PROGRAML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations

Group 6
Zheyu Zhang, Yunchi Lu, Xueming Xu

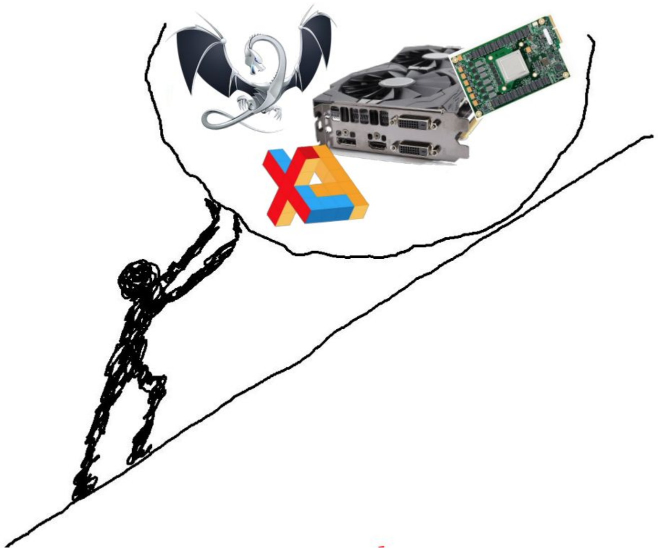
Agenda

- Motivation
- Method
- Experiments & Results

Motivation

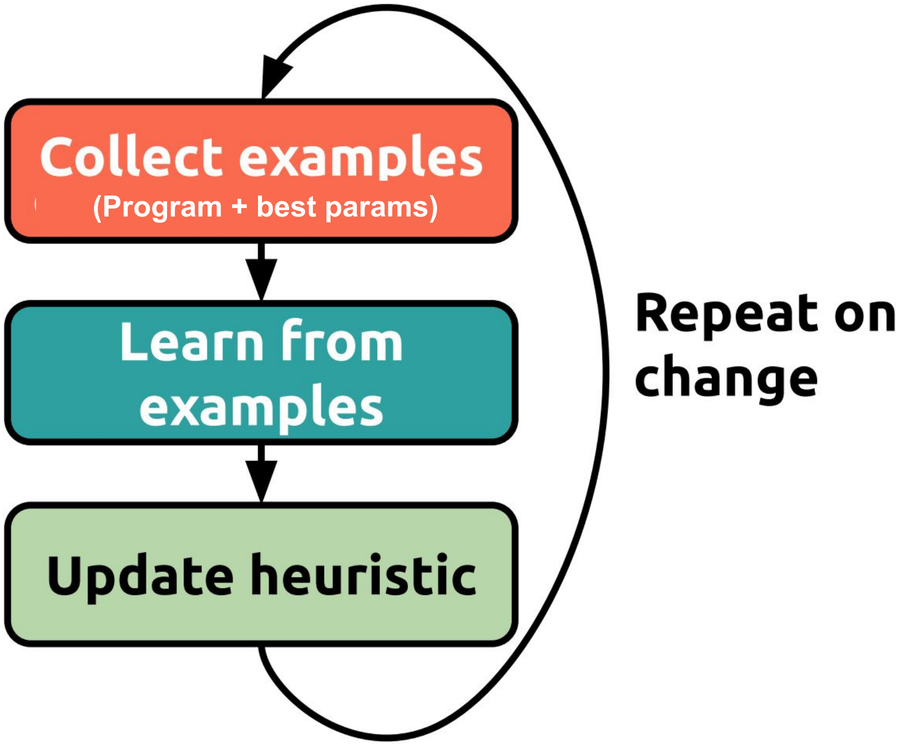
Tuning Compiler Challenge

- 1000s of variables
- Limited by domain expertise
- Compiler/Hardware keeps changing
- Widening performance gap

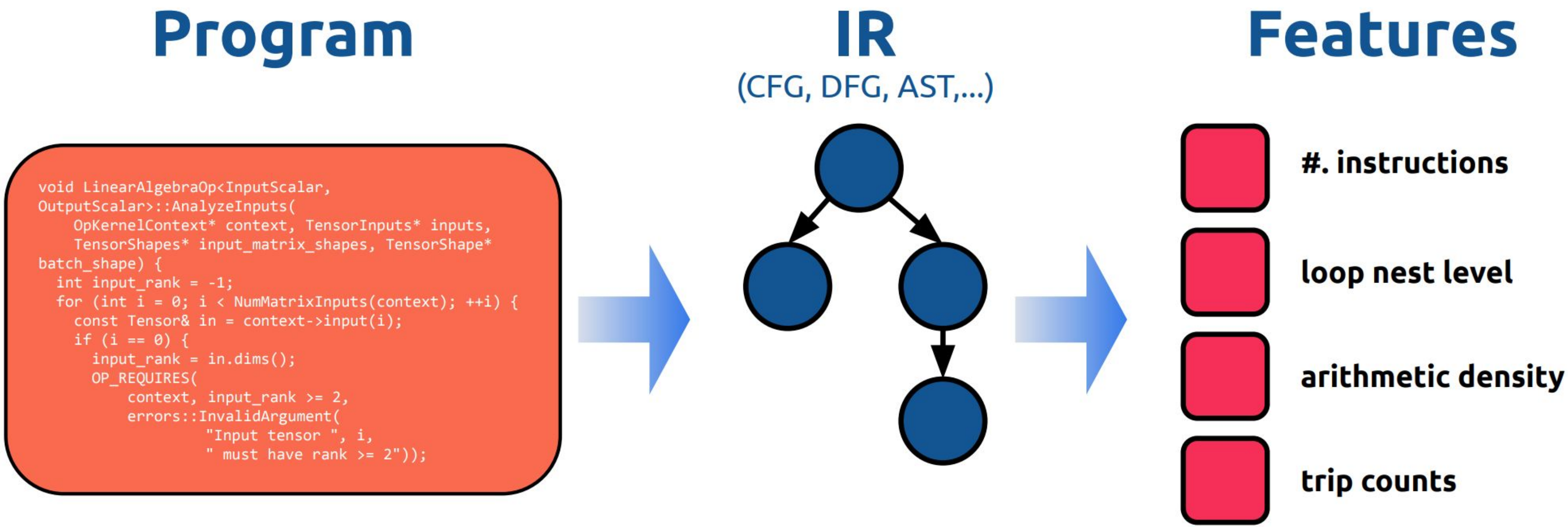


Machine Learning in compiler optimization

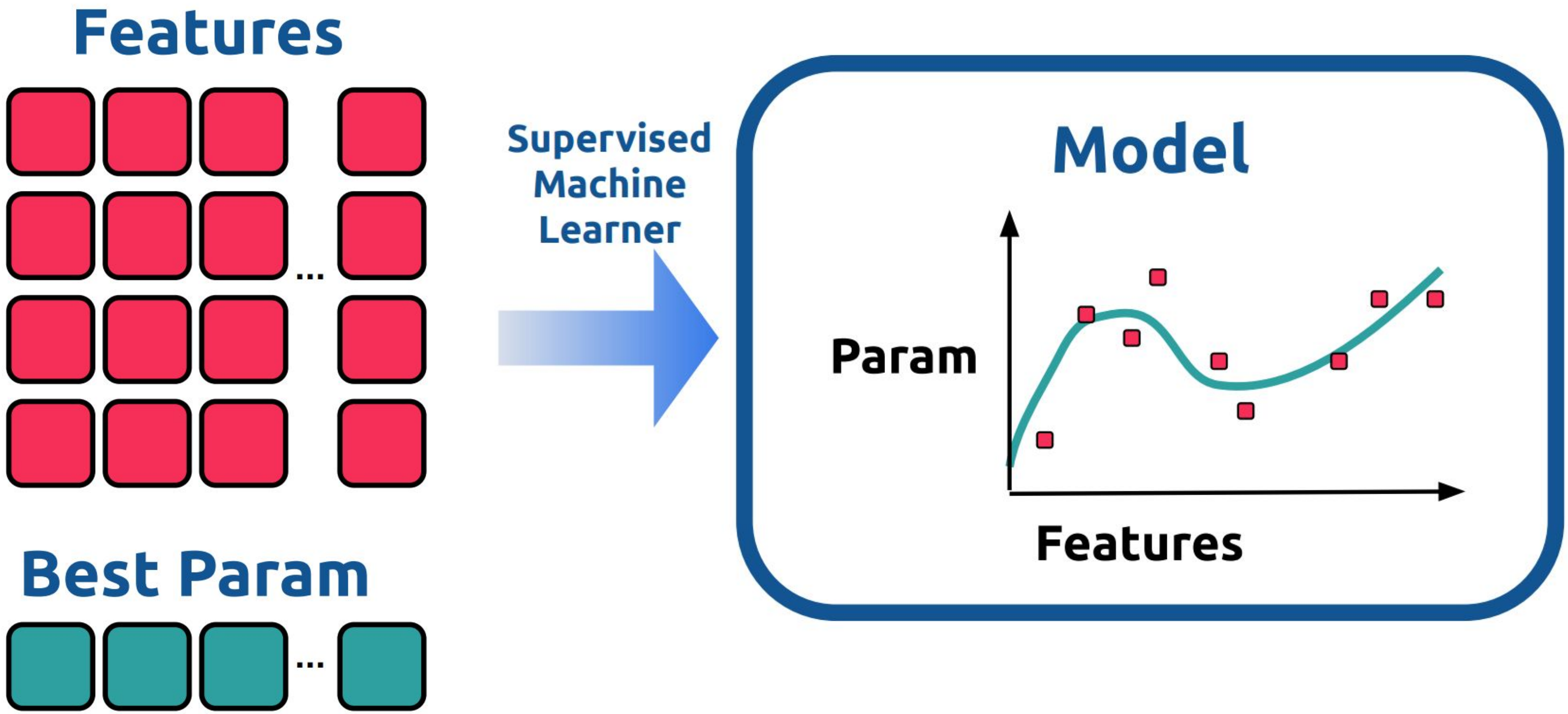
- Turned into Data Science/Machine Learning Problems



