and parameter types. The



```
import {
  write,
  categories,
  messageType
} from "s";
export const animationsTraceCategory = "s";
export const rendererTraceCategory = "s";
export const viewUtilCategory = "s";
export const routerTraceCategory = "s";
export const routeReuseStrategyTraceCategory = "s";
export const listViewTraceCategory = "s";
export function animationsLog ( message: string 100.0% ): void 99.9% {
  write(message, animationsTraceCategory);
}
export function rendererLog (msg): void 53.7% {
  write(msg, rendererTraceCategory);
}
export function rendererError ( message: string 99.5% ): void 99.7% {
  write(message, rendererTraceCategory, messageType.error);
}
export function viewUtilLog (msg): void 100.0% {
  write(msg, viewUtilCategory);
}
export function routerLog ( message: string 99.9% ): void 100.0% {
  write(message, routerTraceCategory);
}
export function routeReuseStrategyLog ( message: string 99.8% ): void 99.98% {
  write(message, routeReuseStrategyTraceCategory);
}
export function styleError ( message: string 99.97% ): void 100.0% {
  write(message, categories.Style, messageType.error);
}
export function listViewLog ( message: string 100.0% ): void 100.0% {
  write(message, listViewTraceCategory);
}
export function listViewError ( message: string 99.93% ): void 100.0% ...
```

```
import {
  ComponentRef,
  ComponentFactory,
  ViewContainerRef,
  Component,
  Type,
  ComponentFactoryResolver,
  ChangeDetectorRef
} from "s";
import {
  write
} from "s";
export const CATEGORY = "s";

function log( message: string 56.95 ) {
  write(message, CATEGORY);
}
@ Component({
  selector: "s",
  template: `template`
}) export class DetachedLoader {
  constructor(private resolver: ViewContainerRef 63.85% (GT: ComponentFactoryResolver) ,
              private changeDetector: ChangeDetectorRef 100.0% ,
              private containerRef: ViewContainerRef 100.0% ) {}
  private loadInLocation (
      componentType<any>: TemplateRef 99.6% (GT: Type)) <ComponentRef<any>>: Promise 100.0% {
    const factory = this.resolver.resolveComponentFactory(componentType);
    const componentRef = this.containerRef.createComponent(
      factory, this.containerRef.length, this.containerRef.parentInjector);
    log("s");
    return Promise.resolve(componentRef);
  }
  public detectChanges() {
    this.changeDetector.markForCheck();
  }
  public loadComponent (
      componentType<any>: TemplateRef 99.9% (GT: Type)) <ComponentRef<any>>: Promise 100.0% {
    log("s");
    return this.loadInLocation(componentType);
  } ...
```

Figure 8: A variant of DeepTyper pre-trained with ContraCode generates type annotations for two held-out programs. The model consistently predicts correct function return types, and often correctly predicts project-specific variable types imported at the top of the file. Metrics are in the top row of Table 8 (not our best performing model).

Method	Precision	Recall	F1
code2vec	10.78%	8.24%	9.34%
code2seq	12.17%	7.65%	9.39%
RoBERTa MLM	15.13%	11.47%	12.45%
Transformer	18.11%	15.78%	16.86%
+ ContraCode	20.34%	14.96%	**17.24%**

Table 4: Results for different settings of **code summarization**: supervised training with 81K functions, masked language model pre-training, training from scratch and contrastive pre-training with fine-tuning.

model correctly predicts that the variable `message` is a string, even though its type is ambiguous without access to the imported `write` method signature. For the second, ContraCode predicts 4 of 8 types correctly including `ViewContainerRef` and `ChangeDetectorRef` from the AngularJS library.

4.3 Extreme Code Summarization

The extreme code summarization task asks a model to predict the name of a method given its body (Allamanis et al., 2016). These names often summarize the method, such as `reverseString(...)`. Summarization models could help programmers interpret poorly documented code. We create a JavaScript summarization dataset using the 81,487 labeled methods in the CodeSearchNet dataset. The name is masked in the method declaration. A sequence-to-sequence model with an autoregressive decoder is trained to

```
function x(url, callback, error) {
  var img = new Image();
  img.src = url;
  if(img.complete){
    return callback(img);
  }
  img.onload = function(){
    img.onload = null;
    callback(img);
  };
  img.onerror = function(e){
    img.onerror = null;
    error(e);
  };
}
```

Ground truth: `loadImage`
Prediction: `loadImage`

Top predictions:

1. `getImageItem`
2. `createImage`
3. `loadImageForBreakpoint`
4. `getImageSrcCSS`

Figure 9: A held-out JavaScript program from CodeSearchNet and method names generated by a Transformer pre-trained with ContraCode. The correct method name is predicted as the most likely decoding.

maximize log likelihood of the ground-truth name, a form of abstractive summarization. All models overfit, so we stop early according to validation loss. As proposed by Allamanis et al. (2016), we evaluate model predictions by precision, recall and F1 scores over the set of method name tokens.

