Program Graphs for Machine Learning

Chris Cummins[1]*, **Zacharias Fisches**[2]*, **Tal Ben-Nun**[2], **Torsten Hoefler**[2],
Hugh Leather[1], **Michael O'Boyle**[3]
[1] Facebook AI Research, [2] ETH Zürich, [3] University of Edinburgh

Abstract

We present PROGRAML† – *Program Graphs for Machine Learning* – a language-independent, portable representation of program semantics that enables analysis through deep learning. We benchmark the capabilities of PROGRAML using traditional data flow analyses that are fundamental to compiler optimizations. We assemble a dataset of 461k Intermediate Representation (IR) files for LLVM, covering five source programming languages, and 8.5B classification labels. By formulating data flow analysis as an MPNN, we show that standard analyses can be learned, significantly outperforming state-of-the-art approaches.

1 Introduction

Compiler architects increasingly look to machine learning when building heuristics for compiler optimization. The promise of automatic heuristic design, freeing the compiler engineer from the complex interactions of program, architecture, and other optimizations, is alluring. However, most machine learning methods cannot replicate even the simplest of the abstract interpretations of data flow analysis [1] that are critical to making good optimization decisions. This must change for machine learning to become the dominant technology in compiler heuristics.

Data flow algorithms act on abstract interpretations of the program, propagating information of interest through the program's control-flow graph until a fixed point is reached [2]. Prior machine learning works, on the other hand, have typically represented the entirety of the program's behavior as a fixed-length, statically computed feature vector, or as a sequence of syntactic tokens [3]. Such representations are unsuited to learning abstract interpretations of programs and so cannot avoid trivial pitfalls such as the addition of dead code [4], which changes the representation without changing the program's behavior or its response to optimizations. We refer readers to Appendix A for details of the data flow analyses used in this paper.

We propose overcoming the limitations of current machine learning techniques by making the program's control, data, and call dependencies a central part of the program's representation *and* a primary consideration when processing it. We achieve this by seeing the program as a graph in which individual statements are connected to other statements through relational dependencies. Each statement in the program is understood only in the context of the statements interacting with it. Through relational reasoning [5], a latent representation of each statement is learned that is a function of not just the statement itself, but also of the (latent) representations of its graph neighborhood. Notably, this formulation has a striking similarity to the IRs used by compilers, and the iterative propagation of information resembles the *transfer functions* and *meet operators* in traditional data flow analyses [1]. Recently proposed techniques for learning over graphs have shown promise in a number of domains [6, 7]. With a suitable representation and graph-based model, we extend these approaches to the domain of compiler analysis, enabling downstream tasks built on top of such graph models to natively incorporate reasoning about data flow into their decision making.

*Both authors contributed equally.
†Code and data available at: https://chriscummins.cc/programl

34th Conference on Neural Information Processing Systems (NeurIPS 2020), ML for Systems Workshop.

We make the following contributions:

- We propose a portable, language-independent representation of programs derived from compiler IRs. PROGRAML is the first representation to capture whole-program control-, data-, and call relations between instructions and operands as well as their order and data types. PROGRAML is a compiler-agnostic design for use at all points in the optimization pipeline; we provide implementations for LLVM and XLA IRs.
- We introduce a benchmark dataset that poses established compiler analysis tasks as supervised machine learning problems. DEEPDATAFLOW comprises five tasks and is constructed from 461k real-world program IRs covering a diverse range of domains and source languages, totaling 8.5 billion data flow analysis classification labels.
- We adapt Gated-Graph Neural Networks (GGNN) to the PROGRAML representation. We show that, within a bounded problem size, our approach achieves ≥ 0.939 F_1 score on all analysis tasks, a significant improvement over state-of-the-art learning techniques.