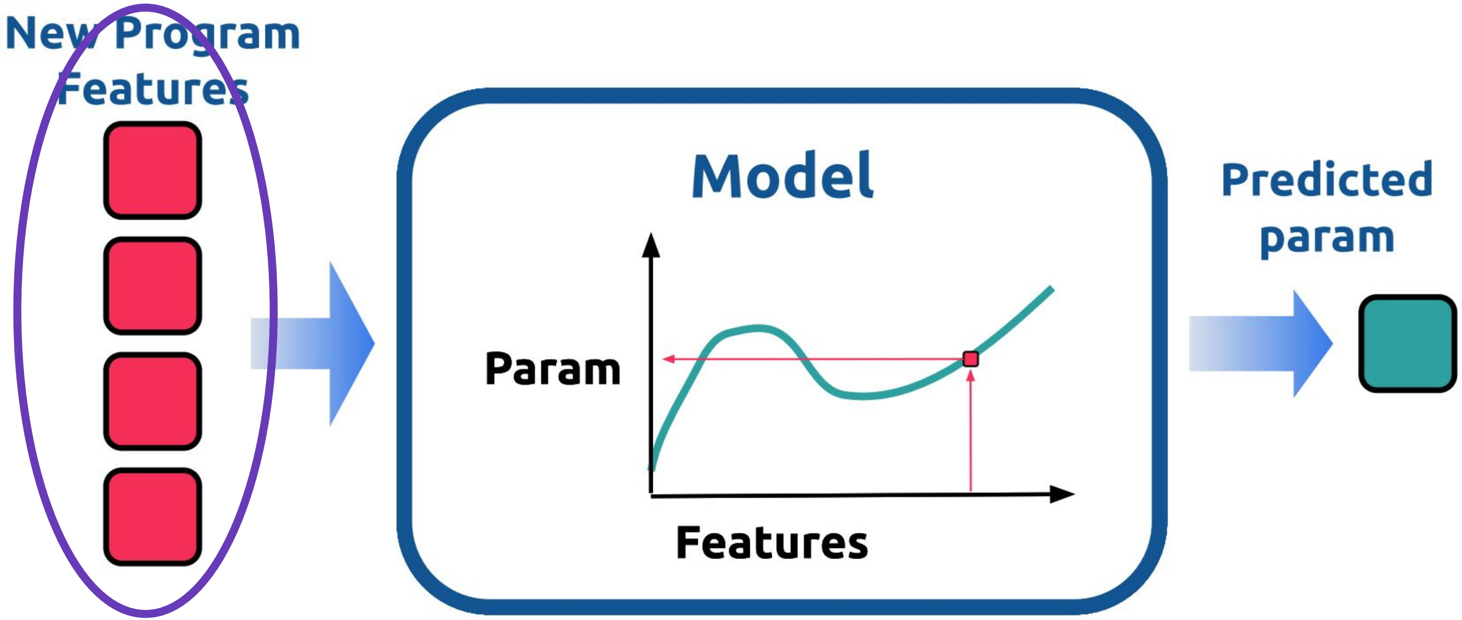
Traditional Machine Learning in compiler optimization



Traditional Machine Learning in compiler optimization



Traditional Machine Learning in compiler optimization



Hard to select!

Very successful! Typically outperforms human experts

[Wang et. al. 2018]

Machine Learning “without” features (Attempt #1)

Treat “Program” as Natural Language (NLP problem) [Cummins et al., PACT 17]

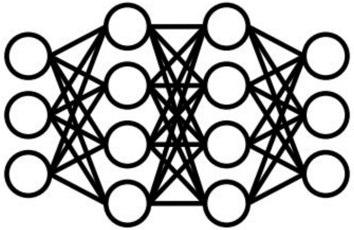
```
kernel void A(global float* a, const float b) {
  a[get_global_id(0)] *= 3.14 + b;
}
```

Token	Index
kernel	0
[space]	1
void	2
A	3
(	4
global	5
float	6
*	7
a	8

Token	Index
,	9
const	10
b	11
)	12
{	13
\n	14
[	15
get_global_id	16
0	17

0 1 2 1 3 4 5 1

LSTM/
Transformer

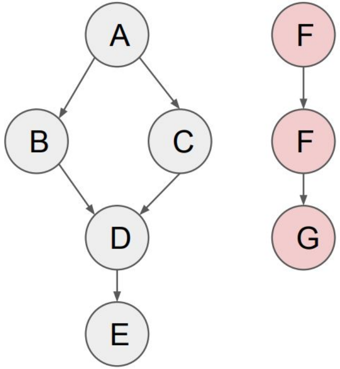


Optimization Decision

Machine Learning “without” features (Attempt #1)

Problem: Source code is highly structured

Feature vectors are easy to fool (e.g. insert dead code)

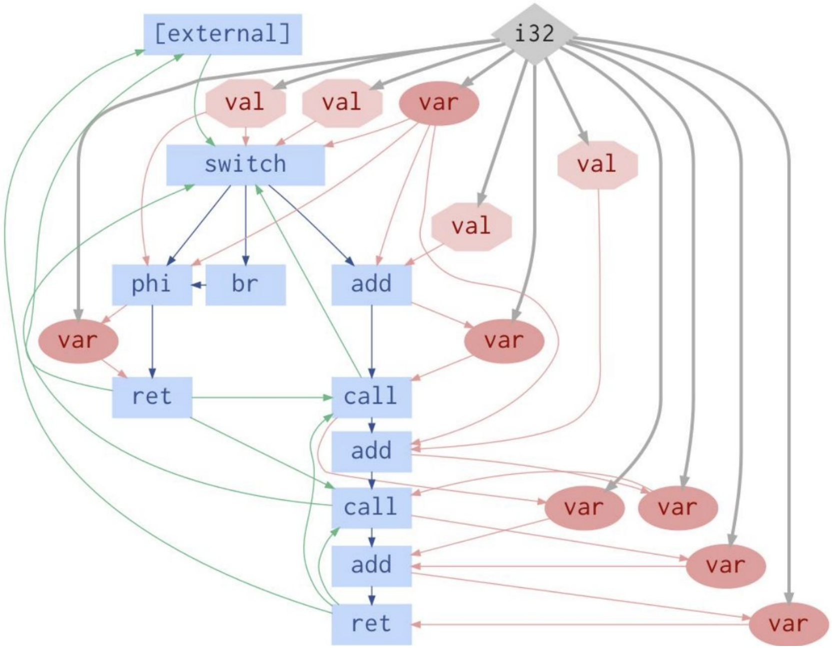


Sequential representations fail on non-linear relations, long-range deps

```
void A(int a) {
  int b = init();
  //
  // ... 1000 lines
  //
  //
  return b - a;
}
```

Machine Learning “without” features (Today’s paper)

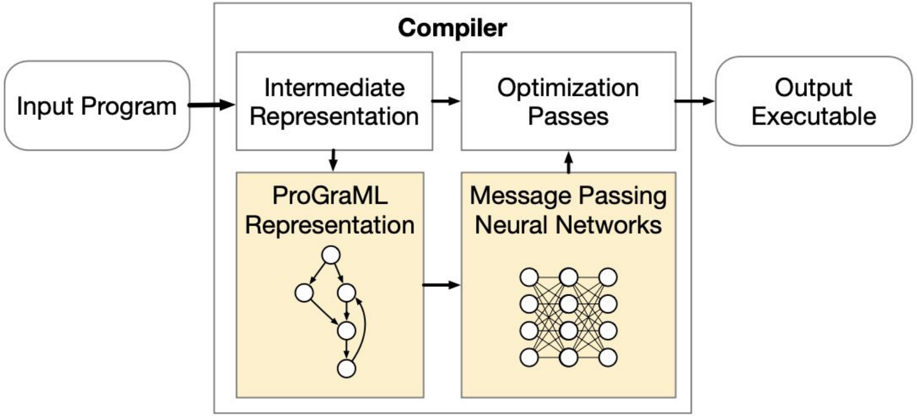
```
int Fib(int x) {
 switch (x) {
  case 0:
   return 0;
  case 1:
   return 1;
  default:
   return Fib(x - 1)
        + Fib(x - 2);
 }
}
```