Table 4 shows results in four settings: (1) supervised training using baseline tree-structured architectures that analyze the AST (code2vec, code2seq), (2) pre-training on all 1.8M programs using MLM followed by fine-tuning on the labeled programs (RoBERTa), (3) training a Transformer from scratch and (4) contrastive pre-training followed by fine-tuning with augmentations.

Contrastive pre-training outperforms code2seq by +8.2% test precision, +7.3% recall, and +7.9% F1 score. ContraCode outperforms self-

Code summarization model	F1
Transformer (Table 4)	**16.86**
+ augmentations	15.65

Type inference model	Acc@1
Transformer (Table 3)	**45.66**
+ augmentations	44.14
DeepTyper (Table 3)	**51.73**
+ augmentations	50.33

Table 5: Compiler data augmentations degrade performance when training supervised models *from scratch*.

Pre-training augmentations	Acc@1	Acc@5
All augmentations (Table 3)	**52.65%**	**84.60%**
w/o identifier modification (-VR, -IM)	51.94%	84.43%
w/o line subsampling (-LS)	51.05%	81.63%
w/o code compression (-T,C,DCE,CF)	50.69%	81.95%

Table 6: Ablating compiler transformations used during contrastive pre-training. The DeepTyper BiLSTM is pre-trained with constrastive learning for 20K steps, then fine-tuned for type inference. Augmentations are only used during pre-training. Each transformation contributes to accuracy.

supervised pre-training with RoBERTa by +4.8% F1. ContraCode also achieves slightly higher performance than the Transformer learned from scratch. While this improvement is smaller, code summarization challenging as identifier names are not consistent between programmers.

Figure 9 shows a qualitative example of predictions for the code summarization task. The JavaScript method is not seen during training. A Transformer pre-trained with ContraCode predicts the correct method name through beam search. The next four predictions are reasonable, capturing that the method processes an image. The 2nd and 3rd most likely decodings, `getImageItem` and `createImage`, use `get` and `create` as synonyms for `load`, though the final two unlikely decodings include terms not in the method body.

4.4 Understanding augmentation importance

We analyze the effect of augmentations on supervised learning and on pre-training.

Supervised learning with augmentations As a baseline, we re-train models from scratch with compiler transforms during *supervised learning* rather than pre-training. Data augmentation artificially expands labeled training sets. For sequence-to-sequence summarization, we apply a variety of augmentations (LS, SW, VR, DCI). These all preserve the method name. For type inference, labels are aligned to input tokens, so they must be realigned after transformation. We only apply token-level transforms (LS, SW) as we can track labels.

Table 5 shows results. Compiler-based data augmentations degrade supervised models, perhaps by creating a training distribution not reflective of evaluation programs. However, as shown in §4.1–4.3, augmenting during ContraCode pre-training yields a more accurate model. Our contrastive learning framework also allows learning over large numbers of unlabeled programs that supervised learning alone cannot leverage. The ablation indicates that augmentations do not suffice, and contrastive learning is important.

Ablating pre-training augmentations Some data augmentations could be more valuable than others. Empirically, pre-training converges faster with a smaller set of augmentations at the same batch size since the positives are syntactically more similar, but this hurts downstream performance. Table 6 shows that type inference accuracy degrades when different groups of augmentations are removed. Semantics-preserving code compression passes that require code analysis are the most important, improving top-1 accuracy by 1.95% when included. Line subsampling serves as a regularizer, but changes program semantics. LS is relatively less important, but does help accuracy. Identifier modifications preserve semantics, but change useful naming information.

5 Conclusion

Large-scale code repositories like GitHub are a powerful resource for learning machine-aided programming tools. However, most current code representation learning approaches need labels, and popular label-free self-supervised methods like RoBERTa are not robust to adversarial inputs. Instead of reconstructing tokens like BERT, learning *what code says*, we learn *what code does*. We propose ContraCode, a contrastive self-supervised algorithm that learns representations invariant to transformations via compiler-based data augmentations. In experiments, ContraCode learns effective representations of functionality, and is robust to adversarial code edits. We find that ContraCode significantly improves performance on three downstream JavaScript code understanding tasks.