2 Related Work

Data flow analysis is a long established area of work. Despite its central role, there has been limited work in learning such analysis. [8] use ASTs and code synthesis to learn rule-sets for static analyses, some of which are dataflow-related. Our approach does not require a program generator or a hand-crafted DSL for rules. [9] use dynamic information (e.g., register snapshots) from instrumented binaries to embed an assembler graph representation. We propose a static approach that does not need runtime features. [10] use a graph embedding of a Static Single Assignment (SSA) form to generate invariants. The lack of function call/return edges means that the representation is not suitable for interprocedural analysis as it stands. [11] explore a large-scale, context-dependent vector embedding. This is done at a token level, however, and is unsuited for dataflow analysis. IR2Vec [12] models part-of-statements as relations. However, in order to compute the values of the embeddings, IR2Vec requires access to the type of data flow analyses that our approach learns from data alone.

Recently there have been attempts to develop program representations that allow fine-grain reasoning. However, representations based on source code and its direct artifacts [13, 14, 15, 16, 17] put unnecessary emphasis on naming and stylistic choices that may not correlate with the functionality of the code. Approaches based on IRs [18, 19, 20, 21] remove such noise but fail to capture information about the program that is important for analysis, e.g. variables [20] and [18] commutativity. In both cases, models are expected to reason about the flow of information in programs using representations that do not directly encode this information. Clearly, a program representation is needed that enables machine learning algorithms to reason about the execution of a program by developing its own data flow analyses.

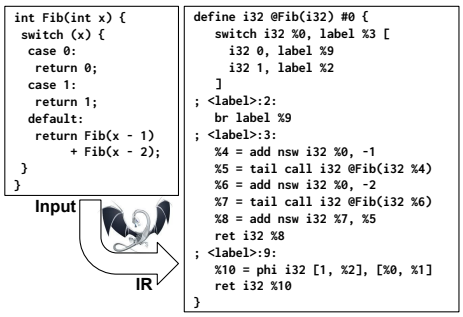
3 A Graphical Representation for Deep Program Analysis

We present PROGRAML, a novel IR-based program representation that closely matches the data structures used traditionally in data flow analysis and can be processed natively by deep learning models. We represent programs as directed multigraphs where instructions, variables, and constants are vertices, and relations between vertices are edges. Edges are typed to differentiate control-, data-, and call-flow. Additionally, we augment edges with a local position attribute to encode the order of operands to instructions, and to differentiate between divergent branches in control-flow.

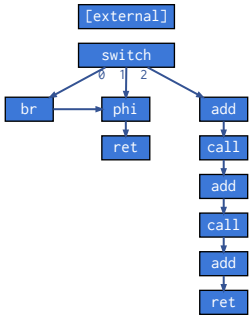
We construct a PROGRAML graph $G = (V, E)$ by traversing a compiler IR. An initially empty graph $G = \emptyset$ is populated in three stages: control-flow, data-flow, and call-flow, shown in Figure 1.

(I) Control Flow We construct the full-flow graph of an IR by inserting a vertex for each instruction and connecting control-flow edges (Fig. 1a, 1b). Control edges are augmented with a numeric position using an ascending sequence based on their order in the list of an instruction's successors.

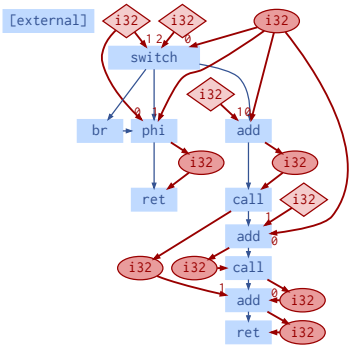
(II) Data Flow We introduce constant values and variables as graph vertices (Fig. 1c). Data-flow edges are inserted to capture the relation from constants and variables to the instructions that use them as operands, and from instructions to produced variables. Data edges have a position attribute that encodes the order of operands for instructions.



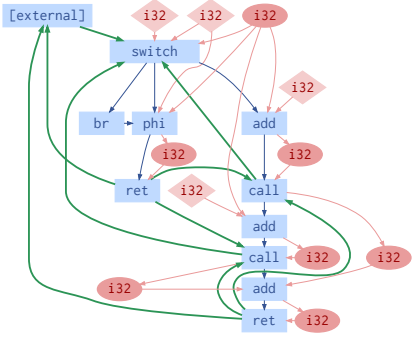
(a) The input program is passed through the compiler front-end to produce an IR. In this example, LLVM-IR is used.



