TECHNISCHE UNIVERSITÄT DRESDEN

FACULTY OF COMPUTER SCIENCE
CENTER FOR ADVANCING ELECTRONICS DRESDEN
CHAIR FOR COMPILER CONSTRUCTION
PROF. DR. JERONIMO CASTRILLON

Master's Thesis

for obtaining the academic degree
Master of Science

A Domain-Specific Generative Model of Code for LLVM

Alexander Thierfelder
(Born 20. May 1993 in Dresden)

Tutor: Andrés Goens

Dresden, January 30, 2020

Task Description

The source code used to describe software applications is generally rich in structure and complex. Generative models in machine learning provide methods to learn the properties of complex structures from examples. Learning a generative model of code can help produce code examples. This has several applications, ranging from benchmark generation to code completion and other constrained generation tasks.

Methods for automated generation of code have been developed and studied for decades. However, the research community has barely scratched the surface of the new deep-learning-based approaches in this field. In particular, state-of-the-art generative models of code [1] use a model at the character-level of the source code. This disregards much of the structure we know to be present in the source code, in the hopes that the neural networks can learn said structure from the character strings. This is particularly problematic when dealing with long-term dependencies, which is for example why CLgen [1] explicitly counts and adds closing brackets (}) manually.

In the realm of generative models based on neural networks, recent advances have been made that allow to train and learn graph-based models [2]. This structure better reflects the structure of code, for example, as expressed by the control- and dataflow graph (CDFG) in the LLVM compiler framework. In this thesis, we shall explore these graph-based methods on LLVM. Specifically, we consider the deep generative models of graphs (DeepGMG) graph-based method and adapt it to learn and generate CDFG from LLVM code.

The LLVM language has a very particular structure. For example, a conditional “br” instruction takes exactly three arguments: a condition, and two labels to jump to depending on the value of the condition. While a graph-based generative model can learn this structure from several examples, we can also explicitly code this into the generative algorithm. This has the advantage of leveraging the structural information we know. We conjecture that by doing so, the model can “focus” on learning the semantical information of the code, instead of the structural information that defines valid LLVM code. In this thesis, we will develop such a domain-specific generative model of code that encodes structural information of the LLVM language. Concretely, the student shall:

1. Familiarize himself with the methods from [1] and [2], as well as the LLVM language.

2. Develop a graph-based representation based on the LLVM CDFG, such that the LLVM code can be fully reconstructed from it.
3. Using available implementations of these methods, and the CLGen data set, develop a generative model using DeepGMG and the graph-based representation above.
4. Analyze the LLVM language and its structure and re-design the DeepGMG algorithm to encode LLVM-specific structures like the conditional “br” instruction from the example.
5. Implement the modified DeepGMG algorithm to produce LLVM code that is mostly structurally correct by construction.
6. Test the different implementations and compare them to the code generated by CLGen on the OpenCL data set from CLGen.

Declaration of authorship

I hereby declare that I wrote this thesis on the subject

A Domain-Specific Generative Model of Code for LLVM

independently. I did not use any other aids, sources, figures or resources than those stated in the references. I clearly marked all passages that were taken from other sources and cited them correctly.

Furthermore I declare that – to my best knowledge – this work or parts of it have never before been submitted by me or somebody else at this or any other university.

Dresden, January 30, 2020

Alexander Thierfelder

Abstract

Graph structured data effectively models relational structures in many different domains, including source code. Recently, graph-based generative models have risen in popularity; however, existing approaches are often too general and computationally expensive to generate large graphs in complex target domains. In this thesis, we implement a domain-specific graph generative model for the generation of the LLVM intermediate representation (IR) based on Li et al.'s DeepGMG. For this purpose, we design a graph representation for LLVM IR and incorporate structural knowledge about the language into our graph generation process. Our approach generates valid samples at a substantially higher rate than the general-purpose DeepGMG. Additionally, we find that our model's samples are considerably richer in structure than those generated by the base model. However, our approach is not able to outperform the state-of-the-art CLgen in eighter of those metrics. Furthermore, we observe various issues of our model, the biggest one being type-correctness. Consequently, our model is biased towards structurally simple types and small graphs. As a result, the graphs generated by our model only partially cover the feature-space present in our training set.

Contents

1 Introduction

Graphs are highly versatile data structures that effectively capture complex relationships between entities in many important scientific, economic, and social domains. One example of such a domain is source code, which is generally rich in structural entities such as instructions, operands, and other language-specific concepts. Recently, graph-based generative models have risen in popularity; however, to our knowledge, not much research has been done on the ability of such models to generate code.

In this thesis, we design and implement a domain-specific model for the generation of the LLVM IR, which is the assembly language used inside the LLVM compiler. We base our model on the general-purpose graph generative model proposed by Li et al. [2] and modify it to specifically accommodate the graph representation of LLVM IR that we develop throughout this thesis.

We begin this work with an analysis of the machine learning concepts required to implement our approach (chapter 3), and a review of the model proposed by Li et al. (chapter 4). In chapter 5, we give a short overview of the LLVM project, analyze the LLVM IR, and subsequently develop a suitable graph representation for it. At this point, we will have fulfilled all necessary prerequisites for the extension of Li et al.'s DeepGMG into our domain-specific approach. Chapter 6, therefore, discusses both our structural modifications of the model, as well as our new graph generation algorithm. In the last part of this thesis, we examine the training procedure (chapter 7), evaluate our model (chapter 8), and give an outlook on possible future work (chapter 9).

Although our goal is the generation of LLVM code, we work atop the training set provided by Cummins et al. [1], which consists of roughly 5600 OpenCL kernels. Throughout this thesis, we develop a pipeline which enables us to transform OpenCL kernels into a graph representation, train our model, and generate LLVM code.

Code Generation Pipeline In order to synthesize LLVM code, various transformation and computation steps are necessary. Figure 1.1 shows the code generation toolchain we developed for this purpose.

The first pipeline step transforms an OpenCL kernel into an LLVM program through Clang [3], which can compile source code directly into LLVM IR. In the next step, we extract all structurally relevant

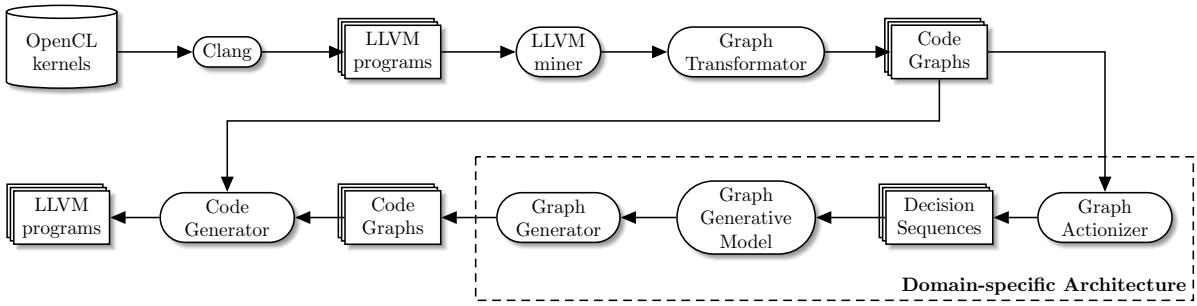


Figure 1.1: LLVM code generation pipeline.

information from the IR and save it to a JSON file. Usually, the LLVM optimizer *opt* uses passes to optimize or collect information about a program; we can use this architecture for our purposes by developing an analysis pass that traverses the IR and collects all necessary data. We can subsequently use this information to construct a code graph, which is a restructured, in-memory representation of the code.

The graph generative model introduced by Li et al. [2], and by extension, our model, trains on decision sequences, which represent a number of actions concerning the generation of a graph. Therefore, we utilize a graph actionizer to transform code graphs into their corresponding decision sequences. We then use these sequences to train our domain-specific generative model, which we develop on top of Brauckmann's implementation [4] of DeepGMG. In the next step, we use a graph generator to transform decision sequences generated by our model back into code graphs, which we subsequently transform into LLVM IR through our code generator.

It is worth noting that all domain-specific aspects of our approach are located in the graph-actionizer, the model itself, and the graph generator. Therefore, when we compare our approach against Li et al.'s model during the evaluation in chapter 8, we only need to adapt these three components of the pipeline.

2 Related Work

Graph neural networks (GNNs) have been initially proposed by Scarselli et al. in 2009 [5], and are a class of neural network models whose unique characteristic is the ability to process graph-structured input directly. Unlike the work done in [2] and [6], Scarselli et al. utilize an output model that maps node embeddings to outputs which are independent per node. In the same year, Micheli [7] proposed a model closely related to GNNs, which differs from them mainly in its usage of transductions to map input graphs to an output domain.

In 2016, Li et al. [6] proposed gated graph sequence neural networks (GG-SNNs), which modify GNNs to predict output sequences. The most significant structural adaptations of GG-SNNs are the usage of gated recurrent units (GRUs) [8] for the computation of node embeddings, and the propagation of information over a fixed number of steps T. Importantly for our work, this change allows for the usage of domain-specific node embedding initializations, since there no longer exists a fixed-point that node embeddings converge to during the information propagation process.

Johnson has described an approach related to GG-SNNs in 2017 [9]. He proposed an extension of GG-SNNs called gated graph transformer neural networks, which construct graphs to use them as an intermediate representation to solve reasoning problems. However, unlike the work done by Li et al. [2], Johnson's approach makes assumptions about the generation process, such as a fixed number of nodes for each input sentence.

Research has also been done towards domain-specific generative models of graphs. In 2018, Li et al. [10] proposed a new approach for the generation of molecules. Compared to the general approach in [1], this method generates graphs in a domain-specific iterative process and scales to larger molecules than those generated in the experiments we analyze in section 4.4.1. This work shows that sacrificing generality for performance in a chosen target domain can be a valuable tradeoff.

To the best of our knowledge, there is no existing work regarding the generation of LLVM code. This is not true for code generation in general, however. In 2015, Chiu et al. described GENESIS [11], a language for the generation of synthetic training programs in machine learning, a use case which is also possible for the samples generated in this thesis. However, unlike our approach, GENESIS relies on user input to annotate specific parts of a template program with possible values and a corresponding statistical

distribution to generate samples.

Graph representations of code have also been used in machine learning tasks. In 2018, Allamanis et al. [12] used a graph representation of C# programs based on their abstract syntax tree with added data-flow edges to reason over variable names and misuses. Unlike the work done in this thesis, however, Allamanis et al. approach is based on GNNs and is consequently not used in generative tasks.

3 Machine Learning Foundations

This chapter introduces the machine learning concepts and techniques required for the implementation of DeepGMG. We examine each technique with regards to its mathematical concepts, merits, and challenges.

3.1 Recurrent Neural Networks

Conventional feed-forward neural networks (NNs) have been applied to a multitude of classification and regression problems successfully. They struggle, however, at tasks that require information persistence. For example, it is unclear how a traditional NN could use its knowledge about previously generated words of a sentence to influence the generation of the next word. Recurrent neural networks (RNNs) address this issue by feeding past information back into themselves, where it is, in turn, used to influence future decisions. Figure 3.1 shows the schematic layout of an RNN in its ordinary and unrolled form.

Their structure makes RNNs naturally suited for sequential tasks, and they have been successfully applied to a large number of problems such as language modeling [13], speech recognition [14] and stock price pattern recognition [15].

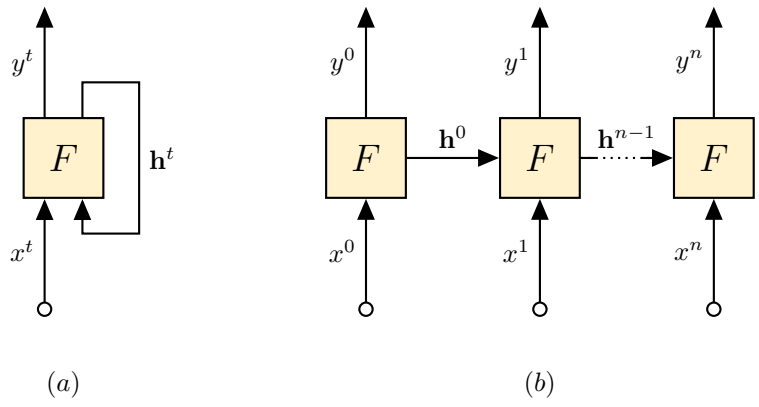


Figure 3.1: Schematic structure of an RNN by itself (a), and unrolled through time (b).

Since an RNN's behavior is temporally dynamic, its output at time t is dependent on all previous calculations up to that point. When training an RNN with a backpropagation algorithm, this temporal structure has to be simulated by unrolling the network with a technique called backpropagation through time (BPTT). This, however, requires a significantly higher amount of memory than traditional backpropagation. Furthermore, conventional RNNs trained with BPTT have been found to have difficulties learning long time dependencies and are inefficient to train due to a phenomenon called the "vanishing gradient problem" [16].

Vanishing gradients occur when RNNs are trained with BPTT over a large number of time steps t. Since individual weights inside the RNN are typically smaller than 1, they, and with them their derivative, decrease exponentially as t increases, which in turn makes the weights slower to update and the network as a whole harder to train.

Since the discovery of these problems, a lot of research on potential solutions has been made, leading to the development of various novel RNN architectures. One of them, the GRU, will be discussed in the next section.

3.2 Gated Recurrent Unit

The GRU is an RNN core introduced by Cho et al. in 2014 [8]. It aims to solve the vanishing gradient problem found in more traditional RNN cores, while at the same time having a simpler, and therefore easier to train internal structure than long short-term memory (LSTM) models. Figure 3.2 illustrates the mechanisms inside a fully gated GRU cell.

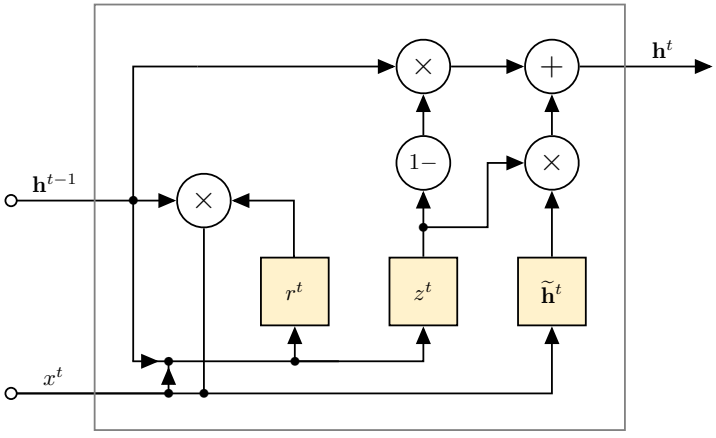


Figure 3.2: Internal structure of a fully gated GRU cell, as proposed by Cho et al. [8].

In this thesis, we use a GRU to predict node embeddings for the GNNs of Li et al.'s generative model [2] discussed in chapter 4.

3.2.1 Architecture

Contrary to LSTMs, a GRU has no need for an output gate, as all past information is carried in the output of the previous time step $t-1$. To achieve this, the cell updates its internal state $\mathbf{h}^t$ using an update gate z and a reset gate r. The computation of $\mathbf{h}^t$ is performed in three steps:

First, the reset gate r is computed. It decides how much of the previous hidden state $\mathbf{h}^{t-1}$ is "remembered", and therefore incorporated into $\mathbf{h}^t$. It is computed in the following way:

$$r^t = \sigma([\mathbf{W}_r \mathbf{x}^t] + [\mathbf{U}_r \mathbf{h}^{t-1}]) \tag{3.1}$$

Here, σ is the logistic sigmoid function, $\mathbf{W}_r$ and $\mathbf{U}_r$ are learned weight matrices, $\mathbf{x}$ is the cell input and $\mathbf{h}^{t-1}$ is the hidden state at time step $t-1$.

In the next step, the update gate z is computed. It controls how much information from $\mathbf{h}^{t-1}$ is carried over into the current hidden state $\mathbf{h}^t$. It acts similar to the memory cell in an LSTM and helps the GRU remember long-term information. z is computed in a similar fashion to r:

$$z^t = \sigma([\mathbf{W}_z \mathbf{x}^t] + [\mathbf{U}_z \mathbf{h}^{t-1}]) \tag{3.2}$$

Analogous to Equation 3.1, $\mathbf{W}_z$ and $\mathbf{U}_z$ are learned weight matrices.

Once both gates are computed, the hidden state $\mathbf{h}^t$ is updated in the following way:

$$\widetilde{\mathbf{h}}^t = \phi([\mathbf{W}x] + [\mathbf{U}(r \odot \mathbf{h}_{t-1}]) \tag{3.3}$$

$$\mathbf{h}^t = z\mathbf{h}^{t-1} + (1-z)\widetilde{\mathbf{h}}^t \tag{3.4}$$

In equation 3.3, the current memory content $\widetilde{\mathbf{h}}^t$ is calculated. Here, ϕ represents the $tanh$-function, $\odot$ is the Hadamard (elementwise) product, and $\mathbf{W}$ and $\mathbf{U}$ are once more learned weight matrices. Equation 3.4 then calculates $\mathbf{h}^t$ using the update gate z to control the ratio of information carried over from the previous hidden state $\mathbf{h}^{t-1}$ versus the information incorporated from the current memory content $\widetilde{\mathbf{h}}^t$.

3.2.2 Evaluation

Since their proposal in 2014, GRUs have been quickly adopted and used in a multitude of scenarios. While they are strictly weaker than LSTMs [17], Chung et al. [18] have found them comparable to LSTMs in terms of performance.

In 2015, Jozefowicz et al. [19] examined how LSTMs, GRUs, and several dynamically generated architectures perform in various tasks. They found that GRUs outperform LSTMs in tasks such as arithmetic integer computations and XML-modelling, while LSTMs outperform GRUs in a language modeling task on the Penn TreeBank[1].

Li et al. [2] have tested DeepGMG both with an LSTM and a GRU and found the results comparable. In addition, the GRU is, thanks to its simpler structure compared to an LSTM, able to alleviate some of DeepGMG's problems that we discuss in section 4.4.2.

This, combined with the results of the research mentioned above, makes the GRU a suitable RNN core for our domain-specific graph generative model.

[1] A treebank is a structured, annoted text corpus.

4 Deep Generative Models of Graphs

This chapter discusses the machine learning model introduced by Li et al. in March 2018 [2]. Since the model is an integral part of the work in this thesis, and we modify and expand it throughout the following chapters, we conduct an in-depth examination of its underlying mechanisms and mathematical foundations.

4.1 Overview

DeepGMG use graph neural networks to learn probability distributions over graphs. They make no assumptions about the structure of the graphs and can, therefore, in principle, be used to generate any arbitrary graph.

This is achieved through a sequential generation process, which splits up the graph generation into a series of structural decisions. During this process, nodes are generated one at a time and are connected to the already existing part of the graph by creating edges one by one. Every action is influenced by the history of structure building decisions before it, and therefore by the current state of the graph. One example of this process is illustrated in Figure 4.1.

The distribution of information inside the graph is achieved through an information propagation process.

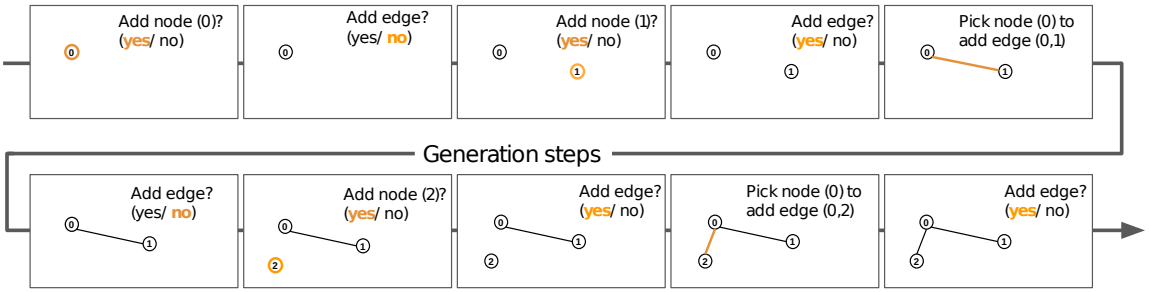


Figure 4.1: Depiction of the steps taken during the generation process, as taken from [2]. Used with permission of Dr. Yujia Li.

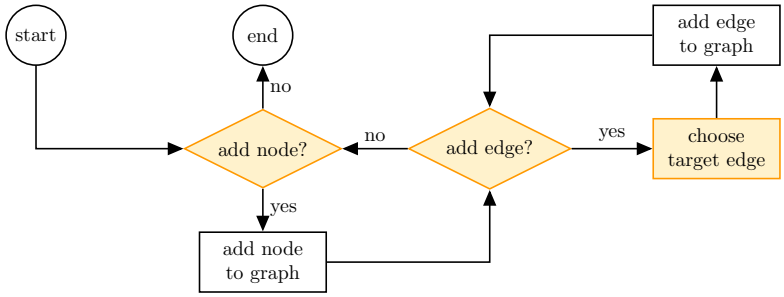


Figure 4.2: Flowchart of the sequential graph generation process. The three boxes highlighted in orange perform predictions using the $f_{addnode}$ (left), $f_{addedge}$ (center), and f_{nodes} modules (right).

4.2 Graph Generation Process

The generative model uses a sequential process to predict structure building decisions one by one. More precisely, the decisions are made by three modules called $f_{addnode}$, $f_{addedge}$, and f_{nodes}. The algorithm works as follows:

(1) Sample whether to add a node of a particular type (in the following called "label") or to terminate the algorithm. If a node label is chosen, add a node v with the according label to the graph G.

(2) Sample whether an edge should be added to the newly added node. If not, go to (1).

(3) Sample the target node u for the edge to be added. Add the edge (u, v) to G. Go to (2).

Figure 4.2 illustrates this process. While this base version of the algorithm does not support labeled or backwards edges, it can quite easily be extended to support these functionalities as well, as shown in chapter 6.

After one iteration of this process, a sequence of decisions and consequently a corresponding graph is generated.