Methods

Program Graphs for Machine Learning (ProGraML)

- **General-purpose** representation of programs for optimization tasks.
- **Task independent** - capture structured relations fundamental to program reasoning (i.e. data flow analysis)
- **Language independent** - derived from compiler IRs
- **Main procedure:**



Building ProGraML: Code to IR

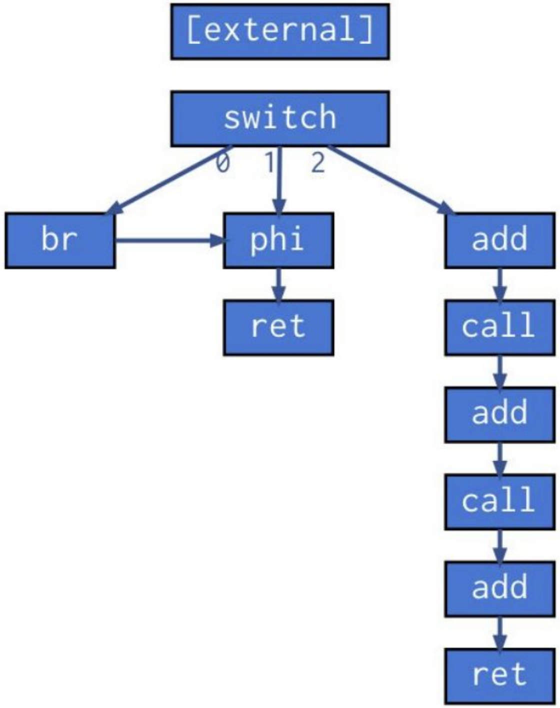
- Input program passed through the compiler front-end to produce an IR (e.g., LLVM IR)
- Why IR?
  - Language agnostic
  - Closer to what compiler sees

```c
int Fib(int x) {
 switch (x) {
  case 0:
   return 0;
  case 1:
   return 1;
  default:
   return Fib(x - 1)
        + Fib(x - 2);
 }
}
```

```llvm
define i32 @Fib(i32) #0 {
 switch i32 %0, label %3 [
   i32 0, label %9
   i32 1, label %2
 ]

; <label>:2:
 br label %9

; <label>:3:
 %4 = add nsw i32 %0, -1
 %5 = tail call i32 @Fib(i32 %4)
 %6 = add nsw i32 %0, -2
 %7 = tail call i32 @Fib(i32 %6)
 %8 = add nsw i32 %7, %5
 ret i32 %8

; <label>:9:
 %10 = phi i32 [ 1, %2 ], [ %0, %1 ]
 ret i32 %10
}
```

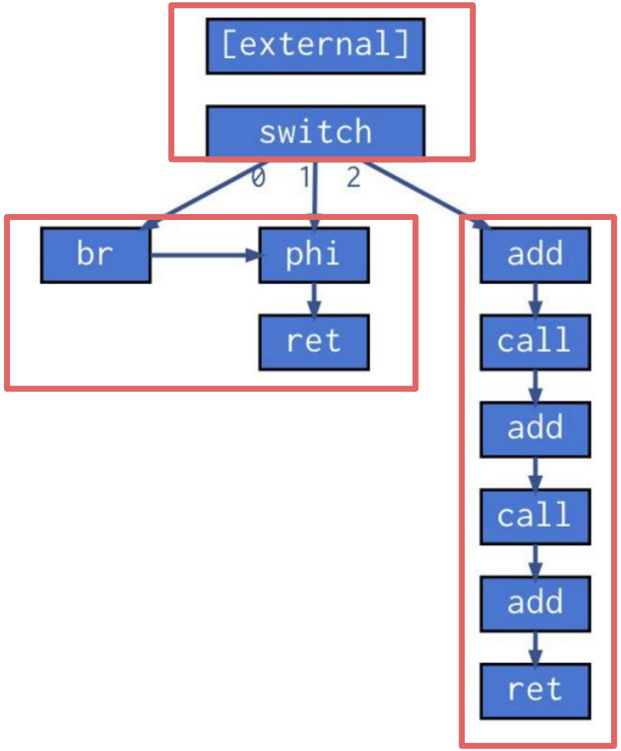
Building ProGraML: Control-flow

- Graph constructed of **instructions** and **control dependencies**.



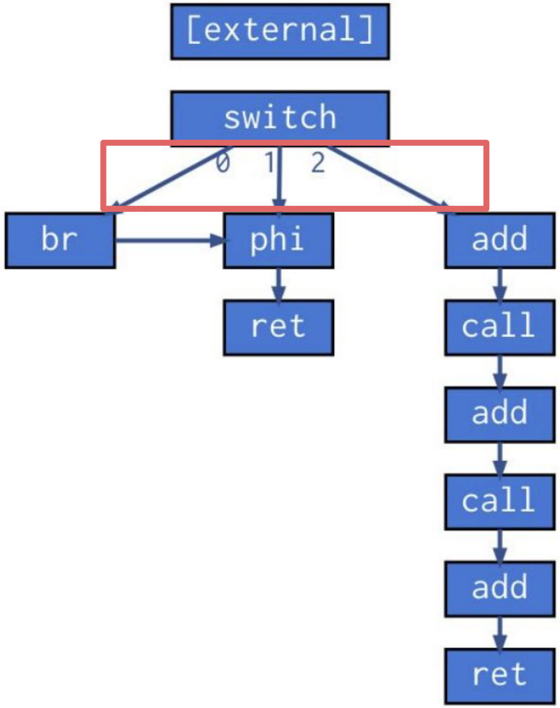
Building ProGraML: Control-flow

- Full-flow-graph
  - **Node** represents **instruction**.
  - **Node label** is the **instruction name**.



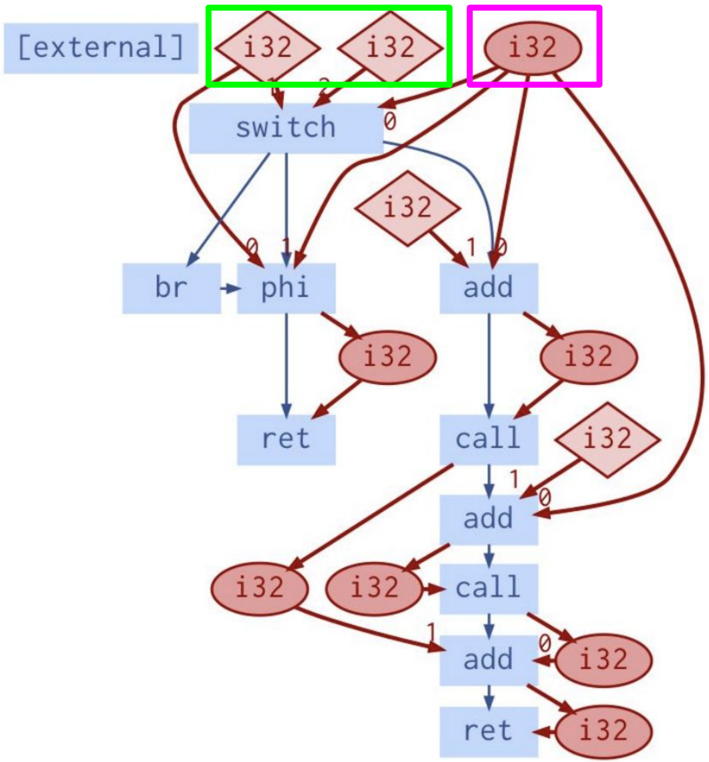
Building ProGraML: Control-flow

- Full-flow-graph
  - Node represents instruction.
  - Node label is the instruction name.
  - **Edges** are **control-flow.**
  - Edge position attribute for **branching control-flow.**



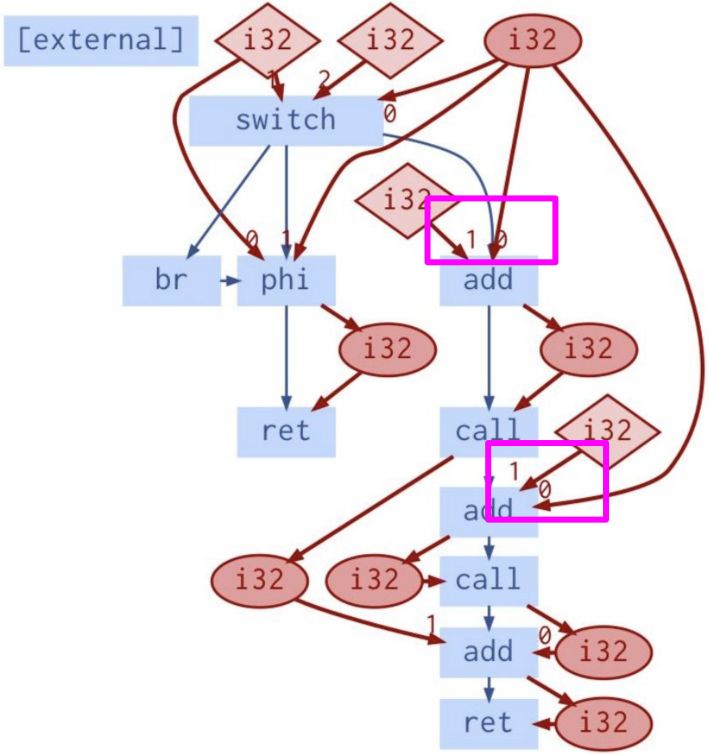
Building ProGraML: Data-flow

- Add nodes for **constants** (diamonds) and **variables** (oblongs).



