Technical University Dresden

Faculty of Computer Science

Institute of Computer Engineering

Chair for Compiler Construction

Learning a Graph Neural Network based model for Probabilistic Alias Analysis

Bachelor Thesis

Simon Meusel

October 27, 2022

Advisor: Julian Robledo Mejia
First examiner: Prof. Dr.-Ing. Jeronimo Castrillon
Second examiner: Prof. Dr. Bjoern Andres

Acknowledgements

I would like to express my deepest appreciation to my primary supervisor, Alexander Brauckmann, who guided me through this project. I would like to extend my sincere thanks to my advisor Julian Robledo Mejia and my examiners Prof. Dr.-Ing. Jeronimo Castrillon and Prof. Dr. Bjoern Andres. I am also thankful for the computational resources provided by the Chair of Compiler Construction and the Center for Information Services and High Performance Computing (ZIH).

Declaration of independent work

I hereby declare that this thesis titled

Learning a Graph Neural Network based model for Probabilistic Alias Analysis

is a work of my own, and that only cited sources have been used. I am submitting this thesis for the first time as a piece of assessed academic work.

S. Meusel

Dresden, October 27, 2022

Abstract

Algorithms for conventional alias analysis compute whether two pointers must always, may sometimes, or must never point to the same memory location. Probabilistic alias analysis tries to enrich these results with information on how likely it is for two pointers to alias, which allows for more aggressive optimizations. In this thesis, we evaluate to what extend graph neural networks can learn to answer probabilistic alias queries. We make the following contributions: First, we present a methodology to dynamically collect probabilistic alias information from a dataset of runnable C programs. We apply our methodology to the Jotai input dataset. Second, we train multiple graph neural network based models and evaluate their performance on the created dataset. Our results show, that graph neural networks can, in general, learn to predict labels of probabilistic alias analysis queries. Furthermore, the trained model based on gated graph neural networks outperforms both random and the multi layer perceptron baseline on the task of predicting conventional and probabilistic aliasing queries obtained from the Jotai dataset.

Contents

Chapter 1

Introduction

1.1 Motivation

Modern software development rely on *compilers* to transform source code from high-level languages to machine code. *Compiler optimizations* try to improve the properties of the generated machine code, such as execution time, memory footprint, or code size. In order to decide when and where to apply optimizations, compilers heavily rely on program analysis. Being required by many common optimizations, there has been a lot of ongoing research on *alias analysis* during the last 30 years. While algorithms for conventional alias analysis (CAA) try to determine whether two pointer variables must not, may or must alias, probabilistic alias analysis (PAA) tries make statements about the statistical probability of two variables aliasing. This information can be used by speculative optimizations where CAA cannot provide a definitive must or no alias result.

Deep learning has been successfully applied to many tasks in the compiler domain, such as compiler option tuning[1], parallelism mapping[2], speedup prediction, and codesize reduction. graph neural networks (GNNs) have been shown to automatically extract relevant features from graph representations of program source code, such as abstract syntax trees (ASTs) or control- and data-flow graphs (CDFGs)[2]. It is the goal of this thesis, to study to what extend GNNs are capable of learning PAA.

1.2 Document structure

This thesis is structured as follows: In chapter 2 we give a short introduction into the field of alias analysis, and provide additional motivation for our research on probabilistic alias analysis. Furthermore, we cover the required technical knowledge about the LLVM[1] compiler toolchain, which we used for the dataset creation. Chapter 3 contains the necessary background about different machine learning models, such as GNNs.

In chapter 4 we outline our methodology for creating a ground-truth dataset for PAA. We start be explaining the quality criteria we imposed on our creation process, continue by describing our design decisions, and provide an overview over our implementation at the end of the chapter.

The result of our dataset creation process is a set of programs, associated PAA queries and their ground-truth results. At that point of the thesis, it remains unclear how to represent the programs and queries in a way that can be processed by GNN-based model. We discuss possible approaches to this problem in chapter 5 and define the architecture of our machine learning model.

In chapter 6 we evaluate the performance of this machine learning model: First, we apply our dataset creation methodology to the Jotai[3] dataset and evaluate its quality. Second, we compare two variations of the GNN model to different baselines and interpret the evaluation outcomes.

Chapter 7 summarizes the results of this thesis and describes possible improvements and open topics.

1.3 Contributions

This thesis makes the following contributions:

- We design a methodology to dynamically collect probabilistic alias information from an input dataset of compilable and runnable C programs.
- We apply and evaluate this methodology on the Jotai input dataset.
- We develop GNN-based machine learning models which are capable of processing alias queries. We evaluate these models in comparison to multiple baselines.

[1]LLVM is the name of the compiler toolchain we used, and is no longer an acronym

Chapter 2

Alias analysis

2.1 Compiler stages: Front end, middle end and back end

A lot of programs are written in languages that are not *directly* executable on computer hardware. It is the task of a compiler to translate source code from an input programming language into executable machine code, that can be run on target hardware. For example, the LLVM compiler framework can produce binary files that can be executed on the $x64$ CPU architecture from source code written in the e.g. C++ programming language. During this process called *compilation*, the compiler should produce *correct* output, in the sense that the generated machine code behaves exactly as defined in the input source code[1].

For almost every input program, there are (infinitely) many correct output programs, out of which the compiler has to generate exactly one. Finding a good machine code for a given target architecture is commonly done by repeatedly applying optimizations to the input program. Because real-world programming langauges and the optimizations that are applied to them can be very complex, compilers are generally divided into multiple stages. We will give an overview of the three-stage compiler model and focus on the parts that are required as background for the later chapters. For a more complete introduction into compilers, we recommend [4].

Compiler front end

In the first stage, the compiler front end converts the input program into a stream of tokens (lexing). Next, this stream is parsed into an AST, which is one

[1]The intended behavior of a programming language is commonly defined in its language specification

derivation tree of the grammar of the input language. The AST is then type checked and converted to an intermediate representation (IR) of the program.

The IR can be seen as an intermediary programming language, that acts as an in-between layer of the input and target language. It is designed to be at an abstraction layer that is easy to analyze and optimize. For example, the IR used by LLVM compiler toolchain, is in single static assignment (SSA) form, where every variable is assigned exactly once. Furthermore, the individual instructions of the LLVM IR are divided into basic blocks. A basic block is a sequence of instructions that can only be jumped to at the first instruction (entry node), and only contain branches or jumps as the last instruction (exit node).

Compiler middle end

In the second stage, the compiler iteratively improves the IR of the program. It does so by repeatedly applying optimizations, that transform parts of the IR. Different analysis passes, such as alias analysis, are used to determine where to optimize, and whether the optimization can be applied without changing the semantics of the input program.

Compiler back end

It is the task of the last stage to convert the IR produced by the middle end into machine code that can be executed a the target architecture. In this process, it has to assign registers, perform instruction scheduling and target-specific optimizations according to the specification of the target architecture, emitting an runnable output program.

Program analysis that is done at compile time, also called static analysis, can, in general, not use run time time information, such as the future program input. Run time information is not available at compile time, because the compiled program should function on any set of inputs.

2.2 Conventional alias analysis

CAA is characterized by the results of aliasing queries having to be correct. In distinction to PAA, other terms used for CAA are traditional, safe or definitive alias analysis.

For an algorithm for CAA to determine that two pointers $NoAlias$, there must not be a single case where the pointers alias at run time, independent of the program inputs. If an algorithm for CAA can not infer a $NoAlias$ or $MustAlias$ relationship for a pair of pointer variables, it has to conservatively answer with

MayAlias. Because of these semantics, optimizations using CAA information can rely on the alias results being correct.[2]

Research on both CAA and PAA is located in the broader field of pointer analysis. With alias analysis being concerned about whether two pointer variables point to the same location, other related analysis also exist. For example, points-to analysis tries to statically infer points-to relationships, describing the variables and memory locations a given variable of pointer type possibly or definitely points to. Some algorithms for CAA can also directly determine points-to relationships.

Theoretical results show, that precise CAA is, in general, undecidable [5]. Algorithms for CAA used in compilers can therefore only compute a subset of the aliasing relationships. Thus, different dimensions for comparing CAA algorithms exist:

Example: Flow sensitivity

Flow-sensitive alias analysis takes the position in the program control flow into account when determining aliasing relationship. Oppositely, flow-insensitive algorithms must always respond with the same aliasing result, independent of the position in the program flow. Therefore, they have to answer with a *MayAlias* result, even if two variables alias at just one possible position in the programs control flow.

Even precise flow-insensitive CAA is known to be NP-hard[6]. Thus, algorithms such as Steensgards[7] and Andersons[8], which are for example used in LLVM, are additionally context-insensitive.

Other categorizations of CAA

Next to flow sensitivity, CAA can also be categorized with respect to field and array sensitivity (whether different fields of structs or array elements can be distinguished), context sensitivity and being demand-driven or exhaustive. For additional background on CAA, see [9].

[2]This means, if an CAA still returns an incorrect results, this is considered a bug in the alias analysis algorithm. If optimizations use the incorrect aliasing information, this can lead to program transformations that do not preserve the semantics of the program (also called compiler bugs).

Focusing CAA sensitivity using heuristics

To improve the scalability of CAA algorithms to large programs, several exciting algorithms[10], [11], [12], [13] and machine learning approaches[14] to focus e.g. the context or field sensitivity on certain parts of the program have been developed.

This allows heuristics to improve the precision of existing algorithms for CAA. Because the results are still always definitive, optimizations can use the existing interfaces for CAA. In contrast to PAA, these approaches do not provide statistical information about the probability of two pointers aliasing.

2.3 Alias analysis in LLVM

LLVM[15] implements multiple algorithms for alias analysis, as well as multiple optimization passes that use aliasing information. LLVM provides the `llvm::AliasAnalysis` class as common interface between the two[3]:

```
AliasResult AAResults::alias(const MemoryLocation &LocA,
                             const MemoryLocation &LocB)
```

An `AliasResult` encodes information about the result of the analysis. Most importantly, it contains whether the two memory locations *NoAlias*, *MayAlias*, *MustAlias* or *PartialAlias*. Each memory location refers to an instruction with a pointer type, and additionally includes information about the size of the pointer (e.g. how many addressable units the the alias analysis should assume the pointed-to values are long)[4]. If the the pointer variables defined by the two memory locations do not point to the exact same address in memory, but still to overlapping objects, as defined by the size of the memory locations, *PartialAlias* is returned.

The public interface of the `llvm::AliasAnalysis` restricts what types of analysis algorithms can be implemented in LLVM. Because the interface does not support expressing information about the control flow, the alias analysis interface is flow-insensitive[5].

LLVM implements multiple alias analysis passes, the providing aliasing information e.g. based on the langauge specifics, algorithms such as Steensgards[7] or even hard-coded information about address spaces for given targets. The

[3]We used LLVM Version 14, the most recent version of LLVM when we started writing this thesis. The alias analysis interface might be different in other versions.

[4]The size information is required especially because LLVM is moving towards opaque pointer types, where the size information is not part of the pointer type itself.

[5]See `https://releases.llvm.org/14.0.0/docs/AliasAnalysis.html` for additional information.

alias analysis passes implemented in LLVM 14 include: `BasicAA`, `CFLSteensAA`, `CFLAndersAA`, `GlobalsAA`, `TypeBasedAA`, `AMDGPUAA`.

To allow multiple different analysis passes to coexists, alias analysis chaining is used. It works as following: Run the first alias analysis pass. If the results is one of *NoAlias*, *MustAlias* or *PartialAlias*, return this alias results. Otherwise, where the first alias analysis pass could only determine a *MayAlias* results, the next CAA pass of the alias chain is consulted. If no alias analysis pass could determine a non-*MayAlias* result, *MayAlias* is returned.

2.3.1 Applications of alias analysis

Alias analysis has many use cases in compilers and program analysis. Examples include common subexpression elimination[16], loop-invariant code motion[16], global value numbering[17], dead store elimination[17] and constant propagation[16]. Additionally, it can improve the results of other analysis algorithms, such live variables and available expressions analysis (as noted in [6]). Furthermore it has applications in program analysis outside compilers, such as in analysing coredumps and reverse execution[18].

2.4 Probabilistic alias analysis

PAA (also called speculative or unsafe alias analysis) enriches CAA by providing a probability of the two pointers aliasing at compile time. The need for PAA arose, because theoretical results ([5], [6]) and the scalability issues of CAA can result in many *MayAlias* responses on real world programs (see e.g. our evaluation results on the Jotai dataset in chapter 6).

There exist a wide range of optimizations that can utilize PAA information: For example, [19] have used PAA information for speculative multithreading. [17] use PAA information to perform speculatively perform code motion for target architecture which support transactions. [18] applied PAA with deep learning using recurrent neural networks to root cause diagnosis of reverse execution.

Chapter 3

Applications of machine learning in compilers

In this chapter, we cover the background for the model design in section 5 and its evaluation in chapter 6. We describe different approaches to extract source code features for machine learning on programs.

3.1 Background on machine learning

Supervised learning is a subfield of machine learning, where we try to learn a function f from labeled training data using a computer algorithm.

Which functions are considered during training is defined by the hypothesis space. For example, we can use all linear functions [1] as the hypothesis space:

$$y = \theta_1 \cdot x + \theta_2$$

Here, "training" means finding optimal or good values for the parameters $\theta = (\theta_1, \theta_2)$, such that the error defined by a loss function is minimized and the model has low complexity as defined by a regularizer.

We will now introduce the perceptron, a basic building block of many neural network architectures.