Acknowledgments

We thank Lisa Dunlap, Jonathan Ho, Koushik Sen, Rishabh Singh, Aravind Srinivas, Daniel Rothchild, and Justin Wong for helpful feedback. In addition to NSF CISE Expeditions Award CCF-1730628, the NSF GRFP under Grant No. DGE-1752814, and ONR PECASE N000141612723, this research is supported by gifts from Amazon Web Services, Ant Financial, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, NVIDIA, Scotiabank, Splunk and VMware.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 143–153, New York, NY, USA. Association for Computing Machinery.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81.

Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Programming Language Design and Implementation (PLDI)*.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (ICML)*.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*.

Sanjeev Arora, Hrishikesh Khandeparkar, Mikhail Khodak, Orestis Plevrakis, and Nikunj Saunshi. 2019. A theoretical analysis of contrastive unsupervised representation learning. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5628–5637. PMLR.

Brenda S. Baker. 1992. A program for identifying duplicated code. *Computing Science and Statistics*.

Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 394–403, New York, NY, USA. Association for Computing Machinery.

Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *NeurIPS*.

Samuel Benton, Ali Ghanbari, and Lingming Zhang. 2019. Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 47–50. IEEE.

Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. *CoRR*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020a. A simple framework for contrastive learning of visual representations. In *International Conference on Macnine Learning*.

Xinlei Chen, Haoqi Fan, Ross Girshick, and Kaiming He. 2020b. Improved baselines with momentum contrastive learning. *arXiv preprint arXiv:2003.04297*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics*.

Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. 2010. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. 2018. A public unified bug dataset for Java. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 12–21.

James Fogarty, Ryan S. Baker, and Scott E. Hudson. 2005. Case studies in the use of roc curve analysis for sensor-based estimates in human computer interaction. In *Proceedings of Graphics Interface 2005*, GI '05, page 129–136, Waterloo, CAN. Canadian Human-Computer Communications Society.

Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2019. Momentum contrast for unsupervised visual representation learning. *arXiv preprint arXiv:1911.05722*.

Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 152–162.

Olivier J Hénaff, Aravind Srinivas, Jeffrey De Fauw, Ali Razavi, Carl Doersch, S. M. Ali Eslami, and Aaron van den Oord. 2019. Data-efficient image recognition with contrastive predictive coding. In *International Conference on Machine Learning*.

Dan Hendrycks, Mantas Mazeika, Saurav Kadavath, and Dawn Song. 2019. Using self-supervised learning can improve model robustness and uncertainty. *Advances in Neural Information Processing Systems (NeurIPS)*.

Danny Hernandez, Jared Kaplan, Tom Henighan, and Sam McCandlish. 2021. Scaling laws for transfer.

Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. 2019. Adversarial examples are not bugs, they are features. In *Advances in Neural Information Processing Systems*, volume 32, pages 125–136. Curran Associates, Inc.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083.

Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 304–314, New York, NY, USA. Association for Computing Machinery.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Pre-trained contextual embedding of source code. *ArXiv*, abs/2001.00059.

Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11.

James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One tool, many languages: Language-parametric transformation with incremental parametric syntax. *Proc. ACM Program. Lang.*, 2(OOPSLA).

Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

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

Henry Massalin. 1987. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, page 122–126, Washington, DC, USA. IEEE Computer Society Press.

Sebastian McKenzie et al. 2020. Babel: compiler for writing next generation javascript. `https://github.com/babel/babel`.

Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, pages 4505–4515.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Dana Movshovitz-Attias and William Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 35–40.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*.

Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. 2020. Opttyper: Probabilistic type inference by optimising logical and natural constraints.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2019. Typewriter: Neural type prediction with search-based validation. *arXiv preprint arXiv:1912.03768*.

Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25.

Md. Rafiqul Islam Rabin and Mohammad Amin Alipour. 2020. Evaluation of generalizability of neural program analyzers under semantic-preserving transformations.

Fábio Santos et al. 2020. Terser: Javascript parser, mangler and compressor toolkit for es6+. `https://github.com/terser/terser`.

Mike Schuster and Kuldip Paliwal. 1997. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45:2673 – 2681.

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, page 476–480, USA. IEEE Computer Society.

Wilson L Taylor. 1953. "Cloze procedure": A new tool for measuring readability. *Journalism Quarterly*, 30(4):415–433.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Ke Wang and Mihai Christodorescu. 2019. COSET: A benchmark for evaluating neural program embeddings. *arXiv preprint arXiv:1905.11445*.

Ke Wang and Zhendong Su. 2019. Learning blended, precise semantic program embeddings. *ArXiv*.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE.

Zhirong Wu, Yuanjun Xiong, Stella X Yu, and Dahua Lin. 2018. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3733–3742.