(b) A full-flow graph is constructed of instructions and control dependencies. All edges have position attributes; for clarity, we have omitted position labels where not required.



(c) Vertices are added for data elements (elliptical nodes are variables, diamonds are constants). Data edges capture use/def relations. `i32` indicates 32 bit signed integers. Numbers on edges indicate operand positions.



(d) Functions have a single entry instruction and zero or more exit instructions. Call edges are inserted from call sites to function entry instructions, and return-edges from function exits to call sites.

Figure 1: PROGRAML construction from a Fibonacci implementation using LLVM-IR.

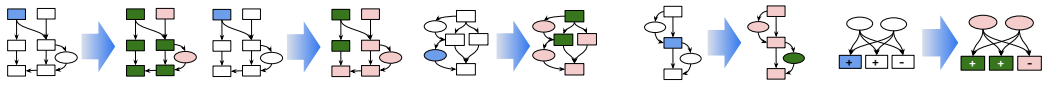
(III) Call Flow Call edges capture the relation between an instruction that calls a function and the entry point of the called function (Fig. 1d). Return edges are added from each of the terminal instructions of a function to the calling statement. For IRs that support external linkage, an additional vertex is created representing an external call site and connected to all externally visible functions.

4 Graph-based Deep Learning for Program Analysis

We formulate our system in a Message Passing Neural Network (MPNN) framework [22, 23]. Our design mimics the *transfer functions* and *meet operators* of classical iterative data flow analysis [2, 24], replacing the rule-based implementations with learnable analogues (message and update functions). Our model is an adaptation of GGNN [22] that consists of three logical phases: input encoding, message propagation and update, and result readout.

(I) Input Encoding Starting from the augmented graph representation $G = (V, E)$, we capture the semantics of the program graph vertices by mapping every instruction, constant, and variable vertex $v \in V$ to a vector representation $h_v^0 \in \mathbb{R}^d$ by lookup in a fixed-size embedding table. The mapping from vertex to learnable embedding vector $f : v \mapsto h_v^0$ must be defined for each IR.

For LLVM-IR, we construct an embedding key from each vertex using the name of the instruction, e.g., `store`, and the data type for variables and constants, e.g., `i32*` (a pointer to a 32-bit integer). In this manner we derive the set of unique embedding keys for training and deployment using the graph vertices of a training set of LLVM-IRs. An *unknown element* embedding is used during deployment to map embedding keys which were not observed in the training data. The embedding vectors are trained jointly with the rest of the model.



(a) REACHABILITY (b) DOMINANCE (c) DATADEP (d) LIVENESS (e) SUBEXPRESSIONS

Figure 2: Example input-output graphs for each of the five DEEPDATAFLOW tasks. A single vertex is randomly selected from as the starting point for computing analysis results, indicated as the blue node. Each output vertexis annotated with a binary value after the analysis has completed. As a supervised classification task, the goal is to predict the output vertex labels given an input graph. These small graphs are illustrative, DEEPDATAFLOW graphs comprise an average 581 vertices.

(II) Message Propagation Each iteration step is divided into a message propagation step followed by a vertex state update. Received messages $M(h_w^{t-1}, e_{wv})$ are a function of neighboring states and the respective edges. Messages are mean-aggregated over the neighborhood after transformation with a custom position-augmented transfer function that scales h_w elementwise with a position-gating vector $p(e_{wv})$:

$$M(h_w^{t-1}, e_{wv}) = W_{\text{type}(e_{wv})}\Big(h_w^{t-1} \odot p(e_{wv})\Big) + b_{\text{type}(e_{wv})}$$

The position-gating $p(e_{wv}) = 2\sigma(W_p \operatorname{emb}(e_{wv}) + b_p)$ is implemented as a sigmoid-activated linear layer mapping from a constant sinusoidal position embedding [25, 26]. It enables the network to distinguish non-commutative operations such as division, and branches of diverging control-flow. We add backward edges for each edge in the graph as separate edge-types to allow for reverse-propagation of information, which is necessary for backward compiler analyses. In all our experiments, we employ Gated Recurrent Units (GRU) [27] as our update function.

