

ProGraML: Graph-based Deep Learning for Program Optimization and Analysis.

Chris Cummins
Facebook AI Research

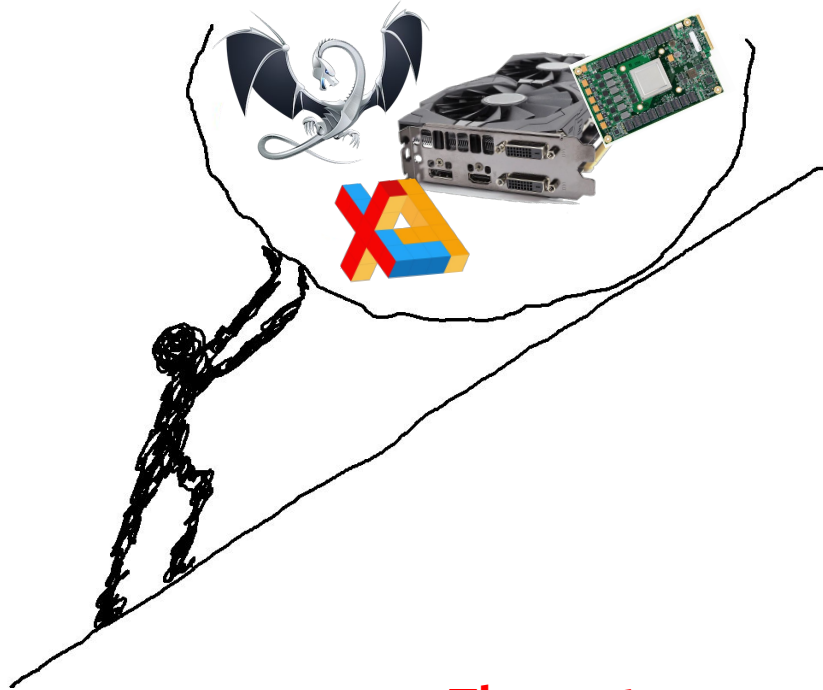
"machine learning for compilers for machine learning"

Compilers



**Machine
Learning**

Tuning optimizing compilers...