4.3 Learning the Graph Generative Model

In order to compute and learn structure building decisions, the model computes a representation $\mathbf{h}_G$ of the graph $G = (V, E)$. For that purpose, every node $v \in V$ in the graph is associated with a node embedding vector $\mathbf{h}_v \in \mathbb{R}^H$, where $H \in \mathbb{N}^+$. These node embeddings are initially computed based on domain-specific inputs $\mathbf{x}_v$ associated with v (e.g., the one-hot encoded node label):

$$\mathbf{h}_v = f_{init}(R_{init}(\mathbf{h}_V, G), \mathbf{x}_v) \tag{4.1}$$

Here, $\mathbf{h}_V$ is the set of all existing node embeddings, $R_{init}(\mathbf{h}_V, G)$ computes a graph representation[1] and f_{init} is an multi layer perceptron (MLP). The use of $R_{init}(\mathbf{h}_V, G)$ as input for the initialization process prevents nodes with the same input features from having the same initial node embeddings.

The node embeddings will then be propagated along the graph to aggregate information on their local neighborhood. Each iteration of this process consists of the following steps:

(1) A "message" vector is computed on each edge.

(2) Every node collects the message vectors of all its incoming edges.

(3) Every node recomputes its own node embedding based on its old representation, the incoming message vectors, and the corresponding feature vectors of the incoming edges.

This process is characterized by the following equations:

$$\mathbf{a}_v = \sum_{u:(u,v)\in E} f_e(\mathbf{h}_u, \mathbf{h}_v, \mathbf{y}_{u,v}), \quad \forall u \in V \tag{4.2}$$

$$\mathbf{h}'_v = f_n(\mathbf{a}_v, \mathbf{h}_v), \quad \forall v \in V \tag{4.3}$$

Here, $f_e(\mathbf{h}_u, \mathbf{h}_v, \mathbf{y}_{u,v})$ computes the message vector from u to v and $\mathbf{y}_{u,v}$ is a feature vector describing the edge (u, v). In the extended model used in later chapters, this vector contains the edge label of (u, v). In the second equation, $\mathbf{h}'_v$ represents the updated node embedding. f_n can be implemented as another MLP, but lends itself to instead be implemented as a RNN core, since it needs to incorporate newly gathered information into the node embedding, while at the same time preserving as much of the known information as possible. In this thesis, we implement f_n as a GRU, as suggested by Li et al [2].

One round of propagation is denoted as $\mathbf{h}'_V = \mathrm{prop}(\mathbf{h}_V, G)$. Multiple rounds of propagation allow for information to travel through increasingly larger parts of the graph. In order to compute $\mathbf{h}_G$, the updated node embeddings $\mathbf{h}_v$ are then mapped to a higher dimensional vector, where f_m is another MLP:

$$\mathbf{h}_v^G = f_m(\mathbf{h}_v) \tag{4.4}$$

This step allows for a higher dimensionality of $\mathbf{h}_G$, which in turn allows for a better representation of the (compared to a node) high amount of information contained in the graph.

The final step in the computation of $\mathbf{h}_G$ is a gated summation over all node embeddings contained in the graph:

$$\mathbf{h}_G = \sum_{v\in V} \mathbf{g}_v^G \odot \mathbf{h}_v^G \tag{4.5}$$

[1] This graph representation is not the same as $\mathbf{h}_G$, but can be calculated identically to equation 4.5. The computation is, however, performed by a different set of MLPs.

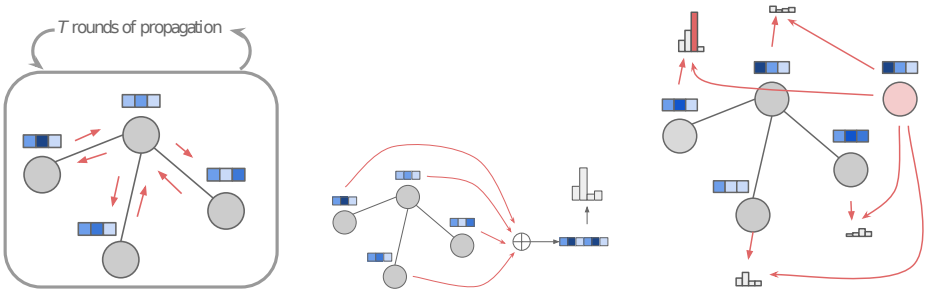


Figure 4.3: Illustration of the graph propagation process (left), graph level predictions using $f_{addnode}$ and $f_{addedge}$ (center), and node selection using the f_{nodes} module (right), as taken from [2]. Used with permission of Dr. Yujia Li.

A separate gating network, which can again be implemented as an MLP, computes $\mathbf{g}_v^G = \sigma(g_m(\mathbf{h}_v))$ for each node, where σ denotes the logistic sigmoid function.

In short, T rounds of propagation (Eq. 4.2 and 4.3) and the subsequent recomputation of the graph representation vector (Eq. 4.4 and 4.5) are denoted as:

$$\mathbf{h}_V^{(T)} = \text{prop}^{(T)}(\mathbf{h}_V, G) \tag{4.6}$$

$$\mathbf{h}_G = R(\mathbf{h}_V^{(T)}, G) \tag{4.7}$$

In order to predict structure building decisions, $\mathbf{h}_G$ can now be used as input to the $f_{addnode}$, $f_{addedge}$ and f_{nodes} modules. Figure 4.3 visualizes the application of these modules during the graph generation process.

4.3.1 Probabilities of Structure Building Decisions

The sequential graph generation process follows the algorithm illustrated in Figure 4.2. Each of the decision steps is modeled by one of the three modules described in this section.

(a) $\mathbf{f_{addnode}}(G)$ This module predicts whether to add a node on the graph level and, if the nodes are labeled, which label the new node should have. This is achieved by predicting a vector of node label probabilities in which one entry indicates the termination of the algorithm. The probabilities are computed as follows:

$$f_{addnode}(G) = \text{softmax}(f_{an}(\mathbf{h}_G)) \tag{4.8}$$

Here, f_{an} is an MLP that maps $\mathbf{h}_G$ to the action output space, and the softmax function transforms the output into a vector of probabilities.

(b) $\mathbf{f_{addedge}}(G, v)$ This module predicts the probability of adding an edge to a newly created node v. Since this decision is made on the node level, the node embedding $\mathbf{h}_v$ of v is also given as input:

$$f_{addedge}(G, v) = \sigma(f_{ae}(\mathbf{h}_G, \mathbf{h}_v^{(T)})) \tag{4.9}$$

f_{an} is implemented as an MLP, σ is the logistic sigmoid function and $\mathbf{h}_v^{(T)}$ indicates $\mathbf{h}_v$ after T rounds of propagation.

(c) $\mathbf{f_{nodes}}(G, v)$ This module first predicts a score s_u for each node u in the G. In the next step, the softmax function is applied to the vector of all scores s:

$$s_u = f_s(\mathbf{h}_u^{(T)}, \mathbf{h}_v^{(T)}), \quad \forall u \in V \tag{4.10}$$

$$f_{nodes}(G, v) = \mathrm{softmax}(\mathbf{s}) \tag{4.11}$$

f_s is once more implemented as an MLP. In order to determine the target node for the edge predicted by $f_{addedge}$, the $argmax$ function is then applied to $f_{nodes}(G, v)$.

4.4 Evaluation

Li et al. have evaluated their graph generative model in multiple experiments. This section briefly illustrates the two experiments applicable to unconditional graph generation found in [2], as well as the model's performance in them. Subsequently, we give a general overview of the strengths and weaknesses of the model, as found by Li et al.

4.4.1 Experiments

Generation of Graphs with Certain Topological Properties In the first experiment, the model is trained on three sets of synthetic undirected graphs: Cycles, trees and Barabasi-Albert graphs,[2] each having between 10 and 20 nodes.

The model is compared against a regular LSTM trained on graph generating sequences and an Erdős-Rényi random graph model [20]. Table 4.1 shows the results of this experiment. The graph generative model distinctly outperforms the LSTM and the E-R-Model at generating cycles and trees and is able to achieve a much smaller Kullback-Leibler (KL) divergence than those models.

Molecule Generation In the second experiment, the model is trained for the task of molecule generation using the ChEMBL [21] molecule database. The number of nodes of graphs in the data set is

[2] A Barabasi-Albert model generates scale-free graphs.

Data Set	Graph Model	LSTM	E-R Model
Cycles	84.4%	48.5%	0.0%
Trees	96.6%	30.2%	0.3%
B-A Graphs	0.0013	0.0537	0.3715

Table 4.1: Percentage of valid samples for three models on data sets for cyclic graphs and trees, and the KL-divergence[3]for Barabasi-Albert graphs.

restricted to 20, and the graph model is again compared against an LSTM trained on graph generating sequences. The model is extended to support labeled edges, and the number of propagation steps T is chosen form {1,2}. In this task, on average, the graph model produces 96.75% valid samples, while the LSTM produces 88.3% valid samples.

These experiments show that the graph model is a powerful approach capable of outperforming conventional graph generation models. Nevertheless, there are still a number of challenges facing it, which we discuss in the next section.

4.4.2 Challenges

A primary challenge for the model is the generation of large graphs. Due to the nature of the sequential generation process, such graphs produce long decision sequences. If available, more conventional forms of graph linearization[4] are typically 2-3 times shorter. This is a significant disadvantage for the graph model, since long decision sequences make training more difficult. In this thesis, we try to alleviate this problem by using domain-specific knowledge to combine multiple decision steps and generation process loops into fewer, or even single steps.

Large graphs also pose challenges regarding the scalability of the information flow inside the model. The number of propagation steps T used to propagate information along the edges of the graph is fixed during training. To achieve sufficient information flow in large graphs, high Ts would be required, which in turn would negatively influence training speed and increase the model's memory intensity.

Lastly, the constant addition of new node embeddings $\mathbf{h}_v$ can lead to unstable training, more so than for typical LSTM models. To alleviate this, Li et al. suggest to use a lowered learning rate.

[3]The Kullback-Leibler divergence is a distance measure between probability distributions, in this case between the degree distributions of the Barabasi-Albert graph data set and the graphs generated by the three models.

[4]For example, in the case of molecule graph generation, SMILES strings can be used to specify molecules more efficiently.

4.4.3 Conclusion

The deep generative model proposed by Li et al. has proved to be a powerful and novel approach for the generation of arbitrary graphs. However, while it has shown promise in certain applications, it also faces challenges, particularly at generating large graphs.

It has become clear that, in order to adapt this model for the creation of (potentially large) LLVM code graphs, substantial structural changes to the model are necessary.

5 LLVM

The LLVM[1] project encompasses a variety of modular compiler and toolchain technologies aiming to deliver fast compile times as well as useful debug information for a variety of popular CPU architectures [22]. This chapter gives a brief overview of the LLVM infrastructure and IR. We then analyze how to map the IR to a graph representation usable to train our domain-specific graph generative model.

5.1 Overview

LLVM originated as a research project at the University of Illinois in 2000. Since then, the project has found success in a wide variety of commercial [23] and open-source [24] projects and has been extensively studied and used in academic research [25][26]. LLVM provides components for every part of the compiler toolchain.

Front end Since version 2.6, LLVM provides a native compiler frontend for languages in the C family (C, C++, etc.) through Clang, which was designed to act as a replacement for the GNU Compiler Collection (GCC). Consequently, Clang aims to support a wide range of GCC features and extensions [3]. Frontends have also been written for a variety of other languages, including Go [27], Javascript [28], and Ruby [29].

Optimizer The LLVM Core libraries contain the modular LLVM optimizer and analyzer *opt*, which works atop LLVM IR. Optimizations are implemented as passes that traverse the IR to eighter transform it or collect information usable by subsequent passes. In order to extract the necessary information for a mapping of IR to a graph representation, we implement an optimization pass as well.

Back end The transformation of LLVM IR into target-specific machine code is handled by the LLVM target-independent code generator. It provides a range of reusable components, including target de-

[1] Despite its appearance, the name "LLVM" is not an acronym, but the full name of the project. While it originally stood for "Low-Level Virtual Machine", this acronym has officially been removed to avoid confusion, since LLVM now contains a

scription interfaces and target-independent algorithms for various phases of native code generation (e.g., register allocation). At the time of writing, it supports many mainstream architectures, including x86, x86-64, and ARM, and is extendable to provide support for arbitrary target architectures.

Alongside these components, LLVM also encompasses a native debugger [30] and various other subprojects positioned at various steps of the compile chain [31][32] and outside of it [33].

5.2 Intermediate Representation

The LLVM IR lies at the core of LLVM and is the code representation all compile-time optimizations are performed upon. It is a target-independent, static single assignment (SSA) language; hence, every value is immutable once assigned. Assigned values are stored in virtual registers, which are, unlike physical registers, unlimited in size and amount. The LLVM IR is representable in three different forms: As an in-memory compiler IR, bitcode, and a human-readable assembly language representation. A line of LLVM IR in human-readable form may look like this:

```
%result = fmul float %var, 4.0
```

In this example, the value of the virtual register `%result` is set to the result of the multiplication of the contents of `%var` and the literal value `4.0`. The following sections will discuss how LLVM IR can be mapped to a graph representation.

5.2.1 Vertex Labels

The first task in designing a mapping from LLVM IR to a graph representation lies with determining the set of possible vertex[2] labels. For instruction opcodes, this is straightforward: Every opcode can be mapped to a corresponding node label. The same holds true for most other IR language constructs, such as structs, vectors, and arrays. The first non-trivial case arises with integer types, a typical usage of which may look like this:

```
%result = add i32 4, %var
```

Most languages differentiate between a small number of different integer bit-widths, e.g., 8, 16, 32, and 64 bit. In LLVM IR, however, integers can be of every bit width from 1 to $2^{23} - 1$. To limit the number of

variety of different subprojects.

[2] In the following, the terms “vertex” and “node” will be used interchangeably.

Figure 5.1: Graph representation of an `add` instruction (a) and an ambiguous , and therefore rejected, representation of instructions with variable operand counts (b).

possible node labels, only the nine integer types found in the training data set[3] have been implemented: `i1`, `i2`, `i6`, `i8`, `i16`, `i31`, `i32`, `i33`, and `i64`.

Mappings for all other IR concepts are mostly straightforward; their implementation, as well as the implementation of edge labels, will be discussed in the following sections. A complete list of all 91 implemented node labels is located in Appendix A.1.

5.2.2 Instructions

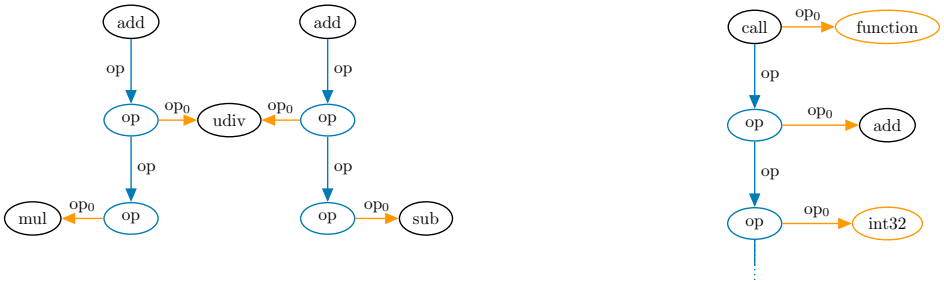
The LLVM IR language reference [34] specifies 64 different instructions opcodes, 47 of which are implemented in this thesis. The remaining 17 opcodes do not appear in the training data set and are, therefore, omitted. For a complete list of the implemented opcodes, please refer to Appendix A.2.

With the exception of `unreachable`, all instructions perform calculations on several constant or variable (i.e., previously calculated) operands. We map constant values to nodes (or in the case of complex values, to subgraphs) labeled with the type of the corresponding constant. Literals are additionally labeled with their normalized[4] value. Figure 5.1a illustrates the usage of both constant and variable operands.

Fixed operand instructions Of the 47 implemented opcodes, 43 have a fixed number of operands between zero and four, which allows for a straightforward mapping of operand order to the graph representation. We connect each operand to its instruction node with an edge labeled according to the operands position, e.g., the second operand of an `add` instruction is connected with an edge labeled "op_1".

[3] Section 7.1 discusses our training set in detail.

[4] The details of this normalization are discussed in section 6.3.2.



(a) (b)

Figure 5.2: Graph representation of the usage of operand nodes (a) and a `call` instruction (b). The node label "operand" is abbreviated with "op".

Variable operand instructions The `phi`, `switch`, `call`, and `getelementptr` instructions differ from other opcodes in that they possess a variable number of operands. Directly mapping operand order to edge labels is impractical for these instructions, as it would introduce a potentially unlimited number of new labels.

As a first approach, we tried directly linking the operands in order of appearance. This, however, leads to ambiguities when multiple instructions perform calculations on the same virtual registers. Figure 5.1b illustrates this problem with two `add`[5] instructions. IR that produces such a graph may look like this:

```
%add1 = add i32 %udiv0, %mul0
%add2 = add i32 %udiv0, %sub0
```

The code establishes `%mul0` and `%sub0` as the second operands of `%add1` and `%add2`, respectively. However, this information is lost in the graph representation, as it is no longer obvious which operand belongs to which `add` instruction. The solution lies in the introduction of a new node and edge label, which we denote as "operand" node and "operand" edge. For the first operand, we connect a node with this label to the instruction; for each subsequent operand, we instead attach such a node to the last operand node in the chain. Additionally, we connect each operand node to the corresponding variable operand with an "op_0" edge. Figure 5.2a shows the code mapped to a representation using these labels.

The opcodes `call`, `switch`, and `phi` also posses a default operand. A `call` instruction, for example, is always linked to a node representing the called function. Since they are present in all instruction instances, we connect default operands directly to instructions with an "op_0" edge, as shown in Figure 5.2b.

[5] `add` instructions have a fixed number of operands; however, their usage simplifies the explanation of this problem. In the actual implementation of the graph mapping, `add` instructions are connected directly to their operands with edges labeled "op_0" and "op_1", as shown in Figure 5.1a.

Figure 5.3: Graph representation of control-flow transition for non-diverting instructions (a) and `br` instructions (b).

5.2.3 Control-Flow

Order of execution is an IR concept that we map to the graph representation through the usage of control-flow edges. For most instruction types, this is achieved by directly linking them to each other in order of appearance, as shown in Figure 5.3a. We represent the entry-point of an LLVM function in a similar way, by creating a "function" node with an outgoing control-flow edge to the first instruction of the function. Two opcodes, though, require a different approach.

Branch instructions Depending on whether a branch is conditional, it can transfer the control-flow to one or two locations in the program; we represent these with the instruction nodes that execute immediately after the jump. The following IR implements a conditional jump:

```
br i1 %icmp0, label %7, label %8
%add0 = add i32 5, 3        ; <label>:7:
...
store i32 2 i32* %alloca0   ; <label>:8:
```

Contrary to many other languages, LLVM declares jump labels implicitly; they appear on the leader of each basic block. Basic blocks themselves are declared implicitly as well; they begin at the first instruction that obtains the control-flow after a so-called "terminator instruction" and extend until the next terminator instruction. Out of the implemented opcodes, `return`, `br`, and `switch` constitute terminator instructions.

Since `br` instructions may have multiple outgoing control-flow edges whose order carries structural information, it is not sufficient to label both of them in the same way. Consequently, we introduce the "cf_1" label for control-flow edges, which enables us to map the order of branch targets to our graph representation, as depicted in Figure 5.3b.

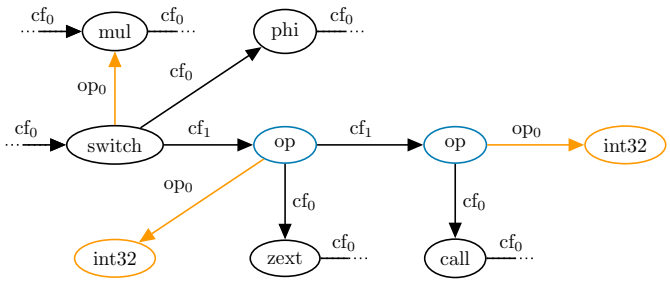


Figure 5.4: Graph representation of a `switch` instruction with two case values.

Switch instructions Whereas a `br` statement allows for a maximum of two jump targets, `switch` instructions can transfer the program's control-flow to one of many possible locations. They are, as mentioned in section 5.2.2, variable operand instructions: Additionally to a comparison variable, they may possess an arbitrary amount of value-label operand pairs. We map these pairs to operand nodes connected with "cf_1" edges. Each operand node is then linked to the value node and the jump target with edges labeled "op_0" and "cf_0" respectively. Figure 5.4 shows the graph presentation of a `switch` statement with a default jump target and two case values.

5.2.4 Types

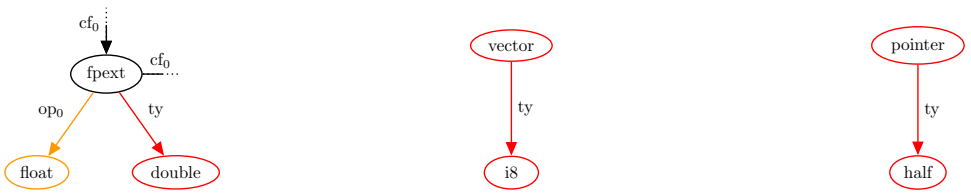
Every global variable, function argument, and instruction result in LLVM has a type associated with it. In the human-readable representation of the IR, these types are explicitly declared in front of the variable or instruction opcode. To keep graph sizes manageable, we omit type nodes wherever doing so does not introduce ambiguities. For most instructions, this is easily achieved, as their result type is a function of the type and value of their operands, and can therefore be deduced from information already present in the graph. As an example, let us consider the following `extractelement` instruction:

```
%result = extractelement <4 x i32> %vec, i32 0
```

The first operand of an `extractelement` instruction is required to be a vector, and its element type constitutes the instruction's result type. In this case, the result type is `i32`.

Implicitly defining types is, however, not possible for function arguments, global variables, and certain instructions[6]. In the following part of this section, we therefore discuss how to represent LLVM types as subgraphs.

[6]The following opcodes require explicit type definitions: `trunc`, `zext`, `bitcast`, `sext`, `alloca`, `sitofp`, `fptoui`, `fpext`, `fptrunc`, `fptosi`, `uitofp`, `ptrtoint`, and `inttoptr`.