3.1.1 Perceptron

The perceptron takes, as input, multiple numbers $x_i \in \mathbb{R}$, weights every number by a pre-specified parameter $w_i \in \mathbb{R}$, sums up the weighted numbers, and then adds a pre-specified bias $b \in \mathbb{R}$. Afterwards, a non-linear activation function

[1]We use the term linear function for polynomial functions of degree zero or one

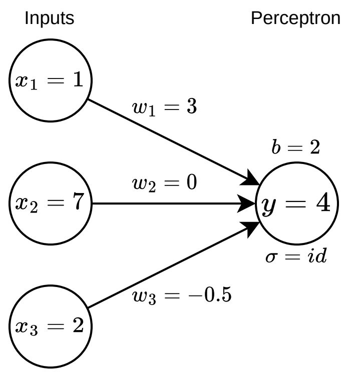


Figure 3.1: Perceptron with 3 inputs (1, 7, 2) and pre-defined parameters, evaluating to 4

$\sigma : \mathbb{R} \to \mathbb{R}$ gets applied to this sum. Possible activation functions are e.g. ReLU or the sigmoid function.

$$y = \sigma\left(\sum_i (w_i \cdot x_i) + b\right)$$

Here, w_i and b are *trainable* parameters θ of the perceptron. The weights impact how relevant each x_i is to the output y of the perceptron. Choosing the right parameters is often hard, and gets done during the training process using gradient descend.

ReLU activation function The rectified linear unit is defined as the following:

$$\sigma(x) = max(x, 0)$$

3.1.2 Multi-layer perceptron

A very basic type of neural network architecture is the multi-layer perceptron (MLP). It is composed of a single input layer h^0, possibly multiple hidden layers $h^1 \ldots h^{l-1}$, and a single output layer h^l.

Each layer consists of one of multiple nodes, which are connected to the nodes of the previous and next layer.

The input layer has one node for every input of the MLP. The nodes of the input layer have a value which is equal to the corresponding input: $h_i^1 = x_i$.

Each hidden/output layer consists of multiple perceptrons. The count is determined each the layers size. The ith perceptron at layer $h^k, k \geq 1$ processes all outputs of the previous layer h_j^{k-1} and produces a single output h_i^{k-1}. Therefore, each layer has as many outputs as its size.

$$h_i^k = \sigma \left(\sum_j w_{(k,i,j)} \cdot h_j^{k-1} + b_{(k,i)} \right)$$

3.1.3 Neural network architectures for processing sequences

Because MLPs have a fixed number of input nodes, they can only process a fixed number of inputs. Adjusting the number of inputs would change the number of connections the from the input layer to the next layer and would also change the number of weights, which would require re-training the network.

In order to process sequences, other neural network architectures exists. For example, the architecture of recurrent neural network (RNN) such as [20] can be unfolded for n time steps to process a sequence of length n, with the weights of the network being shared for every time step. This allows training and evaluation on sequences of variable length.

More sophisticated architectures, such as Long-Short-Term-Memory-based RNNs [21] and gated recurrent units (GRUs)[22] add additional gates which have, for some tasks, shown better results, for example when processing very long sequences.

3.1.4 Neural network architectures for processing graphs

"Graph neural network" is an umbrella term for neural network architectures that process graph-structured data. While for a MLP the number of inputs is a fixed parameter of the architecture, a GNN model can be trained and evaluated on graphs with different amount of nodes and edges.

Although GNNs emerged as a generalization of convolutional neural networks[2] (see[23]), there are multiple perspectives under which GNNs can be viewed. Many GNNs function as *message passing neural networks*, where each node exchanges messages with adjacent nodes and combines the received messages to generate its new representation. The way the messages of the neighboring nodes are aggregated differs between the specific GNN architectures. For example, the GraphSAGE[24] architecture combines the the node embeddings at each layer in the according to[3]:

$$h_v^k = \sigma \left(W^k \cdot CONCAT(h_v^{k-1}, AGGR(\{h_u^{k-1} | u \in \mathcal{N}(v)\})) \right)$$

[2]Convolutional neural networks are used in image processing. While images can be modeled as a grid graph of pixels that can be processed by convolution layers, arbitrary graph structures need a generalization of the convolution operator, as in [23].

[3]Adopted from algorithm 1 of the GraphSAGE paper[24]

Here, the weight matrix W^k is shared between all nodes v of the same layer, which allows training and evaluation of the same network on an arbitrary amount of nodes. h_v^k denotes the node embedding (or hidden feature vector) of node v at GNN layer k. $\mathcal{N}(v)$ is the set of neighbors of node v. Commonly used aggregation function $AGGR$ are element-wise $MEAN$ or SUM.

There exist many adoptions of GNNs that e.g. support edge features, training on a bigger number of layers or use skip connections between multiple layers. During evaluation, we used gated graph neural network (GGNN)[25] and graph attention network (GAT)[26] as possible propagation layers.

The most significant difference of GGNNs to e.g. GraphSAGE is it's use of GRUs in the GNN layers. The GRU is unrolled for as many time steps as there are GNN layers. It combines, at each layer, the node embedding of neighbors with the own node embedding. In contrast to GraphSage and GAT, the weights of the GRU are shared among all GGNN layers. The use of a GRU in the GNN layer allows information of the node embeddings of all previous layers to be used in the following layers. For details on the exact propagation model, see section 3.2 of the GGNNs[25] paper.

In addition to GGNNs, we used GATs, which have achieved similar or better performance than [23] and GraphSAGE on some datasets[26]. The node embeddings at each layer are computed as follows[4]:

$$h_v^k = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{v,u}^k \cdot W^k h_u^{k-1}\right)$$

where h_v^k, W^k and $\mathcal{N}$ are defined as above, and α is the attention of neighbor u with respect to node v and a shared single layer MLP a^k:

$$\alpha_{v,u}^k = \frac{exp(a^k(W_a^k h_v^k, W_a^k h_u^k))}{\sum_{w \in \mathcal{N}(v)} exp(a^k(W_a^k h_v^k, W_a^k h_w^k))}$$

For additional explanations and experiments on the performance of GATs, see the GAT paper[26].

3.2 Program representations

In order to process a program using neural networks, we need to extract features that can be processed by neural networks, from e.g. the program source code.

[4]Adopted from GAT paper[26]

When trying to use machine learning to make predictions at compile time, features need to be extracted statically, because run time information, such as the program inputs, are unknown at compile time.

The source code of a program can be represented in multiple ways:

3.2.1 Fixed-size features

Some features about the program a of fixed size for every possible input program. For example, the counts for every instruction type in the IR representation of a program will always result in i numbers, where i is the number of instructions that the IR language supports. Also information, such as the target architecture can be used as a fixed-size feature. Because of this fixed size, these features can be processed by many machine learning models, e.g. by MLPs. However, selecting the best features for a given task is often not easy. For this reason, RNNs GNNs and can be used on a more complete representation of the input program to automatically learn which features are relevant.

3.2.2 Sequence of lexer tokens

As covered in section 2.1, the lexer produces a list of tokens during compilation. Every token can be represented e.g. by the type of token. The sequence of token representation can then be processed by sequence-based models, such as RNNs and Long-Short-Term-Memorys (LSTMs).

3.2.3 Abstract syntax tree and control- and data-flow graph

During the compilation process, many graph representations of the program can be found. For example, the parser produces an abstract syntax tree (AST). Furthermore, a CDFG can be extracted from the IR of a program. For background information and a comparison of different graph representations, see [2].

Model architecture	Input data	Example program features
Multi-layer perceptron	Fixed amount of features	Instruction counts
Recurrent neural network	Sequence data	Tokens from lexer
Graph neural network	Graph-structured data	AST, CDFG

Table 3.1: Comparison of deep learning architectures in machine learning for code.

Chapter 4

Methodology for creating a probabilistic alias analysis dataset

In order to train a machine learning model for PAA, a ground truth dataset is required. In this chapter, we cover the methodology of the dataset creation. The dataset lays the foundation of our GNN model in chapter 5. We evaluate the quality of our dataset in chapter 6.

We approached the creation of a GNN model for PAA using supervised learning. First, we created a dataset which contains example programs together with aliasing queries and their correct results. Second, we used this data to train machine learning models, providing them with example programs and instructing them to produce the ground truth query results from the dataset. Because of the supervised learning approach, the quality of the dataset puts an upper limit on the performance of the machine learning model.

We choose to create our PAA dataset from an existing dataset of example programs, which we call the *input dataset* in this chapter. We compare different input datasets, all based on real-world programs, in the beginning of chapter 6. Based on the input dataset, we want to extract aliasing data to create the dataset for PAA.

We start this chapter by defining the goals of our dataset creation methodology, and then cover our design decisions and give an overview of our implementation.

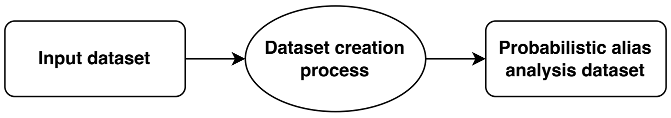


Figure 4.1: Creating a PAA dataset based on an input dataset

4.1 Scope of our approach

Because it would in practice not be possible to implement our dataset creation methodology for every programming language and compiler toolchain, we focused our research on the C programming language and the LLVM compiler. We choose LLVM because it is a modern compiler which can be extended with custom analysis and transformation passes. It is commonly used in machine learning for compilers research (e.g. [27], [28], [1]).

That being said, because of the universality of our approach, we believe that in theory it can be easily adopted for other compilers (such as GCC[29]) or other programming languages with a similar memory model as C (such as C++).

4.2 Requirements

We grouped requirements of the dataset creation process into three categories:

- The first category defines which properties the PAA queries should have, with respect to their use as analysis data for downstream compiler optimizations (see 2.4).
- The second category hosts requirements for training machine learning models using the PAA dataset.
- Our dataset creation process should ideally be universally compatible with all input programs. We formalize this goal into the requirements in category three.

4.2.1 Probabilistic aliasing queries

FQ1: Queries shall operate on compilers intermediate representation

Probabilistic alias analysis can be done at source code level (subjects of the analysis could e.g. be variables in a C program), at the assembler level, or at the level of the compiler-generated IR. As optimizations profiting from PAA (covered in section 2.4) operate in the compiler middle end or back end, they transform the program on the level of the program IR. Therefore, a PAA giving responses on the layer of source language variables or at the level of assembly instructions would be of little use for the compiler, as the relationship between those variables and the compilers IR would not always be clear. For example, new instructions introduced in prior transformation passes do not have a one to one counterpart in the source language.

ID	Requirement description
FQ1	Queries should operate on compilers intermediate representation
FQ2	Query results shall encode statistic probability of aliasing
FD1	Compiler source IR shall be included
FD2	Conventional aliasing results shall be recorded
FD3	The dataset shall be split into train, test and validation sets
FD4	Similar programs shall be in the same dataset split set
FD5	Dataset shall contain correct data
FC1	All possible C files of the input dataset shall be supported
FC2	Non-relevant parts of the input dataset shall be ignored
NF1	Dataset creation shall be adjustable to different input datasets

Table 4.1: Requirements of the dataset generation methodology.

The first requirement of our dataset creation methodology is derived from the above observation. We impose, that the subjects of the PAA queries shall be objects on the level of the compilers IR.

In our case, as we focused our efforts on the LLVM compiler toolchain, this means the subjects shall be memory locations as described in section 2.3. Downstream LLVM optimizations passes could directly use the probabilistic aliasing information by asking queries on the representation that is also used for transformations.

FQ2: Query results shall encode statistic probability of aliasing

The optimizations described in section 2.4 require probabilistic alias responses that indicate the probability of aliasing at runtime to decide when it is worth to optimize. Therefore, the PAA query results shall encode statistic probability of aliasing.

4.2.2 Dataset requirements

FD1: Compiler source IR shall be included

The IR generated from the input dataset shall be included in the generated dataset, because it can be used to extract features about the program (see section 3.2.3).

FD2: Conventional aliasing results shall be recorded

Including the results of CAA queries that got answered during compilation is required to evaluate the differences between PAA and CAA with respect to the input dataset.

FD3: The dataset shall be split into train, test and validation sets

A machine learning model training on the PAA dataset should be evaluated on programs different from those used during training. In particular, our PAA dataset shall include three disjunct sets of programs:

- the *training set* for training a machine learning model using a given set of hyperparameters,
- the *validation set* to compare different hyperparameters and
- the *test set* which can be used to evaluate the performance of a model on programs it has not seen before.

FD4: Similar programs shall be in the same dataset split set

Because the input datasets we considered are derived from real world programs, often using code crawled from GitHub (e.g. [30], [31]), they may contain the same program multiple times. This can occur if source files of a library get copied to other software projects. One example for the Jotai[3] dataset is the file `xutils.c` which is dataset twice, because it was contained in both the `libgit2` repository and `git` repository on GitHub.

Duplicate programs in the dataset can lead to issues with the train/test/validation splits. If the dataset splits get assigned randomly a program could occur in both the training and the validation sets. Thus, a machine learning model could memorize the ground truth labels during training, and achieve good results during evaluation, even tough the model has not generalized at all. In the most extreme case, where every program occurs once in the training set, and once in the validation set, the model could achieve a perfect score just by memorization.

Therefore, not only duplicate, but also similar, programs, shall be as assigned to the same dataset split set (either train, test, or validation).

FD5: Dataset shall contain correct data

The dataset shall not contain any incorrect aliasing query results, because a model trained on incorrect aliasing information would be of little use for downstream optimizations.

4.2.3 Compatibility with input datasets

FC1: All possible C files of the input dataset shall be supported

Because models trained on the PAA dataset should perform well on a many programs as possible, the dataset should include a wide range of programs for training. Therefore, the implementation of the dataset generation shall support as many features of the C language as possible, and thus as many programs of the input dataset as possible.

FC2: Non-relevant parts of the input dataset shall be ignored

Some input datasets, such as Jotai, include some C functions, e.g. the `main` function, that are very similar for every program. It shall be possible to exclude such functions from the alias extraction, because their aliasing data would be very similar, which could lead to a very biased PAA dataset.