Noam Yefet, Uri Alon, and Eran Yahav. 2019. Adversarial examples for models of code. *arXiv preprint arXiv:1910.07517*.

Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30.

Appendices

A Program transformation details

We use the Babel compiler infrastructure (McKenzie et al., 2020) and the `terser` JavaScript library for AST-based program transformations. We perform variable renaming and dead code insertion (variable declaration insertion) using custom Babel transforms, subword regularization with `sentencepiece` Python tokenization library, line subsampling using JavaScript string manipulation primatives and other transformations with `terser`. Terser has two high-level transformation modes, mangling and compression, each with finer grained controls such as formatting, comment and log removal, and dead code elimination. We show an example merge sort with variants in Figure 10.

Reformatting, beautification, compression (R, B, C): Personal coding conventions do not affect the semantics of code; auto-formatting normalizes according to a style convention.

Dead-code elimination (DCE): In this pass, all unused code with no side effects are removed. Various statements can be inlined or removed as stale or unneeded functionality.

Type upconversion (T): In JavaScript, some types are polymorphic & can be converted between each other. As an example, booleans can be represented as `true` or as `1`.

Constant folding (CF): During constant folding, all expressions that can be pre-computed at compilation time can be inlined. For example, the expression `(2 + 3) * 4` is replaced with `20`.

Variable renaming, identifier mangling (VR, IM): Arguments can be renamed with random word sequences and identifiers can be replaced with short tokens to make the model robust to naming choices. Program behavior is preserved despite obfuscation.

Dead-code insertion (DCI): Commonly used no-ops such as comments and logging are inserted.

Subword regularization (SW): From Kudo (2018), text is tokenized in several different ways, with a single word (`_function`) or subtokens (`_func tion`).

Line subsampling (LS): We randomly sample ($p = 0.9$) lines from a method body. While not semantics-preserving, line subsampling serves as a regularizer.

```
// Split the array into halves and merge
    them recursively
function mergeSort (arr) {
  if (arr.length === 1) {
    // return once we hit an array with a
        single item
    return arr
  }
  const middle = Math.floor(arr.length / 2)
  // get the middle item of the array
      rounded down
  const left = arr.slice(0, middle)
  // items on the left side
  const right = arr.slice(middle)
  // items on the right side
  return merge(
    mergeSort(left),
    mergeSort(right)
  )
}
```

Original merge sort program

```
function mergeSort(e) {
  if (e.length === 1) {
    return e;
  }
  const t = Math.floor(e.length / 2);
  const l = e.slice(0, t);
  const n = e.slice(t);
  return merge(mergeSort(l), mergeSort(n));
}
```

Variable renaming, comment removal, reformatting

```
function mergeSort(e) {
  if (1 === e.length) return e;
  const t = Math.floor(e.length / 2), r =
      e.slice(0, t), n = e.slice(t);
  return merge(mergeSort(r), mergeSort(n));
}
```

Combining variable declarations, inlining conditional

Figure 10: Given a JavaScript code snippet implementing the merge sort algorithm, we apply semantics-preserving transformations to produce functionally-equivalent yet textually distinct code sequences. Variable renaming and identifier mangling passes change variable names. Compression passes eliminate unnecessary characters such as redundant variable declarations and brackets.

B How similar are transformed programs?

To understand the diversity created by program transformations, we compute the Levenshtein

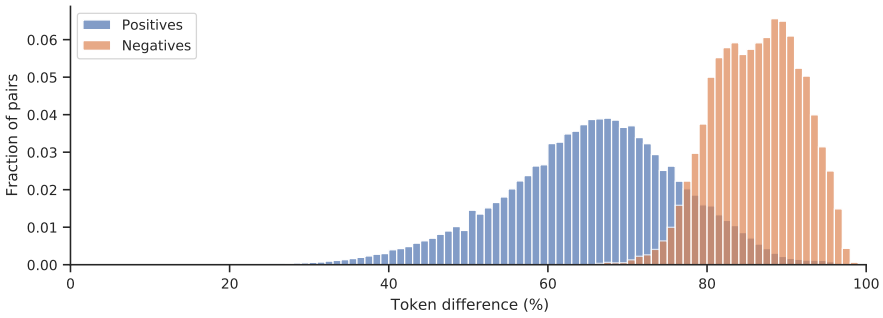


Figure 11: Histogram of pairwise token dissimilarity for contrastive positives (transformed variants of the same method) and negatives (transformed variants of different methods). Code transformations produce positives with dissimilar token sequences.

minimum edit distance between positive pairs in the precomputed pre-training dataset, *i.e.* transformed variants of the same source method. For comparison, we also compute the edit distance between negative pairs: transformed variants of different programs.