(a) (b) (c)

Figure 5.5: Graph representation of an `fpext` instruction (a), the vector type[7] `<8 x i8>` (b), and the pointer type `half*` (c). The edge label "type" is abbreviated with "ty".

Integer and floating-point types As mentioned in section 5.2.2, integer and floating-point literals can be mapped to a single node labeled with their type. The same holds true when representing them as types. With the introduction of the edge label "type", which allows us to distinguish types from constant values, we are able to map simple types to the graph representation. Figure 5.5a showcases this based on a `fpext` instruction.

Vectors, arrays, and pointers Both vectors and arrays are data structures which hold constant values. However, whereas vectors may only contain primitive data types such as `i32` or `float`, arrays may consist of elements of any type with non-zero size. Albeit they are semantically distinct, we represent them both with a root node connected to their element type with a type edge, as depicted in Figure 5.5b.

We employ the same approach to represent LLVM pointers. (Figure 5.5c).

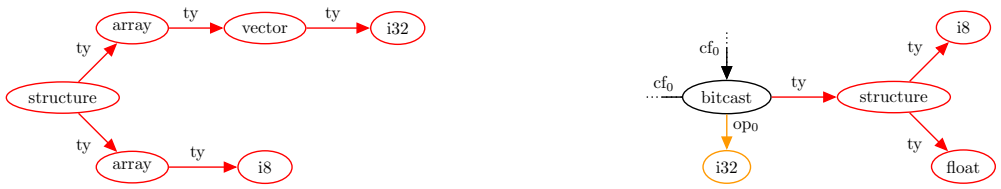
Structures To allow for collections of data types in memory, LLVM incorporates the `structure` type. In our graph representation, we differentiate between the declaration of a structure and its usage as a type or constant value. The IR for a structure declaration may look like this:

```
%struct = type { [3 x <16 x i32>], [64 x i8] }
```

Here, two subtypes are declared: an array of vectors of `i32`, and another array of `i8`. We map this to a `structure` node that, through the usage of type edges, acts as a central hub for all subtypes, as depicted in Figure 5.6a.

In order to represent instances of such a structure, we directly connect them to the corresponding declaration with a "ty" edge (Figure 5.6b).

[7] Although the size `8` of the vector is not depicted in this graphical representation, during training, we give this information to the model as part of the domain-specific input vector $\mathbf{x}_v$, as discussed in section 4.3.



(a) (b)

Figure 5.6: Graph representation of structure declaration (a) and a `bitcast` instruction utilizing a previously declared structure as target type (b).

5.2.5 Constant Values

Instructions can operate on both variable and constant operands, as established in section 5.2.2. In this section, we examine how to map the latter to subgraphs representing the corresponding value.

Integer and floating-point values We map elementary literals to single nodes labeled with the respective type. Their value is given to the generative model as part of the input vector $\mathbf{x}_v$. For details on this process, please refer to section 4.3.

Null pointers and undefined or zero-initialized values LLVM provides the keywords `null`, `undef`, and `zeroinitializer` to initialize constant values in different ways:

- `null` is reserved for pointer values and associates them with no specific address.
- `undef` indicates an unspecified bit-pattern and is used to initialize arbitrary constants. Typically, this keyword emerges from variable declarations without initialization, e.g., "`int i;`" in C.
- `zeroinitializer` initializes a value of any type, including aggregate values, to zero.

We map constants initialized in one of these three ways to nodes labeled with the corresponding keyword. This representation alone, though, does not establish the constant's type, which we consequently incorporate into the graph as well.

Figure 5.7a illustrates this on the basis of a `mul` instruction with two constant `i32` operands, one of which is initialized with the `undef` keyword.

Vectors and arrays We employ the same approach to represent both constant vectors and constant arrays (hereafter also referred to as lists). As a first idea, we simply mapped them to a root `vector` or `array` node connected to a chain of constant values, as depicted in Figure 5.7b. This approach

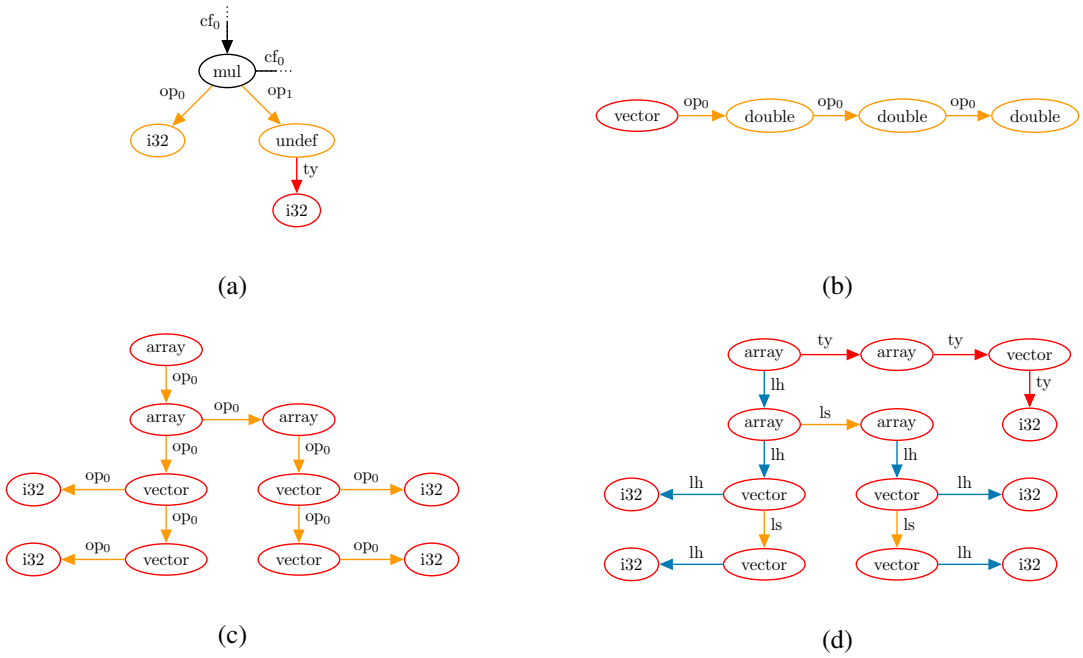
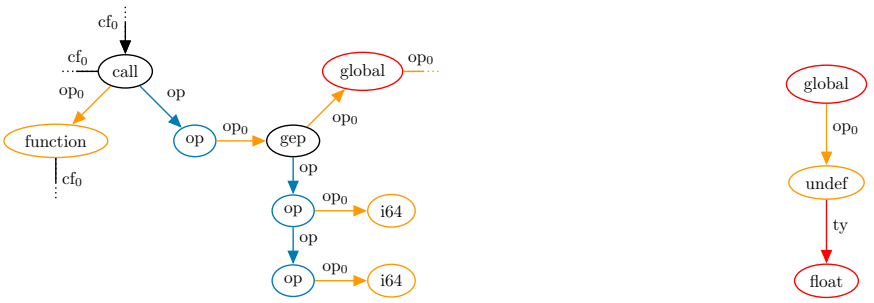


Figure 5.7: Usage of an undefined `i32` constant (a), a rejected graph representation of constant vectors and arrays (b) (c), and the adjusted version of this representation (d). The edge labels "list head" and "list successor" are abbreviated with "lh" and "ls", respectively.

has two problems: Whenever all elements of the chain are initialized by `undef` or a similar keyword, their type is not reconstructable from the graph. Simply creating type nodes for every value that is initialized in this way is not preferable either, as it would generate redundancies and increase graph sizes substantially. Furthermore, this representation is ineffective when applied to nested arrays/vectors, such as the in Figure 5.7c depicted `[2 x [2 x <1 x i32>]]`. Even though the graph produced by this mapping still retains all necessary information, it fails at efficiently representing the relationships between nodes. Especially larger subgraphs would require many rounds of propagation to sufficiently contain the structural information of the nested list inside their node embeddings.

Therefore, we adjust the mapping in two ways: To resolve the problem of missing types, we associate the first node of a constant list with a subgraph representing its type. Additionally, we introduce two new edge labels, "list head" and "list successor" to more effectively represent nested lists. The former of these labels links the root node with the first element, whereas the latter connects all subsequent elements. Figure 5.7d incorporates both of these adjustments.

Global variables Unlike most other languages, LLVM requires global variables to be declared as constant values. We map them to our graph representation by creating a "global" node which we connect to the respective constant value with an edge labeled "op_0", as depicted in Figure 5.8b.



(a) (b)

Figure 5.8: Graph representation of a constant `getelementptr` instruction used as operand of a `call` instruction (a), and the global variable `@A.j = global float undef` (b). The node label "getelementptr" is abbreviated with "gep".

Constant expressions LLVM supports a range of constant expressions, which are instructions operating on constant values that themselves are treated as constant values. They function in the same way as their non-constant counterparts, e.g., a constant `getelementptr` operation may look like this:

```
%result = call i32 @A(i1* getelementptr ([5 x i1], [5 x i1]* @glo, i64 0, i64 0))
```

In this example, an `i1*` pointer is extracted from the global variable `@glo` of type `[5 x i1]*`. We represent constant expressions in the same way we represent normal instructions, and connect them to their parent instruction with the appropriate operand edge. Figure 5.8a depicts the `getelementptr` instruction shown above, as well as a `call` instruction that operates on it.

5.2.6 Unsupported Features

In order to correctly synthesize LLVM programs, it is necessary to transfer them from one of their original representations (e.g., human-readable IR) to our graph representation and vice versa. Ideally, this transition should be achievable without information loss, e.g., the following should hold true:

$$x = f^{-1}(f(x)), \quad \forall x \in X \tag{5.1}$$

Here, X is the set of all possible LLVM programs, and f is the transformation-function to our graph representation. As established in section 5.2.2, we only implemented the LLVM features exhibited by the kernels in our data set. For example, we forgo the implementation of functions with a variable number of arguments, since no kernel utilizes this IR feature. However, even for the kernels in the data set, we are unable to satisfy equation 5.1, as some IR features are not carried over to the graph representation.

For an example of the information loss occurring when translating an LLVM program to our graph representation and back to IR, please refer to Appendix A.3.

To explain why we omit some features of the LLVM IR, we first need to reiterate that one of the main goals of our graph representation is to keep graph sizes as small as possible. Consequently, instead of trying to satisfy equation 5.1, we omit all features that are not required to satisfy the following equation, where $\equiv$ denotes equivalency and Y is the CLgen data set [1]:

$$y \equiv f^{-1}(f(y)), \quad \forall y \in Y \tag{5.2}$$

This approach requires us to define equivalency between LLVM programs. Fortunately, LLVM provides the tool *llvm-diff*, that compares the structure of two LLVM modules and ignores "[i]nsignificant differences, such as changes in the ordering of globals or in the names of local values" [35]. With this, we define two LLVM programs `llvm_0` and `llvm_1` to be equivalent, when the bash command

```
llvm-diff llvm_0 llvm_1
```

produces no output, provided the LLVM environment is installed correctly on the test computer. Using this method, we are able to filter out a sizeable amount of information, that, while being helpful for the optimizer, is not required for the compilation of an LLVM program.

The filtered information can be divided into two main groups: target data and attributes. The former contains information about the target architecture; the latter acts as additional information for optimization passes or the code generator. Both are unrelated to the semantics of the program itself, and can, therefore, be omitted for our purposes.

6 Adapting the Generative Graph Model

In order to synthesize valid LLVM IR through the approach described by Li et al. [2], substantial structural changes to the model are required, as established in section 4.4. In this chapter, we discuss these changes in detail. First, we evaluate which parts of the model need to be modified, before giving a high-level overview of our extended model and graph generation algorithm. In the last part of the chapter, we discuss mathematical foundations and implementation details of our approach and give an overview of the baseline model we use for evaluation purposes in chapter 8.

6.1 Extenstion Points of the Base Model

To briefly reiterate, the structure of the generative graph model can roughly be divided into two phases:

1. Information propagation between nodes, followed by the computation of the node embeddings $\mathbf{h}_V$ and the graph representation $\mathbf{h}_G$.
2. Prediction of structure building decisions using $\mathbf{h}_V$ and $\mathbf{h}_G$.

To accommodate the domain-specific properties of LLVM graphs, we focus on the parts of the model that perform the steps described in phase 2, while leaving the parts involved in phase 1 unchanged[1]. More precisely, we modify the modules $f_{addnode}$, $f_{addedge}$, and f_{nodes} and introduce new modules where necessary.

Modification of existing modules We modify the first of the three base modules, $f_{addnode}$, in order to account for the high number of node labels in our graph representation. More concretely, we split it into the submodules $f_{addinstruction}$, $f_{addtype}$, $f_{addconstant}$ and $f_{predicate}$. These four modules are structurally similar to $f_{addnode}$; however, each of them only predicts a subset of all node labels. With this approach, we separate the creation of instruction nodes, type nodes, constant nodes, and predicates[2] on a structural level, which allows us to devise a more robust graph generation algorithm.

[1] With the exception of extending the input of R_{init} by a value representing type sizes and the numerical value of constants, as discussed in sections 5.2.4 and 5.2.5, respectively.

[2] Predicates specify how `icmp` and `fcmp` instr. compare their operands, e.g., the predicate `eq` indicates a test for equality.

Since we design our algorithm specifically for the construction of LLVM graphs, we are able to make strong assumptions about the structure of certain parts of the graph. This allows us to map multiple actions to one decision step, or even to forgo certain decisions entierly. The "add edge" decision step is most affected by this; with one exception, we are able to derive the decision whether to add edges to a newly created node from the current state of the graph. Consequently, we rarely use $f_{addedge}$ in our model. However, since $f_{addedge}$ predicts binary node level (BNL) decisions, we reuse its structure in a range of new modules that we discuss below.

The function of the last module, f_{nodes}, can be achieved more efficiently by domain-specific modules, which is why we omit it entirely. However, similarly to $f_{addedge}$, we construct new modules by reusing its internal structure.

Introduction of new modules Unlike Li et al. [2], who designed their model around arbitrary graphs, our narrowly defined field of application allows us to incorporate domain-specific modules into the graph generation process. With $f_{addinstruction}$, $f_{addtype}$, $f_{addconstant}$, and $f_{predicate}$, we already discussed four such modules. In order to adapt the model to the IR's structure, we introduce three additional modules that predict LLVM-specific structural decisions on the graph level: $f_{addglobal}$, $f_{addstructure}$ and $f_{addfunction}$. These modules predict whether to add additional global variables, structures, or functions to the graph, respectively. Whenever one of them produces a positive output (i.e., "true"), a subroutine in the graph generation algorithm creates the corresponding graph structure.

As mentioned above, we also introduce several modules that are derived from $f_{addedge}$. Concretely, these modules are $f_{operand}$, $f_{constant}$, and $f_{branchedge}$. Each of them predicts binary decisions on the node level. The $f_{operand}$ module decides whether to add a new operand to a variable operand instruction, whereas $f_{constant}$ predicts if an operand should be a constant value or a variable. The last of these modules, $f_{branchedge}$, computes whether a control-flow edge added to a `br` instruction should be labeled "cf_0" or "cf_1". Explicitly deciding the branch edge order in this way allows for branches where the "cf_0" edge connects to instructions that are generated at a later point in time than the instruction connected with a "cf_1" edge.

Additionally, we substitute the general-purpose module f_{nodes}, which in the base model predicts edge-targets, with three domain-specific modules that each fulfills a more specific task:

- The selection of control-flow edge- and operand edge-targets is handled by $f_{instructionedge}$.
- Whenever the label "structure" is chosen during type creation, f_{struct} predicts which structure declaration is chosen.

- Lastly, whenever a `call` instruction calls a locally defined function, $f_{localfunction}$ selects the function node.

The modules discussed up until now perform tasks that, in principle, are achievable through the base model as well, though in a less efficient way. However, the generation of LLVM graphs requires functionality that is not provided by the base model at all. More precisely, we require the ability to predict numerical values as well as externally defined OpenCL and LLVM functions.

In order to address the first shortcoming, we introduce two new modules f_{number} and $f_{typenumber}$. Whereas the former calculates numerical values of integer and floating-point constants, we utilize the latter for the prediction of integer literals in types, e.g., the `8` in `<8 x i32>`. This distinction is necessitated by the discrepancy between the value ranges of constant literals and type-sizes; e.g., whereas constants may represent almost any value, types only contain (typically small) non-negative integers.

We overcome the second shortcoming through the introduction of the module $f_{externalfunction}$. To explain why the prediction of externally defined OpenCL and LLVM functions is necessary, we have to examine the `call` instruction, which can call both locally and externally defined functions. In the case of local functions, the selection of callable functions is limited to all functions that have been defined in the current graph. This is not the case for external functions, which is why we require a separate module for them. When a call to an external function is about to be generated, $f_{externalfunction}$ selects a function out of 920 LLVM and OpenCL functions present in our training set.

6.2 Graph Generation Process

Our modified model sacrifices the simplicity of Li et al.'s general-purpose approach to more effectively synthesize LLVM graphs. Consequently, our graph generation algorithm is far more complex than the one of the base model (Figure 4.2). In this section, we introduce the concept of "active nodes", discuss each subroutine of the generation process, and conclude with an analysis of the main algorithm.

6.2.1 Active Nodes

In the base model, any node level action, e.g., any action that takes a node embedding $\mathbf{h}_v$ with $v \in V$ as input, performs its calculations on the node that was added last to the graph. Our model, in contrast, often "returns" to a node after performing several actions not directly related to that node. For example, we may generate an "add" node u, construct a subtree representing its first operand, and then create an edge (u, w) to a previous instruction w that represents u's second operand. Because of this, we introduce

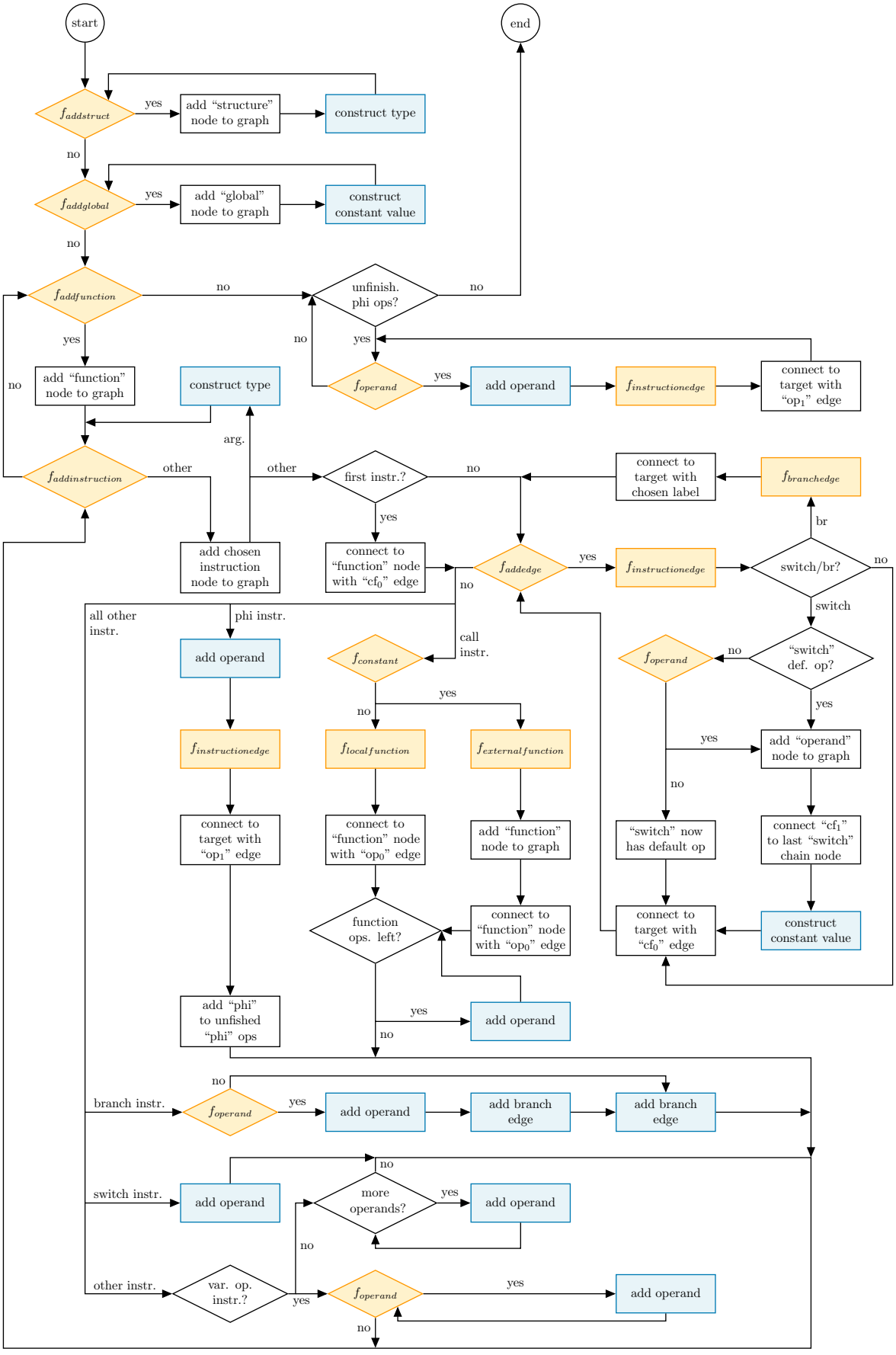


Figure 6.1: Flowchart of our modified graph generation process. Blue elements signify subroutines.

the concept of an active node, which represents the node v whose embedding $\mathbf{h}_v$ will be used whenever node level actions are performed. During graph generation, we change the active node to the last added node whenever a node is generated, as well as to other nodes whenever necessary. In the example above, we set u as the active node after the creation of the first operand.

6.2.2 Subroutines

For clarity, we abbreviated multiple procedures in Figure 6.1 with blue boxes. In order to comprehensibly discuss our modified graph generation algorithm in the next section, we first examine each of those subroutines, as well as the modules they use.

Construct type subroutine The generation of type-subgraphs is required numerous times during graph synthetisation; Figure 6.2 depicts a flowchart of this process. We differentiate between structure declarations and other type declarations.