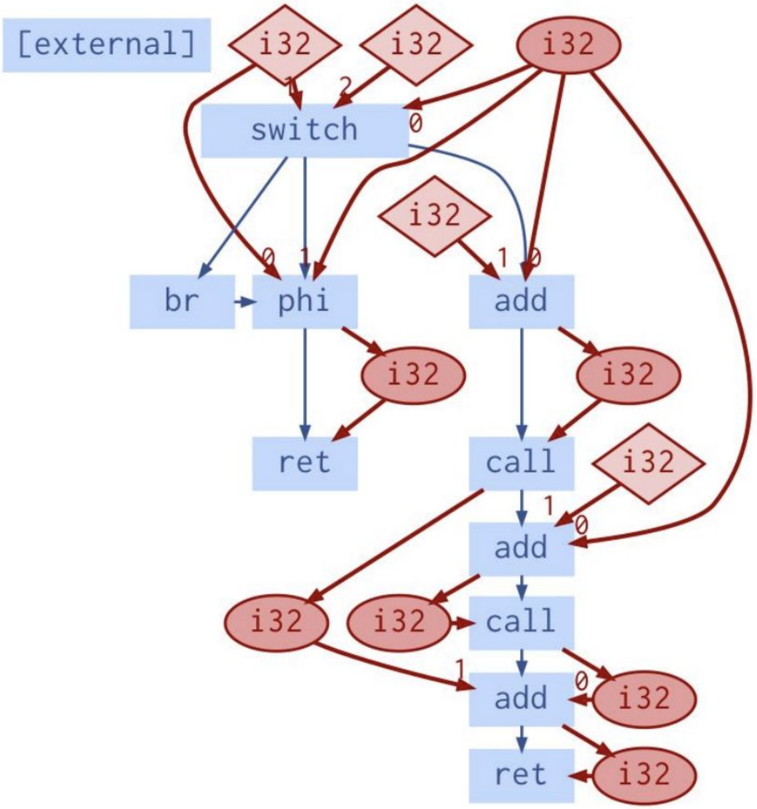
Building ProGraML: Data-flow

- Add nodes for constants (diamonds) and variables (oblongs).
- **Edges** are **data-flow**.
- **Edge position attribute for operand order.**



Building ProGraML: Data-flow

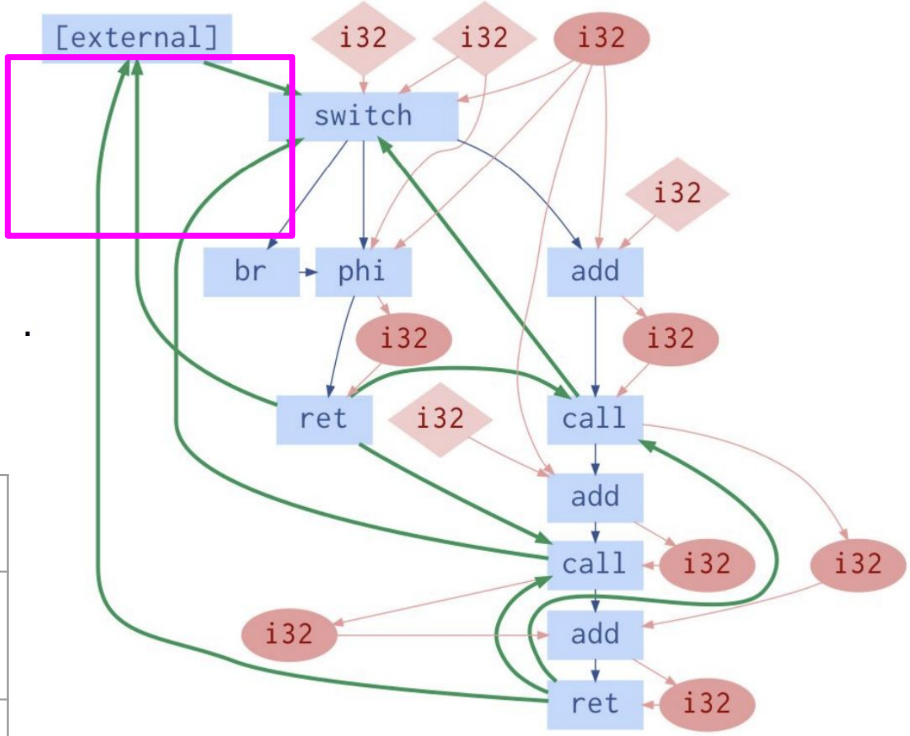
Elliptical	Variables
Diamonds	Constants
Data edges	Use/def relations
i32	32 bit signed integers
Numbers on edges	Operand positions



Building ProGraML: Call-flow

- **Edges** are **call-flow.**
- Inbound edge to **function entry instruction.**
- Outbound edge from (all) **function exit instruction**(s)

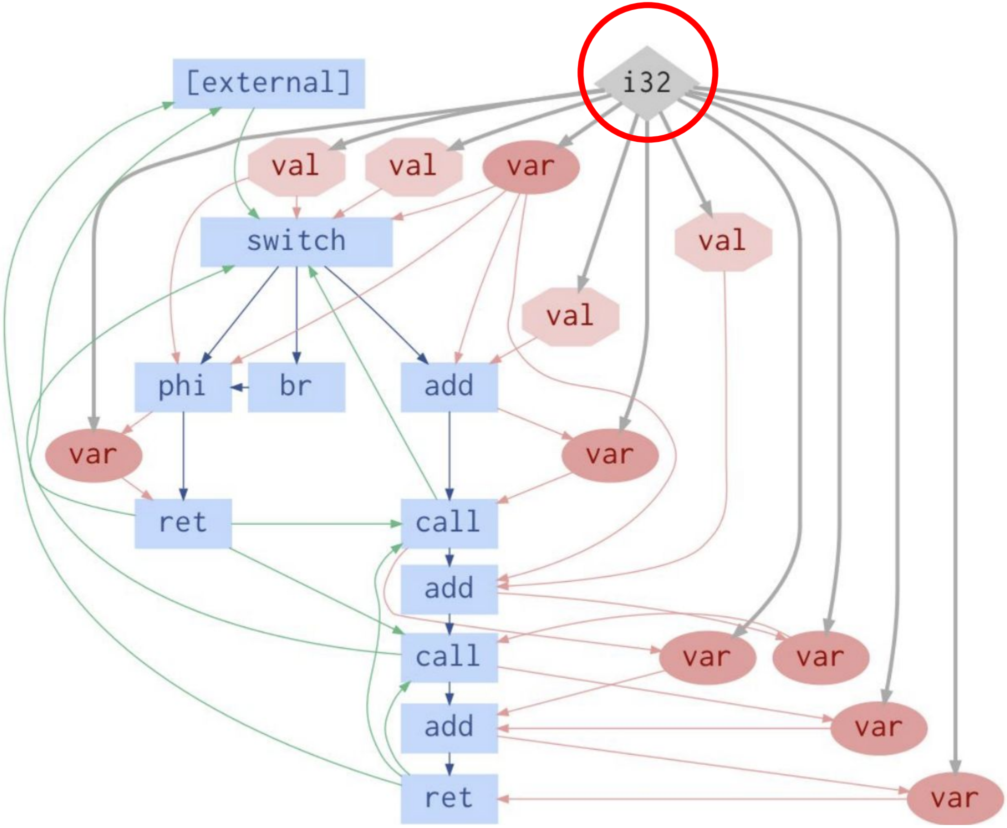
	from	**to**
Call edges	Call sites	Function entry instructions
Return edges	Function exits	Call sites



Building ProGraML: Types

- Nodes represent **types**, Edges are **instances**.

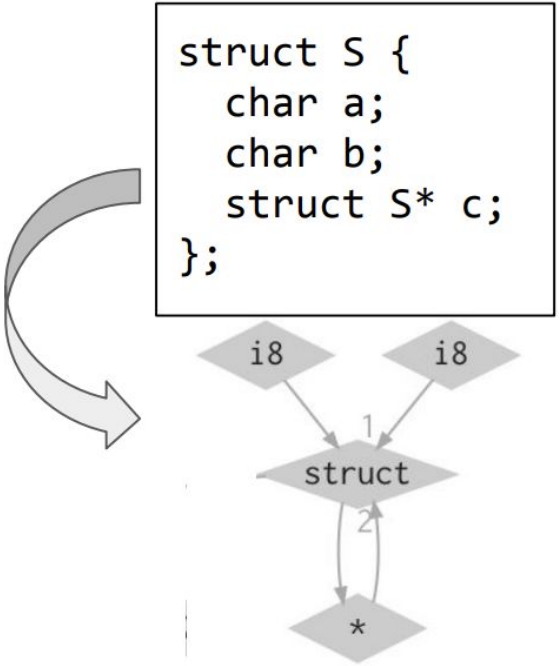
- Types are composable.
- Edge position per field.