4.2.4 Non-functional requirements

NF1: Dataset creation shall be adjustable to different input datasets

The programming landscape has been constantly evolving over the last decades, with new ways to write code, new libraries and different design patterns being established.

Furthermore, over the last years, there has been a lot of innovation around datasets for machine learning on code ([31], [3]). As the quality of the PAA dataset highly depends on the quality of the input dataset, being able to process different input datasets is key to adapt to new developments. Being universally compatible with all datasets is not possible, because currently there is no standard format being used by the datasets. Still, the dataset creation methodology shall be easily adjustable to different and new input datasets.

This could also be key to adapt the probabilistic aliasing data to a particular field of an application, and fine-tune the machine learning model by using e.g. transfer learning.

4.3 Semantics of probabilistic aliasing results

Different options to define the syntax and semantics of probabilistic aliasing results exist:

1. $result \in \{NoAlias, MayAlias, MustAlias\}$

   One possibility would be to mirror the interface for conventional aliasing queries, but instead of giving definite answers, the model could return the query result which is most likely. Contrary to CAA, the model would not need to prove that it's result is correct, which would allow for more aggressive no-alias and must-alias responses, which could possibility allow for more aggressive downstream optimizations.

2. $result \in \{\{NoAlias, MayAlias, MustAlias\} \rightarrow [0, 1]\}$

   A problem with the first approach is that the alias result would give no information about how likely it is that two variables alias. Therefore, we could update the semantics as follows: When the model outputs a likelihood of 0.9 for *NoAlias* ten times, the actual result for the alias queries would only be *NoAlias* nine of the ten times. Downstream optimizations could benefit from this information, in order to decide whether it is useful to do a particular performance optimization.

3. $result \in [0, 1]$ When assessing whether an optimization should be done in a particular case, the relevant factor often isn't how likely a *NoAlias* or *MustAlias* response is, but instead, what the probability of the two variables actually aliasing at run time is. It could still be beneficial to perform optimizations even if the model determines that there is a single case where the subjects alias, but the run time probability of that case is very low. Thus, better semantics for the query results would be to encode the statistic probability of the two variables actually aliasing at run time. 0 as the result would mean the model determined a statistic probability of 0% for the variables aliasing at run time, and 1 a statistic probability of 100%.

For the reasons outlined above and in to we choose the third option as the semantics for out PAA interface.

4.4 Approaches to gather probabilistic aliasing data

There exists multiple options to gather the ground truth aliasing data from the input dataset. In this section we discuss to up- and downsides of the approaches we considered, and explain the reasons why we chose to gather the aliasing data using instrumentation.

We considered three approaches to gather ground truth aliasing data from the input dataset: Static analysis, ORAQL and program instrumentation.

4.4.1 Statically using conventional alias analysis

The first approach is to statically analyze a program using existing CAA implementations, and to use this data for training a machine learning model. It could then try to classify new aliasing queries.

The upsides of this approach are:

- Existing implementations of CAA can be used.
- Machine learning models could possibly produce similar results as existing CAA implementations of high computational complexity faster.
- The input dataset would only require compilable, not runnable, programs. Furthermore, because the aliasing results are determined statically, they would universally remain true completely independent of the input.

The downsides of this approach are:

- The last pro argument is also a downside: Because no inputs are required, the aliasing results can never output a low probability of aliasing even when there is only a single, however unlikely, case of aliasing.
- The data collected through CAA would not contain information about the statistical likelihood of aliasing at run time.
- Because the data does not encode statistical likelihoods of aliasing at run time, the only theoretical benefit of using a machine learning model here would be speed. Else, one could theoretically just run the original CAA implementation to obtain the results.

4.4.2 Collecting aliasing data at run time

Another approach to gather probabilistic aliasing data is to measure aliasing taking place at run time. This can be done by using debugging tools, changing the program to output its aliasing behavior at run time, or using a modified interpreter for running the program.

The upsides of this approach are:

- The measured aliasing behavior does exactly represent the percentage of aliasing during run time. This takes the common program inputs of the input dataset into account.
- Changes the programs and input of the input dataset allows adjusting to a particular field of application.

The downsides of this approach are:

- Runnable input datasets are required and the quality of the gathered aliasing data directly depends on the inputs.
- Depending on the way aliasing measurements are taken, only certain input programs may be supported, aliasing measurements may be incorrect, or performance of the program could be degraded.

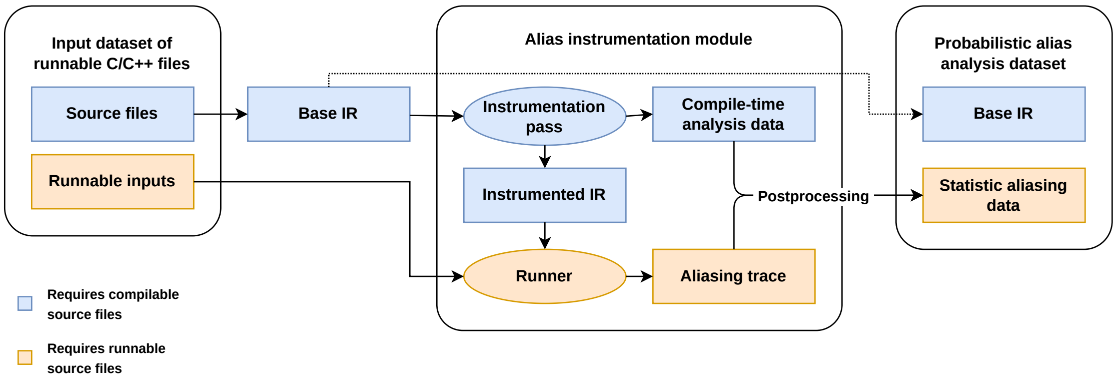
Additionally, the aliasing queries could be collected using the ORAQL[32] tool. Instead of using *MayAlias* as fallback responses in the LLVM alias analysis chain, ORAQL responds with *NoAlias* to queries that no other alias analysis can find an answer to. This can result in an incorrect compilation, because the assumed *NoAlias* response might be wrong. Therefore, ORAQL iteratively refines the alias responses in multiple compilations, until the compiled program produces its correct output for one particular input. Using this approach, ORAQL can produce almost optimal *NoAlias* responses for a single input, while still ensuring the correct program outputs. However, like the CAA results, the obtained aliasing data contains no information about the probability of aliasing happening at runtime. Furthermore, the optimistically answered queries might lead to undefined behavior, which is undetected as long as the program produces the correct outputs.

4.5 Our approach: Collect data using instrumentation

To be able to gather aliasing data that accurately represents run time behavior and encodes the statistical likelihood of aliasing, we choose to measure it at run time.

This section describes how we designed and implemented our *Alias Instrumentation Module*. It comprises the following steps, that get run on every program of the input dataset:

1. Convert the input program to LLVM IR.



Input dataset of runnable C/C++ files
Source files
Runnable inputs
Base IR
Alias instrumentation module
Instrumentation pass
Compile-time analysis data
Instrumented IR
Postprocessing
Runner
Aliasing trace
Probabilistic alias analysis dataset
Base IR
Statistic aliasing data
Requires compilable source files
Requires runnable source files

Figure 4.2: Overview of the alias instrumentation module

2. At compile time, ask conventional aliasing queries, to compare them with the results of PAA (see requirement FD2).

3. Instrumentation: Run compile time transformation pass to instrument relevant instructions.

4. Runner: Execute the instrumented program to gather run time data.

5. Postprocessing: Analyze run time data to compute statistical aliasing percentages.

6. Detect similar programs (see requirement FD4)

7. Create dataset splits (see requirement FD3)

4.5.1 Compilation of the input program to LLVM IR

A prerequisite for the following steps, and for requirement FD1, is the conversion of the program to LLVM IR. Both, the extraction of CAA data and the measurement of PAA data, require the program IR.

During the typical compilation process, a C program passes through many IRs, because the optimization passes can change parts or even the general structure of the program.

In order to compare the results of CAA and PAA, it is important that both extractors work on exactly the same IR of the program. Alias queries can not directly be compared after applying different optimizations, because these can lead to different programs, with different instructions and possibly different aliasing behavior. For the same reasons it is important to use the same IRs as input representation of the program for the machine learning model, too.

Therefore, we compile the program into IR using LLVM, and then use this IR without applying any additional optimizations for CAA and PAA, and also as the input representation for machine learning. We call this the *base* IR of the program.

For simplicity, we only process every program once, and use its IR without any LLVM optimizations applied. It would also be possible to run the dataset extraction process on every program multiple times, with different optimization passes used to generate the IR. This could be used as a data augmentation method, would increase the size of the dataset, and could allow the trained machine learning model to perform better on IR in different stages of the compilation process. To limit the scope of the dataset generation methodology, we did not implement this yet.

4.5.2 Collection of conventional alias analysis data

Before running our transformation pass, we collect the results of CAA queries. To achieve this, we modified the `llvm::AAResults::alias` function. We adopted the `-aa-trace` option of the LLVM `opt` command, and updated the logging to include the information we need. Our logging is done after a conventional alias result has been determined by the alias analysis chain (recall section 2.3) and before is returned to the caller of the alias analysis.

In addition to storing the LLVM aliasing result (e.g. `NoAlias`, `MayAlias`, `MustAlias`, `PartialAlias`) we also collect information about the subjects of the query, such as an identification of the instructions, and the sizes of the two LLVM memory locations. To uniquely identify instructions while collecting CAA data and before running the instrumentation pass, we use a tuple $id \in \mathbb{N}^5$ containing the index of the instruction in its basic block, the index of the basic block in its function, the name of the function, and the name of its module. This allows us to compare the CAA and PAA results of a particular aliasing query.

We use LLVMs `AliasAnalysisEvaluator` to generate aliasing queries between every pair of pointer instructions. To collect the aliasing data of a particular program given its base IR, we run the command described in table 4.2. Its output gets passed to the post-execution analysis script through Linux pipes.

4.5.3 Compile time instrumentation pass

In order to instrument C/C++ programs, we have developed a custom LLVM transformation pass. It analyzes the functions of an input program and adds additional LLVM IR instructions to the programs base IR. Output is what we call the *instrumented* IR, which should produce the required aliasing information at run time.

Program base IR:

```
store i32 5, i32* %2, align 4
store i32 10, i32* %3, align 4
store i32* %2, i32** %4, align 8
store i32* %3, i32** %5, align 8
%10 = load i32*, i32** %4, align 8
%11 = call i32 @process(i32* %10)
store i32 %11, i32* %6, align 4
%12 = load i32*, i32** %5, align 8
%13 = call i32 @process(i32* %12)
```

Instrumented IR:

```
store i32 5, i32* %5, align 4
store i32 10, i32* %7, align 4
store i32* %5, i32** %9, align 8
store i32* %7, i32** %11, align 8
%21 = load i32*, i32** %9, align 8
%22 = call i32 (i8*, ...)
        @printf([...], i32 29, i32* %21, i32 %1, [...])
%23 = call i32 @process(i32* %21)
store i32 %23, i32* %13, align 4
%24 = load i32*, i32** %11, align 8
%25 = call i32 (i8*, ...)
        @printf([...], i32 32, i32* %24, i32 %1, [...]))
%26 = call i32 @process(i32* %24)
```

Figure 4.3: Alias instrumentation pass applied to a list of LLVM IR instructions.

Command and its arguments	Description
`opt`	Run our modified version of the LLVMs optimizer
`-O0` `-disable-output`	Do not run any additional optimization passes Do not print LLVM IR again

`-aa`
`-scoped-noalias-aa`
`-cfl-anders-aa`
`-cfl-steens-aa`
`-basic-aa`
	Use a chain of LLVMs existing CAA implementations (see section 2.3)

`-aa-trace` `-aa-eval`	Enable logging of CAA results as described in section 4.5.2 Enable LLVMs alias evaluation to generate aliasing queries
`program_base_ir.ll`	Path to input file containing the programs IR

Table 4.2: The command used to collect CAA data.

We restricted the scope of our implementation to only LLVM variables inside functions, and ignored global variables. The transformation pass first identifies all variables which return a pointer type, because these are the relevant subjects of alias analysis, conventional or probabilistic. To track when the variables alias at run time, we monitor the variables for changes. Because LLVM IR is in SSA form, every variable is initialized by exactly one LLVM IR instruction, and not assigned to afterwards. Therefore, we only need to track the LLVM IR variable after its initialization. To accomplish this, we add IR instructions recording the following information when a pointer variable is assigned to:

- The memory address where the pointer variable is pointing to.

  This information allows to calculate the statistic aliasing percentages during postprocessing.

- A number uniquely identifying the instruction.

  This number can be used to associate the instruction with e.g. the CAA queries collected during compile time.

- A number uniquely identifying the function context.

  Variables in LLVM functions can exist multiple times during execution, in separate stack frames. This can happen when a function is called multiple times in sequence, or during recursion where multiple instances of a variable can exist at the same time. Therefore, we assign a unique, numerical ID to every function execution at run time, and, in our post-execution analysis script, only consider aliasing between variables in the same function context.

To uniquely identify each function context, we add a global variable to the instrumented program, which is implemented as a counter. It keeps track of the numerical ID that should be used for the next function context. We add additional instructions to every function of the instrumented IR, which increment the global counter variable and keep track of the ID of the current context.

4.5.4 Execution of instrumented programs

The instrumentation pass described in the last section is used to generate an executable binary file starting from the base IR:

Command and its arguments	Description
`clang`	Run the LLVM compiler front end for C
`-O0`	Do not run any additional optimization passes
`-Xclang -fpass-plugin="libAliasInstrumentation.so"`	Load custom instrumentation pass
`program_base_ir.ll`	Path to the input file that should be compiled
`-o instrumened.o`	Path to the output file

Table 4.3: The command used to compile the base IR into an instrumented executable.