For structure declarations, the process is straightforward: The $f_{operand}$ module predicts whether to add another subtype to the structure, which is, in the case of a positive outcome, constructed trough recursively executing the "construct type" routine (When entering "construct type" in this way, the subprocess is not considered to be a structure declaration). The inquiry of $f_{operand}$ and the subsequent type generation continues until $f_{operand}$ produces a negative outcome. This allows for empty structure declarations, which are valid in LLVM IR.

For non-structure types, we first predict one of 17 possible type labels through $f_{addtype}$. Depending on the result, we construct the type in different ways: If "structure instance" is chosen, we select a structure declaration node through f_{struct}. In the next step, we create a type edge from the start node (the node that was active before the "construct type" routine was entered) to the chosen structure node. In the

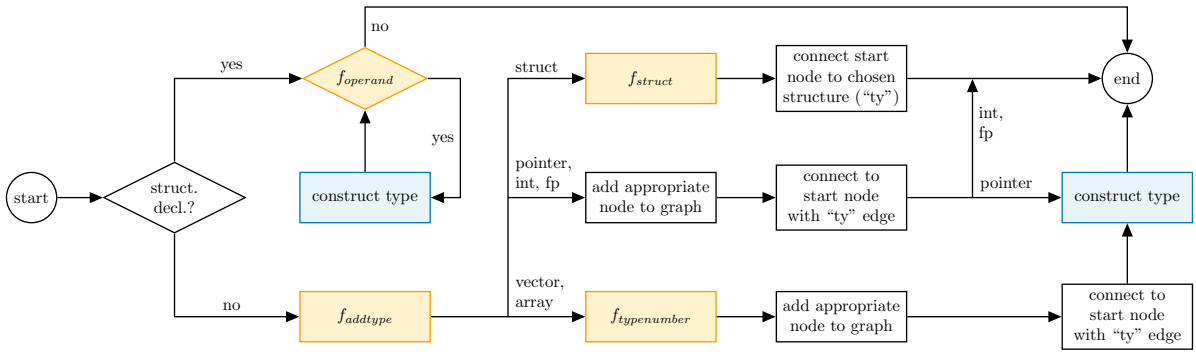


Figure 6.2: Flowchart of the "construct type" subroutine referenced in Figure 6.1.

following, we call the node which is active when a subroutine is entered "start node" and denote it as v_s.

In the case $f_{addtype}$ predicts one of the two labels "vector" or "array", we first predict the size of the list using $f_{typenumber}$. We then create an appropriately labeled node v, as well as the type edge (v_s, v). The previously predicted integer value is used as input for R_{init} during the initialization of v (please refer to section 4.3 for details). In the last step, we construct the subtype of the list by recursively executing the "construct type" routine.

Two additional cases may occur: Eighter $f_{addtype}$ predictes the "pointer" label or one of 12 different floating-point or integer labels. In both cases, we first create an appropriately labeled node v and connect the starting node to v with the type edge (v_s, v). Additionally, if "pointer" is the selected label, we create the pointer's subtype by recursively executing the "construct type" routine.

Construct constant value subroutine Similarly to types, constant values may be constructed at multiple stages of the graph generation process. When this subroutine is entered, the parent step passes along an edge label k, which we use to label the edge connecting the start node v_s to the first node generated by this routine. The value of k depends on the context in which the routine is entered, e.g., when generating the first operand of an add instruction, we pass along the label "op$_0$". The first step of the generation process consists of predicting one of 23 possible node labels through the $f_{addconstant}$ module. Depending on the chosen node label, we generate the constant value in different ways.

In the case that one of the labels "undef", "null" or "zeroinitializer" is selected, we add a node v with the corresponding label to the graph and connect it to the start node with an edge (v_s, v) labeled k. Since this node contains no type information, we additionally execute the "construct type" routine.

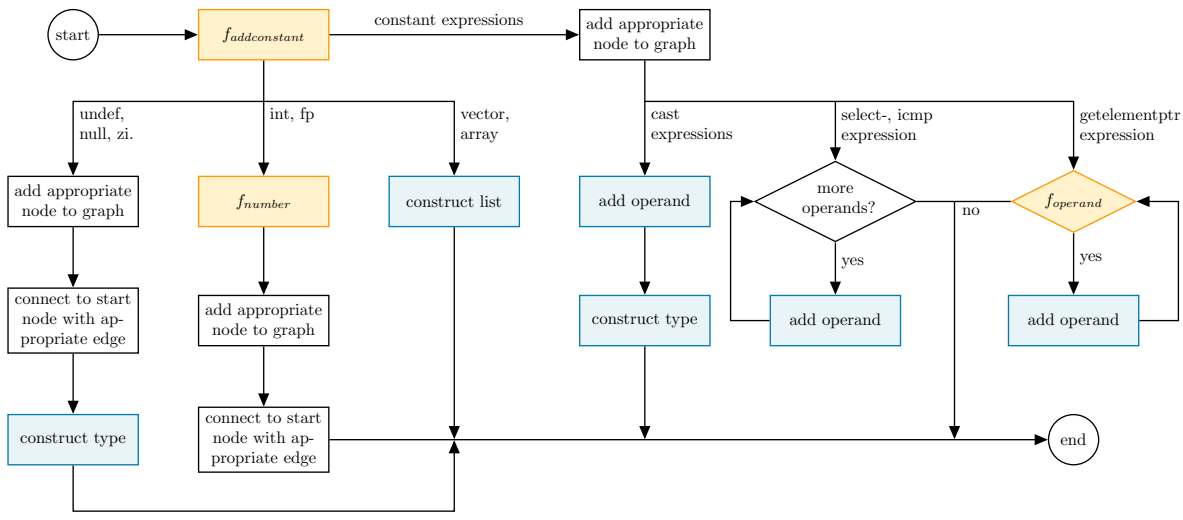


Figure 6.3: Flowchart of the "construct constant value" subroutine referenced in Figure 6.1.

If instead, one of 12 integer or floating-point labels is selected, we predict its numerical value with the f_{number} module. Subsequently, we once more create a node v with the respective label and connect it to the start node v_s with an edge (v_s, v) labeled k. The predicted numerical value is given to the model as input to R_{init}.

Since the generation of constant vectors and arrays is somewhat complex, we execute a separate "construct list" routine whenever one of the two associated labels is chosen. When entering the routine, we pass along k (we discuss the reason for this below). The remaining six labels predicable by $f_{addconstant}$ all represent constant expressions. Three of them, "bitcast", "icmp", and "inttoptr", are structurally similar and can, therefore, be generated in the same way. If one of these labels is selected, we create a node labeled with it and execute the "add operand" routine a number of times equal to the number of the expression's operands.

Out of the three remaining labels representing constant expressions, two have fixed operands ("select", "icmp") and one has variable operands ("getelementptr"). We construct them by generating as many operands as required by their opcode and as predicted by $f_{operand}$, respectively.

Construct list subroutine As discussed in section 5.2.5, lists may be nested, which makes their generation process somewhat complex compared to other constant values. Fortunately, we are able to once more apply the same approach for the generation of both constant vectors and arrays. When calling the "construct list" routine, the caller (either "construct constant value" or "construct list" itself) passes along an edge label k, which is used to connect the first generated node to the start node.

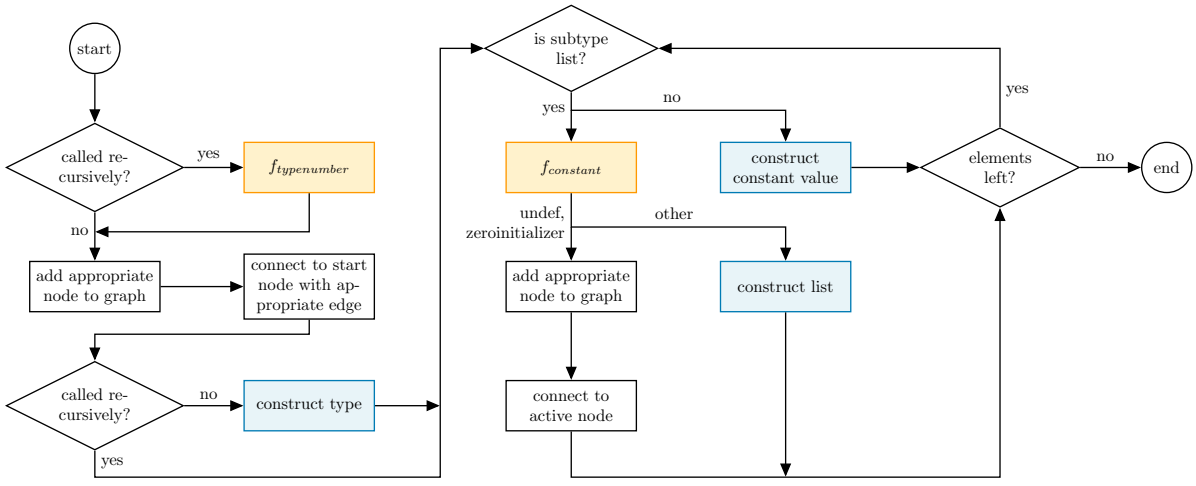


Figure 6.4: Flowchart of the "construct list" subroutine referenced in Figure 6.1.

Upon entering the routine, we first check whether it was called recursively, indicating the generation of a nested list. If this is not the case, we determine the list size through the $f_{typenumber}$ module. If the routine was called recursively, this step is not necesessary since all type information is provided by the caller (we discuss this in detail below). Following this step, we create a "vector" or "array" node v, and, if computed, pass the list size to R_{init}. We connect the generated node to the start node v_s through an edge (v_s, v) with the label k. If "construct list" is not called recursively, we subsequently generate the lists element type by executing the "construct type" routine. This concludes the construction of the list base; the next task is the generation of the list's elements.

To distinguish between simple and nested lists, we examine the previously calculated element type (if the routine is called recursively, the caller passes this type information to the callee). If the element type is neither a "vector" nor an "array" type, we construct the list element by executing the "construct constant value" routine, passing along the edge label "list head" as argument (For details on why this edge label is necessary, please refer to section 5.2.5). If, however, the element type is a list type itself, we predict whether the corresponding constant value is undefined or zero-initialized through the $f_{constant}$ module. To accomplish this, we first tried to apply a mask to $f_{constant}$ that nullifies all outputs not representing either "undef", "zeroinitializer", "vector" or "array". This approach led to an overrepresentation of undefined/zero-initialized list elements in synthesized graphs when compared to the graphs in the training data set. Therefore, we chose to instead check specifically for "undef" and "zeroinitializer", and treat all other outputs as an indication for a regularly initialized constant value. With this method, the model is able to learn the probability of undefined/zero-initialized list elements that exists within the training set. If either "undef" or "zeroinitializer" is selected by $f_{constant}$, we add a corresponding node v to the graph and connect it to the base node v_b with a "list head" edge (v_b, v). Since the lists global type information is already contained in the graph, we are able to forgo the local type generation of v.

If neither "undef" nor "zeroinitializer" is selected, we generate the list element by recursively executing the "construct list" routine. We pass two arguments to the routine: First, the edge label "list head", which is used to connect the resulting list to the current list base node. Additionally, we pass along the subtype of the previously predicted element type (e.g., if we earlier predicted the type `[3 x <16 x i32>]`, we pass the type `<16 x i32>`), thereby allowing the routine to use this information for the process described in the previous paragraph. With this, one list element is generated; we now set the node which was added to it first as the active node. Depending on whether the routine is executed normally or recursively, we determine the remaining number of elements to be generated with the list size predicted by $f_{typenumber}$ or by the type information provided by the caller, respectively. However, for all subsequent elements, we substitute all uses of the edge label "list head" with "list successor".

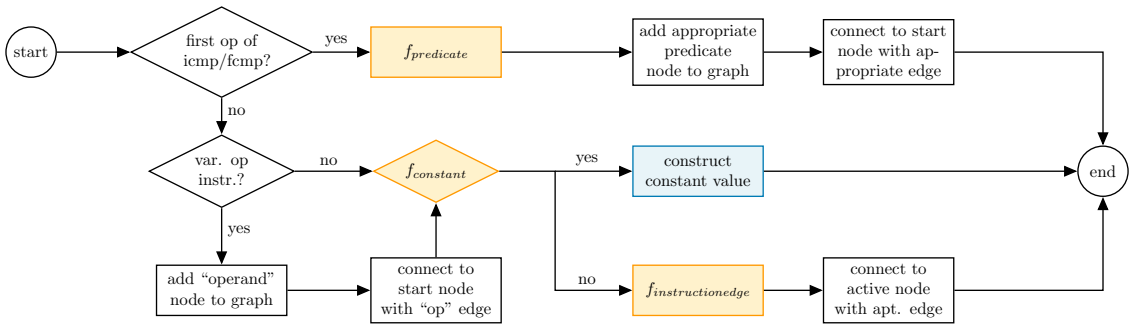


Figure 6.5: Flowchart of the "add operand" subroutine referenced in Figure 6.1.

Add operand subroutine This subroutine handles the generation of instructions operands, including constant instructions, i.e., constant expressions. When it is executed, the caller passes along the parent instructions opcode as well as the edge label k, which we use to connect the resulting operand to said parent instruction. Additionally, a predefined LLVM type t may be passed by the caller. We explain the reason for this below.

We first address a special case: If the routine is executed in order to generate the first operand of an `icmp` or `fcmp` instruction, the result is required to be a predicate. We, therefore, select one of 17 possible predicates through the $f_{predicate}$, add an accordingly labeled node v to the graph, and connect it to the parent instruction (i.e., the start node) v_s through an edge (v_s, v) that we label k.

For all other argument pairs, we first check whether the passed opcode belongs to a variable operand instruction. If this is the case, we create an "operand" node and connect it with an incoming "op" edge to eighter the parent instruction, or if it already has at least one operand, to the last node in the chain of "operand" nodes connected to the parent instruction. The procedure then converges, making the following steps identical for both variable and fixed operand instructions. We proceed by predicting whether the resulting operand is constant through the $f_{constant}$ module. If this produces a positive result, we handle the operand generation through the "construct constant value" routine, passing k as argument. A negative result from $f_{constant}$, however, indicates a variable operand. In this case, we select a node v with the $f_{instructionedge}$ module and connect it to the active node v_a (either the parent instruction or the added "operand" node) with an edge (v_a, v) labeled k. If an LLVM type t is passed to "add operand" as argument, we limit the possible results of $f_{instructionedge}$ to instruction nodes with the same associated type. We use this during the creation of a `call` statement's function operands, where the function's signature predetermines the type of each operand.

Add branch edge subroutine The last subroutine we introduce concerns control-flow edges originating from `br` instructions; more specifically, we use it to predict jump targets as well as jump order

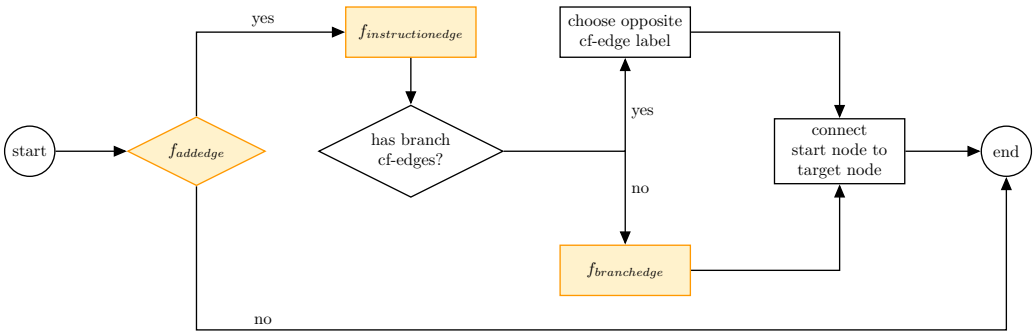


Figure 6.6: Flowchart of the "add branch edge" subroutine referenced in Figure 6.1.

for newly generated nodes representing `br` statements. This process is comparatively simple: First, we utilize $f_{addedge}$ to predict whether to add an edge at all. If this produces a negative result, the routine terminates. We allow for this possibility, since several potential jump targets of `br` instructions may not exist in the graph at the time of their generation. Therefore, we allow this routine to leave the graph unaltered, effectively reducing the space of potential jump targets to nodes generated at a later point in time. Next, we choose a target node v_i through $f_{instructionedge}$. After this, we check whether the `br` instruction in question already has outgoing control-flow edges. If this is the case, we add an edge (v_s, v_i) to the graph, where v_s is the start node as well as the node representing the `br` instruction. We set the label of this edge to be the one in $\{cf_0, cf_1\}$ that is not used by the already existing control-flow edge originating from v_s.

If, however, v_s has no outgoing control-flow edges, we choose an edge label k through $f_{brachedge}$. Subsequently, we once more add an edge (v_s, v_i) to the graph and label it with k.

6.2.3 Main Algorithm

After discussing the subroutines of our modified graph generation algorithm, we now examine its high-level structure. In this section, we will give a brief overview of the main algorithm, followed by a more thorough examination of each part in order of its occurrence during graph generation.

Overview We begin the graph synthetization by generating structures and global variables. Next, we predict whether to add an LLVM function to the graph. If no function is created, we complete all previously created "phi" instructions (we discuss the details of this below) and terminate the algorithm.

If instead, a new function is created, we add a "function" node v_f to the graph and begin with the generation of an instruction. In this process, we first select a label with the $f_{addinstruction}$ module. If "none" is chosen, the algorithm returns to the prediction of new functions. If any other label is chosen, we

add a node v_i with the according label to the graph. We predict both function arguments and instruction opcodes with $f_{addinstruction}$. If the label "argument" is selected, we add an accordingly labeled node to the graph and generate its type. If, however, an opcode is selected, we handle it differently depending on whether it represents the first instruction of the current function. If this is the case, we additionally construct a control-flow edge (v_f, v_i). For all other instructions, we select a predecessor instruction v_{i-1}, and connect it to v_i with a control-flow edge (v_{i-1}, v_i); the exact procedure varies depending on the label of v_{i-1}. It is also possible to predict multiple predecessor instructions, since it is possible for LLVM instructions to be the target of multiple jumps.

The algorithm then generates the instruction's operands. In this step, we handle "call", "phi", "br", and "switch" separately from other instruction opcodes. After the generation of operands is complete, the algorithm returns to the start of the instruction generation process. In the following part of this section, we discuss each step of the algorithm in detail.

Generation of structures and global variables As mentioned above, the first task of synthesizing LLVM graphs lies in the generation of structures, followed by global variables. We construct them in this order, since it is possible for global variables to be instances of previously defined structures. In both cases, a node with the corresponding label is added to the graph, followed by the generation of a type or constant value, respectively.

Adding functions and instructions Next, we predict whether to add a new LLVM function to the graph through the $f_{addfunction}$ module. If this produces a positive result, we add a "function" node to the graph. The steps taken if no function should be added (concerning the completion of "phi" instructions) are discussed later.

We now begin with the generation of the new function's instructions. For this purpose, we select one of 49 possible node labels through the $f_{addinstruction}$ module. If "none" is selected, we consider the function complete and go back to predicting whether to add another one with $f_{addfunction}$. However, in order to prevent the premature termination of the instruction generation procedure, we apply a mask to the output of $f_{addinstruction}$ that nullifies the probability of selecting "none" if not all basic blocks are complete, e.g., end with a terminator instruction (`return`, `br` or `switch`). Since no explicit basic block information is contained in the graph, we compute it separately each time a node is added to the graph. Aside from 47 opcodes and "none", $f_{addinstruction}$ may also select the node label "argument". We include it in this module, since function arguments, similarly to instructions, have associated types and may be used as instruction operands. If this label is selected, we add an "argument" node to the graph and generate its associated type through the "construct type" subroutine.

The majority of $f_{addinstruction}$'s result space, however, represents instruction opcodes. If one of these labels is selected, we add a node v_i with the corresponding label to the graph and connect it to the existing graph through control-flow edges. If the current function does not contain any instructions, we construct a "cf_0" edge originating from the current "function" node v_f to v_i. If it, however, contains at least one instruction, the procedure is considerably more complex. In this case, we first predict whether to add a control-flow edge through $f_{addedge}$. If this produces a positive result, we choose a predecessor instruction v_{i-1} through the $f_{instructionedge}$ module. In order to prevent nodes that have reached their maximum of outgoing control-flow edges from being selected, we only apply $f_{instructionedge}$ to nodes that may still receive such edges. For example, we ignore an `add` instruction with an outgoing "cf_0" edge and a `br` instruction with outgoing "cf_0" and "cf_1" edges, but always predict scores for `switch` instructions, since they may have an arbitrary number of jump targets (greater than zero). Depending on the label of v_{i-1}, we handle edge construction in different ways:

- For most labels, a "cf_0" edge from v_{i-1} to v_i is added to the graph.
- If v_{i-1} represents a `br` instruction, we label the new edge "cf_0" or "cf_1", depending on the prediction of $f_{branchedge}$.
- Lastly, if v_{i-1}'s label is "switch", we first check whether v_{i-1} already has a default jump target. If not, we utilize $f_{operand}$ to predict whether one should be added. If this is the case, we consider v_i to be the default operand of the `switch` instruction represented by v_{i-1}, and add a "cf_0" edge from v_{i-1} to v_i. If, however, v_{i-1} already has a default jump, or $f_{operand}$ predicts that none should be added, we regard v_i as a conditional jump target of v_{i-1}. In this case, we add an "operand" node to the graph and connect it to v_{i-1} (or, if v_{i-1} already has at least on non-default operand, we connect it to the last "operand" node in the chain of nodes with this label originating from v_{i-1}) with an incoming "cf_1" edge. Subsequently, we construct an "cf_0" edge from the "operand" node to v_i. For every conditional jump target, a `switch` instruction also possesses a constant comparison value to check whether this target should be jumped to or not. We generate this value through the "construct constant value" subroutine and pass it the edge label "op_0" as argument.

We repeat the process of adding control-flow edges until $f_{addedge}$ produces a negative result.