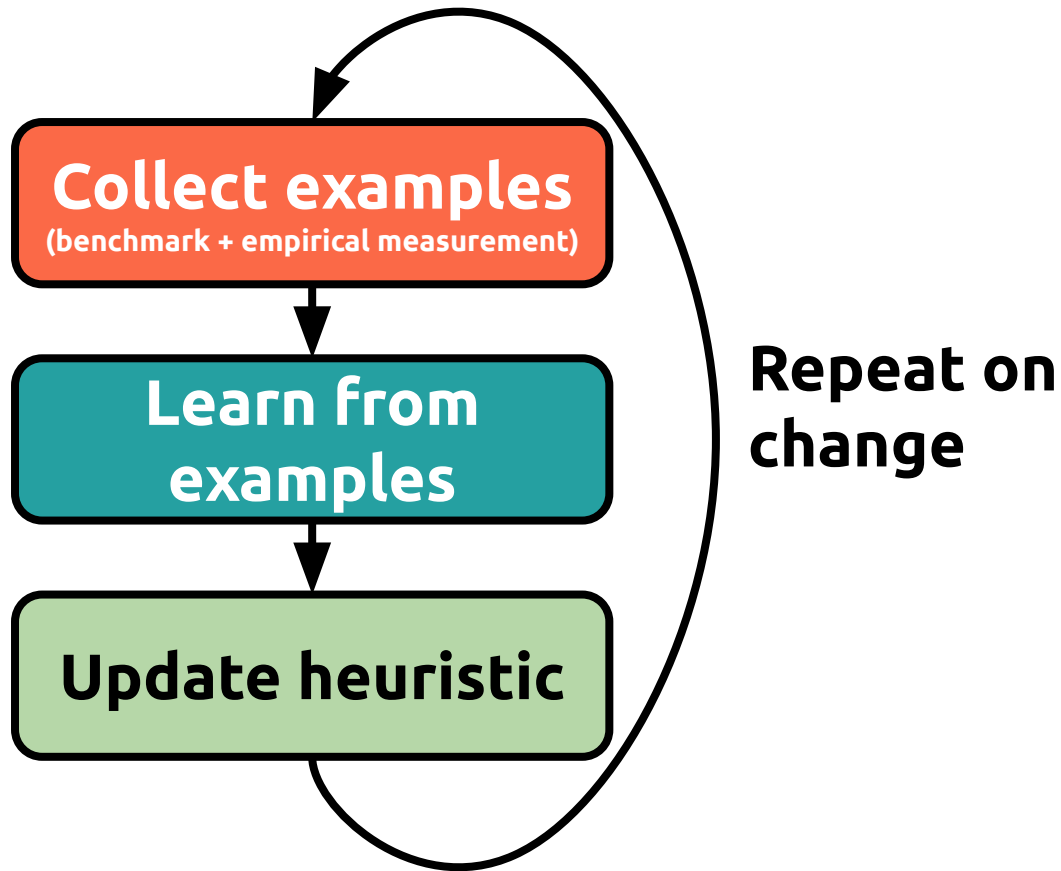
The problem

- 1000s of variables
- Limited by domain expertise
- Compiler / HW keeps changing

The cost

- Bad heuristics
- Wasted energy, \$\$\$
- Widening performance gap

"Build an optimizing compiler, your code will be fast for a day.
Teach a compiler to optimize ... "



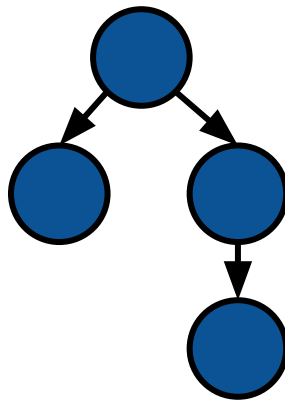
Summarize the program

Program

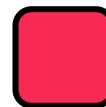
```
void LinearAlgebraOp<InputScalar,
OutputScalar>::AnalyzeInputs(
    OpKernelContext* context, TensorInputs* inputs,
    TensorShapes* input_matrix_shapes, TensorShape*
batch_shape) {
    int input_rank = -1;
    for (int i = 0; i < NumMatrixInputs(context); ++i) {
        const Tensor& in = context->input(i);
        if (i == 0) {
            input_rank = in.dims();
            OP_REQUIRES(
                context, input_rank >= 2,
                errors::InvalidArgument(
                    "Input tensor ", i,
                    " must have rank >= 2"));
        }
    }
}
```



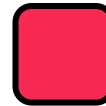
IR (CFG, DFG, AST,...)