The binary gets run with the aliasing information being printed to the standard output by default. We use a signature to disambiguate program output from aliasing information.

Depending on the type of library and system calls present in the input dataset, it may be beneficial to run the binaries as a separate user, in a Linux container or in a virtual machine, to prevent damage to the host system and to make the runs more reproducible. Some dataset may also include non-terminating programs, which can be approached by restricting the execution time of the programs.

4.5.5 Post-execution analysis

To calculate the PAA query results, we replay the alias trace produced by the instrumented binary at runtime. We do this for every program input. For every function context, we keep track of the locations where pointer variables are pointing to. When a pointer variable gets assigned to during the replay, the address it is pointing to is compared to all other pointer variables in the current function context. If the memory locations that the pointer variables point to overlap, the count of *DidAlias* counter for the corresponding query is incremented. If the memory locations did not overlap, the *DidNotAlias* counter is incremented. At the end of the replays, the PAA query result is calculated as $\frac{DidAlias}{DidAlias+DidNotAlias}$.

4.5.6 Detection of similar programs using instruction k-grams

To measure program similarity, we extract all instruction k-grams from every programs uninstrumented IR. Here, a k-gram is a sequence of k successive instructions in the linearized IR. After extracting, we can then check whether a program A contains duplicated code from a program B by calculating how many of Bs k-grams are also included in B.

By using k-grams we can ensure that the code is actually similar, as single instructions being present in both programs does not imply similarity. Furthermore, this approach is resilient to reordering of large code blocks, because the matching k-grams inside each code block can still be detected.

Using k-grams is an adoption of techniques used in document fingerprinting and code plagiarism detection[33].

A downside of extracting k-grams from linearized IR is that reordering of short instruction sequences, such as basic blocks, can significantly change the programs k-gram set. The effect of basic block reordering could possibly be reduced by instead of extracting k-grams, extracting instruction neighborhoods, which are invariant to the basic block order.

We define two programs as similar, if at least 70% of the instruction k-grams of one program are included in the other program. We build a graph, where similar programs are connected by an edge, and extract the connected components. All programs in the same connected component have to be assigned to the same dataset split.

When creating the dataset splits, we used 80% of the programs for the train set, 10% for the validation and 10% for the test set. Starting with the connected component with the most programs, we randomly assigned each connected components to a dataset split that had enough space for all the programs in it.

Chapter 5

Graph-Neural-Network based model for alias analysis

This chapter covers the methodology used to create a machine learning model for probabilistic alias analysis.

In the last chapter, we laid the foundation for training GNN-based models for PAA. The output of our dataset creation methodology contains programs, their base IR and a list of probabilistic aliasing queries including their ground-truth results. In addition to the above, the dataset also includes CAA queries and their statically obtained ground-truth results.

In this chapter we describe the design of our GNN-based model. More precisely, we designed an architecture that allows training models for both CAA and PAA queries. In the next chapter, we evaluate these models both on the CAA queries obtained by our dataset creation methodology and also on the dynamically gathered PAA queries.

Thus, our model architecture has to be adjustable to both types of queries. While the CAA query labels are of categorical type (one of `NoAlias`, `MayAlias`, `MustAlias` and `PartialAlias`), PAA query labels are of continuous nature: A number in $[0, 1]$ encoding the statistical probability whether two variables alias at runtime. To unify both types of queries into one framework, we discretize PAA query labels into categories.

At the beginning of this chapter we describe how we represent the input programs and queries, and how we discretize the PAA query results. We continue by presenting our multi-phased model architecture that can operate on the input representation. We show how it can output predictions of CAA or PAA query results.

Towards the end of this chapter, we take a closer look at our training methodology, optimizer and regularization techniques. Furthermore, we present details about the implementation of the model architecture and training pipeline.

5.1 Overview of the training methodology

The machine learning models should be able to predict the correct query label category of PAA or CAA queries given an input program and input query. More precisely, they should be able to answer aliasing queries from the *test* set after being trained on a separate set of aliasing queries from the *train* set. The different dataset splits should be unique, but should be drawn from the same (unknown) probability distribution[1].

Therefore, the training procedure should find good model parameters $\theta \in \Theta$, that:

1. can predict the query labels of the training set well, given the input programs and query information. More formally, this means that negative log likelihood loss should get minimized by the training procedure:

   $$l_q = weight(c) \cdot \hat{y}_{q,c}$$

   where l_q is the loss of query q with true label c, and $\hat{y}_{q,c}$ is the log-probability of category c as obtained from the readout layer of the neural network. We calculate the total loss as the mean of the losses of the individual queries.

2. have low complexity. We discuss the regularization techniques we applied to minimize the model complexity later in this chapter, in section 5.5.

The second property is required to penalize the model for overfitting the training set. Instead, if the model learns to generalize, we can also apply it to the validation and test with good results.

In order to iteratively improve the training parameters θ, we use the Adam optimizer instead of stochastic gradient descend. We describe this in more detail in section 5.5 of this chapter.

In addition to the trainable parameters of the model, we also need to specify multiple *hyperparameters*. They we can not be optimized using stochastic gradient descend, because no gradient of the loss function with respect to the hyperparameters can be calculated. Examples of hyperparameters include the count and sizes of the GNN layers of the model.

We use the validation set to compare models with different hyperparameters trained exclusively on the train set. During *hyperparameter search*, we select the models that performed best on the validation set.

[1]For this reason we implemented the program similarity check in chapter 4, but also randomly assigned the dataset splits, not assigning similar programs to the same split

5.2 Representation of programs and alias queries

With an overview of the training pipeline in mind, we now describe how we represented the input programs and queries in a way that can be processed by machine learning models.

As explained in the last chapter, the PAA dataset includes the base IR of every program, including a set of probabilistic aliasing queries and their results. This dataset is, however, not in a representation that can be understood by a GNN. A key ingredient for training GNNs to learn alias analysis is this representation of programs and the aliasing queries, because it defines what information the machine learning model can use as input.

Thus, we now cover how obtain a representation of the inputs. In the next section we describe in more detail how this representation is used by the multi-phase GNN model to predict the query labels.

5.2.1 Graph representation of the programs IR

We formalize a input program either with respect to CAA or PAA queries, because the model architecture has to be slightly different for CAA and PAA, because the PAA queries need to be discretized.

The representation p of an input program is:

$$p = (G, t, Q)$$

where G is the graph representation of the programs base IR, Q is the set of aliasing queries associated with the program, and $t \in \{CAA, PAA\}$ is the type of the aliasing queries.

We transform the base IR into an multi-graph $G = (V, E)$ with labeled edges.

5.2.2 Nodes

The nodes V of the graph represent the instructions of the IR.

If an instructions returns a value, its node in the graph also directly represents this return value. This means there are no separate nodes for the LLVM variables returned by the instructions. Separate nodes for the LLVM variables would be redundant and would increase the steps required to traverse the graph, which would require more GNN layers.

For each node $v \in V$ we extract features from the IR:

- LLVM instruction opcode (e.g. `add`, `alloca`, `store`)
- LLVM type of the return value of the instruction (e.g. `f32`, `i32*`)

We convert both the opcode and the return type to a textual representation. We filter out values that occur in less that 0.01% of the instructions, because the training data would be too limited to tune the associated parameters. We replace such values with a separate category of infrequent values. For the return type, we do not yet support complex struct types.

We concatenate the one-hot encoding of the opcode and the return type. We call the resulting vector $x_v \in \mathbb{R}^{D_V}$ the node feature vector of node v.

5.2.3 Edges

We extract three types of edges from the IR:

- Control-flow edges: For every successor j of instruction i: An edge (v_i, v_j) with label *control*
- Data edges: For every operand of instruction i using a value of instruction j: An edge (v_i, v_j) with label *data*
- Memory dependency edges: For every memory access of an instruction i depending on instruction j: An edge (v_i, v_j) with label *memory*

5.2.4 Representation of alias queries

We define the set of aliasing queries Q with respect to either CAA or PAA of a program $p = ((V, E), Q)$ to contain queries $q = (v_a, v_b, s_a, s_b, y)$ with $v_a, v_b \in V$, where

- v_a, s_a represent the instruction and size associated with memory location a,
- v_b, s_b represent the instruction and size associated with memory location b and
- y is the label of the query. For CAA queries, this is a categorical value $y \in \{NoAlias, MayAlias, MustAlias, PartialAlias\}$. For PAA queries, we divide the result space $[0, 1]$ into $|C|$ categories of the same size and assign each probabilistic alias query to its category depending on its query result (see figure 5.1).

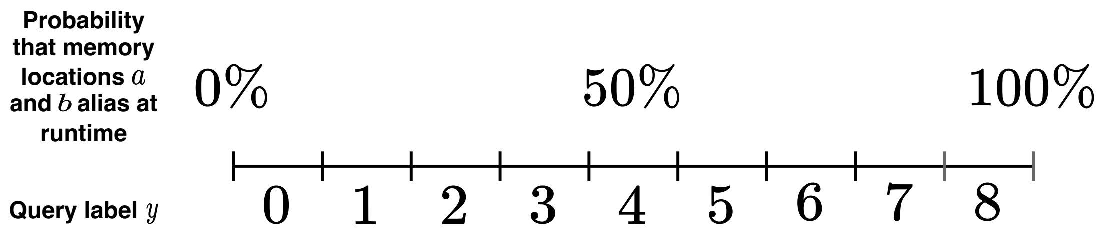

Probability that memory locations a and b alias at runtime
0%
50%
100%
Query label y
0
1
2
3
4
5
6
7
8

Figure 5.1: Conversion of continuos probabilistic aliasing responses to discrete categories for $|C| = 9$.

5.3 Task

5.4 Model architecture

Input:

Directed input graph $G = (V, E)$, where $V = 1, 2, \ldots, |V|$ and $E \subseteq V \times V$. The node feature vector $x_v \in \mathbb{R}^{D_V}$. A probabilistic aliasing query $q \in \mathcal{L} \times \mathcal{L}$ is made up of two memory locations. A memory location $l = (v, x_l)$ with $v \in V \wedge x_l \in R^{D_L}$.

We divided our model architecture into three phases:

5.4.1 Embedding phase

In the embedding phase, each node and memory location feature vector is processed on its own. No propagation along graph edges is done. It is goal of the embedding phase is to transform the initial one-hot encoding of the input feature vectors into a more compact representation.

We used a MLP for embedding the node input features, with the MLP parameters $\Theta_V^{\mathcal{E}}$ being shared among all nodes. This means, that the features of every node get embedded in the same way, and that the embedding can handle any amount of input nodes.

We used a MLP with separate parameters $\Theta^{\mathcal{E}_L}$ to embed the feature vectors of the memory locations. The weights are shared between both memory locations of the aliasing query.

5.4.2 Propagation phase

The propagation phase takes the node features produced by the embedding phase, and applies one or multiple GNN layers to produce the node embeddings. These embeddings can additionally contain aggregated information about the nodes in the (multi-step) neighbourhood of each node and it's graph structure.

The propagation phase leaves information about the aliasing queries, such as the memory location embeddings of the embedding layer, untouched. This information is not used at all during the propagation phase.

The propagation phase can be done through one of following propagation types: GGNN, GAT and MLP. The GGNN and GAT function as defined in section 3.1.4, along the edges E of graph G.

For the MLP propagation type, no GNN layers are used. Instead, the node features produced by the embedding phase get processed by multiple MLP layers. In the evaluation chapter, we use this propagation type as a baseline to compare the GNN propagation layers to.

5.4.3 Readout phase

The readout phase combines node embeddings obtained by propagation phase and the feature embeddings of the memory locations obtained during the embedding phase and produces a query result. This happens in three steps:

1. The two node embeddings of the query instructions, which are the results of the propagation phase, get aggregated. We use element-wise summation as the aggregation method. While also other aggregation methods such as concatenation can be used, using e.g. summation has the benefit that is permutation invariant. This induces the bias, that the order of memory locations does not matter, into the network.

2. The two feature embeddings of the of the memory locations, which are the results of the embedding phase, get aggregated, also using element-wise summation.

3. The aggregated node and memory location feature embeddings get concatenated.

4. A MLP gets applies to the concatenated embeddings. The output layer of the readout MLP is of size $|C|$. Before calculating the negative log likelihood loss, we apply the softmax and logarithm functions to the outputs, to obtain log-probabilities[2].

[2]This is the same as applying cross entropy loss.

5.5 Regularization and optimization algorithm

Instead of using stochastic gradient descend to iteratively improve/update the parameters Θ of the model, we use the Adam algorithm described in [34].

As a regularization technique, we define the weight decay as an additional hyperparameter of the model, which specifies the L2 penalty on the parameters θ.

Furthermore, we use dropout as described in [35] in the MLPs of the embedding and readout phase and in the propagation phase for the MLP layer. If GAT is used as propagation phase of the model, dropout is also applied "to the normalized attention coefficients" [26], in addition to the inputs of the GAT layers. For implementation reasons, we do not use dropout between the propagation layers for the GGNN propagation phase.

Dropout works as a regularization technique by "preventing complex co-adaptations on the training data. On each presentation of each training case, each hidden unit is randomly omitted from the network with a probability of 0.5, so a hidden unit cannot rely on other hidden units being present."[35] Because dropout randomly omits certain units of the network, it reduces the performance of the network. Thus, it is only applied during training, not during evaluation. For the same reason, dropout can lead to higher training loss that validation loss during the training process of a neural network.

5.6 Training methodology

5.6.1 Implementation

There are multiple libraries for training GNNs (see table 5.2).

After experimenting with and implementing efficient GNN layers directly with Tensorflow using sparse matrices, we decided to use a GNN library instead. This has the advantage that different types of GNN layers can easily be tested and compared (e.g. GAT and GGNN), and that their implementation is widely used and well tested.

We decided to use PyTorch Geometric[39], because PyTorch[38] is commonly used in research on GNNs and on machine learning for code. It furthermore has implemented many different GNN layers, including GAT and GGNN layers.