Generation of call operands Now that an instruction node v_i has been added to and connected with the graph, the next task lies with the construction of its operands. Here, we differentiate between `call`, `phi`, `br`, `switch` and the remaining opcodes. If v_i represents a `call` statement, we proceed by predicting whether an external function or a local function is called through $f_{constant}$. We use this module since the generation process for an external function very much resembles the creation of a

constant value: First, we predict the specific function through $f_{externalfunction}$. This module selects one of 920 predefined OpenCL and LLVM functions that we extracted from the training set, such as *get_global_id*. It should be noted, that many of these functions are mangled[3] versions of the same base function. We chose to include mangled functions instead of demangled ones, to reduce the number of actions required for the construction of external functions; unmangled functions would require additional actions to specify their signature. After an external function is selected, we add a "function" node v_f to the graph. Additionally, we create an "operand" node for each argument α in the function's signature and connect it to v_f (or the last "operand" node in the chain originating in v_f) with an incoming "op" edge. Thereafter, we construct the corresponding operand with the "add operand" subroutine, which we pass the LLVM type of α as argument.

If $f_{constant}$ instead produces a negative result, we choose a previously generated, local function as the callee of the `call` instruction represented by v_i. We achieve this by selecting a "function" node v_f through the $f_{localfunction}$ module and subsequently adding an edge (v_i, v_f) with the label "op$_0$" to the graph. In the next step, we extract the chosen function's signature from the graph and proceed with it in the same way we did with the external function's signature.

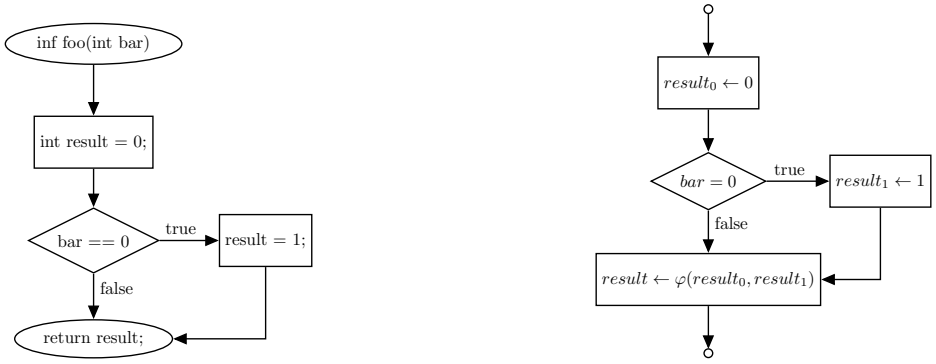
Generation of phi operands If, however, $f_{addinstruction}$ selects the label "phi" for v_i, we proceed by generating exactly one `phi` operand. To explain our reason for doing so, a short examination of SSA languages (to which LLVM belongs) and the usage of φ in SSA graphs is necessary. First, let us clarify what exactly SSA form is:

Definition 6.2.1. *A language that is in static single assignment form requires each variable to be assigned exactly once.*

With this definition in mind, let us examine the following C function:

```
int foo(int bar){
    int result = 0;
    if(bar == 0){
        result = 1;
    }
    return result;
}
```

[3]Name mangling is a process in which information about the signature and namespace of an entity (e.g., a function) is encoded into its identifier. This is done to avoid name clashes of different entities at compilation time. An example applicable to our domain is the LLVM function *round*, which may be mangled to "llvm.round.f32" or "llvm.round.v4f32", depending on



(a) (b)

Figure 6.7: Control-flow graph of the C function depicted on page 45 (a), and its corrected SSA graph representation (b).

Figure 6.7a shows the control-flow graph of this function. As illustrated by the graph, two control-flow paths exist, resulting in two possible values for the variable `result`. Even though such behavior is typical for most languages, it directly contradicts definition 6.2.1, since a value is assigned to `result` more than once. In order to translate this code into an SSA language such as LLVM, a new statement is required, which examines all possible reaching definitions of `result` and creates a new variable containing its actual value during program execution, which depends on the executed control-flow path. The function performing this task is called φ. Figure 6.7b depicts the SSA graph of the code above and contains an application of φ to `result`.

In the graph, no variable is assigned more than once; instead, substitute variables are created in order to adhere to the SSA form. The two possible values of `result` are then unified through the φ function. In LLVM, the functionality of φ is achieved through the `phi` statement. We, therefore, can deduce the following:

1. Whenever a `phi` statement is executed, at least one possible control-flow path leading up to it has been traversed; hence, one possible value of it has been computed.

2. Since the actual value of `phi` during program execution depends on the program's control-flow, the order of its operands has no semantic significance. E.g., the following lines of IR are equivalent:

```
%var2 = phi i64 [ %var0, %label0 ], [ %var1, %label1 ]
%var2 = phi i64 [ %var1, %label1 ], [ %var0, %label0 ]
```

From this follows, that during graph synthetisation, whenever a node v_i representing a `phi` instruction is constructed, one of its operands already exists within the graph. Therefore, we can always predict v_i's

whether it is applied to a floating-point value or a vector of floating-point values.

first operand. Such an operand is of the form `[ %var, %label ]` which consists of two parts: The actual value to be assigned to the `phi` instruction (`%var`), followed by the label of the basic block that needs to be traversed in order for this assignment to be performed (`%label`). Since we do not translate labels to our graph representation, we represent them with an edge to the first instruction after them.

We construct the first part of the `phi` operand by executing the "add operand" subroutine, passing it the edge label "op_0". Unfortunately, even if this results in a variable (i.e., a previously generated instruction node), we cannot deduce the second part of the `phi` operand from it, since both sub-operands may be located in different basic blocks. Consequently, we select a "label" v_l through $f_{instructionedge}$ and add an edge (v_o, v_l) labeled "op_1" to the graph, where v_o represents the "operand" node created by the "add operand" subroutine. Lastly, we add v_i to a set S_{phi}, which contains all unfinished phi instructions.

Generation of branch operands The third instruction requiring a unique approach regarding its operand generation is `br`. LLVM differentiates between unconditional and conditional branches; we predict to which of those two categories a newly generated `br` statement belongs through the $f_{operand}$ module. This approach is possible since a conditional jump has, unlike an unconditional one, a boolean operand that determines which of its two jump targets is selected during program execution. Therefore, if $f_{operand}$ produces a positive result, we know that v_i represents a conditional `br` instruction. We consequently generate v_i's operand through the "add operand" subroutine. Additionally, we execute the "add branch edge" subroutine twice, thereby generating between zero and two jump targets. If $f_{operand}$ instead produces a negative result, we can conclude that v_i represents an unconditional `br` statement; hence, we do not generate an operand, and instead execute "add branch edge" once.

Generation of switch operands Unlike `br` instructions, a `switch` is required to dominate[4] each of its jump targets. Since we generate instruction nodes in a topological ordering, it follows that, at the time of a `switch`'s construction, the graph contains none of its jump targets. However, this is not the case for the comparison value that determines which jump is executed during runtime. Therefore, we construct this value through the "add operand" routine and pass it the edge label "op_0" as argument.

Default operand generation procedure Each of the four opcodes discussed until now requires a unique approach regarding the generation of its operands. For the 43 other opcodes selectable by $f_{addinstruction}$, however, we can apply the same approach, since they are structurally similar. We begin by determining whether the selected label represents a variable operand opcode. If this is the case, we utilize $f_{operand}$ to predict whether to construct a new operand. We then execute the "add operand"

[4] In a control-flow graph, a node v_d dominates a node v_n, if every control-flow path from the entry node to v_n goes through v_d.

subroutine as long as $f_{operand}$ produce a positive result. If the chosen label represents a constant operand opcode, we instead execute the subroutine as long as v_i has fewer operands then specified by its opcode. After eighter $f_{operand}$ produces a negative result or v_i has the number of operands specified by its opcode, the operand generation procedure is completed.

The five separate paths handling the generation of instruction operands now converge again, and the algorithm continues by predicting whether to add a new instruction through the $f_{addinstruction}$ module.

Completion of phi instructions On the previous pages, we discussed our approach for the generation of instruction nodes and their operands. This part of the algorithm repeats, as long as $f_{addinstruction}$ produces a positive result. However, if $f_{addinstruction}$ predicts the label "none", indicating the termination of the instruction generation process, the algorithm returns to the decision step performed by $f_{addfunction}$. We also discussed the actions taken if this module predicts the addition of another function to the graph. If $f_{addfunction}$ instead produces a negative result, no more functions are created, leaving only one remaining task: The completion of previously generated phi instructions.

On page 46, we stated that during the initial operand generation process of a node representing a `phi` instruction, exactly one operand is constructed. As discussed, the reason for this is that at this point, one possible `phi` value is already present in the graph. However, this also means that we cannot guarantee the existence of any other such value at that stage of the graph generation process. At the current stage of the algorithm, however, all functions and instructions have been generated; thus, we are able to finish operand generation for the previously constructed `phi` nodes. For this purpose, we select the earliest generated node v_i from the set S_{phi} of nodes representing unfinished `phi` instructions and set it as the active node. The actual operand generation process is identical to the one for newly created `phi` nodes: We construct the "value part" of the operand with the "add operand" subroutine and the "label part" of it with the $f_{instructionedge}$ module. Lastly, we add an "op$_l$" edge (v_o, v_l) from the "operand" node generated by "add operand" to the chosen "label" v_l and remove v_i from S_{phi}. We repeat this procedure until S_{phi} is empty. After the construction of the remaining `phi` instructions is finished, the algorithm terminates, resulting in one generated LLVM graph.

In this section, we discussed the logical structure of our graph generation algorithm as well as the purpose of the modules comprising it. However, in order to not divert any attention from the algorithm itself, we refrained from examining the mathematical structure of those modules. Now that our process of graph generation has been laid out, we will discuss their architecture in detail.

6.3 Architecture of Structure Building Modules

Our modified graph generative model is comprised of 17 modules predicting structure building decisions. In the previous section, we discussed the semantics of these modules; here, we examine their mathematical structure.

6.3.1 Binary Graph Level Modules

In the graph generation process, it is often necessary to predict binary decisions that are not associated with specific nodes (one example of this would be the decision whether to add another function node to the graph). We categorize modules making such decisions as graph level modules, since they only take the graph representation vector $\mathbf{h}_G$ as input. Out of the 17 implemented modules, $f_{addstruct}$, $f_{addglobal}$ and $f_{addfunction}$ constitute graph level modules. They compute binary decisions in the following way:

$$f_{bglm}(G) = \sigma(f_{bn}(\mathbf{h}_G)) \tag{6.1}$$

Here, f_{bglm} represents each of the three binary graph level (BGL) modules, as they are structurally identical. The MLP f_{bn} computes a binary output from the graph representation vector $\mathbf{h}_G$, which is then normalized by the logistic sigmoid function σ. It should be noted, that $f_{addstruct}$, $f_{addglobal}$ and $f_{addfunction}$ each have a separate f_{bn}. We discuss the exact dimensions of $\mathbf{h}_G$ and f_{bn} in section 7.3.3, which discusses the hyperparameter optimization in our model.

6.3.2 N-ary Graph Level Modules

Apart from the binary modules described in the previous section, our model comprises six n-ary graph level (NGL) modules: The four domain-specific substitutes of the $f_{addnode}$ base module $f_{addinstruction}$, $f_{addtype}$, $f_{addconstant}$, and $f_{predicate}$, as well as f_{number} and $f_{typenumber}$.

Add node modules The two modules $f_{addconstant}$ and $f_{predicate}$ share an identical structure:

$$f_{node}(G) = \mathrm{softmax}(f_{an}(\mathbf{h}_G)) \tag{6.2}$$

Similarly to f_{bglm} in Eq. 6.1, f_{node} represents both modules. The MLP f_{an} has, depending on which module it belongs to, 23 or 17 output neurons, respectively. After f_{an} maps $\mathbf{h}_G$ to the action output space, the softmax function is used to transform the output into a vector of probabilities. The actual node label is then selected by applying the $argmax$ function to $f_{node}(G)$.

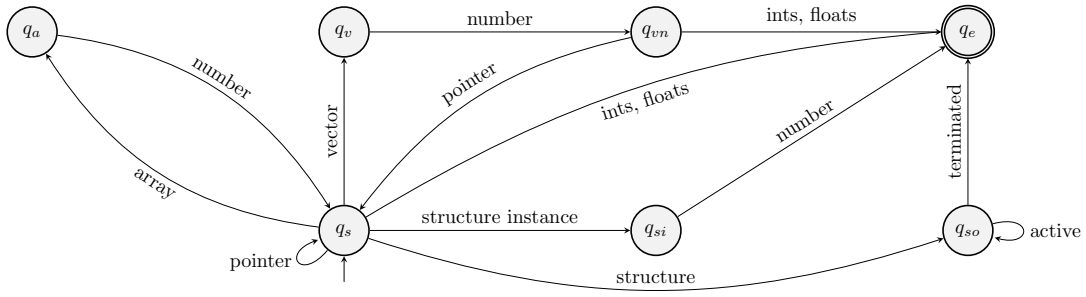


Figure 6.8: State diagram of the type generation process.

We construct $f_{addinstruction}$ and $f_{addtype}$ in a similar way, however, we additionally filter their output with a bitmask $\mathbf{m}$:

$$f_{addtype/instr.}(G) = \text{softmax}(f_{ati}(\mathbf{h}_G)) \odot \mathbf{m} \tag{6.3}$$

Here, $f_{addtype/instr.}$ once more represents both modules, $\odot$ is the Hadamard (elementwise) product and $\mathbf{m}$ is a vector that satisfies $\mathbf{m}^T \in \{0,1\}^n$, where n is the number of type labels (17), or instruction labels (49) selectable by the MLP f_{ati}, respectively[5]. Regarding $f_{addinstruction}$, we set all values of $\mathbf{m}$, except the one representing the label "none" to 1. If all basics blocks in the current graph are complete, we set the value corresponding to "none" to 1, otherwise to 0 (we discussed the reason for this in section 6.2.3). To explain how we determine the elements of $\mathbf{m}$ in $f_{addtype}$, we need to re-examine the type generation subroutine (Figure 6.2). During this process, we utilize $f_{addtype}$ repeatedly. Every time we do this, as well as every time we predict a number through $f_{typenumber}$, we feed the results from these modules to a state machine S. Figure 6.8 depicts the state diagram of S[6]. For any state q_n, the following holds:

$$\mathbf{m}_i = \begin{cases} 1, & \text{if } (q_n, l_i) \in \delta \\ 0, & \text{else} \end{cases} \qquad i \in \{0,1,2,\ldots,16\} \tag{6.4}$$

Here, l_i is the label represented by the i-th output neuron of $f_{typenumber}$, and δ represents the state-transition function of S. For example, if S is in the state q_{vn}, we set the entries of $\mathbf{m}$ representing integer or floating-point labels to 1, and all others to 0.

This approach guarantees that the resulting LLVM type t is syntactically correct. However, this does not mean that t is also semantically correct, for example, during the target-type generation for a `bitcast` instruction, the resulting type might not be castable to the statements operand type. Initially, we tried to

[5] In this section, T refers to transposition, not to rounds of information propagation

[6] If we reach the state q_{so}, we initialize another state machine S_1 that is structurally identical to S. As long as S_1 does not terminate, or $f_{operand}$ (see Figure 6.2) produces a positive result, we stay in q_{so}. If both of these conditions are not satisfied, we advance to the accepting state q_e.

implement a state machine of the form $S_{universal}(o, n) = \mathbf{m}$, which takes an opcode o and an integer $n \in \mathbb{N}_0$ representing the operand number, and produces an according bitmask $\mathbf{m}$. However, this proved too time-consuming. We were also unable to implement a similar state machine for the generation of constant values; once more due to time constraints.

Add number modules The modules f_{number} and $f_{typenumber}$ are graph level modules, even though they predict numerical values of nodes representing constant values and types, respectively. The reason for this is that we pass their results to R_{init} (as discussed in sections 5.2.4 and 5.2.5); thus, we compute this value before the affected node is present in the graph.

Before training, we normalize the numerical values that act as target labels for both modules through min-max normalization, thereby constraining them into the range $[0, 1]$:

$$f_{norm}(x) = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{6.5}$$

Here, the values $min(x)$ and $max(x)$ represent the lowest and highest numerical values found in the training data for each module; hence, $min(x)$ and $max(x)$ are different for f_{number} and $f_{typenumber}$.

Both modules, here represented by f_{num}, operate in the following way:

$$f_{num}(G) = \sigma(f_n(\mathbf{h}_G)) \tag{6.6}$$

Since the modules only predict one value between 0 and 1, we implement f_n as an MLP with one output neuron and normalize its output with the logistic sigmoid function σ. To obtain the actual predicted value, we denormalize the module's output with the inverse normalization function f_{norm}^{-1}.

6.3.3 Binary Node Level Modules

Modules operating on the node level take the graph representation vector $\mathbf{h}_G$, as well as the node embedding $\mathbf{h}_v$ of a node v as input. In our model, four such modules predict binary structure building decisions: $f_{addedge}$, $f_{branchedge}$, $f_{operand}$, and $f_{constant}$. They operate as follows:

$$f_{binnode}(G, v) = \sigma(f_{bn}(\mathbf{h}_G, \mathbf{h}_v)) \tag{6.7}$$

Similarly to Eq. 6.1, f_{bn} is an MLP with two output neurons, and $f_{binnode}$ represents each of the four BNL modules.

6.3.4 N-ary Node Level Modules

The last class of modules predicts node level decisions with an n-ary action output space. We implement four such modules: $f_{instructionedge}$, $f_{localfunction}$, f_{struct}, and $f_{externalfunction}$.

Score-based modules Out of those four modules, three are score-based. More precisely, the modules $f_{instructionedge}$, $f_{localfunction}$, and f_{struct} each predict a score s_v for each node $v \in V$ in the graph $G = (V, E)$ and subsequently select the node $u \in V$ with the highest score s_u.

The two modules $f_{localfunction}$ and f_{struct}, here represented by f_{nodes}, share the following structure:

$$s_u = f_s(\mathbf{h}_u, \mathbf{h}_v), \quad \forall u \in S \tag{6.8}$$

$$f_{nodes}(G, v) = \mathrm{softmax}(\mathbf{s}) \tag{6.9}$$

Here, f_s is again implemented as an MLP, $\mathbf{s}$ is a vector containing all scores computed in Eq. 6.8, and S represents a subset of nodes in the graph. Each of the two modules possesses a separately denoted S containing different nodes:

- In the case of $f_{localfunction}$, the set $S_{function}$ contains all "function" nodes that represent the entry point of an LLVM function ($S_{function}$ does not include, for example, the "function" operand of another `call` instruction).
- f_{struct} utilizes the set S_{struct}, which contains all root nodes of a subtree representing a structure. It does not include "structure" nodes that are themselves part of a larger type-subtree.

The third module, $f_{instructionedge}$, has a structure similar to the one described in Eq. 6.8 and Eq. 6.9. However, the MLP f_{ie}, that computes a score for a given node pair has one additional input:

$$s_u = f_{ie}(\mathbf{h}_u, \mathbf{h}_v, \mathbf{x}_{target}), \quad \forall u \in S_{instruction} \tag{6.10}$$

$$f_{instructionedge}(G, v) = \mathrm{softmax}(\mathbf{s}) \tag{6.11}$$

Here, the set $S_{instruction}$ contains instruction nodes, and $\mathbf{x}_{target} \in (\{0,1\}^2)^T$ is a one-hot encoded vector. We incorporate $\mathbf{x}_{target}$ into f_{ie} to accommodate for two different use cases of $f_{instructionedge}$: If the module selects a control-flow edge source node, which happens at a single point in the graph generation process, shortly after a new instruction node is generated, we define $\mathbf{x}_{target} = (0,1)^T$. In all other cases, $f_{instructionedge}$ instead predicts a control-flow edge target node; hence, we set $\mathbf{x}_{target} = (1,0)^T$. The content of $S_{instruction}$ also varies depending on the application of the module:

- If $f_{instructionedge}$ predicts a control-flow edge source, $S_{instruction}$ contains all instruction nodes that may still receive outgoing control-flow edges.
- If the module is instead utilized for the prediction of a control-flow edge target, $S_{instruction}$ contains all instruction nodes of the graph.
- During the "add operand subroutine", if a type t is passed as an argument, and $f_{instructionedge}$ selects an instruction operand, $S_{instruction}$ only contains nodes whose associated type matches t.

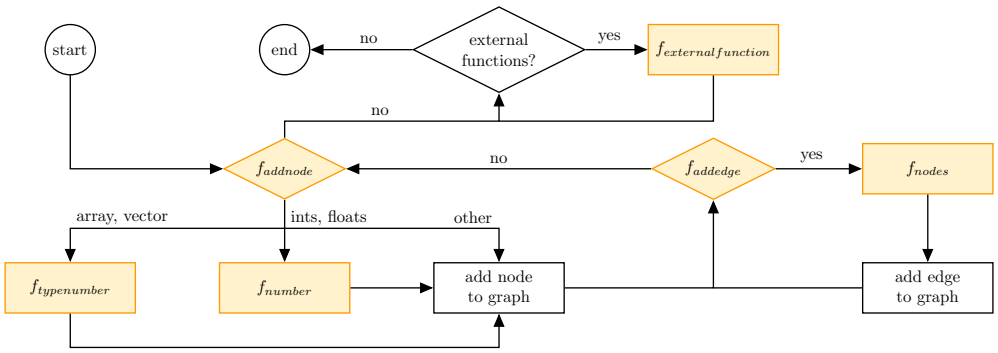


Figure 6.9: Flowchart of the base model's graph generation process, extended to allow for the generation of LLVM graphs.

External function module The last of the four n-ary node level (NNL) modules, $f_{externalfunction}$, does not predict a score for nodes in the graph. Instead, it selects one of 920 classes representing external LLVM and OpenCL functions:

$$f_{externalfunction}(G, v) = \sigma(f_{ef}(\mathbf{h}_G, \mathbf{h}_v)) \tag{6.12}$$

Structurally, the only difference between $f_{externalfunction}$ and the BNL modules characterized by Eq. 6.7 is the dimensionality of their respective MLP's last layer.

6.4 Baseline model

In section 8.1.1, we compare the performance of our domain-specific model against the performance of the base model proposes by Li et al. [2]. This comparison necessitates the implementation of the LLVM graph generation process for the base model. In section 6.1, however, we established that the base model's functionalities are not sufficient for the domain of LLVM graphs, as it does not allow for the selection of external functions or the prediction of numerical values. To allow for a comparison of the two approaches, we altered the base model in a way that keeps most of its structure intact, while still extending it for the generation of LLVM graphs.