Step (II) is iterated T times to extract vertex representations that are contextualized with respect to the given graph structure.

(III) Result Readout We support per-instruction and per-variable classification tasks using a *readout head* on top of the iterated feature extraction, mapping, for each vertex, the extracted vertex features h_v^T to probabilities $R_v(h_v^T, h_v^0) = \sigma\left(f(h_v^T, h_v^0)\right) \cdot g(h_v^T)$, where $f(\cdot)$ and $g(\cdot)$ are linear layers and $\sigma(\cdot)$ is the sigmoid activation function.

5 Learning Data Flow Anlayses

We pose a suite of data flow analyses as supervised learning tasks. These particular data flow analyses can already be perfectly solved by non-ML techniques. Here, we use them to benchmark the capabilities of machine learning techniques for reasining about optimizations.

Dataset We assembled DEEPDATAFLOW, a 256M-line corpus of LLVM-IR files from a variety of sources and produced labeled datasets using five traditional data flow analyses, described in Appendix A. Each of the 15.4M analysis examples consists of an input graph in which a single vertex is annotated as the root node for analysis, and an output graph in which each vertex is annotated with a binary label corresponding to its value once the data flow analysis has completed (Fig. 2). A 3:1:1 ratio is used to divide the examples for the five problems into training, validation, and test instances. We release DEEPDATAFLOW for open use.

Models We evaluate the effectiveness of our approach against two contrasting state-of-the-art approaches for learning over programs: *inst2vec* [18], an LSTM-based approach for whole-program classification that we extend to per-variable predictions, and *CDFG* [20], a graph-based approach.

For CDFG and PROGRAML we use 32 dimensional embeddings as in [20]. Input *vertex-selectors*, encoded as binary one-hot vectors, are used to mark the starting point for analyses and are concatenated to the initial random embeddings. MPNNs typically use a small number of propagation steps out of practical consideration for efficiency [23, 22, 20]. In contrast, data flow analyses iterate until a fixed point is reached. In this work we iterate for a fixed number T of message passing steps and exclude from the training set graphs for which a traditional implementation of the analysis task

Table 1: F_1 scores for data flow analysis.

Analysis	Example Optimization	inst2vec	CDFG	PROGRAML		
		DDF-30	DDF-30	DDF-30	DDF-60	DDF
Reachability	Dead Code Elimination	0.012	**0.998**	**0.998**	0.997	0.943
Dominance	Global Code Motion	0.004	0.999	**1.000**	0.991	0.123
DataDep	Instruction Scheduling	—	—	**0.997**	0.993	0.965
Liveness	Register Allocation	—	—	**0.937**	0.939	0.625
Subexpressions	Global Common Subexpression Elimination	0.000	0.009	**0.996**	0.967	0.959

requires greater than T iterations to solve. We set $T = 30$ for training all models, using 1M graphs, evaluated on a fixed 10k validation set at 10k intervals for the first 50k training graphs, and at 100k intervals thereafter. The checkpoint with the greatest validation F_1 score is used for testing.

5.1 Evaluation

Vocabulary Coverage Each of the three approaches uses a vocabulary to produce embeddings that describe the instructions and operands of a program. Our approach to deriving a vocabulary from training graphs provides 98.3% coverage of programs in the test set, an improvement of $2.9\times$ and $2.1\times$ over the vocabularies used by inst2vec and CDFG, respectively.

DDF-30 We initially limit our testing to the subset of each task's test set which can be solved using a traditional analysis implementation in ≤ 30 steps, denoted DDF-30. Table 1 summarizes the performance of inst2vec, CDFG, and PROGRAML. The relational representation of our approach shows excellent performance. Neither CDFG or inst2vec representations enable per-variable classification, so are incapable of the DATADEP and LIVENESS tasks. CDFG, which also captures control-flow, achieves comparable performance on two of the tasks. However, the lack of operand vertices, positional edges, and data types renders poor performance on the SUBEXPRESSIONS task. PROGRAML correctly labels $4.50\times$ and $1.12\times$ more nodes than the state-of-the-art approaches.