Our implementation can be divided into the following phases:

5.6.2 Data preparation

1. Extracting the representation of the input program and aliasing queries from the PAA dataset. To extract the graph information from the program base

Hyperparameter name	Description
Learning rate	Determines the step size of the optimizer
Batch size	Size of minibatches during training
Weight decay	L2 penalty applied by the Adam optimizer
Embedding phase	
Dropout	Dropout probability
MLP layer counts	Layer count of node and memory location embedding layers
MLP layer sizes	Size of feature vectors produced by every MLP layer
Propagation phase	
Propagation type	Type of GNN layer used for propagation of node features
Dropout	Dropout probability
Layer count	Count of propagation layers
Layer sizes	Size of every propagation layer
Readout phase	
Dropout	Dropout probability
MLP layer count	Count of hidden readout layers
MLP layer sizes	Size of feature vectors produced by every hidden readout layer

Table 5.1: Overview of hyperparameters.

Deep learning library	GNN library
TensorFlow [36]	Spektral [37]
PyTorch [38]	PyTorch Geometric [39]
PyTorch [38]	DGL [40]
Jax [41]	Jraph [42]

Table 5.2: Comparison of software for training GNNs.

IR of the input programs, we used a custom visitor based on the ComPy**??** library. To ensure that the extraction finishes in reasonable time on large datasets, we use the Ray Core library for parallelization.

2. Enumerating over all programs and collecting the possible feature categories for the node and memory location features. Furthermore we calculated the counts of the query labels, which we used to calculate the category weights

used the loss function.

3. Converting node and memory location to one-hot encoding using the categories collected in the last step and transforming the graph structure into the data structures used by PyTorch Geometric[39].

5.6.3 Training

1. Generation of all possible hyperparameters that should be used hyperparameter search. To execute multiple training runs in parallel we use the Ray Tune library.

2. Batching multiple graphs into a single graph to allow effient use of minibatching. In addition to the graph collation implemented by PyTorch Geometric, we adjust the correct indices of the nodes used by the aliasing queries when merging multiple graphs on the GPU.

3. Training of CAA and PAA models. We use PyTorch and PyTorch Geometric for implementing out model including the embedding, propagation and readout phases. During training we log the evaluation results on the training and validation dataset split on every epoch.

5.6.4 Evaluation

1. Evaluating the best models on the training, validation and/or test dataset split.

2. Visualizing the training and evaluation results.

Chapter 6

Evaluation and discussion

In the last two chapters, we described our methodology for creating a dataset for PAA and defined machine learning architectures for GNN models.

We continue by applying the dataset creation methodology to the Jotai input dataset. We then train and evaluate the GNN-based models on the CAA and PAA queries of the obtained dataset. To find out how well GNN-based models can perform on the task of predicting the alias analysis queries of the dataset, we compare them to multiple baselines and discuss the results.

6.1 Alias analysis dataset

6.1.1 Input dataset

In order to choose a input dataset, we compared multiple options used in the field of machine learning for code. As covered in chapter 4, out methodology requires not only compilable, but also runnable input programs. See table 6.1 for a comparison of different input datasets.

We choose the Jotai[3] dataset, because with over runnable $36K$ programs it provides a lot of data for training the machine learning models. The programs were collected from a large amount of real-world software projects on GitHub. The Jotai dataset is based on AnghaBench[30], but additionally contains inputs to run the programs. The inputs have been checked for undefined behavior, using `AddressSanitizer` and `KCC`[43]. Note that when we worked on the dataset creation, the programs and inputs of the ExeBench dataset were not yet released. The Jotai dataset contains 36223 programs with 69623 inputs in total.

We created the alias analysis dataset from the Jotai input dataset according to our methodology explained in chapter 4. While converting the input programs to LLVM IR, 35580 out of the 36223 programs could be compiled. We continued

Name	Programs	Compilable	Runnable	Release year
AnghaBench[30]	$1M$ program samples extracted from GitHub	Yes	No	2021
Project CodeNet [44]	$4M$ accepted C++ programs for $4K$ coding problems	Yes	Yes	2022
Jotai[3]	$36K$ leaf functions extracted from GitHub	Yes	Yes	2022
ExeBench[31]	$4.5M$ compilable, $0.8M$ runnable samples extracted from GitHub	Yes	Yes	2022

Table 6.1: Comparison of different input datasets.

with the compilable programs. We filtered out 262 programs of the Jotai dataset (`extr_hashcatsrcmodulesmodule_XXXXX.c_module_init_Final.c`, with `XXXXX` being a six-digit number), which are initialization functions from the `hashcat` GitHub repository. We removed them, because they contain autogenerated struct initialization code and would on their own account for 84.3% of PAA queries of the whole dataset.

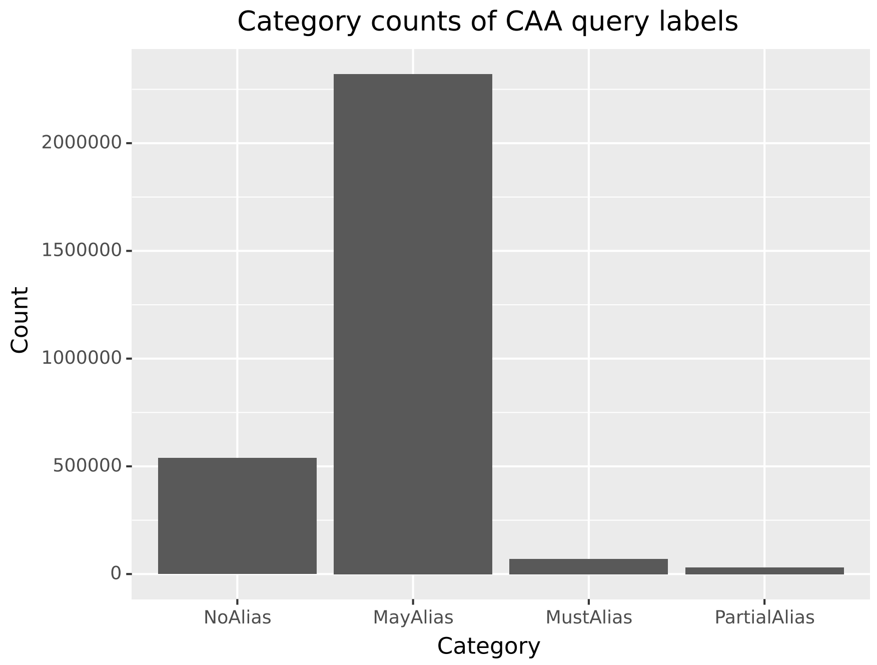
6.2 Graph-Neural-Network-based models

We evaluated the model on the Jotai dataset. This means, we can not make any statement about how good the model learned PAA in general, but only how well the model performed on the Jotai dataset.

We now describe out baselines, metrics and training methodology used for the evaluation. After presenting the results, we continue by discussing the performance of the GNN models and baselines.

6.2.1 Baselines

We use the following baselines during evaluation:



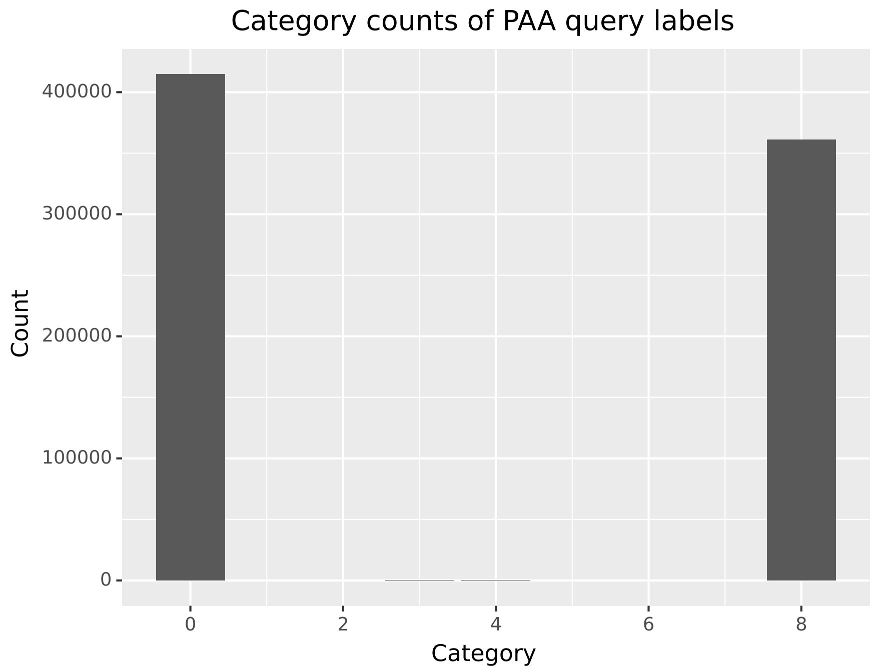


Figure 6.1: CAA and PAA category counts over all dataset splits. PAA categories are discretized as described in section 5.2. PAA query category 3, which includes query $y = \frac{1}{3}$, is contained once in the whole dataset, PAA query category 4, which includes query $y = \frac{1}{2}$, is contained twice.

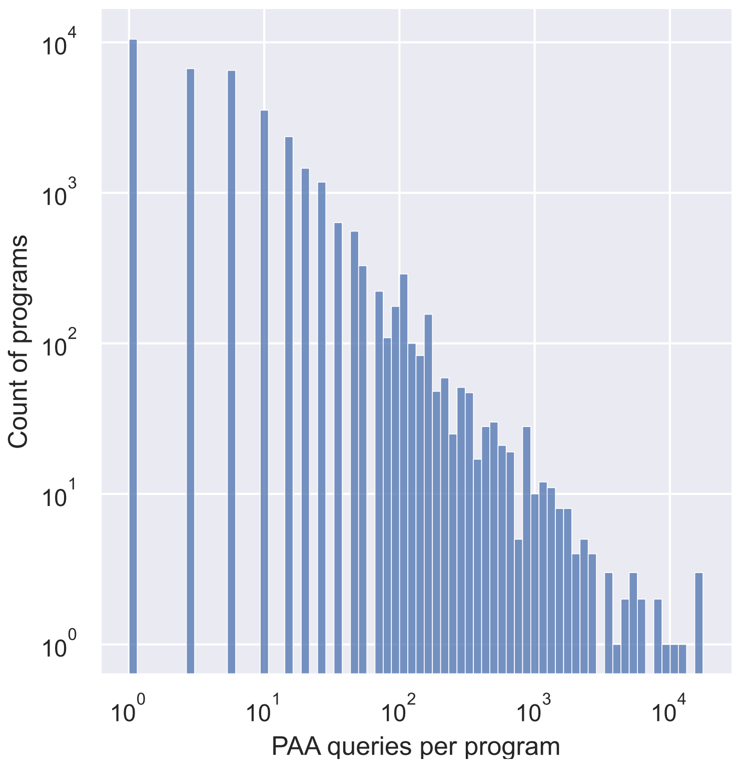


Figure 6.2: Histogram showing how many programs with a given count of PAA aliasing queries there are in the generated dataset. For example, the leftmost bar shows that there are approximately 10^4 programs with 1 alias query. The scales are logarithmic.

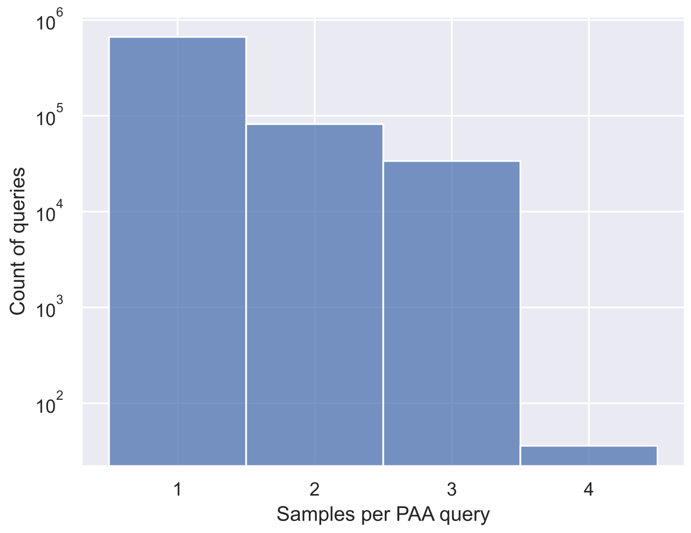


Figure 6.3: Histogram showing how many PAA queries of a given sample count exist in the dataset. The sample count of a query is calculated as the sum of the $DidAlias$ and $DidNotAlias$ count (see section 4.5.5).

Uniform random sample

As a first baseline, we record the results of a random model. It does not take any information about the PAA query or the program into account. Instead, it responds to an aliasing query by uniformly sampling from the $|C|$ possible query label categories:

$$p(c \in C) = \frac{1}{|C|}$$

Random sample from training distribution

This baseline records, during training, the distribution of the query labels and independently sample from this distribution during evaluation:

$$p(c \in C) = \frac{count(c)}{\sum_{c \in C} count(c)}$$

where $count(c)$ denotes how often a label c occurs in the training set.

Model with MLP propagation phase (no message passing)

In addition to the random baselines, we also compare the GNN models the a neural baseline. We use the architecture described in chapter 5, but use a MLP instead of the GNN propagation phase. Therefore, this baseline does not have the ability to aggregate features from instructions other than the instructions of the aliasing query. In contrast to the GNN-based models, it use the graph structure of the programs IR.

6.2.2 Evaluation metrics

During evaluation, we used the following metrics:

Cross-entropy loss

To show training progress and noise, we evaluated the loss on the train and validation set. As described section 5.5, during the training procedure, the loss on the train set can be higher than the loss on the validation set, because of the use of dropout in the embedding, readout or propagation layers.