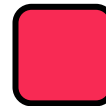
Features



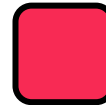
#. instructions



loop nest level



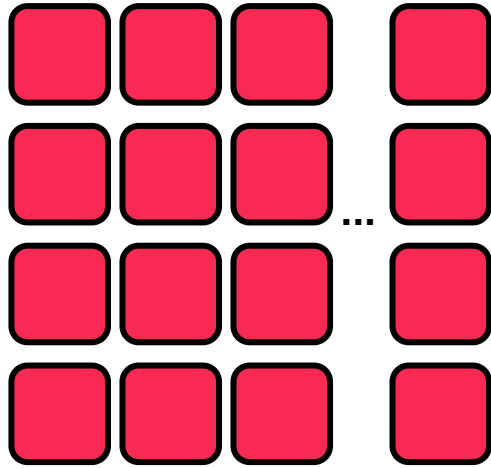
arithmetic density



trip counts

Collect examples

Features

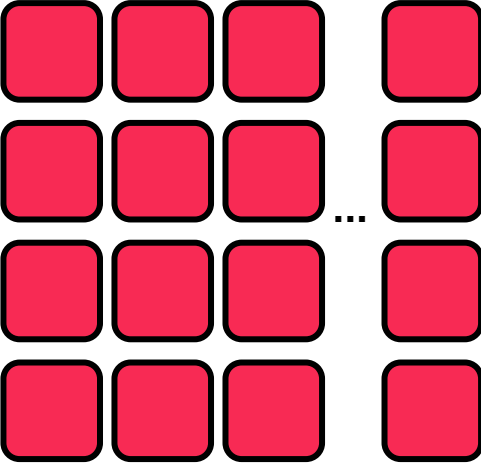


Best Param

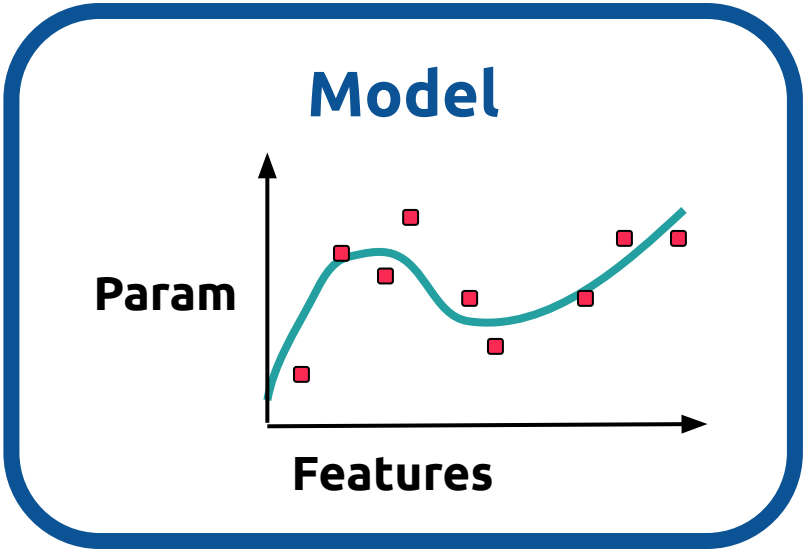
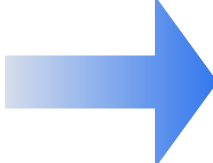


Learn from examples

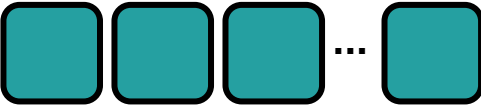
Features



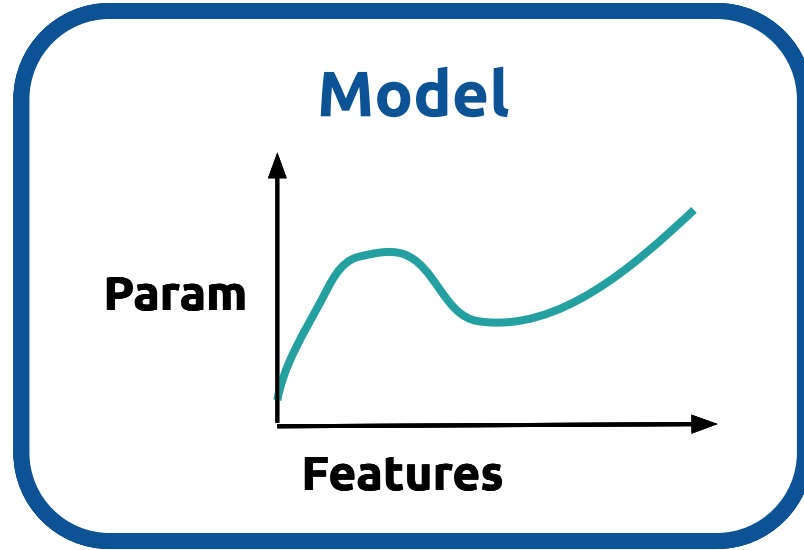
Supervised
Machine
Learner



Best Param

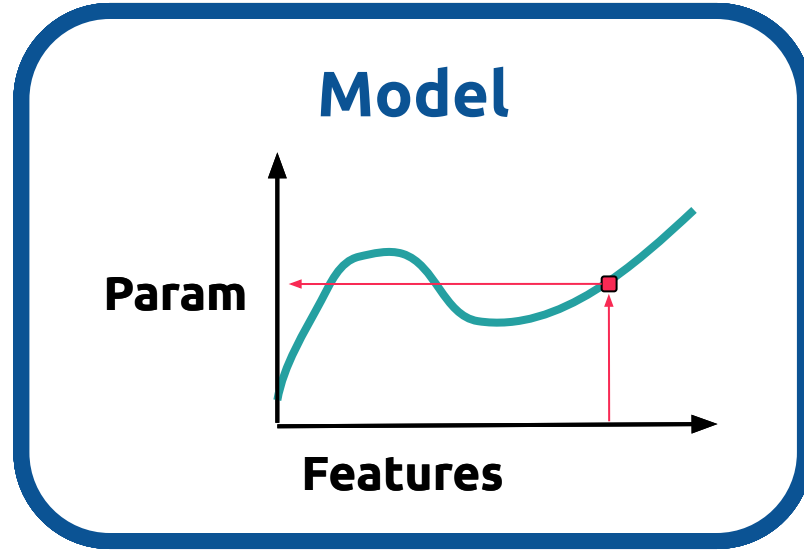
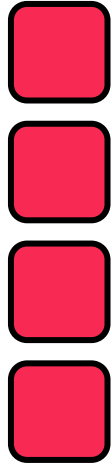