Building ProGraML: Types

- Nodes represent types, Edges are instances.

- Types are **composable**. Edge position corresponds to the field.



Learning with ProGraML: Input encoding

- Map instruction, constant, and variable node to vector embedding
- Use node labels as embedding keys

- **Derive** vocab from set of unique vertex labels on **training graphs**.
- Separate type/instruction nodes leads to **compact vocab**
  - excellent coverage on unseen programs compared to prior approaches

	Vocabulary Size	Vocabulary Test Coverage
inst2vec	8,565	34.0%
CDFG	75	47.5%
PROGRAML	2,230	**98.3%**

Without types

- inst2vec:
  combined instruction+operands

```
i32 <id> = a<id> <int8>
```

- CDFG:
  uses only instructions for vocab, ignores data

Learning with ProGraML:
Message propagation: Gated Graph Neural Networks (GGNNs)

Position gating to differentiate control branches and operand order

- **Message Passing function**

$$m_v^t = \sum_{w \in \mathcal{N}(v)} W_{\text{type}(e_{wv})}\left(h_w^{t-1} \odot p(e_{wv})\right) + b_{\text{type}(e_{wv})}$$

- **Update function (Gated Recurrent Unit (GRU))**

$$h_v^t = \text{GRU}\left(h_v^{t-1}, m_v^t\right)$$

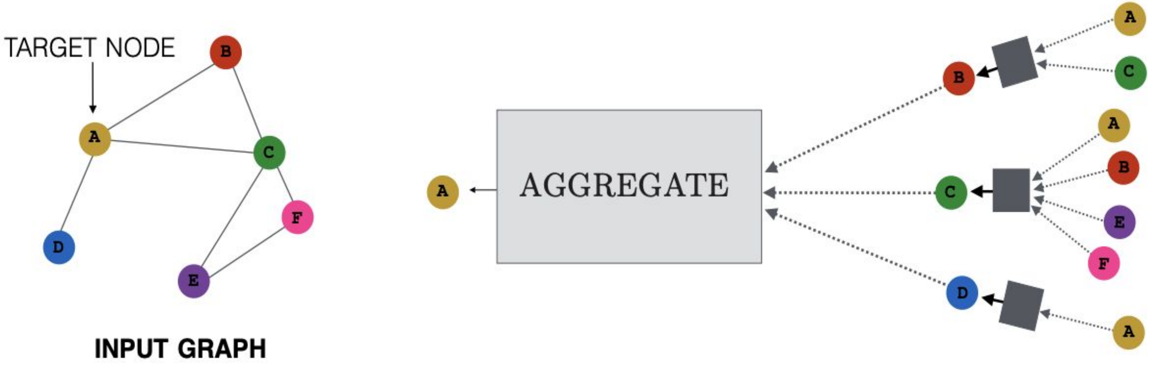


Figure: Message passing example. Adapted from (Hamilton, 2022)

Learning with ProGraML: Result Readout

- **Readout Head**
  - **Node-level inference** (e.g., per-statement/identifier classification)

$$R_v\left(h_v^T, h_v^0\right) = \sigma\left(i\left(h_v^T, h_v^0\right)\right) \cdot j\left(h_v^T\right)$$

per-node prediction after T message-passing steps

sigmoid function

feed-forward NNs

  - **Graph-level classification** (e.g., whole-program classification)

$$R_G\left(\{h_v^T, h_v^0\}_{v \in V}\right) = \sum_{v \in V} R_v\left(h_v^T, h_v^0\right)$$

Experiments & Results

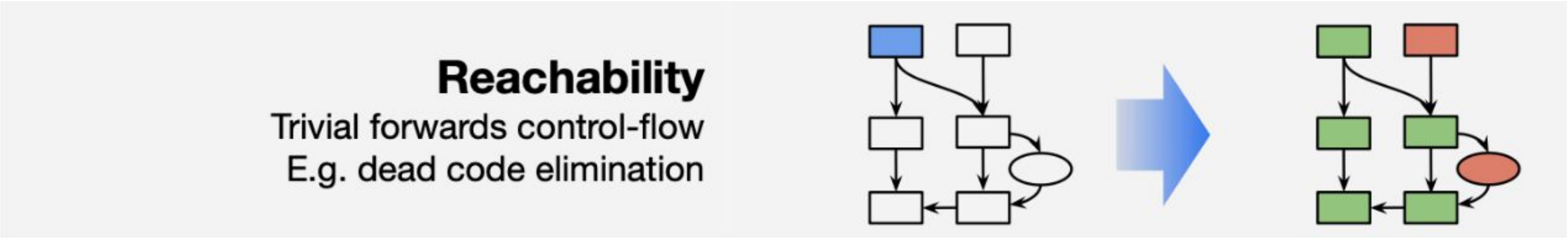
Exp: Deep Data Flow Analysis

- Model: Gated-Graph Neural Networks

Exp: Deep Data Flow Analysis

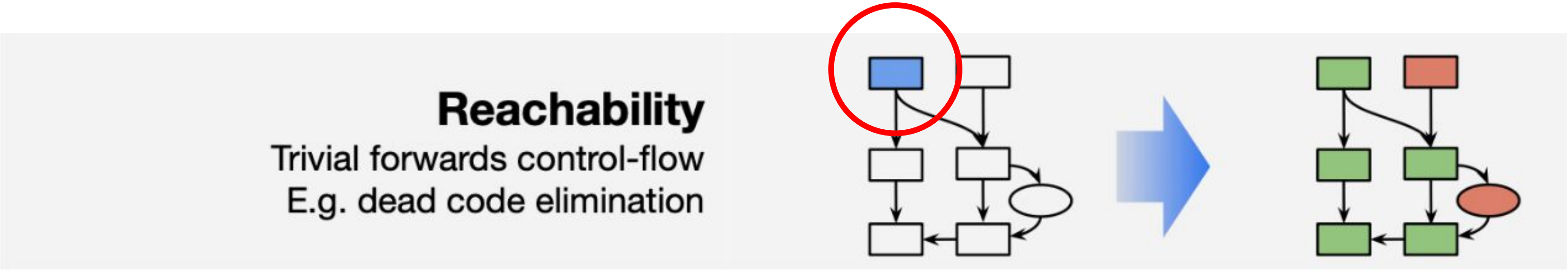
- Model: Gated-Graph Neural Networks
- Training Type: Supervised Classification Task