Precision, recall, F1-score

Because the dataset has imbalanced query labels categories, especially for the CAA queries, we used precision, recall and F1-score during as evaluation metrics. For a binary classification task, they are defined as follows:

$$precision = \frac{tp}{tp + fp}$$

$$recall = \frac{tp}{tp + fn}$$

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

where tp is the number true positives, fp is the number of false positives and fn is the number of false negatives. We calculated all three metrics for every category $c \in C$, only taking the queries with c as *label* into account. We then calculated the *macro* average $metric_{macro}$ of each metric $metric \in precision, recall, F1$ as follows:

$$metric_{macro} = \frac{1}{|C|} \sum_{c \in C} metric_c$$

For each metric we calculated the *weighted* average as follows:

$$metric_{weighted} = \frac{1}{\sum_{c \in C} count(c)} \sum_{c \in C} count(c) \cdot metric_c$$

where $count(c)$ is the number of queries with label c.

Accuracy

We calculated accuracy as the number of queries with a correctly predicted label divided by the number of all queries. Note that the accuracy metric can be misleading if the label categories are unbalanced, which is especially the case for the CAA queries.

6.2.3 Experiments

We conducted two rows of experiments: On the CAA queries of our dataset and on the PAA queries. For each type of aliasing query, our evaluation consisted of the following steps:

1. First, we searched the hyperparameters space of each propagation type: MLP, GGNN and GAT. We started a single training run for every possible set of hyperparameters, and evaluated every models best epoch on the *validation* set. We choose the best set of hyperparameters according to validation loss.

2. For each propagation type, we continued with the best set of hyperparameter that have been found. This resulted in three models, one for each propagation type. We conducted five training runs for each of the models, to compare the training noise for the fixed sets of hyperparameters.

3. Of the five training runs, we selected the *median run* (in this case third-best run) by average loss on the validation set.

4. We evaluated the median models performance on the *train* set.

Note that for every training run in the above steps, we choose the model parameters θ at the epoch which had the lowest validation loss, which does not have to be the last epoch.

Hyperparameter search space

We evaluated the MLP, GGNN and GAT models on the hyperparameters shown in table 6.2. For GAT propagation layers we used a single attention head. We did an exhaustive search of the hyperparameter space, with a single training run per combination of hyperparameters. We selected the best set of hyperparameters by comparing the validation set loss of the models.

Hyper parameter name	Domain
Learning rate	$\{0.01\}$
Batch size	$\{1024\}$
Weight decay	$\{0.0005\}$
Embedding phase	
Dropout	$\{0.5\}$
MLP layer counts	$\{2\}$
MLP layer sizes	$\{64\}$
Propagation phase	
Propagation type	$\{MLP, GGNN, GAT\}$
Dropout	$\{0\}$ for GGNN, else $\{0, 0.5\}$
Layer count	$\{2, 4, 6\}$
Layer sizes	$\{64, 256, 512\}$
Readout phase	
Dropout	$\{0.5\}$
MLP layer count	$\{2\}$
MLP layer sizes	$\{64\}$

Table 6.2: Search space of CAA and PAA hyper parameter evaluation.

6.3 Discussion

We will first discuss the aliasing queries collected from the Jotai dataset. As shown in figure 6.3 there are only very few samples collected per PAA query from the Jotai dataset. The few samples that there are likely mostly come from the multiple inputs of each program, not from e.g. loops in the Jotai dataset. This indicates that collected aliasing data is not very complex. Figure 6.1 shows that only PAA queries with aliasing percentage 0% and 100% exist, except for 3 queries in the whole dataset with different categories. We therefore excluded categories 1 through 7 from the evaluation, because training neural networks typically requires more than 1 to 2 samples per category. Furthermore, none of the 3 queries has been assigned to the test set, which could make recall and precision ill-defined during evaluation (because for categories 1 to 7 there are e.g. no true positives in the test set). We therefore decided to do evaluation only on PAA query label categories 0 and 8. As shown in figure 6.1, the categories of the CAA are heavily unbalanced, with the majority of the queries having label $MayAlias$. This means, that the majority of queries could not be answered using the CAA algorithms implemented in LLVM. For exactly these cases, PAA could possibly still give an useful probabilistic response.

We did an exhaustive search of the hyperparameter search space. This means we also trained on the best set of hyperparameters during the hyperparameter search. Because the search space is limited, there could still be better hyperparameter outside the hyperparameter space. For example, the results of both hyperparameter searches show, that the best performing GGNN model uses a propagation layer count of 6, which was the maximum count in the search space, together with a propagation feature size of 64, which was the minimum size in the search space (see tables 6.3 and 6.4). Therefore, extending the hyperparameter search space, to e.g. more layers, could lead to better results. However, training even deeper GNN might require additional techniques to combat vanishing gradients during training. We discuss this further in the outlook (section 7.2). Similarly, the MLP and GAT models sometimes used layer counts or feature counts at the limit of the search space (2 or 6). Here, widening the search space could lead to improvements as well. Additionally, we did only evaluate different hyperparameters of the propagation layer. It could, however, be the case, that certain types of propagation phases perform significantly better/worse paired with a different embedding or readout phase. This could be solved by a hyperparameter search that also takes non-propagation hyperparameters into account. Another source that could influence the validity of the results is training noise. Having evaluated one training run per set of hyperparameters, it could be that the best set of hyperparameter averaged over more training runs has not been found because of noise during training, that can be caused by the randomly initialized weights of the network or the randomly chosen minibatches. Another results that can be observed, is that the GGNN consistently takes the first places ranked by validation loss, both for CAA queries, but especially for PAA queries. This could indicate that for the GGNN model, the concrete choice of hyperparameters is not as important.

Figures **??** and **??** show that there is some noise during the training process. The graphs also show, that the noise is smaller that the performance difference between the models, with the shaded area between different propagation type not overlapping much, except for some outlier runs. To mitigate the impact of outlier runs, we used the median model during evaluation on the test set. The figures furthermore show, that the MLP likely converged during the 100 training epochs, but the GAT and especially GGNN could potentially benefit from more epochs.

The evaluation scores of the models trained on CAA queries in figure 6.5 allows the following observations and conclusions:

- The GGNN outperforms both the MLP and GAT models in all metrics. This means the GGNN generalize best from the train and validation set to the test set.
- The GGNN outperforms the all random baselines in most metrics, most importantly in the macro and weighted F1-score.
- The neural networks perform better than the random baselines in macro precision, recall and F1-score, but have worse weighted macro, precision and recall scores. The reason for this is, that the neural networks got trained using a loss function where every category gets the same weight, and is equally important. In contrast, in the weighted precision, recall and F1-score, categories with more samples in the dataset have a higher impact. Better performance of the neural networks on the weighted scores could be achieved by training without a weighted loss function, but this would lead to lower macro scores.

The evaluation scores of the models trained on PAA queries in figure 6.7 allows the following observations and conclusions:

- The GGNN outperforms both the MLP and GAT models in almost all metrics (except category 8 recall), but most importantly in the macro and weighted F1-score. This means the GGNN generalize best from the train and validation set to the test set.
- All neural network models outperform the all random baselines in all metrics, except for category 0 where the static baseline per definition has a precision of 1.

Overall, the results show that, on the hyperparameters we evaluated, the GGNN model performed best on the dataset. In other words, the GGNN model generalized best from the train and validation sets to the test set. This indicates, that the architecture of repeatedly applying the same aggregation of neighbouring features (because weights are shared between the different GGNN propagation layers) is a good bias for the task of predicting CAA and PAA queries of the dataset best.

Prop. type	Prop. layers	Prop. feature size	Prop. dropout	Trainable params	Avg. val. loss	Avg. val. accuracy	Global loss rank
MLP	2	64	0	28612	**0.334321**	0.470492	**7**
MLP	2	256	0	115396	0.336512	0.429347	8
MLP	2	64	0.5	28612	0.337396	0.415565	9
MLP	2	512	0	345796	0.338110	0.521025	10
MLP	2	256	0.5	115396	0.339645	0.464891	11
MLP	2	512	0.5	345796	0.339976	0.416176	12
Details about MLP models with rank 6 to 15 omitted							
MLP	4	512	0.5	873156	0.402931	0.290466	30
MLP	6	512	0.5	1400516	0.404667	0.240478	31
GGNN	6	64	0	69700	**0.253530**	0.732258	**1**
GGNN	4	64	0	61508	0.259293	0.705427	2
GGNN	2	64	0	53316	0.264208	0.704289	3
GGNN	2	256	0	558276	0.313204	0.579577	4
GGNN	4	256	0	689348	0.333001	0.582222	6
GGNN	2	512	0	2149060	0.381326	0.531902	22
GGNN	4	512	0	2673348	0.455251	0.231432	37
GGNN	6	256	0	820420	0.457750	0.230905	39
GGNN	6	512	0	3197636	0.480498	0.231087	40
GAT	2	64	0	29252	**0.318106**	0.536847	**5**
GAT	2	256	0	117956	0.349123	0.414470	17
GAT	2	512	0	350916	0.363584	0.357991	18
GAT	4	64	0	38340	0.379023	0.324738	21
GAT	4	256	0	252612	0.383273	0.262970	23
GAT	6	64	0	47428	0.386328	0.338562	24
Details about GAT models with rank 6 to 15 omitted							
GAT	6	256	0	387268	0.897915	0.209601	44
GAT	6	512	0	1413828	0.900790	0.202828	45

Table 6.3: Results of hyperparameter search for CAA queries. The propagation phase of type *Prop. type* contains *Prop. layers* many layers, each outputting node embeddings of size *Prop. feature size. Dropout* denotes the dropout probability used only for the propagation phase, all other phases use dropout of 0.5. Loss and accuracy of the best epoch of every model are evaluated on the validation set and are averaged over all queries. Only half as many models were evaluated for GGNN, because the current implementation based on PyTorch Geometric does not support dropout.

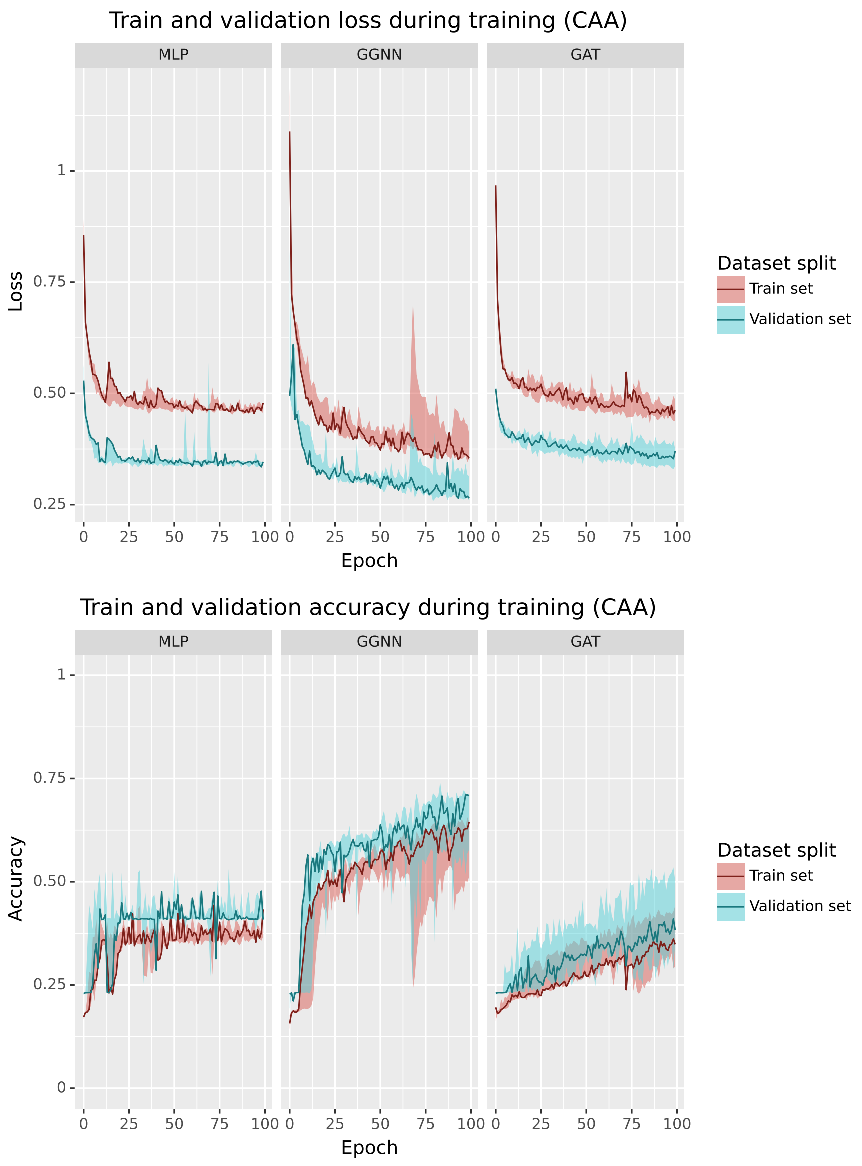


Figure 6.4: For each propagation layer type, we trained 5 models with the best hyperparameters from the hyperparameter search. The darker lines show the train/validation loss/accuracy of the model with the median (third best) performance of the 5 models. The area between the best/worst of the 5 models at each epoch is shaded, to visualize noise between different training runs (no smoothing applied). Lower loss and higher accuracy is better. Note that the training loss/accuracy can be worse than the validation counterpart because dropout was used during training (see section 5.5). Also note, that the classes are heavily unbalanced.