The model is the heuristic

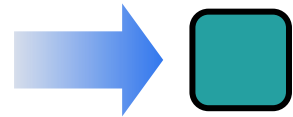


The model is the heuristic

New Program
Features

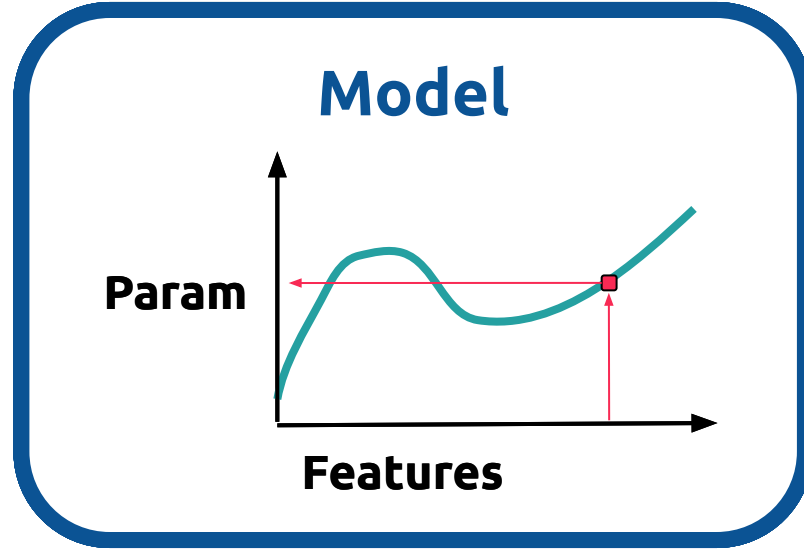
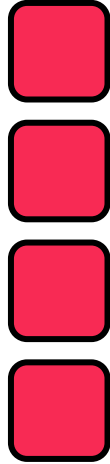