DDF-60 and DDF The DDF-30 set excludes 28.7% of DEEPDATAFLOW graphs that require more than 30 steps to compute ground truth labels. To test whether the learned models can generalize to solve larger problems, we repeated the experiments using the DDF-30 models on on all graphs which require ≤ 60 analysis steps (excluding 19.6%), doubling the number of inference steps. We observe that performance is consistent on this larger problem set, demonstrating that MPNNs generalize to problems larger than those they were trained on. Finally, we test the DDF-30 models on all graphs, shown as DDF. We use $T = 200$ inference message passing iterations to test the limits of stability and generalization. We see substantial degradations of model performance in line with two challenges of forumlating data flow analysis in an MPNN framework: first, that using a fixed number of message passing iterations across each and every edge leads to unnecessary work for problems that can be solved in fewer iterations or by propagating only along a dynamic subset of the edges at each timestep (the maximum number of steps required by a graph in DDF is 28,727). Second, models that compute correct results for a graph when processed for an appropriate number of steps may prove unstable when processed for an excessively large number of steps. We believe that *dynamically-sparse* message passing strategies and an adaptive number of iterations could address these scalability challenges. We will pursue an extension to the MPNN formulation in future work.

6 Conclusions

The evolution of ML for compilers requires more expressive representations. We show that current techniques cannot reason about simple data flows which are at the core of all compilers. We present PROGRAML, a graph-based representation of programs derived from compiler IRs that enables enhanced program reasoning through machine learning[1]. We are releasing the DEEPDATAFLOW dataset as a community benchmark for evaluating approaches to learning over programs. PROGRAML and DEEPDATAFLOW open up new directions for research towards more flexible and useful program analysis. PROGRAML outperforms the state-of-the-art, but is limited by scalability issues imposed by MPNNs. As future work, we will investigate how MPNNs could be improved to learn efficient and stable fixed-point algorithms, regardless of input graph size.

[1] In prior work we show improved performance on downstream optimization reasoning tasks [28].

References

[1] G. A. Kildall. A Unified Approach to Global Program Optimization. In *POPL*, 1973.

[2] J. B. Kam and J. D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7(3), 1977.

[3] Z. Wang and M. O'Boyle. Machine learning in Compiler Optimization. *Proceedings of the IEEE*, 106(23), 2018.

[4] F. Barchi, G. Urgese, E. Macii, and A. Acquaviva. Code Mapping in Heterogeneous Platforms Using Deep Learning and LLVM-IR. In *DAC*. ACM, 2019.

[5] P. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational Inductive Biases, Deep Learning, and Graph Networks. *arXiv:1806.01261*, 2018.

[6] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. Modeling Relational Data with Graph Convolutional Networks. In *ESWC*, 2018.

[7] Z. Ziwei, P. Cui, and W. Zhu. Deep Learning on Graphs: A Survey. *TKDE*, 2020.

[8] P. Bielik, V. Raychev, and M. Vechev. Learning a Static Analyzer from Data. In *CAV*, 2017.

[9] Z. Shi, K. Swersky, D. Tarlow, P. Ranganathan, and M. Hashemi. Learning Execution through Neural Code Fusion. In *ICLR*, 2020.

[10] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. Learning Loop Invariants for Program Verification. In *NeurIPS*, 2018.

[11] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. Learning and Evaluating Contextual Embedding of Source Code. In *ICML*, 2020.

[12] V. Keerthy S, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Spkant. IR2Vec: A Flow Analysis based Scalable Infrastructure for Program Encodings. *arXiv:1909.06228*, 2019.

[13] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end Deep Learning of Optimization Heuristics. In *PACT*. IEEE, 2017.

[14] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning Distributed Representations of Code. In *POPL*, 2018.

[15] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt. Learning to Represent Edits. *arXiv:1810.13337*, 2018.

[16] A. Haj-Ali, N. K. Ahmed, T. Willke, S. Shao, K. Asanovic, and I. Stoica. NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning. *CGO*, 2020.

[17] C. Cummins, P. Petoumenos, W. Zang, and H. Leather. Synthesizing Benchmarks for Predictive Modeling. In *CGO*. IEEE, 2017.