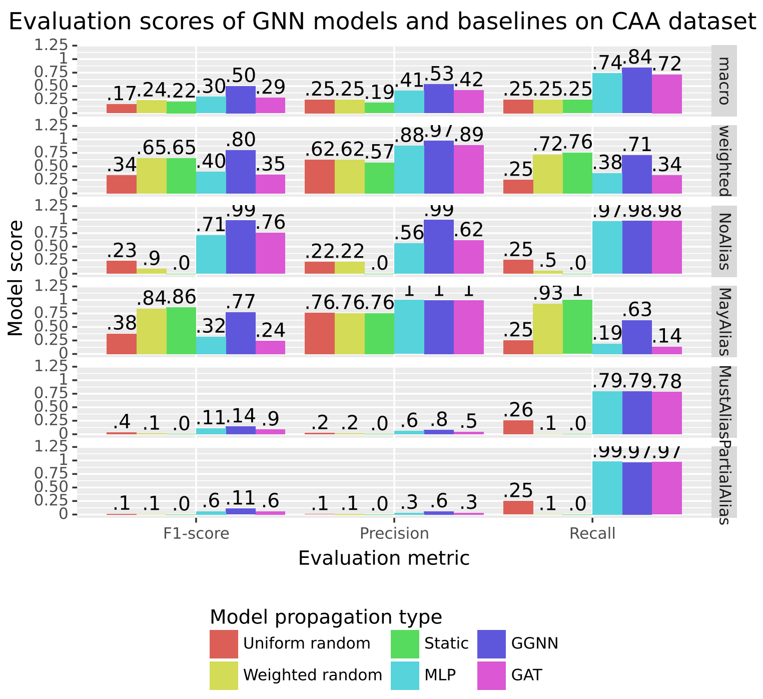
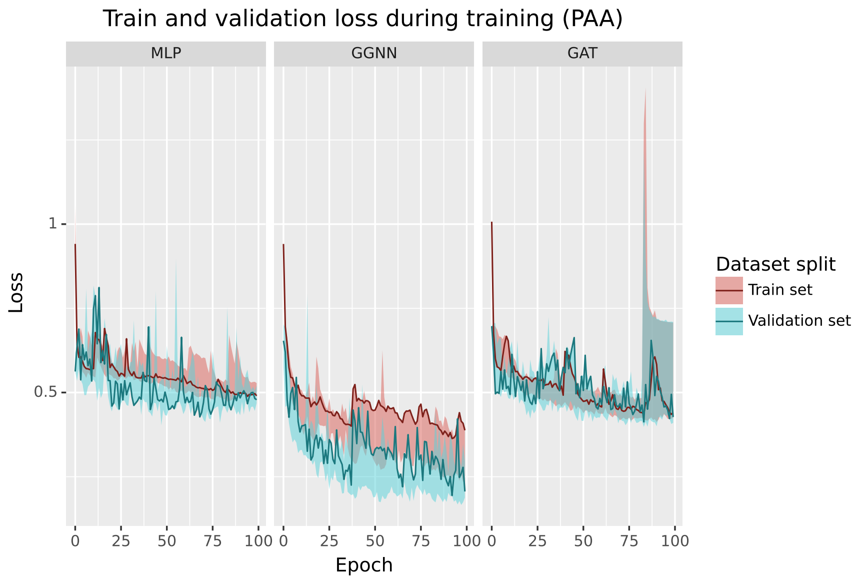


Figure 6.5: For each propagation layer type: Performance, on the test set, of the third-best (median) of the five models, each trained with the best hyperparameters found. Random/best static baselines for comparison. F1-score, precision and recall are measured for each category individually, and additionally the macro average (equal weight per category) and weighted average (weighted by number of queries in each category) is shown. Only relevant categories of the dataset shown, as explained above. Higher precision/recall/F1-score is better. Note that the categories are heavily unbalanced, as described in section 6.1.1.

Prop. type	Prop. layers	Prop. feature size	Prop. dropout	Trainable params	Avg. val. loss	Avg. val. accuracy	Global loss rank
MLP	6	256	0	380937	**0.398494**	0.799996	**9**
MLP	2	64	0	28937	0.406424	0.798500	12
MLP	6	64	0.5	46089	0.409413	0.805453	14
MLP	2	64	0.5	28937	0.412525	0.825592	15
MLP	4	64	0.5	37513	0.418223	0.801578	16
MLP	6	64	0	46089	0.419695	0.847723	17
Details about MLP models with rank 6 to 15 omitted							
MLP	6	256	0.5	380937	0.455150	0.738157	30
MLP	6	512	0	1400841	0.483404	0.692583	42
GGNN	6	64	0	70025	**0.205494**	0.929443	**1**
GGNN	2	64	0	53641	0.219895	0.897291	2
GGNN	4	64	0	61833	0.223433	0.910627	3
GGNN	6	256	0	820745	0.267755	0.800265	4
GGNN	4	256	0	689673	0.287579	0.871770	5
GGNN	2	256	0	558601	0.324307	0.781934	6
GGNN	6	512	0	3197961	0.340842	0.858982	7
GGNN	2	512	0	2149385	0.363189	0.863740	8
GGNN	4	512	0	2673673	0.447975	0.740902	29
GAT	2	512	0	351241	**0.398510**	0.782547	**10**
GAT	2	64	0	29577	0.404198	0.767101	11
GAT	2	256	0	118281	0.408323	0.723292	13
GAT	2	64	0.5	29577	0.441378	0.782816	26
GAT	2	256	0.5	118281	0.461627	0.765885	31
GAT	6	64	0.5	47753	0.465003	0.698826	32
Details about GAT models with rank 6 to 15 omitted							
GAT	6	256	0	387593	0.500299	0.693207	44
GAT	6	512	0.5	1414153	0.633014	0.723410	45

Table 6.4: Results of hyperparameter search for PAA queries. The propagation phase of type *Prop. type* contains *Prop. layers* many layers, each outputting node embeddings of size *Prop. feature size*. *Dropout* denotes the dropout probability used only for the propagation phase, all other phases use dropout of 0.5. Loss and accuracy of the best epoch of every model are evaluated on the validation set and are averaged over all queries. Only half as many models were evaluated for GGNN, because the current implementation based on PyTorch Geometric does not support dropout.



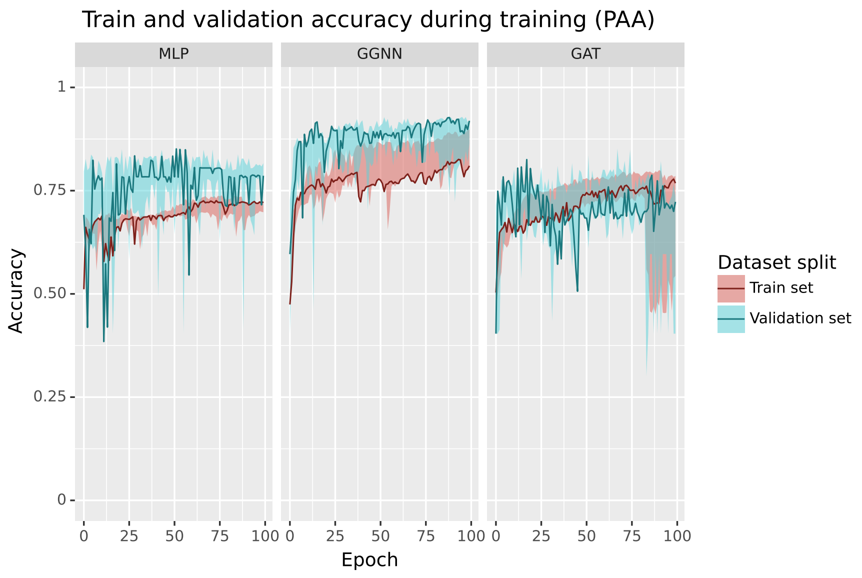
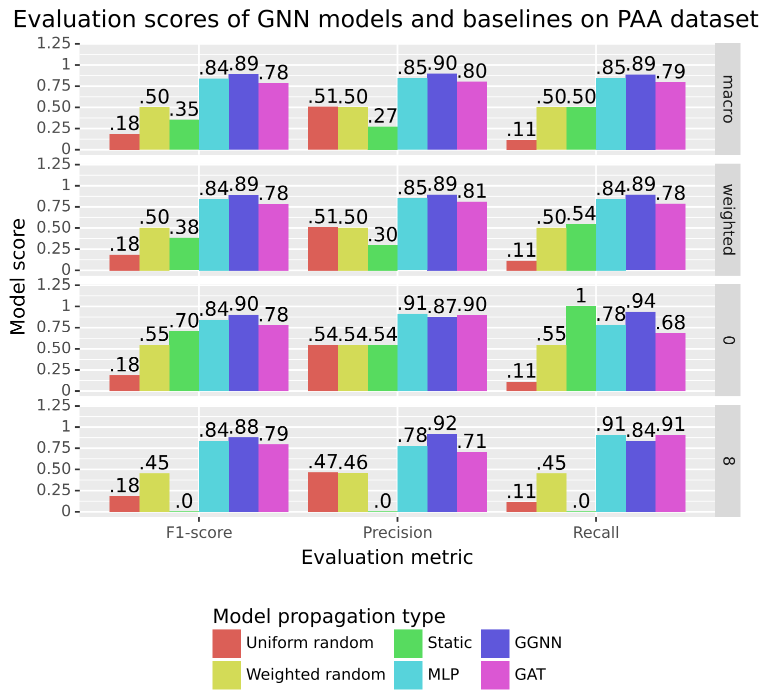


Figure 6.6: For each propagation layer type, we trained 5 models with the best hyperparameters from the hyperparameter search. The darker lines show the train/validation loss/accuracy of the model with the median (third best) performance of the 5 models. The area between the best/worst of the 5 models at each epoch is shaded, to visualize noise between different training runs (no smoothing applied). Lower loss/higher accuracy is better. Note that the training loss/accuracy can be worse than the validation counterpart because dropout was used during training (see section 5.5).



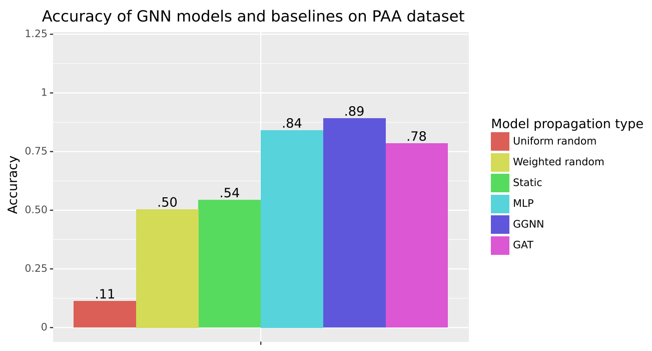


Figure 6.7: For each multiphase model: Performance, on the test set, of the third-best of the five models, each trained with the best hyperparameters found. Random/best static baselines for comparison. F1-score, precision and recall are measured for each category individually, and additionally the macro average (equal weight per category) and weighted average (weighted by number of queries in each category) is shown. Only relevant categories of the dataset shown, as explained above. Higher accuracy/precision/recall/F1-score is better. Note that the categories are not perfectly balanced, as described in section 6.1.1.

Chapter 7

Epilogue

7.1 Conclusion

In this thesis, we presented and implemented a dataset creation methodology that, using instrumentation, produces PAA queries for a given input dataset of C programs, and additionally includes statically gathered CAA queries.

We introduced GNN architectures that can process alias queries, both probabilistic and conventional, to give a prediction about the query result. For this purpose, we extended existing graph-based representations of input programs to include information about alias queries.

Additionally, we created a novel dataset of aliasing queries by applying our dataset creation methodology to the Jotai[3] input dataset. Having evaluated our GNN-based models on this dataset, our results show that, in general, it is possible for GNNs to learn PAA. Furthermore, the GGNN model consistently outperformed both random and MLP baselines on the task of predicting CAA and PAA queries. However, to show the viability of using GNNs for PAA in compilers, evaluation on additional, more challenging datasets are required.

7.2 Outlook

Input dataset To further study the strengths and limitations of GNN, more challenging input datasets are necessary. For example, input datasets with more complex control-flow-graphs, multiple functions, and more inputs per program, could provide additional information about the performance of machine learning models.

Dataset creation methodology We designed a dataset creation methodology for the C language. However, in the scope of future work, we can think of multiple promising options on how to extend this methodology:

- Applying different layers of optimization: Currently, the instrumentation pass gets run only on unoptimized code. To allow the trained machine learning models to predict aliasing of IR at different stages in the optimization pipeline of the compiler, the input programs could be optimized at different levels. As a measure of data augmentation, this could also improve the generalization performance of the machine learning models.
- Processing of additional input languages: Because we use the LLVM, the instrumentation pass can also be used for other source languages supported by LLVM, such as C++, Fortran or Rust.
- The PAA queries produces from the Jotai dataset mostly included aliasing probabilities of 0% or 100%. To improve the results of the PAA on a wider range of program inputs, the inputs could be deliberately chosen to produce different aliasing behaviour.

Use in compilers If GNNs continue to perform well on more challenging datasets, the viability of using PAA in compilers can evaluated. We are excited by the prospect of using GNNs for PAA in combination with existing optimizations, such as speculative multithreading, in a compiler toolchain. This will also allow measurements of the speedups obtained by the optimizations.

GNN model and training procedure We compared the GNN-based to random and the MLP baselines, however additional baselines, such as a RNN baseline based on the tokens of the IR can be evaluated.

As part of our evaluation methodology, we conducted a search of the hyperparameters of the propagation phase. We believe, that by including the parameters of the embedding and readout phase and other training hyperparameters, such as learning rate, batch size, weight decay, the performance of the GNN models can be improved. Furthermore, the best performing model based on GGNNs used the maximum number of layers of our hyperparameter search, training with additional layers could improve the performance especially on more challenging datasets. As number of propagation layers directly limits the range in which the GNN can gather information, techniques such as batch normalization[45] and skip connections are viable options to increase the perceptive field of the GNN and combat vanishing gradients. Other promising approaches exist, such as using a variable number of GNN layers during evaluation, possibly different from the number of layers using during training (e.g. as in [46]).