Figure 6.9 shows this extended base model. We altered its structure in two ways: First, After $f_{addnode}$ predicts a node label l, we check whether l represents a list label ("vector" or "array") or an integer or floating-point label. If eighter is the case, we apply $f_{typenumber}$ or f_{number}, respectively, to predict a numerical value, which we then feed into R_{init}. If l is not one of those labels, the procedure remains unchanged. Our second structural modification takes effect after the graph is already generated: At this point, any external function represented in the graph takes the form of a "call" node u connected to a "function" node v, as well as to a (potentially zero-length) chain of "operand" subtrees (Figure

5.2b shows an example of this). However, since there are multiple LLVM and OpenCL functions with the same signature, this representation may not always uniquely identify them. Therefore, we apply a modified version of the $f_{externalfunction}$ module to the "function" node v in each of those constructions:

$$f_{externalfunction}(G, v) = \sigma(f_{ef}(\mathbf{h}_G, \mathbf{h}_v)) \odot \mathbf{b} \tag{6.13}$$

This version of $f_{externalfunction}$, unlike the one described in Eq. 6.12, incorporates the bitmask $\mathbf{b}$, which satisfies $\mathbf{b}^T \in \{0, 1\}^{920}$. We determine $\mathbf{b}$'s value the following way:

$$\mathbf{b}_i = \begin{cases} 1, & \text{if } f_i \in F_v \\ 0, & \text{else} \end{cases} \qquad i \in \{0, 1, 2, \ldots, 919\} \tag{6.14}$$

Here, f_i is the function represented by the i-th output neuron of f_{ef}, and F_v is the set of external functions that are correctly represented by the subtree of the "call" node u that v is connected to. For example, the subtree depicted in Figure 5.2b may describe a function with two `i32` operands[7] (if we assume the length of the "operand" node chain to be two). In this case, F_v would contain all external functions whose signature matches `(i32, i32)`.

We use the result selected by $f_{externalfunction}$ to determine which function is represented by the subtree emerging from u. This approach assumes that $F_v \neq \emptyset$, as this would imply that the graph has been generated incorrectly. In this case, we abort the graph generation algorithm.

[7] From this visual representation, it is not clear what bit-width the integer type of the first "add" operand has. In the actual graph, however, this information is provided by the operands of this node.

7 Training the Generative Graph Model

Until now, we focused primarily on the structural and mathematical modifications necessary to extend Li et al.'s [2] model for the generation of LLVM graphs. In this chapter, we focus on the technical details of training this adapted model. First, we conduct an examination of our training data and discuss how we transform its kernels into decision sequences. Next, we focus on our model's hyperparameters and how to optimize them. In the last part of the chapter, we discuss the training procedure of both the final model and the baseline model.

7.1 Training Data

For the training of our model, we rely on the same data set used by Cummins et al. to train CLgen [1]. It contains around 5700 OpenCL kernels that were scraped from various GitHub repositories. However, not all kernels contain syntactically correct code, and even fewer are semantically correct, i.e., can be executed successfully. To avoid learning semantically erroneous graphs, we reduce the training set to kernels that can be successfully executed by *cldrive* [36], which is a tool that can execute arbitrary OpenCL kernels. Out of the approximately 5700 kernels in the data set, only 1257 meet this requirement. Additionally, many of them produce long decision sequences, which is problematic for two reasons:

- We conduct all training on the high-performance computing (HPC) cluster Taurus[1] at the TU Dresden ZIH on NVIDIA Tesla V100-SXM2 GPUs. The implementation of our model is not parallelized; therefore, we are only able to utilize one GPU during training, which limits us to 32GB of memory. During training, our model is unrolled in memory because we use BPTT; therefore, longer decision sequences directly result in higher memory utilization.
- Our jobs on Taurus are restricted to a runtime of 24 hours. Although it is possible to suspend training after this period and resume it on another job, during hyperparameter optimization, we adhere to this limit in order to increase the number of constellations we can test. However, since an increase in decision sequence length entails a proportional increase in model parameters and, therefore, in training time, we are compelled to limit their length.

[1] `https://tu-dresden.de/zih/hochleistungsrechnen/hpc`

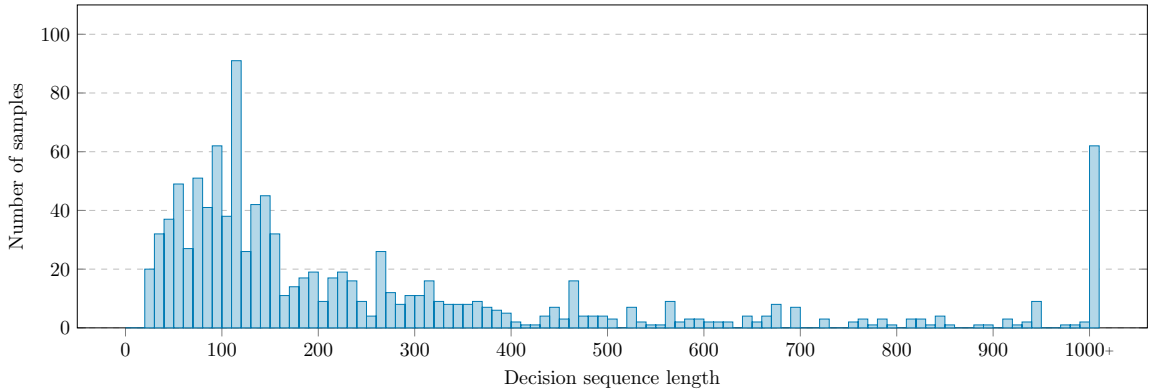


Figure 7.1: Histogram of the decision sequence length of the kernels in our training set with a bin-size of 10. Sequences that exceed 1000 actions are grouped in the same bin.

Especially the second point is problematic, since it restricts the number of parameters we can incorporate into a model while still achieving meaningful training progress. Therefore, we introduce two restrictions towards the graphs in our training set: During hyperparameter optimization, we only utilize graphs with a corresponding decision sequence containing at most 100 actions[2]. For the training of the final model, we increase this limit to 200, since we train it only once and can, therefore, use more time. This limit, unfortunately, entails a low amount of complex sub-structures in the training graphs. For example, both training sets do neither contain nested lists nor complex constant global variables. However, this is not a fundamental shortcoming of our approach and could, in principle, be circumvented with more computation time and memory.

Figure 7.1 displays the decision sequence length of the graphs produced by the 1257 usable kernels in the training set. It becomes clear that, even with a length limit of 200 actions, a large number of kernels are not usable for training. More precisely, 203 kernel's decision sequences are short enough to be used during hyperparameter optimization, and 600 kernels fulfill the requirement to be used in the final model's training. Ideally, we would like to increase the number of training samples significantly; due to time constraints, we leave this task for future work.

7.2 Actionizing the Training Data

Before we can use a kernel k for training, we need to translate it to a decision sequence S_k, which we denote by $S_k = \text{actionize}(k)$. For this purpose, we define an order in which we traverse k, since any non-

[2]The initialization of a new node through R_{init} also counts towards this action limit, even though we did not depict it in the Flowcharts describing our graph generation process in chapter 6.

trivial graph can be constructed in a multitude of ways. Although the translation process is structurally similar to our graph generation algorithm discussed in chapter 6, it lacks certain fail-safe mechanisms, as we can assume every training kernel to be syntactically correct. To avoid redundancies, the following description of the procedure only represents an overview (for more information, please refer to Appendix B):

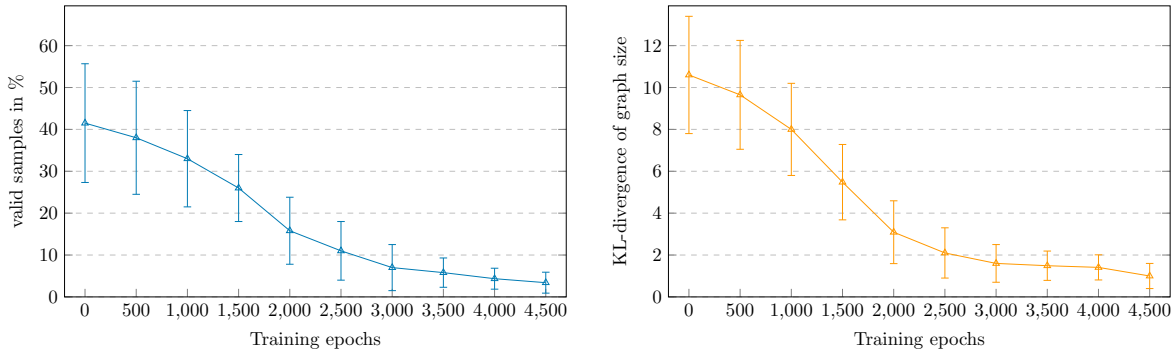
1. Actionize each structure in k in the order of its appearance. Subsequently, do the same for each global variable in k.
2. Sort the basic blocks of k with breadth-first search (BFS) based on their appearance in the kernel's control-flow graph (CFG).
3. For each function f in k:
   a) Actionize f.
   b) For each basic block b in f:
      i. Actionize each instruction i.
      ii. If i is a `phi` instruction, add it to the ordered set I_{phi} and actionize its default operand. If not, actionize all operands of i.
4. For each `phi` instruction p in I_{phi}, actionize all operands except the default operand.

7.3 Hyperparameter Optimization

The rather complex nature of our domain-specific approach results in our model having a large number of tweakable hyperparameters. In order to find a suitable set of values for them, we apply Bayesian optimization [37]. However, since each additional hyperparameter multiplies the search space, and both our resources and time are limited, it is necessary to define a subset of parameters to optimize for. In this section, we first discuss how to find such a subset, followed by an examination of the actual optimization process.

7.3.1 Optimization Target

The first task of hyperparameter optimization lies in determining the target metric. Intuitively, we would want to aim for the highest possible ratio of correctly generated LLVM graphs. However, multiple tests on models with randomized hyperparameters have shown that, as the number of training epochs increases, the ratio of valid sample graphs consistently decreases, as depicted in Figure 7.2a.



(a) (b)

Figure 7.2: Percentage of valid samples (a) and the KL-divergence between the size distributions of generated samples and training graphs (b) for an increasingly trained model.

Even though this discovery seems unintuitive, it makes sense upon closer inspection of the graph model structure. In a randomly initialized model, binary decisions (such as the choices of whether to add a new global variable, function, or edge) have an approximately even chance to predict eighter a positive or negative result. Therefore, on average, the length of the graphs generated by such a model is significantly shorter than the length of graphs in the training set. For example, the following minimal decision sequence produces an empty, but valid, LLVM graph:

```
<do not add structure>
<do not add global variable>
<do not add function>
```

It is composed of three binary decisions, and therefore has, on average, a 12.5% chance of being produced by an untrained model. However, simply ignoring this particular decision sequence does not solve the problem. As the size of generated samples approaches the mean size of the training set graphs during training, each generated graph's corresponding decision sequence also increases in size, and therefore in its susceptibility to erroneous decisions. A whole code graph can become invalid from just one such decision; therefore, a decrease in the percentage of correctly generated graphs is a natural consequence of an increase in their size.

Consequently, we require a different metric for measuring training success. Since our generative model learns a probability distribution over the graphs in the training set, a probability-distance measure between those graphs and the samples generated by the model is a more useful measure than the ratio of valid samples. However, since it would be unreasonable to try to model the entire probability space of LLVM graphs, we instead focus on the graph order (i.e., graph size) distribution. Our metric of choice is

the KL-divergence[3], which, for discrete distributions, is calculated in the following way:

$$D_{KL}(P(x) \parallel Q(x)) = \sum_{i=0}^{|P(x)|} P_i(x) * log_2 \left(\frac{P_i(x) + \epsilon}{Q_i(x) + \epsilon} \right) \tag{7.1}$$

Here, $P(x)$ and $Q(x)$ are two discrete probability distributions defined over the same probability space $\mathcal{X}$, and ϵ is a small positive number that ensures $Q_i(x)+\epsilon > 0$ to avoid divisions through 0. Additionally, it prevents the term $log_2(0)$ from occurring.

During training, as the model learns the probability distribution of the graphs in the training set, the KL-divergence between generated samples and those graphs should decrease, which is indeed what happens, as depicted in Figure 7.2b. Although a multi-faceted metric (e.g., a combined distance-measure of the graph size distributions and the node degree distributions) might be even more useful in determining training progress, it would also introduce problems. For example, it is not clear how each component of such a merged metric should be weighted in order not to distort its significance. Because of this, as well as for time reasons, we rely on the KL-divergence of the graph size distributions for the evaluation of models with separate sets of hyperparameters.

7.3.2 Reducing the Search Space

Our model consists of 17 decision-making modules, as well as the components that compute the node embeddings $\mathbf{h}_V$ and the graph representation $\mathbf{h}_G$, as discussed in section 4.3. In order to reduce the number of possible hyperparameter configurations, it is necessary to limit the number of optimization parameters. Therefore, we define a range of (partially abstract) hyperparameters:

- We establish $h_s = |\mathbf{h}_v|$ as the parameter that determines the node embedding's size. Intuitively, this value should be at least equal to the number of node labels, which is 91. However, since not all labels are used in the training data, we designate $\{60, 61, 62, \ldots, 130\}$ as the search space of h_s.
- The parameter h_{bn} determines the shape of BGL modules, such as $f_{branchedge}$. Later in this section, we define a number of hidden neurons n for each MLP, which allows us to shift the optimization process towards the shape of MLPs rather than their size. The shape is then determined by h_{bn}: We construct the MLP with h_{bn} hidden layers of size $\frac{n}{h_{bn}}$. For example, if $n = 100$, and $h_{bn} = 5$,

[3]Technically, the KL-divergence is not a measure of distance, as $D_{KL}(P(x) \parallel Q(x))$ is generally not equal to $D_{KL}(Q(x) \parallel P(x))$), for two probability distributions $P(x)$ and $Q(x)$. However, in our case, it still produces the desired effect of quantifying the difference between two probability distributions.

the MLP has 5 hidden layers of 20 neurons each. We define $[1, 5]$ as the discrete search space of h_{bn}.

- We additionally introduce h_{bg}, h_{nn} and h_{ng}, which act as equivalents of h_{bn} for BGL, NNL, and NGL modules, respectively. We chose a discrete search interval of $[1, 5]$ for each of these parameters as well.
- Lastly, we introduce h_{fg}, which governs the shape of the MLPs involved in the computation of $\mathbf{h}_G$ in Eq. 4.5. Once more, we designate $[1, 5]$ as the distinct search interval for h_{bn}.

This assortment of values, however, only represents a subset of the model's hyperparameters. For example, we predefine the number of propagation rounds during the node embedding update procedure to be $T = 2$, since this value is used by Li et al. [2] during the experiments discussed in section 4.4.1, and any increase in it substantially slows down training. Additionally, we select 100 as our model's mini-batch size, since this value facilitates a high training speed. In three tests with otherwise identical models with the batch sizes 25, 50, and 100 over 2000 epochs, we have not observed meaningful differences in the KL-divergence of generated samples. For time reasons, we also do not optimize the shape of the MLPs contained in f_{number} and $f_{typenumber}$. In the mentioned mini-batch size test, MLPs with three hidden layers containing 300 neurons each have shown to be a reasonable solution for both modules.

As most of the computations in our model are performed by MLPs, each having a virtually unlimited number of possible shapes, it is necessary to structure our search. Therefore, we define a certain overall size of the model, as mentioned above. This approach allows us to use hyperparameters such as h_{bn} and h_{fg} to focus our search on the shape of the MLPs. We assume that a higher number of model parameters translates into an improvement of the model's performance, or at least not into a performance decrease. Due to time reasons, we are unable to test this hypothesis; however, the fact that the model becomes unstable once n becomes too small[4] suggests that this assumption holds true up to a certain point. Consequently, we require a set of MLP sizes which allows us to perform a substantial amount of training during the hyperparameter optimization phase (where we limit the decision sequence length of training samples to 100 and the training duration to 24 hours). Additionally, the same set of parameters should allow for a complete training run of the final model (which uses training samples with up to 200 actions) within a week. Therefore, we define the following parameters:

- For BGL modules, we define $n_{bg} = 900$. Similarly to the parameters described in the following, we select this value empirically based on a small number of test runs.

[4]This phenomenon seems to be related to the size of the node embeddings h_s. With $h_s = 60$, the size for BNL modules $n_{bn} = 600$, and $h_{bn} = 3$, we experienced exploding gradients during training. However, once we decreased $h_s = 30$, training proceeded without problems.

- We define n_{ng}, which determines the size of NGL modules, as $n_{ng} = 1800$. The reason for this high value compared to n_{bg}, is the large output space of NGL modules of up to 49 states, as opposed to the two output states of BGL modules.
- For BNL modules, we choose $n_{bn} = 1500$, since the higher dimensionality of their input vector leads to unstable training for low n_{bn}'s.
- Additionally, we define n_{nn}, which governs the size of NNL modules, as $n_{nn} = 1500$, as they have the same input vector size as BNL modules.
- For the MLPs involved in the computation of $\mathbf{h}_G$, we select $n_{fg} = 1500$ as well, since the high dimensionality of $\mathbf{h}_G$ leads to a large output space of both MLPs.

7.3.3 Bayesian Optimization

In order to find a suitable set of values for the hyperparameters defined in the previous section, we apply Bayesian optimization. This search strategy was first introduced by Močkus in 1974 [37] and attempts to find a global optimum of a black-box function $f(x)$ in a minimum number of steps. To briefly summarize, Bayesian optimization utilizes a surrogate model that approximates $f(x)$. This model updates its approximation every time a new sample point of $f(x)$ is calculated. In order to decide which parameter constellation x to sample, the model utilizes an acquisition function that aims to strike a balance between the exploitation of known parameter constellations and the exploration of new ones (depending on the acquisition function, one or the other might be favored).

For our purposes, we rely on a Gaussian surrogate model and the expected improvement acquisition function. We train each model over 5000 epochs. For the training in this section as well as the following sections, we use an Adam Optimizer with a learning rate of $\eta = 0.0000075$ and the L2 loss function for all decision modules. Table 7.1 contains the result of the hyperparameter search. For the complete data set of the Bayesian optimation process, please refer to Appendix C.

Parameter	h_s	h_{bg}	h_{ng}	h_{bn}	h_{nn}	h_{fg}	KL-divergence
Value	68	1	3	1	3	1	0.72

Table 7.1: Result of the hyperparameter search with Bayesian optimization over 16 iterations with three initial evaluations.

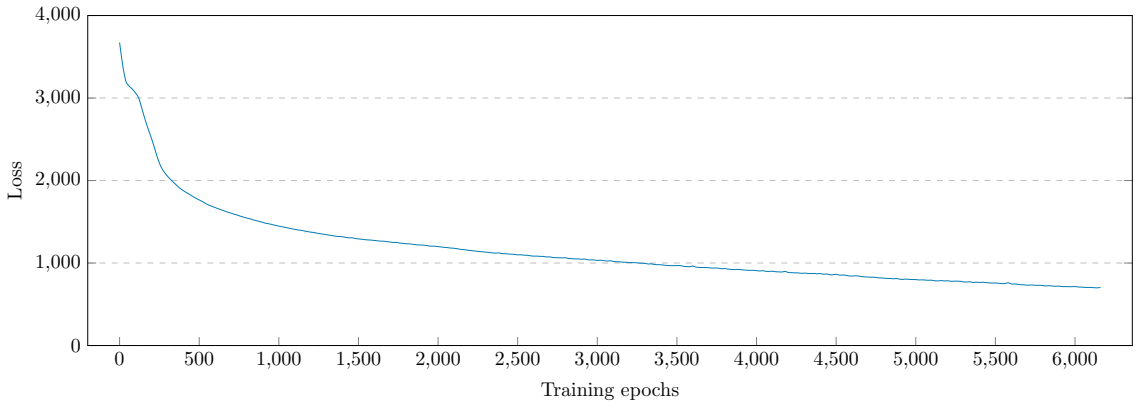


Figure 7.3: Loss of the final model during training.

7.4 Training the Model

For the training of the final model, we rely on the same training parameters that we used in the previous section, except for the number of epochs: We continue training as long as the following inequality, where m_n denotes our model after n training epochs, is satisfied:

$$\text{loss}(m_{n-100}) > \text{loss}(m_n) * 1.01 \tag{7.2}$$

With this approach, training lasts for 6162 epochs, as depicted in Figure 7.3, and we achieve a KL-divergence (sampled over 1000 graphs) between the graphs contained in the training set and the final model of $D_{KL}(P_{dataset}(x) \parallel P_{model}(x)) = 0.72$.

7.5 Training the Baseline Model

Due to time reasons, we are unable to conduct a hyperparameter search for the base model. Therefore, we reuse the hyperparameters h_{ng} and h_{nn}, as well as the corresponding values n_{ng} and n_{nn} for the modules $f_{typenumber}$ and $f_{externalfunction}$, respectively. Additionally, we introduce h_{an}, which governs the shape of the MLP contained in the $f_{addnode}$ module (Eq. 4.8). We set $h_{an} = h_{ng}$, since $f_{addnode}$ is also a NGL module. However, since f_{an} has an output space of 91 actions (one for each label), we define $n_{an} = 3600$.

We employ a similar approach for $f_{addedge}$ and f_{nodes}. For both modules, we adopt the corresponding hyperparameters found in section 7.3.3. Additionally, we once more utilize the inequality characterized by Eq. 7.2 to determine the number of training epochs. The baseline model achieves a KL-divergence (sampled over 1000 graphs) of $D_{KL}(P_{dataset}(x) \parallel P_{baseline}(x)) = 7.07$.

8 Evaluation

After focusing on the design of our model and subsequently on its training, in this chapter, we evaluate our approach based on its ability to generate LLVM code. We begin by examining generated samples, and subsequently compare our model to Li et al.'s general-purpose approach [2] and Cummins et al.'s CLgen [1].