Exp: Deep Data Flow Analysis

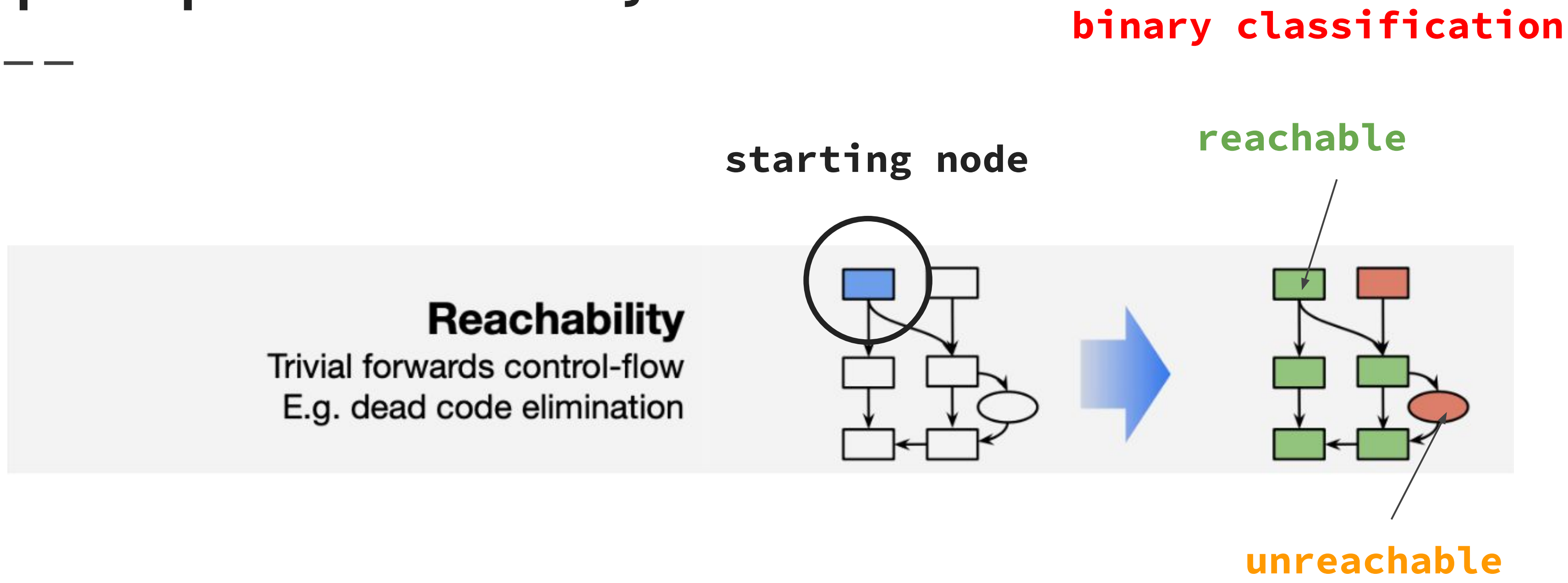


Exp: Deep Data Flow Analysis

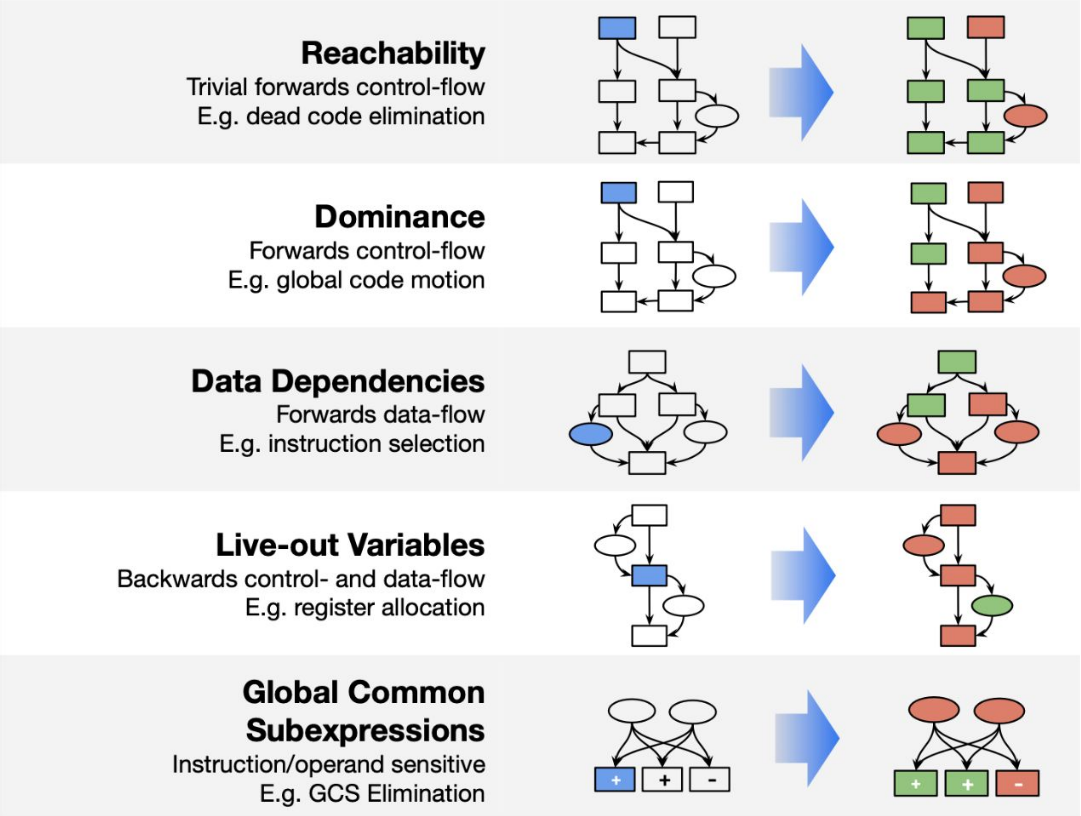
starting node



Exp: Deep Data Flow Analysis



Exp: Deep Data Flow Analysis

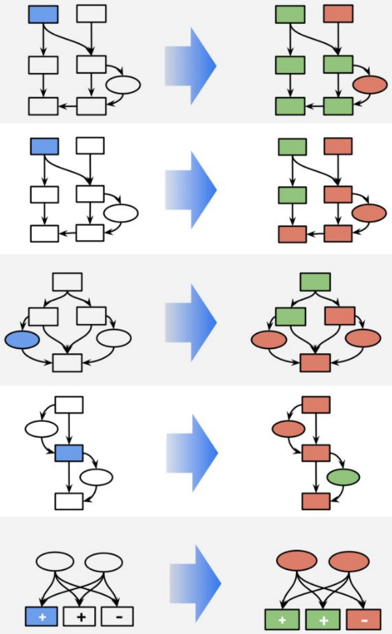


$$\text{F1 Score} = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

Exp: Deep Data Flow Analysis

Deep Data Flow

Dataset: 450k LLVM-IRs covering 5 programming languages



	F1 scores		
	inst2vec	CDFG	ProGraML
Reachability Trivial forwards control-flow E.g. dead code elimination	0.012	**0.998**	0.998
Dominance Forwards control-flow E.g. global code motion	0.004	0.999	1.000
Data Dependencies Forwards data-flow E.g. instruction selection	-	-	0.997
Live-out Variables Backwards control- and data-flow E.g. register allocation	-	-	0.937
Global Common Subexpressions Instruction/operand sensitive E.g. GCS Elimination	0.000	0.009	0.996

$$\text{F1 Score} = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

Exp: Deep Data Flow Analysis

Deep Data Flow

Dataset: 450k LLVM-IRs covering 5 programming languages

	F1 scores: inst2vec	CDFG	ProGraML
Reachability Trivial forwards control-flow E.g. dead code elimination	0.012	**0.998**	0.998
Dominance Forwards control-flow E.g. global code motion	0.004	0.999	1.000
Data Dependencies Forwards data-flow E.g. instruction selection			0.997
Live-out Variables Backwards control- and data-flow E.g. register allocation			0.937

cannot reason about variables

inst2vec: combined instruction+operands

```
i32 <id> = a<id> <int8>
```

CDFG: uses only instructions for vocab, ignores data

Exp: Deep Data Flow Analysis Caveat: Limited Problem Size

- Traditional iterative dataflow analysis: iterate until a fixed point is reached
  - meet functions & transfer functions

Exp: Deep Data Flow Analysis Caveat: Limited Problem Size

- Traditional iterative dataflow analysis: iterate until a fixed point is reached
  - meet functions & transfer functions
- **T := number of iterations to solve dataflow analysis for a piece of program**

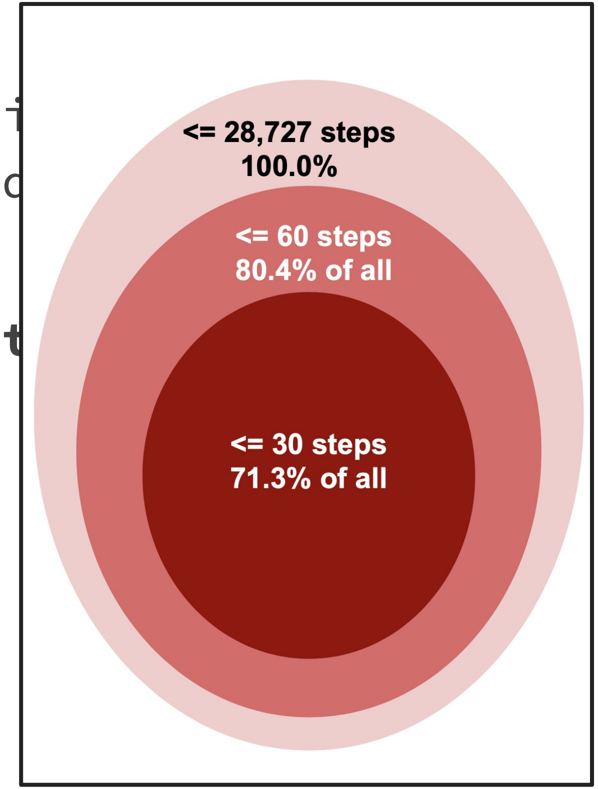
Exp: Deep Data Flow Analysis Caveat: Limited Problem Size

- Traditional iterative dataflow analysis: iterate until a fixed point is reached
  - meet functions & transfer functions
- **T := number of iterations to solve dataflow analysis for a piece of program**
- Restriction of previous results:
  - **only trained on examples with T <= 30**
  - inference step set to be T = 30
  - excluding 28.7% larger programs