Acronyms

AST abstract syntax tree. 4, 6, 7, 15

CAA conventional alias analysis. 4, 7–10, 19, 21–25, 27, 30–33, 38, 39, 41, 44–49, 55

CDFG control- and data-flow graph. 4, 15

GAT graph attention network. 14, 35, 36, 45–48

GGNN gated graph neural network. 14, 35, 36, 45–49, 52, 55, 56

GNN graph neural network. 4, 5, 13–16, 30–32, 35–37, 39, 40, 44, 47, 55, 56

GRU gated recurrent unit. 13, 14

IR intermediate representation. 7, 15, 17, 18, 23–30, 32, 33, 37, 39, 44, 56

LSTM Long-Short-Term-Memory. 13, 15

MLP multi-layer perceptron. 12–15, 34–37, 44–48, 55, 56

PAA probabilistic alias analysis. 4, 5, 7–10, 16–21, 24, 25, 29–33, 36, 38–43, 45–48, 52, 55, 56

RNN recurrent neural network. 13, 15, 56

SSA single static assignment. 7, 27

Bibliography

[1] Chris Cummins et al. “CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research.” In: *arXiv:2109.08267 [cs]* (Dec. 22, 2021). arXiv: `2109.08267`. URL: `http://arxiv.org/abs/2109.08267` (visited on 05/12/2022).

[2] Alexander Brauckmann et al. “Compiler-based graph representations for deep learning models of code.” In: *Proceedings of the 29th International Conference on Compiler Construction*. CC ’20: 29th International Conference on Compiler Construction. San Diego CA USA: ACM, Feb. 22, 2020, pp. 201–211. ISBN: 978-1-4503-7120-9. DOI: `10.1145/3377555.3377894`. URL: `https://dl.acm.org/doi/10.1145/3377555.3377894` (visited on 04/22/2022).

[3] Cecília Conde Kind, Michael Canesche, and Fernando M Quintão Pereira. “Jotai: a Methodology for the Generation of Executable C Benchmarks.” In: (2022), p. 12.

[4] Alfred V. Aho, ed. *Compilers: principles, techniques, & tools*. 2nd ed. OCLC: ocm70775643. Boston: Pearson/Addison Wesley, 2007. 1009 pp. ISBN: 978-0-321-48681-3.

[5] Thomas Reps. “Undecidability of context-sensitive data-dependence analysis.” In: *ACM Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pp. 162–186. ISSN: 0164-0925, 1558-4593. DOI: `10.1145/345099.345137`. URL: `https://dl.acm.org/doi/10.1145/345099.345137` (visited on 10/27/2022).

[6] Susan Horwitz. “Precise flow-insensitive may-alias analysis is NP-hard.” In: *ACM Transactions on Programming Languages and Systems* 19.1 (Jan. 1997), pp. 1–6. ISSN: 0164-0925, 1558-4593. DOI: `10.1145/239912.239913`. URL: `https://dl.acm.org/doi/10.1145/239912.239913` (visited on 05/17/2022).

[7] Bjarne Steensgaard. “Points-to analysis in almost linear time.” In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’96*. the 23rd ACM SIGPLAN-SIGACT symposium. St. Petersburg Beach, Florida, United States: ACM Press, 1996,

pp. 32–41. ISBN: 978-0-89791-769-8. DOI: 10.1145/237721.237727. URL: http://portal.acm.org/citation.cfm?doid=237721.237727 (visited on 05/24/2022).

[8] Lars Ole Andersen. “Program Analysis and Specialization for the C Programming Language.” In: (1994), p. 43.

[9] Yannis Smaragdakis and George Balatsouras. “Pointer Analysis.” In: *Foundations and Trends® in Programming Languages* 2.1 (2015), pp. 1–69. ISSN: 2325-1107, 2325-1131. DOI: 10.1561/2500000014. URL: http://www.nowpublishers.com/article/Details/PGL-014 (visited on 10/27/2022).

[10] Tian Tan et al. “Making pointer analysis more precise by unleashing the power of selective context sensitivity.” In: *Proceedings of the ACM on Programming Languages* 5 (OOPSLA Oct. 20, 2021), pp. 1–27. ISSN: 2475-1421. DOI: 10.1145/3485524. URL: https://dl.acm.org/doi/10.1145/3485524 (visited on 05/25/2022).

[11] Yue Li et al. “A Principled Approach to Selective Context Sensitivity for Pointer Analysis.” In: *ACM Transactions on Programming Languages and Systems* 42.2 (May 27, 2020), pp. 1–40. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/3381915. URL: https://dl.acm.org/doi/10.1145/3381915 (visited on 05/25/2022).

[12] Dongjie He, Jingbo Lu, and Jingling Xue. “Context Debloating for Object-Sensitive Pointer Analysis.” In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). Melbourne, Australia: IEEE, Nov. 2021, pp. 79–91. ISBN: 978-1-66540-337-5. DOI: 10.1109/ASE51524.2021.9678880. URL: https://ieeexplore.ieee.org/document/9678880/ (visited on 05/25/2022).

[13] Yue Li et al. “Precision-guided context sensitivity for pointer analysis.” In: *Proceedings of the ACM on Programming Languages* 2 (OOPSLA Oct. 24, 2018), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3276511. URL: https://dl.acm.org/doi/10.1145/3276511 (visited on 05/26/2022).

[14] Minseok Jeon, Myungho Lee, and Hakjoo Oh. “Learning graph-based heuristics for pointer analysis without handcrafting application-specific features.” In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA Nov. 13, 2020), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3428247. URL: https://dl.acm.org/doi/10.1145/3428247 (visited on 05/25/2022).

[15] C. Lattner and V. Adve. “LLVM: A compilation framework for lifelong program analysis & transformation.” In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* International Symposium on Code Generation and Optimization, 2004. CGO 2004. San Jose, CA, USA: IEEE, 2004, pp. 75–86. ISBN: 978-0-7695-2102-2. DOI: 10.1109/CGO.2004.1281665. URL: http://ieeexplore.ieee.org/document/1281665/ (visited on 08/20/2022).

[16] David R Chase, Mark Wegman, and F. Kenneth Zadeck. “Analysis of Pointers and Structures.” In: (1990), p. 15.

[17] Wonsun Ahn, Yuelu Duan, and Josep Torrellas. “DeAliaser: alias speculation using atomic region support.” In: *ACM SIGPLAN Notices* 48.4 (Apr. 23, 2013), pp. 167–180. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2499368.2451136. URL: https://dl.acm.org/doi/10.1145/2499368.2451136 (visited on 06/01/2022).

[18] Dongliang Mu et al. “RENN: Efficient Reverse Execution with Neural-Network-Assisted Alias Analysis.” In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). 4 citations (Semantic Scholar/DOI) [2022-05-31]. San Diego, CA, USA: IEEE, Nov. 2019, pp. 924–935. ISBN: 978-1-72812-508-4. DOI: 10.1109/ASE.2019.00090. URL: https://ieeexplore.ieee.org/document/8952186/ (visited on 05/26/2022).

[19] Peng-Sheng Chen et al. “Compiler support for speculative multithreading architecture with probabilistic points-to analysis.” In: *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP ’03.* the ninth ACM SIGPLAN symposium. 67 citations (Semantic Scholar/DOI) [2022-05-31]. San Diego, California, USA: ACM Press, 2003, p. 25. ISBN: 978-1-58113-588-6. DOI: 10.1145/781498.781502. URL: http://portal.acm.org/citation.cfm?doid=781498.781502 (visited on 05/25/2022).

[20] Jeffrey L. Elman. “Finding Structure in Time.” In: *Cognitive Science* 14.2 (Mar. 1990), pp. 179–211. ISSN: 03640213. DOI: 10.1207/s15516709cog1402_1. URL: http://doi.wiley.com/10.1207/s15516709cog1402_1 (visited on 10/24/2022).

[21] Haşim Sak, Andrew Senior, and Françoise Beaufays. “Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition.” In: (2014). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.1402.1128. URL: https://arxiv.org/abs/1402.1128 (visited on 10/24/2022).

[22] Kyunghyun Cho et al. “On the Properties of Neural Machine Translation: Encoder-Decoder Approaches.” In: (2014). Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.1409.1259. URL: https://arxiv.org/abs/1409.1259 (visited on 10/24/2022).

[23] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks.” In: *arXiv:1609.02907 [cs, stat]* (Feb. 22, 2017). arXiv: 1609.02907. URL: http://arxiv.org/abs/1609.02907 (visited on 04/09/2022).

[24] William L. Hamilton, Rex Ying, and Jure Leskovec. “Inductive Representation Learning on Large Graphs.” In: *arXiv:1706.02216 [cs, stat]* (Sept. 10, 2018). arXiv: 1706.02216. URL: http://arxiv.org/abs/1706.02216 (visited on 04/06/2022).

[25] Yujia Li et al. *Gated Graph Sequence Neural Networks.* Number: arXiv:1511.05493. Sept. 22, 2017. arXiv: 1511.05493[cs,stat]. URL: http://arxiv.org/abs/1511.05493 (visited on 05/18/2022).

[26] Petar Veličković et al. “Graph Attention Networks.” In: *arXiv:1710.10903 [cs, stat]* (Feb. 4, 2018). arXiv: 1710.10903. URL: http://arxiv.org/abs/1710.10903 (visited on 04/10/2022).

[27] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. “Neural Code Comprehension: A Learnable Representation of Code Semantics.” In: *arXiv:1806.07336 [cs, stat]* (Nov. 29, 2018). arXiv: 1806.07336. URL: http://arxiv.org/abs/1806.07336 (visited on 04/25/2022).

[28] Alexander Brauckmann, Andres Goens, and Jeronimo Castrillon. “ComPy-Learn: A toolbox for exploring machine learning representations for compilers.” In: *2020 Forum for Specification and Design Languages (FDL).* 2020 Forum for Specification and Design Languages (FDL). Kiel, Germany: IEEE, Sept. 15, 2020, pp. 1–4. ISBN: 978-1-72818-928-4. DOI: 10.1109/FDL50818.2020.9232946. URL: https://ieeexplore.ieee.org/document/9232946/ (visited on 05/18/2022).

[29] GCC Developers. *The GNU Compiler Collection.* Version 12.2. Aug. 19, 2022. URL: https://gcc.gnu.org/.

[30] Anderson Faustino da Silva et al. “ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction.” In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Seoul, Korea (South): IEEE, Feb. 27, 2021, pp. 378–390. ISBN: 978-1-72818-613-9. DOI: 10.1109/CGO51591.2021.9370322.

URL: https://ieeexplore.ieee.org/document/9370322/ (visited on 09/21/2022).

[31] Jordi Armengol-Estapé et al. "ExeBench: an ML-scale dataset of executable C functions." In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming.* MAPS '22: 6th ACM SIGPLAN International Symposium on Machine Programming. San Diego CA USA: ACM, June 13, 2022, pp. 50–59. ISBN: 978-1-4503-9273-0. DOI: 10.1145/3520312.3534867. URL: https://dl.acm.org/doi/10.1145/3520312.3534867 (visited on 07/05/2022).

[32] Jan Huckelheim and Johannes Doerfert. "ORAQL — Optimistic Responses to Alias Queries in LLVM." In: (), p. 12.

[33] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. "Winnowing: Local Algorithms for Document Fingerprinting." In: (2003), p. 10.

[34] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." In: (2014). Publisher: arXiv Version Number: 9. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980 (visited on 10/22/2022).

[35] Geoffrey E. Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors." In: (2012). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.1207.0580. URL: https://arxiv.org/abs/1207.0580 (visited on 10/21/2022).

[36] TensorFlow Developers. *TensorFlow.* Version v2.8.2. May 23, 2022. DOI: 10.5281/ZENODO.4724125. URL: https://zenodo.org/record/4724125 (visited on 09/19/2022).

[37] Daniele Grattarola and Cesare Alippi. "Graph Neural Networks in TensorFlow and Keras with Spektral." In: (2020). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2006.12138. URL: https://arxiv.org/abs/2006.12138 (visited on 09/19/2022).

[38] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: (2019). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.1912.01703. URL: https://arxiv.org/abs/1912.01703 (visited on 09/19/2022).

[39] Matthias Fey and Jan Eric Lenssen. "Fast Graph Representation Learning with PyTorch Geometric." In: (2019). Publisher: arXiv Version Number: 3. DOI: 10.48550/ARXIV.1903.02428. URL: https://arxiv.org/abs/1903.02428 (visited on 09/19/2022).

[40] Minjie Wang et al. “Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks.” In: (2019). Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.1909.01315. URL: https://arxiv.org/abs/1909.01315 (visited on 09/19/2022).

[41] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs.* Version 0.3.13. 2018. URL: http://github.com/google/jax.

[42] Jonathan Godwin* et al. *Jraph: A library for graph neural networks in jax.* Version 0.0.1.dev. 2020. URL: http://github.com/deepmind/jraph.

[43] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. “Defining the undefinedness of C.” In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI ’15: ACM SIGPLAN Conference on Programming Language Design and Implementation. Portland OR USA: ACM, June 3, 2015, pp. 336–345. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737979. URL: https://dl.acm.org/doi/10.1145/2737924.2737979 (visited on 08/21/2022).

[44] Ruchir Puri et al. “Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks.” In: (2022), p. 21.

[45] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” In: *arXiv:1502.03167 [cs]* (Mar. 2, 2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167 (visited on 04/06/2022).

[46] Jan Tönshoff et al. “Graph Neural Networks for Maximum Constraint Satisfaction.” In: *Frontiers in Artificial Intelligence* 3 (2021). ISSN: 2624-8212. DOI: 10.3389/frai.2020.580607. URL: https://www.frontiersin.org/article/10.3389/frai.2020.580607.