8.1 Generated Samples

To evaluate our model, we generate 10000 samples, six of which are depicted in Figure 8.1. Excluding empty graphs, 2.3% of these samples represent unique, valid LLVM programs, i.e., can be transformed into human-readable LLVM IR by our code generator and compiled to bitcode with the tool *llvm-as* [38].

However, the mean decision sequence length of these samples is 32, whereas graphs in the training set require, on average, 98 actions to be constructed. To understand this discrepancy, a re-examination of our code generation pipeline (Figure 1.1) is necessary. During the generation process, we first construct a graph, which we subsequently transform into LLVM IR. Therefore, a sample has to undergo both stages as well as a successful compilation by *llvm-as* (which transforms human-readable IR to bitcode) before it can be considered valid. Since an entire decision sequence can be invalidated by one erroneous action, all of these stages favor short sequences, as those are statistically less likely than long ones to contain such an action. Therefore, the mean decision sequence length of samples decreases after each generation stage. This observation is reaffirmed by the mean decision sequence length of the samples passing the stages of graph generation and code generation, which are 75 and 60, respectively. Out of all attempts, 37% manage to advance past the graph generation stage, 23.4% successfully pass through the code generator, and 2.3% constitute valid LLVM programs.

The reason so many attempts fail at the graph generation stage is (for the most part) the type-computation we perform for each newly added node[1]. Even after the graph generation stage, types remain the primary

[1] For example, the type-computation of `getelementptr` instructions has a particularly high failure rate, since each operand following the first (which is of an aggregate type) indexes a sub-type. Therefore, each index has to stay in the bounds of its corresponding sub-type, e.g., a structure with two elements only allows for the two operands `i32 0` or `i32 1`.

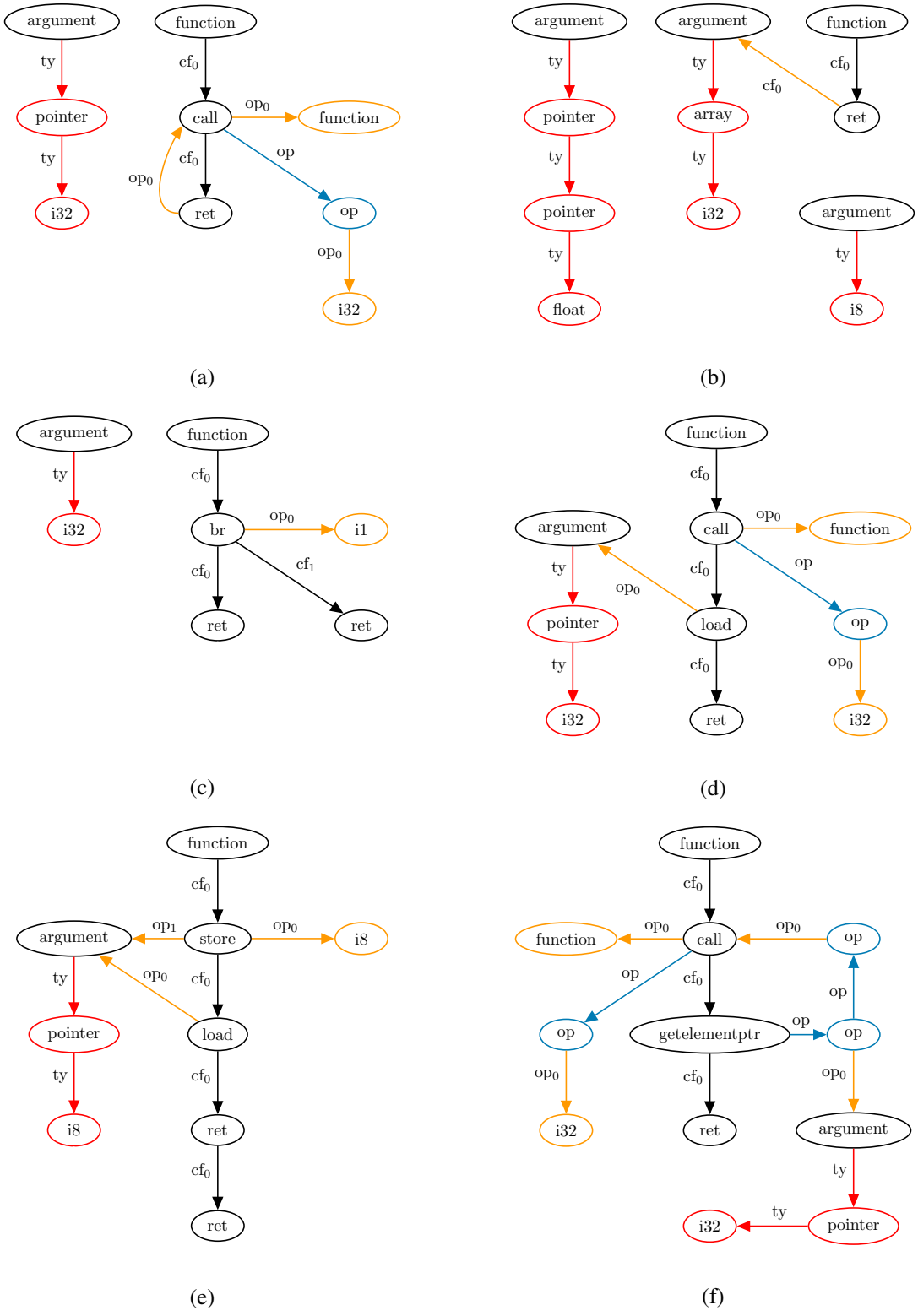


Figure 8.1: Six valid LLVM graphs generated by our domain-specific model. The corresponding LLVM programs are located in Appendix D.1.

challenge for our approach. Whereas the IR produced by our code generator constitutes (with rare exceptions) structurally correct LLVM, 6% of these LLVM programs contain empty function bodies (and are, therefore, invalid), another 10% constitute valid LLVM programs, and the remaining 84% contain type-related semantic errors. For example, let us examine the following generated IR:

```
define void @A(i1**) {
; <label>:1:
   store i32 211, i1** %0
   ret void
}
```

Here, the `store` instruction is structurally correct; however, its types do not match, as a variable of type t requires a storage address of the corresponding pointer type t*. In this case, the constant `i32 211` requires an address of type `i32`*; therefore, the program is invalid.

Low rate of valid samples Since our goal in this work is the development of a model that, by design, creates mostly structurally correct LLVM graphs, the question arises as to why our model only generates valid samples at a ratio of 2.3%. As stated above, the main weakness of our approach in its current form is type-consistency. Our graph generation process primarily applies structural domain-specific knowledge concerning, for example, the number of edges in an `add` instruction, or the termination of basic blocks with terminator opcodes. Although these limitations lead to a high number of structurally correct LLVM graphs, they fail to prevent semantic errors, such as the one depicted in the code sample above. If we, however, focus on samples that have been transformed into LLVM IR but have not yet been compiled by *llvm-as*, we ignore such semantic errors. Out of all generation attempts, 23.4% advance to this stage. As stated above, these samples have a mean decision sequence length of 60. Although this is a severe oversimplification, a success rate of 23.4% over 60 actions translates into an average success rate of roughly 97.5% per individual action. However, since different decision-making modules have separate action output spaces, this number can only serve as a rough estimate.

In summary, we observe that our approach succeeds in significantly reducing structural errors, but struggles with semantic errors. Nevertheless, we believe that this issue is not insurmountable, especially since we successfully integrated a type state machine (Figure 6.8) into the graph generation process. Similar state machines could be used to implement type-consistency checks. We leave this for future work.

Opcode distribution Since our model learns a probability distribution over the training graphs, in theory, their opcode distribution should resemble the distribution of opcodes contained in generated samples. Figure 8.2 shows that this is not entirely the case. For example, our model produces a higher

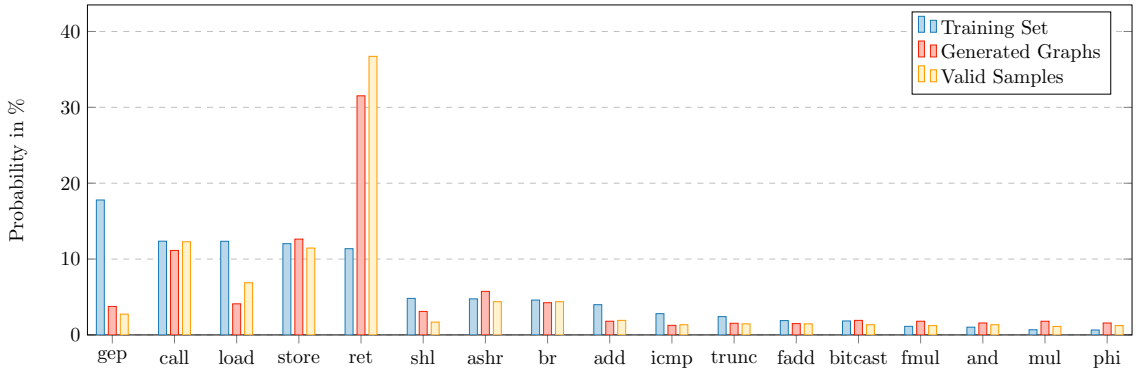


Figure 8.2: Probability of appearance for the 17 most common opcodes. The opcode `getelementptr` is abbreviated with "gep". The values for generated graphs (red) and valid samples (orange) have been calculated with 10000 generated samples. Probabilities for all opcodes are located in Appendix D.2.

number of `ret` statements than expected. This is a direct consequence of the low decision sequence length of the generated samples. Since every basic block is required to end with a terminator instruction, and, out of these opcodes, `ret` is the simplest one, samples which few and short basic blocks terminating with `ret` pass the generation process disproportionally often.

In the opcode distribution of graphs and valid samples, we can additionally observe an underrepresentation of `getelementptr` instructions, which is a consequence of the structural complexity of this opcode. The same underrepresentation can be observed for `load` instructions, although we are not certain about the reason for this, especially since the structurally more complex `store` is generated at a rate more closely resembling its probability of appearance in the training set. Although we have observed a general tendency of the model towards void instructions (to which `load` does not belong to) and structurally simple types, as those avoid type-related errors, a more thorough examination of `load`'s underrepresentation might provide valuable insights. Due to time reasons, we leave this for future work.

8.1.1 Baseline Model

For the evaluation of the baseline model, we once more attempt to generate 10000 samples. However, since only an exceedingly small percentage of generation attempts pass the code generation stage, it is infeasible to synthesize such a large number of samples. Therefore, we instead analyze the results of 10000 generation attempts.

Out of those attempts, 65 (0.65%) reach the code generation stage, 35 (0.35%) successfully pass through our code generator, and 2 (0.02%) constitute valid, non-empty LLVM IR. The two valid samples both

use 9 actions and represent the same LLVM program:

```
define void @A() {
; <label>:0:
   ret void
}
```

The samples passing the graph generation and code generation stages have a mean decision sequence length of 49 and 43, respectively. The ability of the baseline model to produce arbitrary graphs proves to be a significant disadvantage in a precise and highly error-sensitive domain such as LLVM graph generation. For example, even the generation of a function argument proves to be difficult without structural restrictions, as illustrated by the following generated sample:

```
define void @A(operand*) {
; <label>:0:
   ret void
}
```

Out of the 65 samples that pass the graph generation stage, many contain "argument" nodes connected to usually small subgraphs with "ty" edges, which suggests that the model at least partially learned how to generate function arguments. However, most of the constructs connected to "argument" nodes in such a way do not represent valid LLVM types, as illustrated by the nonsensical construct `operand*`. Furthermore, instructions and operands entail an even higher structural complexity than function arguments, and consequently decrease the probability of a correctly generated sample exponentially.

Therefore, we conclude that, at least with our small number of training samples and limited computation time, the baseline model does not represent a useful approach for the generation of LLVM graphs. Additionally, due to the low amount of valid samples, we exclude the baseline model from the analyses performed in the following section.

8.2 Evaluation against CLgen

After examining the samples generated by our model in isolation, we now compare them against the samples generated by Cummins et al.'s CLgen [1]. For this purpose, we define a set of features that we use to provide an overview of the different approaches. Subsequently, we perform a principal component analysis (PCA) over these features to gain a more in-depth insight into the different models.

	valid sample %	decision sequence length
Baseline	0.02	9.00
Graph	2.31	32.34
CLgen	16.20	156.37

Table 8.1: Sample rate and mean decision sequence length of valid samples generated by the baseline model, our model, and CLgen.

8.2.1 Valid Sample Rate

During the stand-alone analysis of our model, we observe the generation of valid samples at a rate of 2.3%. Concerning CLgen, neither [1] nor [39] contains information about its valid sample rate; therefore, we sample 1000 kernels with the model. To ensure a fair comparison, we disable CLgen's dynamic checker[2] and set the temperature hyperparameter to 1. With these settings, 16.2% of the samples generated by CLgen constitute valid OpenCL kernels. Once we transform them into our graph representation and analyze the corresponding decision sequences, we observe an average of 156 actions per kernel, which is more than three times the size of the average valid sample generated by our model. However, unlike our model, which directly produces LLVM, CLgen generates OpenCL, which, on average, is $3.6\times$ longer[3] once it is transformed into LLVM. Furthermore, CLgen manually predefines a function signature, thereby usually adding between 10 and 20 "pseudo-actions" to any given sample once we transform it into our graph representation. The comparatively small size of our training-graphs decreases the size of our model's samples further. Consequently, a comparison between the action-count of both model's samples can only act as a guideline. For a better comparison of both approaches, we need to retrain CLgen on our reduced training set. Due to time limitations, we leave this for future work as well.

In the following sections, we analyze the differences between the approaches with a set of structural features characterizing their graph representation.

8.2.2 Graph Features

Similarly to Grewe et al. [40], we require a set of features that capture significant characteristics of code samples. However, since our goal in this work is the generation of code, not its execution, we utilize features that describe a samples structure rather than its behavior during runtime. Since all samples

[2]This means that a sample's correctness is only evaluated during compile-time, which is the same standard that we apply to the samples generated by our model.

[3]Measured on the character level on the kernels in the CLgen dataset.

	# nodes	# edges	# instr.	# struct.	# globals	# arguments	# unique labels
Training Set	30.11	37.98	9.91	0.01	0.02	2.64	13.62
Graph	10.39	9.86	3.17	0.00	0.03	1.42	8.41
CLgen	41.06	53.13	14.57	0.01	0.01	4.07	18.78

Table 8.2: Averaged graph features for our training set, our domain-specific model, and CLgen.

eighter exist in, or can easily be transformed into, a graph representation, we use graph metrics in our comparison. For every graph $G = (V, E)$, we measure the vertex cardinality $|V|$ and the edge cardinality $|E|$. Additionally, we measure the number of instructions, non-nested structures, global variables, and arguments that the graph contains. Lastly, we measure the number of unique node labels in G.

Table 8.2 contains the mean of these metrics for all training samples, as well as for the valid samples generated by our model and CLgen. As expected after the previous analyses, we observe lower values across the board for our model compared to both the training set and CLgen. Notably, we also find that the mean vertex degree $\deg(V) = \frac{|E|}{|V|}$ of our model's graphs is 0.95, whereas it is 1.26 and 1.29 for the training set graphs and CLgen's graphs, respectively. We can largely attribute this finding to the low rate of variable (e.g., previously calculated) instruction operands, as well as to the many unconnected argument sub-trees produced by our model (as illustrated by the graphs in Figures 8.1a to 8.1c). Although the same cannot be said about the former pattern, the latter is present in the training set as well.

If we turn our attention to the mean vertex and edge cardinalities of the graphs generated by CLgen, we find that both values are significantly higher then their training set equivalents. Although we have not analyzed the model's training procedure in-depth, it is highly likely that this is a consequence of the smaller mean size of our training graphs, compared to the ones used in CLgen's training.

8.2.3 Principal Component Analysis

To visualize the differences between the graphs from the training set, our model, and CLgen, we perform a PCA on all samples with the features illustrated in Table 8.2. For this purpose, we first collect the corresponding data for each sample and scale it appropriately. We choose a two dimensional PCA, as it explains 94.79% of the variance in the collected data (as illustrated by Figure 8.3a), and is easily representable in a two-dimensional graph.

Figure 8.3b shows the results of the PCA for 40 randomly selected samples per data set. For the training set samples, we observe a comparatively even spread over a wide range of values for both principal components, whereas the graphs generated by CLgen are more concentrated around the negative values

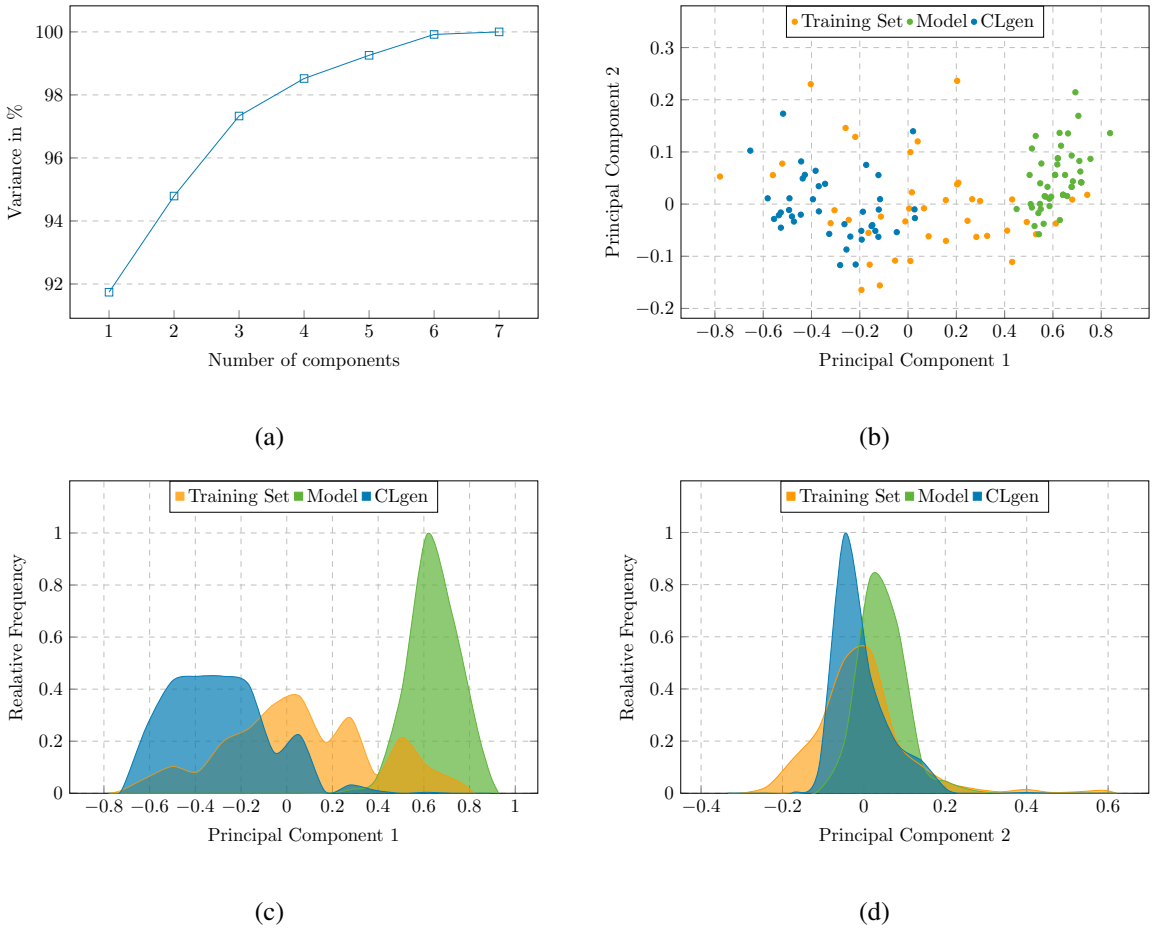


Figure 8.3: Explained variance for the features described in section 8.2.2 as a function of the number of PCA components (a), the two-dimensional PCA for 50 randomly selected samples per data set (b), and the relative frequency of all samples for both principal components (c) (d).

of principal component 1. The graphs generated by our model, however, are concentrated exclusively around a relatively small area around the point $(0.61, 0.05)$. This shows that our model is only able to produce graphs covering a sub-area of the training sets feature space. Notably, there is almost no overlap in principal component 1 for our model and CLgen, which is likely a consequence of the vastly different decision sequence lengths of the graphs generated by both models.

8.3 Conclusion

The analyses in the previous sections have shown that our domain-specific approach is able to produce valid LLVM code at an acceptable, albeit lower rate than CLgen. Whereas the generated samples are mostly structurally correct, our approach struggles with the semantic aspects of LLVM, such as type-correctness. Additionally, our domain is highly error-sensitive, as one action can invalidate an entire

decision sequence. Consequently, our model is biased towards smaller samples, as shown by the consistent decrease in decision sequence length following each generation stage.

As a result, our model only partially covers the feature space present in the training set. However, despite its shortcomings, our extension of Li et al.'s generative model is beneficial to its performance in our specific target domain, since it produces valid LLVM IR at a higher rate and variety than the general-purpose approach. In the next chapter, we discuss possible ways to alleviate many of the weaknesses of our approach.

9 Conclusion and Outlook

In this thesis, we developed a new domain-specific generative model for LLVM, based on Lit et al.'s DeepGMG [2]. We have shown that our approach is capable of generating valid LLVM programs at a rate surpassing this general-purpose model.

However, we also found shortcomings of our model. While we were able to successfully incorporate domain-specific knowledge of LLVM into our graph generation procedure, thereby significantly decreasing the probability of structural errors in generated samples, we struggled to achieve the same for the semantic aspects of the language. We identified type-related errors as the most significant issue of our approach in its current state. Consequently, our approach is biased towards simple (e.g., `void`) types and shorter graphs. As a result, the model only covers a sub-area of the feature space present in the training graphs and generates correct samples at a lower rate than CLgen [1].

Nevertheless, we are confident that many of these problems can be alleviated in future work. Since we already successfully implemented a state machine that ensures context-independent type-correctness, the implementation of similar mechanisms that guarantee context-dependent type-correctness is certainly feasible.