[18] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *NeurIPS*, 2018.

[19] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device Placement Optimization with Reinforcement Learning. In *ICML*, 2017.

[20] A. Brauckmann, S. Ertel, A. Goens, and J. Castrillon. Compiler-Based Graph Representations for Deep Learning Models of Code. In *CC*, 2020.

[21] C. Cummins. *Deep Learning for Compilers*. PhD thesis, University of Edinburgh, 2020.

[22] Y. Li, R. Zemel, M. Brockscmidt, and D. Tarlow. Gated Graph Sequence Neural Networks. *arXiv:1511.05493*, 2015.

[23] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural Message Passing for Quantum Chemistry. In *ICML*. PMLR, 2017.

[24] K. D. Cooper, T. J. Harvey, and K. Kennedy. Iterative Data-flow Analysis, Revisited. Technical report, Department of Computer Science, Rice University, 2004.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. In *NIPS*, 2017.

[26] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. Convolutional Sequence to Sequence Learning. In *ICML*. PMLR, 2017.

[27] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *EMNLP*, 2014.

[28] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather. ProGraML: Graph-based Deep Learning for Program Optimization and Analysis. *arXiv:2003.10536*, 2020.

A Data Flow Analysis

Data flow analysis is an iterative technique for gathering information about possible sets of values at various points in a computer program. The value sets and iteration rules depend on the particular analysis being performed. Our initial release of Deep Data Flow contains five data flow analyses:

(I) REACHABILITY: Reachable Instructions Control reachability is a fundamental compiler analysis which determines the set of points in a program that can be reached from a particular starting point. Given succ(n), which returns the control successors of an instruction n, the set of reachable instructions starting at root n can be found using forward analysis:

$$\text{Reachable}(n) = \{n\} \cup_{p \in \text{succ}(n)} \text{Reachable}(p) \tag{1}$$

(II) DOMINANCE: Instruction Dominance Instruction n dominates statement m if every control-flow path the from the program entry n_0 to m passes through n. Like reachability, this analysis only requires propagation of control-flow, but unlike reachability, the set of dominator instructions are typically constructed through analysis of a program's reverse control-flow graph Lengauer1979,Blazy2015:

$$\text{Dom}(n) = \{n\} \cup (\cap_{p \in \text{pred}(n)} \text{Dom}(p)) \tag{2}$$

Where pred(n) returns the control predecessors of instruction n. We formulate the DOMINANCE problem as: Given a root instruction vertex n, label all vertices m where $n \in \text{Dom}(m)$.

(III) DATADEP: Data Dependencies The data dependencies of a variable v is the set of predecessor instructions that must be evaluated to produce v. Computing data dependencies requires traversing the reverse data-flow graph:

$$\text{DataDep}(n) = \text{defs}(n) \cup (\cup_{p \in \text{defs}(n)} \text{DataDep}(p)) \tag{3}$$

Where defs(n) returns the instructions that produce the operands of n.

(IV) LIVENESS Live-out variables A variable v is live-out of statement n if there exists some path from n to a statement that uses v, without redefining it. Given uses(n), which returns the operand variables of n, and defs(n), which returns defined variables, the live-out variables can be computed forwards using:

$$\text{LiveOut}(n) = \cup_{s \in \text{succ}(n)} \text{uses}(s) \cup \big(\text{LiveOut}(s) - \text{defs}(s)\big) \tag{4}$$

(V) Global Common Subexpressions The identification of common subexpressions is an important analysis for optimization. For compiler IRs we define a subexpression as an instruction and its operands, ordered by either their position (for non-commutative operations), or lexicographically (for commutative operations). We thus formulate the common subexpression problem as: Given an instruction (which forms part of a subexpression), label any other instructions in the program which compute the same subexpression. This is an inter-procedural analysis, though operands must obey their scope. Common subexpressions are typically identified using available expression analysis:

$$\text{Avail}(n) = \text{uses}(n) \cup (\cap_{p \in \text{pred}(n)} \text{Avail}(p)) - \text{defs(n)} \tag{5}$$

Where uses(n) return the expressions used by instruction n, and defs(n) returns the expressions defined by n.