The edit distance $D(x_q, x_k)$ computes the minimum number of token insertions, deletions or substitutions needed to transform the tokenized query progrm x_q into the key program x_k. To normalize by sequence length $|\cdot|$, let

$$\text{dissimilarity}_D(x_q, x_k) = \frac{D(x_q, x_k)}{\max(|x_q|, |x_k|)} \quad (2)$$

Dissimilarity ranges from 0% for programs with the same sequence of tokens, to 100% for programs without any shared tokens. Note that whitespace transformations do not affect the metric because the tokenizer collapses repeated whitespace. For the positives, we estimate dissimilarity by sampling one pair per source program in the CodeSearchNet dataset (1.6M source programs with at least one pair). We sample the same number of negative pairs.

Fig. 11 shows a histogram of token dissimilarity. Positive pairs have 65% mean dissimilarity, while negatives have 86%. Negatives are more dissimilar on average as source sequences could have different lengths, idioms and functionality. Still, the transformations generated quite different positive sequences, with less than half of their tokens shared. The 25th, median and 75th percentile dissimilarity is 59%, 66% and 73% for positives, and 82%, 87% and 90% for negatives.

C Experimental setup

Architectures The Transformer encoder has 6 layers (23M parameters) in all experiments. For code summarization experiments, we add 4 decoder layers with causal masking to generate the natural language summary. We leverage the default positional embedding function (sin, cos) as used in the original Transformer architecture. The network originally proposed in DeepTyper (Hellendoorn et al., 2018) had 11M parameters with a 300 dimensional hidden state. We increase the hidden state size to 512 to increase model capacity, so our BiLSTM for type prediction has 17.5M parameters. During fine-tuning, across all experiments, we optimize parameters using Adam with linear learning rate warmup and decay. For the Transformer, the learning rate is linearly increased for 5,000 steps from 0 to a maximum of 10^{-4}. For the bidirectional LSTM, the learning rate is increased for between 2,500 and 10,000 steps to a maximum of 10^{-3}. Type inference hyperparameters are selected by validation top-1 accuracy.

ContraCode pre-training The InfoNCE objective is minimized with temperature $t = 0.07$ following He et al. (2019). Also following He et al. (2019), the key encoder's parameters are computed with the momentum update equation $\theta_k \leftarrow m\theta_k + (1-m)\theta_q$, equivalent to an EMA of the query encoder parameters θ_q. To pretrain a Transformer using the ContraCode objective, we first embed each token in the program using the Transformer. However, the InfoNCE objective is defined in terms of a single embedding for the full program. The ContraCode Transformer is pre-trained with a batch size of 96. Our model averages the 512-dimensional token embeddings across the sequence, then applies a two-layer MLP

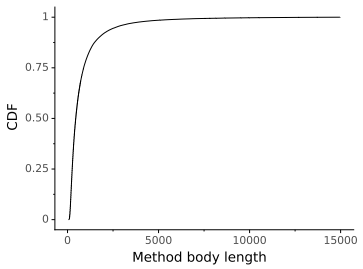
with 512 hidden units and a ReLU activation to extract a 128-dimensional embedding for the loss.

The DeepTyper bidirectional LSTM architecture has two choices for extracting a global program representation. We aggregate a 1024-dimensional representation of the program by concatenating its four terminal hidden states (from two sequence processing directions and two stacked LSTM layers), then apply the same MLP architecture as before to extract a 128-dimensional representation. Alternatively, we can average the hidden state concatenated from each direction across the tokens in the sequence before applying the MLP head. We refer to the hidden-state configuration as a global representation and the sequence averaging configuration as a local representation in Tab. 8. We pre-train the BiLSTM with large batch size of 512 and apply weight decay.

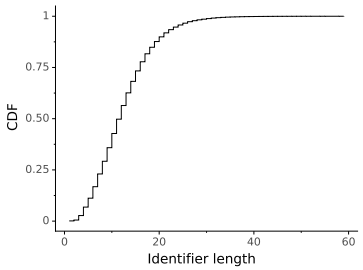
Code clone detection on HackerRank programs Figure 7 shows two programs sampled from the HackerRank clone detection dataset. These programs successfully solve the same problem, so they are clones. We report metrics that treat code clone detection as a binary classification task given a pair of programs. 2065 pairs of programs solving the same HackerRank problem and 2065 pairs of programs solving different problems are sampled to construct an evaluation dataset. We use the area under the Receiver Operating Characteristic (AUROC) metric and Average Precision (AP) metrics. The standard error of the AUROC is reported according to the Wilcoxon statistic (Fogarty et al., 2005). Average Precision is the area under the Precision-Recall curve. AUROC and AP are both computed using the `scikit-learn` library (Pedregosa et al., 2011).