Exp: Deep Data Flow Analysis

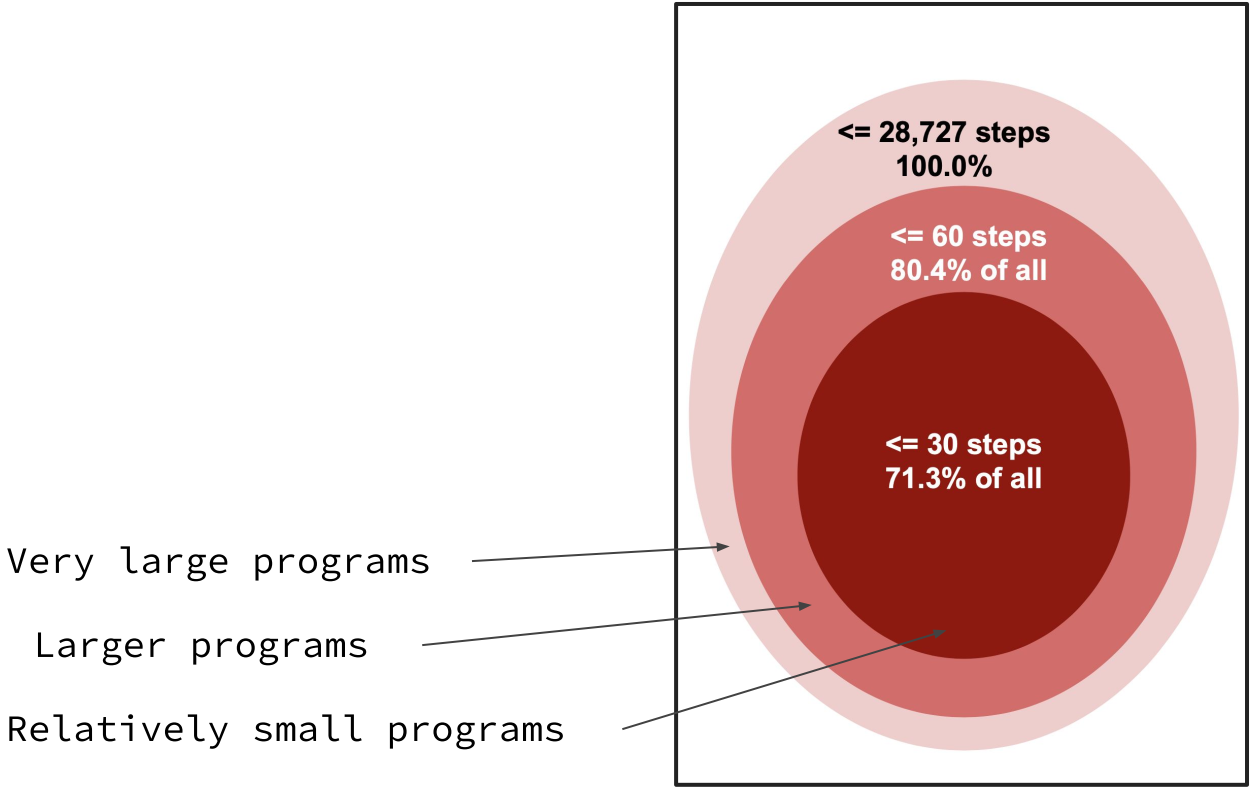
Caveat: Limited Problem Size

- Traditional iterative dataflow analysi
  iterate until a fixed point is reache
  - meet functions & transfer functions
- **T := number of iterations to solve dat**
  a piece of program
- Restriction of previous results:
  - **only trained on examples with T <= 30**
  - inference step set to be T = 30
  - excluding 28.7% larger programs



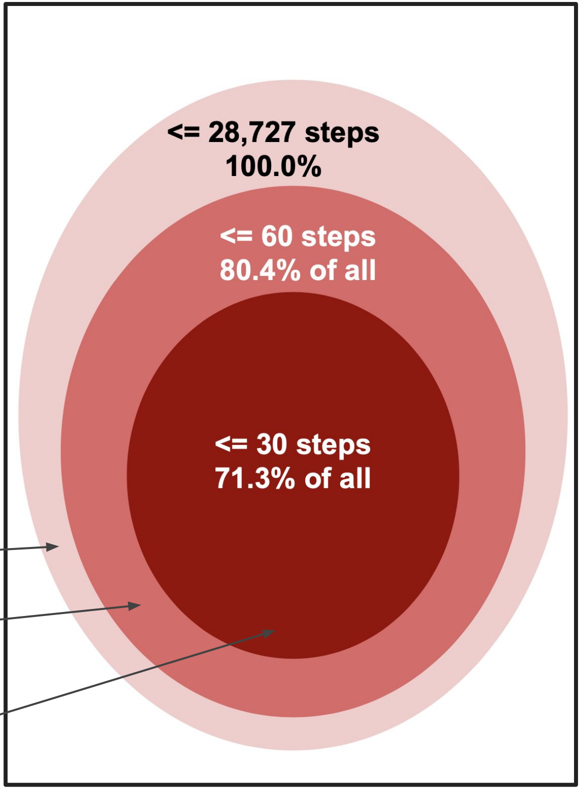
Exp: Deep Data Flow Analysis

Caveat: Limited Problem Size



Exp: Deep Data Flow Analysis Caveat: Limited Problem Size

Q: Is the prediction still accurate when the input program is larger?



Very large programs

Larger programs

Relatively small programs

Exp: Deep Data Flow Analysis

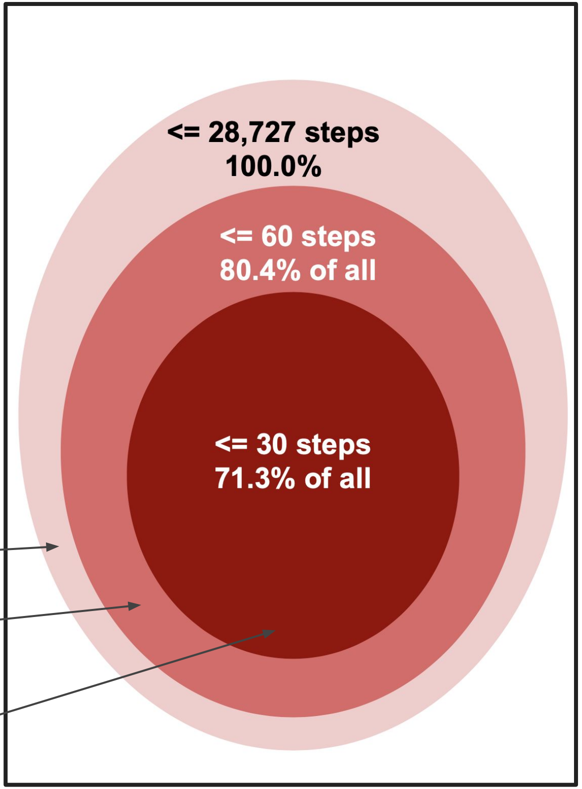
Caveat: Limited Problem Size

Q: Is the prediction still accurate when the input program is larger?

- Model is unchanged: only trained on T = 30
- During inference step is set T = 60, T = 200

Validation
- Very large programs
- Larger programs

Relatively small programs



Exp: Deep Data Flow Analysis — Scaling to Larger Problems

Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

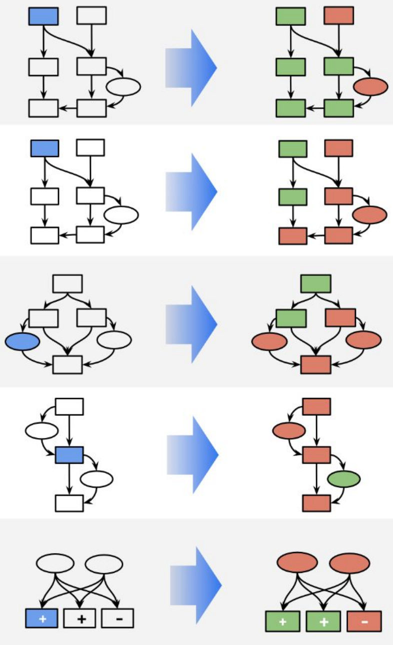
		F1 scores		
		30 timesteps	60 timesteps	200 timesteps
Reachability Trivial forwards control-flow E.g. dead code elimination		**0.998**	0.997	0.943
Dominance Forwards control-flow E.g. global code motion		**1.000**	0.991	0.123
Data Dependencies Forwards data-flow E.g. instruction selection		**0.997**	0.993	0.965
Live-out Variables Backwards control- and data-flow E.g. register allocation		0.937	**0.939**	0.625
Global Common Subexpressions Instruction/operand sensitive E.g. GCS Elimination		**0.996**	0.967	0.959

Exp: Deep Data Flow Analysis — Scaling to Larger Problems

Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

	F1 scores: 30 timesteps	60 timesteps	200 timesteps
Reachability Trivial forwards control-flow E.g. dead code elimination	**0.998**	0.997	0.943
Dominance Forwards control-flow E.g. global code motion	**1.000**	0.991	0.123
Data Dependencies Forwards data-flow E.g. instruction selection	**0.997**	0.993	0.965
Live-out Variables Backwards control- and data-flow E.g. register allocation	0.937	**0.939**	0.625
Global Common Subexpressions Instruction/operand sensitive E.g. GCS Elimination	**0.996**	0.967	0.959