Furthermore, in this work, we placed restrictions on the decision sequence length of training samples to decrease training time, thereby shrinking our training set to only 600 kernels. A substantial increase in the number of training samples would likely benefit the model. If time is less of an issue, such an increase could be achieved by relaxing the limits we imposed. Additionally, techniques such as truncated backpropagation through time could be implemented to alleviate the memory intensity of our model.

In conclusion, we found that our extension of Li et al.'s general-purpose model shows promise in the domain of LLVM graph generation, and we are hopeful that many of its current issues could be mitigated in future work. However, it remains to be seen whether graph generative models are the best approach for the complex and highly error-sensitive domain of code generation.

Bibliography

[1] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *CGO*. IEEE, 2017.

[2] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *CoRR*, abs/1803.03324, 2018.

[3] Clang - Features and Goals. `https://clang.llvm.org/features.html#gcccompat`, (accessed November 1, 2019).

[4] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th ACM SIGPLAN International Conference on Compiler Construction (CC 2020)*, CC 2020, New York, NY, USA, February 2020. ACM.

[5] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, Jan 2009.

[6] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[7] A. Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, March 2009.

[8] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[9] Daniel D. Johnson. Learning graphical state transitions. In *ICLR*, 2017.

[10] Yibo Li, Liangren Zhang, and Zhenming Liu. Multi-objective de novo drug design with conditional graph generative model. *Journal of Cheminformatics*, 10(1):33, 2018.

[11] Alton Chiu, Joseph Garvey, and Tarek S. Abdelrahman. Genesis: a language for generating synthetic training programs for machine learning. In *CF '15*, 2015.

[12] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[13] Tomas Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, 2010.

[14] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, May 2013.

[15] K. Kamijo and T. Tanigawa. Stock price pattern recognition-a recurrent neural network approach. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 215–221 vol.1, June 1990.

[16] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994.

[17] Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision RNNs for language recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 740–745, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[18] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

[19] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. *Journal of Machine Learning Research*, 2015.

[20] Paul. Erdős and Alfred Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5:17, 1960.

[21] ChEMBL molecule database. `https://www.ebi.ac.uk/chembl/`, (accessed October 15, 2019).

[22] The LLVM Compiler Infrastructure Project. `https://llvm.org/`, (accessed November 1, 2019).

[23] LLVM Compiler Overview. `https://developer.apple.com/library/archive/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/index.html`, (accessed November 1, 2019).

[24] Breno Campos Ferreira Guimarães, Gleison Souza Diniz Mendonca, and Fernando Magno Quintão

Pereira. Dawncc: a source-to-source automatic parallelizer of c and c++ programs. 2016.

[25] 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.*, pages 75–86, March 2004.

[26] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In John Field and Michael Hicks, editors, *POPL*, pages 427–440. ACM, 2012.

[27] llgo: LLVM-based compiler for Go. `https://github.com/go-llvm/llgo`, (accessed November 1, 2019).

[28] Emscripten. `https://emscripten.org/`, (accessed November 1, 2019).

[29] rubinius: The Rubinius Language Platform. `https://github.com/rubinius/rubinius`, (accessed November 1, 2019).

[30] The LLDB Debugger. `http://lldb.llvm.org/`, (accessed November 1, 2019).

[31] "libc++" C++ Standard Library. `http://libcxx.llvm.org/`, (accessed November 1, 2019).

[32] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[33] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[34] LLVM Language Reference Manual. `https://llvm.org/docs/LangRef.html`, (accessed November 4, 2019).

[35] llvm-diff - LLVM structural 'diff'. `https://llvm.org/docs/CommandGuide/llvm-diff.html`, (accessed November 13, 2019).

[36] cldrive - Run arbitrary OpenCL kernels. `https://github.com/ChrisCummins/cldrive`, (accessed December 19, 2019).

[37] J. Močkus. *On Bayesian Methods for Seeking the Extremum*, pages 400–404. Springer Berlin Heidelberg, Berlin, Heidelberg, 1975.

[38] llvm-as - LLVM assembler. `https://llvm.org/docs/CommandGuide/llvm-as.`

html, (accessed January 2, 2020).

[39] Synthesizing Benchmarks for Predictive Modeling. https://github.com/ChrisCummins/paper-synthesizing-benchmarks, (accessed January 8, 2020).

[40] D. Grewe, Z. Wang, and M. F. P. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10, Feb 2013.

List of Figures

Acronyms

BFS	Breadth-first search
BGL	Binary graph level
BNL	Binary node level
BPTT	Backpropagation through time
CDFG	Control- and dataflow graph
CFG	Control-flow graph
DeepGMG	Deep generative models of graphs
GCC	GNU Compiler Collection
GG-SNN	Gated graph sequence neural network
GNN	Graph neural network
GRU	Gated recurrent unit
HPC	High-performance computing
IR	Intermediate representation
KL	Kullback-Leibler
LSTM	Long short-term memory
MLP	Multi layer perceptron
NNL	N-ary node level
NGL	N-ary graph level
NN	Neural network
PCA	Principal component analysis
RNN	Recurrent neural network
SSA	Static single assignment

A LLVM IR Implementation Details

The following sections contain implementation details of the LLVM IR to graph mapping used throughout this thesis.

A.1 Node Labels

Node label	Description
none	Used to signal the termination of a subroutine in the graph generation algorithm.
phi	Opcode.
call	Opcode.
trunc	Opcode.
add	Opcode.
icmp	Opcode.
and	Opcode.
or	Opcode.
mul	Opcode.
br	Opcode.
zext	Opcode.
getelementptr	Opcode.
bitcast	Opcode.
load	Opcode.
store	Opcode.
shl	Opcode.
ret	Opcode.
ashr	Opcode.
sdiv	Opcode.
sub	Opcode.
sext	Opcode.

Node label	Description
insertelement	Opcode.
shufflevector	Opcode.
srem	Opcode.
urem	Opcode.
select	Opcode.
fadd	Opcode.
fmul	Opcode.
fsub	Opcode.
xor	Opcode.
alloca	Opcode.
lshr	Opcode.
extractelement	Opcode.
fdiv	Opcode.
fcmp	Opcode.
sitofp	Opcode.
fptoui	Opcode.
udiv	Opcode.
fpext	Opcode.
fptrunc	Opcode.
fptosi	Opcode.
insertvalue	Opcode.
extractvalue	Opcode.
switch	Opcode.
uitofp	Opcode.
ptrtoint	Opcode.
unreachable	Opcode.
inttoptr	Opcode.
function	Specifies the beginning of a function.
struct	Specifies instances of a struct definied earlier in the code.
i1	Integer type with bit-width 1.
i2	Integer type with bit-width 2.

Node label	Description
i6	Integer type with bit-width 6.
i8	Integer type with bit-width 8.
i16	Integer type with bit-width 16.
i32	Integer type with bit-width 32.
i33	Integer type with bit-width 33.
i64	Integer type with bit-width 64.
half	Floating-point type with bit-width 16.
float	Floating-point type with bit-width 32.
double	Floating-point type with bit-width 64.
void	Type label that does not represent any value and has no size.
pointer of	Type label for pointer types.
array of	Type label for arrays.
vector of	Type label for vectors.
struct of	Type label for struct definitions.
undef	Specifies constant undefined values of any type.
zeroinitializer	Initializes a constant value of any type to zero.
null	Initializes a pointer type with no address.
operand	Used with variable operand opcodes. Specifies the next operand.
argument	Specifies an argument of a function.
global var	Specifies a global variable.
eq predicate	“equal” predicate for icmp-instructions.
ne predicate	“not equal” predicate for icmp-instructions.
ugt predicate	“unsigned greater than” predicate for icmp-instructions.
uge predicate	“unsigned greater or equal” predicate for icmp-instructions.
ult predicate	“unsigned less than” predicate for icmp-instructions.
ule predicate	“unsigned less or equal” predicate for icmp-instructions.
sgt predicate	“signed greater than” predicate for icmp-instructions.
sge predicate	“signed greater or equal” predicate for icmp-instructions.
slt predicate	“signed less than” predicate for icmp-instructions.
sle predicate	“signed less or equal” predicate for icmp-instructions.
uge predicate	“ordered and greater than” predicate for fcmp-instructions.

Node label	Description
ult predicate	“ordered and less than” predicate for fcmp-instructions.
ule predicate	“ordered and equal” predicate for fcmp-instructions.
sgt predicate	“ordered and greater than or equal” predicate for fcmp-instructions.
sge predicate	“ordered and less than or equal” predicate for fcmp-instructions.
slt predicate	“unordered or not equal” predicate for fcmp-instructions.
sle predicate	“unordered or equal” predicate for fcmp-instructions.

A.2 Implemented LLVM opcodes

The following opcode descriptions are modified versions of the opcode descriptions found in [34].

Opcode	Description
phi	Implements the φ node of the SSA graph representing the program.
call	Implements a function call.
trunc	Truncates its operand to the smaller integer type.
add	Returns the sum of its two operands.
icmp	Compares its two operands and returns a boolean value or a vector of boolean values.
and	Returns the result of a bitwise logical “and” operation of its two operands.
or	Returns the result of a bitwise logical “or” operation of its two operands.
mul	Returns the product of its two integer operands.
br	Transfers the control-flow to a different basic block in the current function.
zext	Zero-extends its operand to the target type.
getelementptr	Extracts the address of a subelement of a data structure.
bitcast	Converts a value to a target type without changing any bits.
load	Reads from memory.
store	Writes to memory.
shl	Returns its operand left-shifted by the specified number of bits.
ashr	Returns its operand right-shifted by the specified number of bits with sign extension.
sdiv	Returns the quotient of its two integer operands.
sub	Returns the difference of its two integer operands.
sext	Sign-extends its operand to a target type.
insertelement	Inserts an element into a vector at a specified index.

Opcode	Description
shufflevector	Constructs a permutation of elements from two input vectors.
srem	Returns the remainder of the signed division of its two operands.
urem	Returns the remainder of the unsigned division of its two operands.
select	Chooses one of its two operands based on a condition.
fadd	Returns the sum of its two floating-point operands.
fmul	Returns the product of its two floating-point operands.
fsub	Returns the difference of its two floating-point operands.
xor	Returns the result of a bitwise logical “exclusive or” operation of its two operands.
alloca	Allocates memory.
lshr	Returns its operand right-shifted by the specified number of bits with zero fill.
extractelement	Extracts an element from a vector at a specified index.
fdiv	Returns the quotient of its two floating-point operands.
fcmp	Compares its two floating-point operands and returns a boolean value or vector.
sitofp	Regards its operand as a signed integer and converts its to a floating-point type.
fptoui	Converts a floating-point operand to its unsigned integer equivalent of a target type.
udiv	Returns the quotient of its two operands, regarding them as unsigned integers.
fpext	Extends a floating-point operand to a larger floating-point type.
fptrunc	Casts a floating-point operand to a smaller floating-point type.
fptosi	Converts a floating-point operand to a target type, regarding it as an signed integer.
insertvalue	Inserts a value into a member field of an aggregate value.
extractvalue	Extracts an element from a vector at a specified index.
switch	Transfers the control-flow to one of several different places.
uitofp	Regards its operand as an unsigned integer and converts it to a target type.
ptrtoint	Converts the pointer or a vector of pointers to a target type.
inttoptr	Converts an integer value to a pointer type.
unreachable	Informs the optimizer that a particular portion of the code is not reachable.

A.3 Lossy Transformation between IR and Graph Representation

The following OpenCL kernel demonstrates that the transformation of LLVM programs to our graph representation is not lossless:

OpenCL kernel

```
void kernel __attribute__((reqd_work_group_size(1, 1, 1))) A(global int2* a,
    global float4* b, int c) {
  *b = ((global float4*)a)[c];
}
```

Compiled to LLVM IR

```
source_filename = "-"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: minsize norecurse nounwind optsize uwtable
define spir_kernel void @A(<2 x i32>* nocapture readonly, <4 x float>* nocapture,
    i32) local_unnamed_addr #0 !kernel_arg_addr_space !3 !kernel_arg_access_qual
    !4 !kernel_arg_type !5 !kernel_arg_base_type !6 !kernel_arg_type_qual !7 !
    reqd_work_group_size !8 {
  %4 = bitcast <2 x i32>* %0 to <4 x float>*
  %5 = sext i32 %2 to i64
  %6 = getelementptr inbounds <4 x float>, <4 x float>* %4, i64 %5
  %7 = load <4 x float>, <4 x float>* %6, align 16, !tbaa !9
  store <4 x float> %7, <4 x float>* %1, align 16, !tbaa !9
  ret void
}
attributes #0 = { minsize norecurse nounwind optsize uwtable "correctly-rounded-
    divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad
    "="false" "no-frame-pointer-elim"="false" "no-infs-fp-math"="false" "no-jump-
    tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "
    no-trapping-math"="false" }

!llvm.module.flags = !{!0}
!opencl.ocl.version = !{!1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 1, i32 0}
!2 = !{!"clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)"}
!3 = !{i32 1, i32 1, i32 0}
!4 = !{!"none", !"none", !"none"}
!5 = !{!"int2*", !"float4*", !"int"}
```

```
!6 = !{!"int __attribute__((ext_vector_type(2)))*", !"float __attribute__((↩
    ext_vector_type(4)))*", !"int"}
!7 = !{!"", !"", !""}
!8 = !{i32 1, i32 1, i32 1}
!9 = !{!10, !10, i64 0}
!10 = !{!"omnipotent char", !11, i64 0}
!11 = !{!"Simple C/C++ TBAA"}
```

Transformed to our graph representation and from there back to LLVM IR

```
define void @A(<2 x i32>*, <4 x float>*, i32) {
; <label>:3:
   %4 = bitcast <2 x i32>* %0 to <4 x float>*
   %5 = sext i32 %2 to i64
   %6 = getelementptr <4 x float>, <4 x float>* %4, i64 %5
   %7 = load <4 x float>, <4 x float>* %6
   store <4 x float> %7, <4 x float>* %1
   ret void
}
```

The details of this information loss are discussed in section 5.2.6.

B Translating Training Samples to Decision Sequences

The following algorithm describes the translation algorithm discussed in section 7.2 in greater detail:

Algorithm 1 Translation of a training sample to a decision sequence

```
1:  procedure ACTIONIZE(k)
2:      S_k ← ∅                                  ▷ Initial decision sequence is an empty ordered set
3:      I_phi ← ∅                                ▷ No unfinished phi instructions exist at the start
4:      for each s ∈ k do                        ▷ Loop through all structures s
5:          S_k ← S_k ∪ a_structure              ▷ Create "add structure" action
6:          S_k ← S_k ∪ ACTION(s_type)           ▷ Create actions for the type of s
7:      end for
8:      S_k ← S_k ∪ a_nostructure                ▷ Create "do not add structure" action
9:      for each g ∈ k do                        ▷ Loop through all global variables g
10:         S_k ← S_k ∪ a_structure              ▷ Create "add global" action
11:         S_k ← S_k ∪ ACTION(g_value)          ▷ Create actions for the value of g
12:     end for
13:     S_k ← S_k ∪ a_nostructure                ▷ Create "do not add structure" action
14:     for each f ∈ k do                        ▷ Loop through all functions f
15:         S_k ← S_k ∪ a_function               ▷ Create "add function" action
16:         for each a ∈ f do                    ▷ Loop through all function arguments a
17:             S_k ← S_k ∪ ACTION(a)            ▷ Create "add instruction node" action
18:             S_k ← S_k ∪ ACTION(a_type)       ▷ Create actions for the type of a
19:         end for
20:         B ← BASICBLOCKS(f)
21:         B ← BFS(B)                           ▷ Order basic blocks breath-first
22:         for each b ∈ B do
23:             for each i ∈ b do                ▷ Loop through all instructions i
24:                 S_k ← S_k ∪ a_instruction    ▷ Create "add instruction" action
25:                 S_k ← S_k ∪ ACTION(i_edge)   ▷ Create edge related actions for i
26:                 if i_opcode = phi then
27:                     S_k ← S_k ∪ ACTION(op_0) ▷ Create actions for the default operand
28:                     I_phi ← I_phi ∪ i        ▷ Add i to unfinished phis
29:                 else
30:                     for each op ∈ i do       ▷ Loop through all operands op
31:                         S_k ← S_k ∪ ACTION(op)  ▷ Create actions to construct op
32:                     end for
33:                 end if
34:             end for
35:             S_k ← S_k ∪ a_noinstruction      ▷ Create "do not add instruction" action
36:         end for
37:     end for
```

38: $S_k \leftarrow S_k \cup a_{nofunction}$ ▷ Create "do not add function" action
39: **for each** $p \in I_{phi}$ **do**
40: **for each** $op \in p \setminus \{op_0\}$ **do** ▷ Loop through all operands except the default operand
41: $S_k \leftarrow S_k \cup \text{ACTION}(op)$ ▷ Create actions to construct op
42: **end for**
43: **end for**
44: **return** S_k
45: **end procedure**

In order to keep the algorithm readable, we abbreviated multiple procedures with the function ACTION. Although our graph generation algorithm and the algorithm described here are not identical, they share far-reaching similarities. For more details on the abbreviated procedures, please refer to the discussion of our graph generation algorithm in section 4.2. Please note, however, that the explanations provided in this section may differ from Algorithm 1 in some minor points.

C Hyperparameter Optimization

The following table contains the parameters and results of the Hyperparameter search through Bayesian optimization performed in section 7.3.3.

Iteration	h_s	h_{bg}	h_{ng}	h_{bn}	h_{nn}	h_{fg}	KL-divergence	comment
0	83	2	1	1	4	2	1.87	Initial eval.
1	99	3	3	3	1	1	1.05	Initial eval.
2	69	3	3	3	4	1	0.90	Initial eval.
3	69	4	4	2	3	1	1.49	
4	84	1	1	2	4	3	0.91	
5	69	2	3	1	3	1	0.88	New opt.
6	69	1	4	1	4	2	1.42	
7	70	3	4	3	4	1	1.24	
8	83	1	1	1	4	2	0.75	New opt.
9	68	4	4	2	2	2	1.82	
10	69	4	3	4	4	1	1.13	
11	69	4	4	1	2	1	2.22	
12	69	3	3	1	3	2	0.83	
13	68	1	3	1	4	1	0.92	
14	68	1	3	1	3	1	0.72	New opt.
15	71	4	4	3	3	1	1.72	
16	68	4	3	1	2	2	1.19	
17	69	3	3	2	3	1	0.92	
18	00	0	0	0	0	0	0.98	
result	68	1	3	1	3	1	0.72	Iteration 14

Our setup tries to maximize the KL-divergence; therefore, we multiply each result by -1 before passing it to the Bayesian optimization model.

D Model Evaluation

D.1 LLVM IR of Generated Samples

In chapter 8, we evaluate the graphs generated by our domain-specific model. In section 8.1, six such graphs are depicted in Figure 8.1. The following LLVM IR is the code produced by our code generator for each of those graphs.

LLVM IR for Figure 8.1a

```
declare i64 @get_global_id(i32)
define i64 @A(i32*) {
; <label>:1:
   %2 = call i64 @get_global_id(i32 376)
   ret i64 %2
}
```

LLVM IR for Figure 8.1b

```
define [3 x i32] @A(float**, [3 x i32], i8) {
; <label>:3:
   ret [3 x i32] %1
}
```

LLVM IR for Figure 8.1c

```
define i32 @A(i32) {
; <label>:6:
   br i1 87, label %2, label %3
; <label>:7:
   ret i32 %0
   ret i32 -641
}
```

LLVM IR for Figure 8.1d

```
declare i64 @get_group_id(i32)
define void @A(i32*) {
; <label>:1:
   %2 = call i64 @get_group_id(i32 376)
   %3 = load i32, i32* %0
   ret void
}
```

LLVM IR for Figure 8.1e

```
define void @A(i8*) {
; <label>:1:
   store i8 174, i8* %0
   %2 = load i8, i8* %0
   ret void
; <label>:2:
   ret void
}
```

LLVM IR for Figure 8.1f

```
declare i64 @get_global_id(i32)
define void @A(i32*) {
; <label>:1:
   %2 = call i64 @get_global_id(i32 376)
   %3 = getelementptr i32, i32* %0, i64 %2
   ret void
}
```

D.2 Opcode Distributions

The following table contains the probability of each LLVM opcode appearing in the training set as well as in graphs and valid samples generated by our model in %.

Opcode	Training Set	Generated Graphs	Valid Samples
getelementptr	17.79	3.76	2.74
call	12.36	11.14	12.33
load	12.34	4.10	6.85

Opcode	Training Set	Generated Graphs	Valid Samples
store	12.03	12.63	11.37
ret	11.37	31.52	36.58
shl	4.82	3.10	1.64
ashr	4.75	5.74	4.27
br	4.60	4.25	4.27
add	3.99	1.80	1.92
icmp	2.80	1.26	1.37
trunc	2.42	1.53	1.51
fadd	1.89	1.49	1.51
bitcast	1.84	1.92	1.37
fmul	1.13	1.80	1.23
and	1.02	1.57	1.37
mul	0.68	1.80	1.10
phi	0.65	1.57	1.23
select	0.50	0.76	0.82
zext	0.37	0.43	0.82
uitofp	0.21	0.23	0.69
fsub	0.21	0.23	0.41
sext	0.21	0.57	0.41
insertelement	0.20	0.64	0.55
shufflevector	0.20	0.46	0.55
sitofp	0.16	0.46	0.41
sub	0.17	0.12	0.00
extractelement	0.17	0.00	0.27
fdiv	0.14	0.57	0.00
alloca	0.11	0.12	0.00
sdiv	0.11	0.12	0.41
fpext	0.11	0.34	0.00
fptosi	0.08	1.15	0.14
udiv	0.08	0.00	0.00
urem	0.08	0.00	0.41

Opcode	Training Set	Generated Graphs	Valid Samples
fptrunc	0.08	0.23	0.14
lshr	0.08	0.12	0.00
xor	0.06	0.00	0.00
fcmp	0.06	0.21	0.00
srem	0.03	0.23	0.00
fptoui	0.03	0.12	0.00
switch	0.03	0.80	0.27
or	0.03	0.00	0.41
unreachable	0.00	0.80	0.27
ptrtoint	0.00	0.34	0.14
inttoptr	0.00	0.12	0.27

Copyright Information