A Transformer predicts contextual embeddings of each token in a program, but our thresholded cosine similiarity classifier requires fixed length embeddings of whole programs. To determine if two programs that may differ in length are clones, we pool the token representations across the sequence. We evaluated both mean pooling and max pooling the representation. For the hybrid model pre-trained with both RoBERTa (MLM) and contrastive objectives, mean pooling achieved the best AUROC and AP. For other models, max pooling performed the best.

Type prediction Following DeepTyper (Hellendoorn et al., 2018), our regenerated dataset for type prediction has 187 training projects with



(a) Character length per code sample



(b) Character length per method name

Figure 12: CodeSearchNet code summarization dataset statistics: (a) The majority of code sequences are under 2000 characters, but there is long tail of programs that span up to 15000 characters long, (b) JavaScript method names are relatively short compared to languages like C♯ and Java.

15,570 TypeScript files, totaling 6,902,642 tokens. We tune hyperparameters on a validation set of 23 distinct projects with 1,803 files and 490,335 tokens, and evaluate on a held-out test set of 24 projects with 2,206 files and 958,821. The training set is smaller than originally used in DeepTyper as several projects were made private or deleted from GitHub before May 2020 when we downloaded the data, but we used the same commit hashes for available projects so our splits are a subset of the original. We have released the data with our open-source code to facilitate further work on a stable benchmark as more repositories are deleted over time. We perform early stopping to select the number of training epochs. We train each model for 100 epochs and select the checkpoint with the minimum accuracy@1 metric (all types, including `any`) on the validation set. Except for the model learned from scratch, the Transformer architectures are pre-trained for 240K steps. Models with the DeepTyper architecture converge faster on the pre-training tasks and are pre-trained for 20K iterations (unless otherwise noted).

Extreme code summarization by method name prediction We train method prediction models using the labeled subset of CodeSearch-

Net. Neither method names nor docstrings are provided as input to the model: the docstring is deleted, and the method name is replaced with the token 'x'. Thus, the task is to predict the method name using the method body and comments alone.

To decode method names from all models except the code2vec and code2seq baselines which implement their own decoding procedures, we use a beam search with a beam of size 5 and a maximum target sequence length of 20 subword tokens. We detail the cumulative distribution of program lengths in Figure 12. The ContraCode summarization Transformer only needed to be pre-trained for 20K iterations, with substantially faster convergence than RoBERTa (240K iterations). During fine-tuning, we apply the LS,SW,VR,DCI augmentations to ContraCode.

D Baselines

Baselines for code summarization and type prediction trained their models on an inconsistent set of programming languages and datasets. In order to normalize the effect of datasets, we selected several diverse state-of-the-art baselines and reimplemented them on the JavaScript dataset.

AST-based models The authors of code2vec (Alon et al., 2019b) and code2seq (Alon et al., 2019a), AST-based code understanding models, made both data and code available, but train their model on the Java programming language. In order to extend the results in their paper to JavaScript for comparison with our approach, we generated an AST path dataset for the CodeSearchNet dataset. The sensitivity of path-mining embeddings to different datasets is documented in prior work, so published F1 scores are not directly comparable; F1 scores for code2vec (Alon et al., 2019b) vary between 19 (Alon et al., 2019a) and 43 (Alon et al., 2019b) depending on the dataset used. Therefore, we use the same dataset generation code as the authors for fair comparison. We first parse the source functions using the Babel compiler infrastructure. Using the original code on these ASTs, up to 300 token-to-token (leaf-to-leaf) paths are extracted from each function's AST as a precomputed dataset. Then, we generate a token and AST node vocabulary using the same author-provided code, and train the models for 20 epochs, using early stopping for code2seq. We observed that code2vec overfits after 20 epochs, and longer training was not beneficial.

DeepTyper (Hellendoorn et al., 2018) DeepTyper uses a two layer GRU with a projection over possible classes, with an embedding size of 300 and hidden dimension of 650. However, we found improved performance by replacing the GRU with a bidirectional LSTM (BiLSTM). We normalize the LSTM parameter count to match our model, and therefore use a hidden dimension size of 512. We also use subword tokenization rather than space delimited tokens according to Kudo (2018), as subwords are a key part of state-of-the-art models for NLP (Sennrich et al., 2015).