Exp: Deep Data Flow Analysis Scaling to Larger Problems

Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

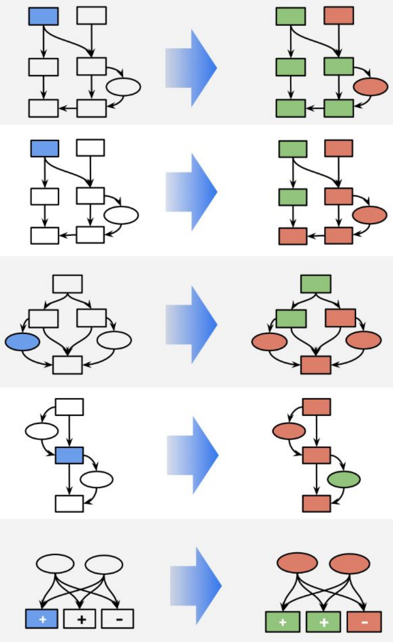
		F1 scores		
		30 timesteps	60 timesteps	200 timesteps
Reachability Trivial forwards control-flow E.g. dead code elimination		**0.998**	0.997	0.943
Dominance Forwards control-flow E.g. global code motion		**1.000**	0.991	0.123
Data Dependencies Forwards data-flow E.g. instruction selection		**0.997**	0.993	0.965
Live-out Variables Backwards control- and data-flow E.g. register allocation		0.937	**0.939**	0.625
Global Common Subexpressions Instruction/operand sensitive E.g. GCS Elimination		**0.996**	0.967	0.959

Exp: Deep Data Flow Analysis   Scaling to Larger Problems

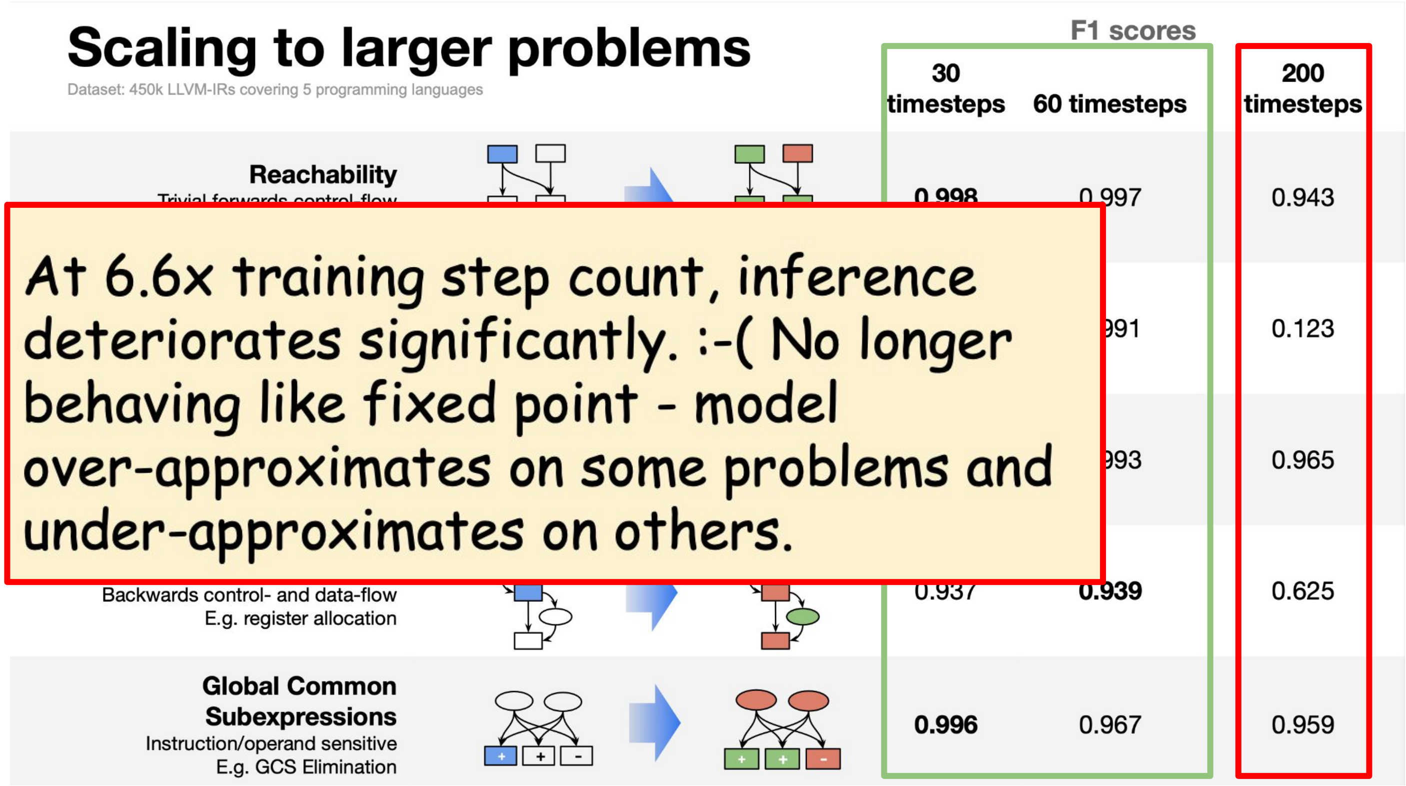
Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

	F1 scores		
	30 timesteps	60 timesteps	200 timesteps
Reachability Trivial forwards control-flow E.g. dead code elimination	**0.998**	0.997	0.943
Dominance Forwards control-flow E.g. global code motion	**1.000**	0.991	0.123
Data Dependencies Forwards data-flow E.g. instruction selection	**0.997**	0.993	0.965
Live-out Variables Backwards control- and data-flow E.g. register allocation	0.937	**0.939**	0.625
Global Common Subexpressions Instruction/operand sensitive E.g. GCS Elimination	**0.996**	0.967	0.959



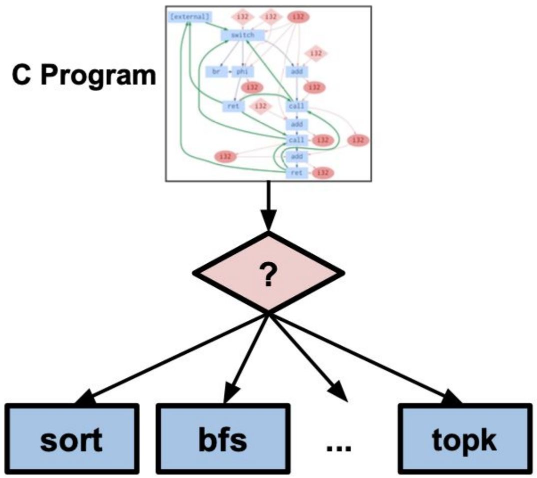
Exp: Deep Data Flow Analysis   Scaling to Larger Problems



At 6.6x training step count, inference deteriorates significantly. :-(No longer behaving like fixed point - model over-approximates on some problems and under-approximates on others.

Exp: Downstream Tasks

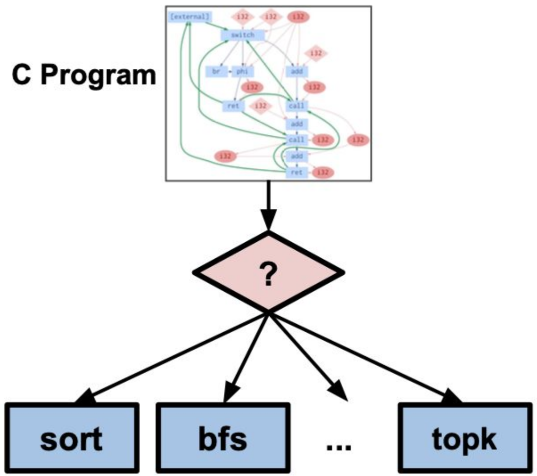
1. Algorithm Classification



1.35× improvement over state-of-art

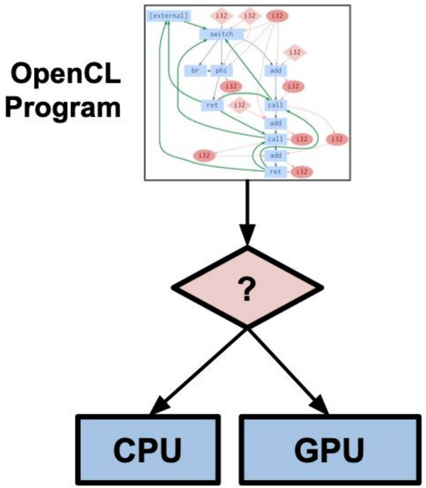
Exp: Downstream Tasks

1. Algorithm Classification



1.35× improvement over state-of-art

2. Heterogeneous Device Mapping



1.20× improvement over state-of-art

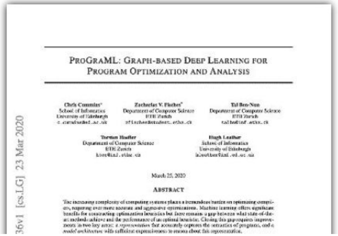
Conclusion

Conclusion

- ProGraML is expressive
- Compact embedding vocab with high test coverage
- Significant improvement on data flow analysis
- Limited by scalability issues imposed by MPNNs

References

- Paper
  https://chriscummins.cc/pub/2021-icml.pdf


PROGRAML: GRAPH-BASED DEEP LEARNING FOR PROGRAM OPTIMIZATION AND ANALYSIS

ABSTRACT

- Author Slides
  https://spcl.inf.ethz.ch/Publications/.pdf/programl-icml21-slides.pdf

- Author Lecture
  https://www.youtube.com/watch?v=cHElgMSOPFs

YouTube

Q&A