RoBERTa We pre-trained an encoder using RoBERTa's masked language modeling loss on our augmented version of CodeSearchNet, the same data used to pre-train ContraCode. This model is then fine-tuned on downstream datasets. Unlike the original BERT paper which cuBERT (Kanade et al., 2020) is based on, hyperparameters from RoBERTa have been found to produce better results during pre-training. RoBERTa pre-trains using a masked language modeling (MLM) objective, where 15% of tokens in a sentence are masked or replaced and are reconstructed by the model. We did not use the BERT Next Sentence Prediction (NSP) loss which RoBERTa finds to be unnecessary. We normalize baseline parameter count by reducing the number of Transformer layers from 24 to 6 for a total of 23M parameters.

E Additional results and ablations

Code clone detection ROC, PR curves Fig. 13 plots true postive rate vs false positive rate and precision vs recall for different zero-shot classifiers on the code clone detection downstream tasks. These classifiers threshold a similarity score given by token-level edit distance for the heuristic approach or cosine similarity for the neural network representations. The hybrid self-supervised model combining ContraCode's contrastive objective and MLM achieves better tradeoffs than the other approaches. Fig. 14 shows the AUROC and Average Precision of four Transformer models on the same task under adversarial transformations of one input program. Untrained models as well as models pre-trained with RoBERTa's MLM objective are not robust to these code transformations. However, the model pre-trained with ContraCode preserves much of its performance as the adversarial attack is strengthened.

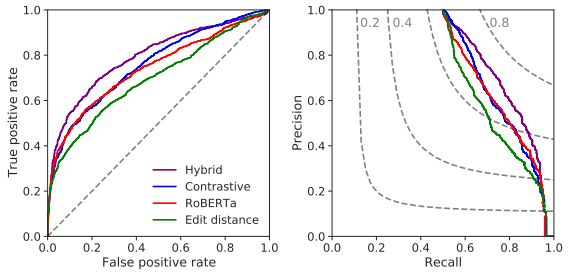


Figure 13: Receiver Operating Characteristic (ROC) and Precision-Recall (PR) curves for non-adversarial classifiers on the code clone detection task. Equal F1 score curves are shown on right.

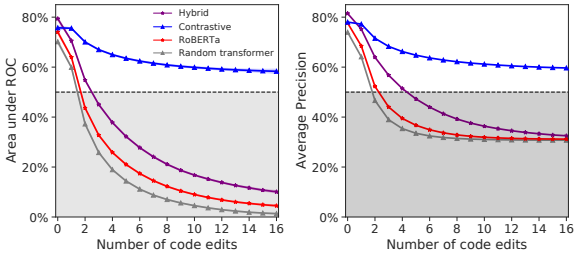


Figure 14: Adversarial AUROC and Average Precision for four models on the code clone detection task: a randomly initialized transformer, and transformers pre-trained on code with the RoBERTa MLM objective, our contrastive objective, or both. Representations learned by the contrastive model transfer robustly.

Which part of the model should be transferred? SimCLR (Chen et al., 2020a) proposed using a small MLP head to reduce the dimensionality of the representation used in the InfoNCE loss during pre-training, and did not transfer the MLP to the downstream image-classification task. In contrast, we find it beneficial to transfer part of the contrastive MLP head to type inference, showing a 2% improvement in top-5 accuracy over transferring the encoder only (Table 7). We believe the improvement stems from fine-tuning both the encoder and MLP which allows feature adaptation, while SimCLR trained a linear model on top of frozen features. We only transferred the MLP when contrasting the mean of token embeddings during pre-training, not the terminal hidden states, as the dimensionality of the MLP head differs. These representations are compared next.

Should we pre-train global or local representations? We compare pre-training DeepTyper with two variants of ContraCode. We either use the mean of token hidden states across the program (averaging local features), or the terminal hidden states as input to the MLP used to extract the contrastive representation $q = f_q(x)$ (global features). Token-level features might capture more syntactic details, but averaging pooling ignores order. Table 8 shows the accuracy of a BiLSTM pre-trained with each strategy. Using the global features for pre-training yields significantly improved performance, +2.38% acc@1 after 10K iterations of pre-training (not converged for the purposes of ablation). The global pre-training strategy achieves our best results.

Table 7: If local representations are learned, transferring part of the Contrastive MLP head improves type inference. The encoder is a 2-layer BiLSTM (d=512), with a 2-layer MLP head for both pre-training purposes and type inference. The mean hidden state representation is optimized for 10K iterations for the purposes of this ablation.

Warm-started layers	Acc@1	Acc@5
BiLSTM	**49.32%**	80.03%
BiLSTM, 1 layer of MLP	49.15%	**82.58%**

Do pre-trained encoders help more with shallow decoders? For the sequence-to-sequence code summarization task, ContraCode only pre-trains the encoder of the Transformer. In Table 9, we ablate the depth of the decoder to understand how much shallow decoders benefit from contrastive pre-training of the encoder. Similar experiments were performed in a vision context by (Erhan et al., 2010), where different numbers of layers of a classifier are pre-trained. After 45k pre-training steps, the 4-layer decoder achieves 0.50% higher precision, 0.64% higher recall and 0.77% higher F1 score than the 1-layer model, so additional decoder depth is helpful for the downstream task. The 1-layer decoder model also benefits significantly from longer pre-training, with a 6.3% increase in F1 from 10k to 45k iterations. This large of an improvement indicates that ContraCode could be more helpful for pre-training when the number of randomly initialized parameters at the start of fine-tuning is small. For larger decoders, more parameters must be optimized during-finetuning, and the value of pre-training is diminished.

Contrastive representation learning strategies In Figure 15, we compare two strategies of refreshing the MoCo queue of key embeddings (the dictionary of negative program representations assumed to be non-equivalent to the batch of positives). In the first strategy, we add 8 items out

Table 8: Contrasting global, sequence-level representations outperforms contrasting local representations. We compare using the terminal (global) hidden states of the DeepTyper BiLSTM and the mean pooled token-level (local) hidden states.

Representation	Optimization	Acc@1	Acc@5
Global	InfoNCE with terminal hidden state, 20K steps	**52.65%**	**84.60%**
	InfoNCE with terminal hidden state, 10K steps	51.70%	83.03%
Local	InfoNCE with mean token rep., 10K steps	49.32%	80.03%

Table 9: Training time and decoder depth ablation on the method name prediction task. Longer pre-training significantly improves downstream performance when a shallow, 1 layer decoder is used.

Decoder	Pre-training (1.8M programs)	Supervision (81k programs)	Precision	Recall	F1
Transformer, 1 layer	MoCo, 10k steps	Original set	11.91%	5.96%	7.49%
Transformer, 1 layer	MoCo, 45k steps	Original set	**17.71%**	**12.57%**	**13.79%**
Transformer, 4 layers	MoCo, 45k steps	Original set	**18.21%**	**13.21%**	**14.56%**

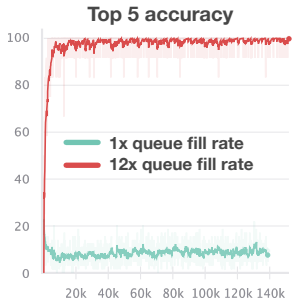


Figure 15: Pre-training quickly converges if negative programs in the queue are frequently changed.

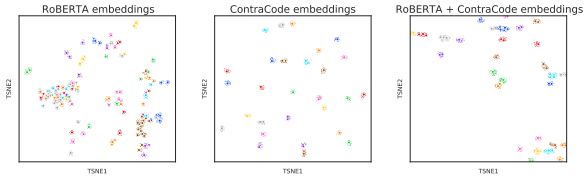


Figure 16: t-SNE (Maaten and Hinton, 2008) plot of mean pooled program representations learned with masked language modeling (RoBERTa), contrastive learning (ContraCode), and a hybrid loss (RoBERTa + ContraCode). Transformed variants of the same program share the same color. Note that colors may be similar across different programs.

of the batch to the queue ($1\times$), while in the second we add 96 items ($12\times$). In addition, we use a larger queue (65k versus 125k keys) and a slightly larger batch size (64 versus 96). We observe that for the baseline queue fill rate, the accuracy decreases for the first 8125 iterations as the queue fills. This decrease in accuracy is expected as the task becomes more difficult due to the increasing number of negatives during queue warmup. However, it is surprising that accuracy grows so slowly once the queue is filled. We suspect this is because the key encoder changes significantly over thousands of iterations: with a momentum term $m = 0.999$, the original key encoder parameters are decayed by a factor of 2.9×10^{-4} by the moving average. If the queue is rapidly refreshed, queue embeddings are predicted by recent key encoders, not old parameters. This also indicates that a large diversity of negative, non-equivalent programs are helpful for rapid convergence of ContraCode pre-training.

t-SNE visualization of representations We qualitatively inspect the structure of the learned representation space by visualizing self-supervised representations of variants of 28 programs using t-SNE (Maaten and Hinton, 2008) in Figure 16. Representations of transformed variants of the same program are plotted with the same color. ContraCode (BiLSTM) clusters variants closely together. Indeed, contrastive learning learns representations that are invariant to a wide class of automated compiler-based transformations. In comparison, the representations learned by masked language modeling (RoBERTa) show more overlap between different programs, and variants do not cleanly cluster. With a hybrid loss combining masked language modeling and contrastive learning, representations of variants of the same program once again cluster.