Predicted
param

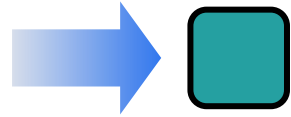


The model is the heuristic

New Program
Features



Predicted
param



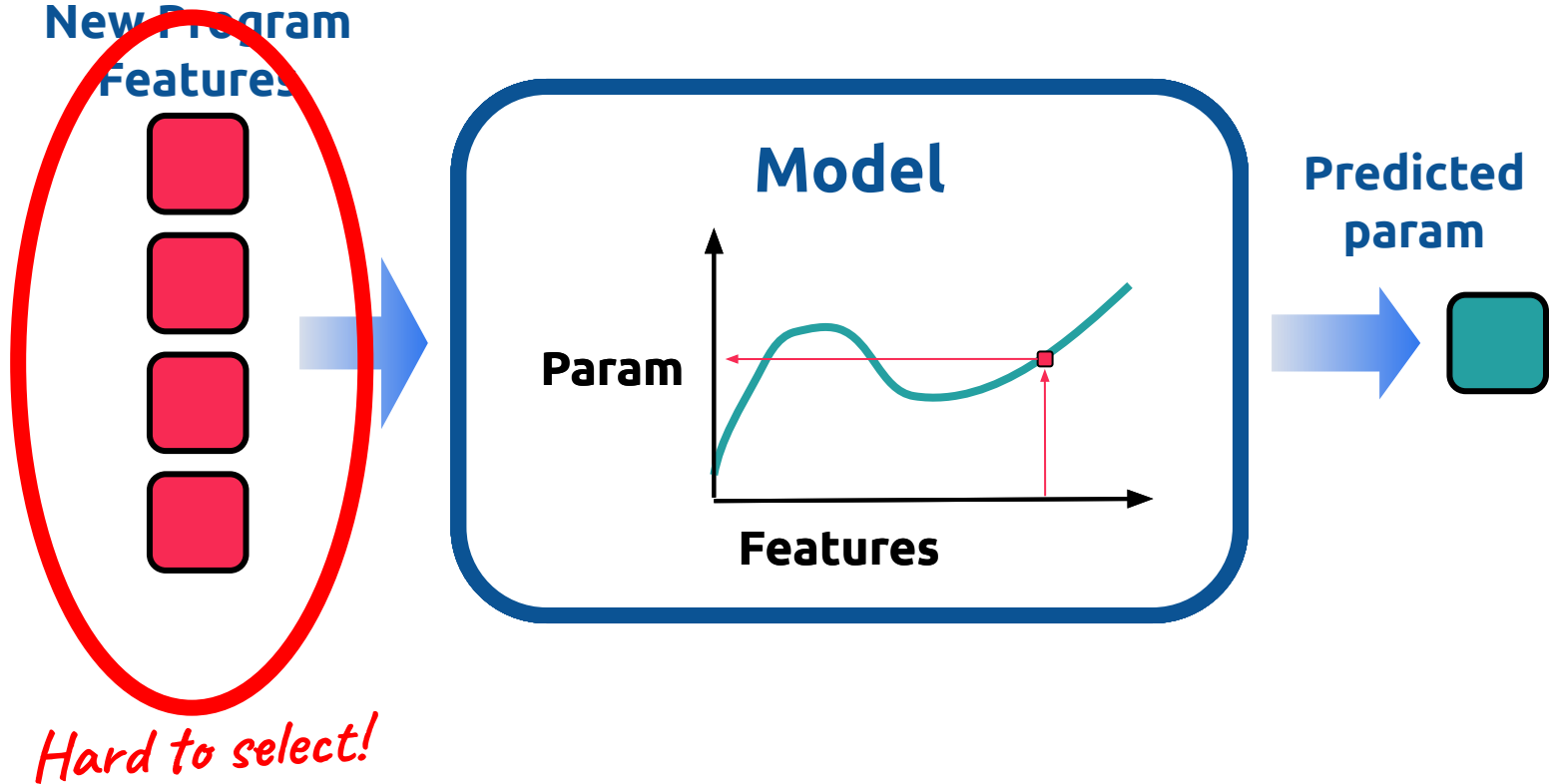
Very successful!

Huge performance gains to be had. Typically outperforms human expert.

[[Wang et. al. 2018](#)]

**Why aren't our
compilers full of
ML?**

The model is the heuristic



Learning without features

(Cummins et al., PACT 17)
"End-to-end Deep Learning of
Optimization Heuristics"

1. Input

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

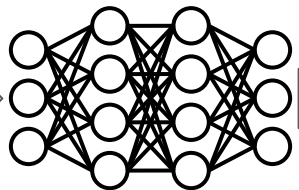
2. Vocab

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

181 tokens

3. Encoded



LSTM

Optimization
Decision

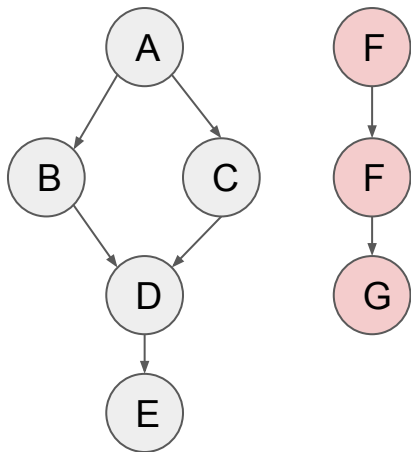


The problem with code representations

Source code is *highly structured*

It isn't a vector of numbers

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



It isn't a sequence of tokens

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

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

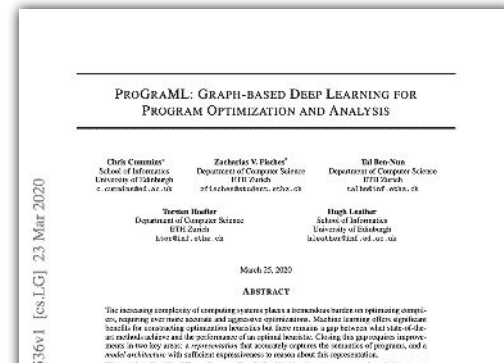
**Can we make ML
think like a
compiler?**

Program Graphs for Machine Learning

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



Building ProGraML: IR

Derive **IR** from input program (here, LLVM)

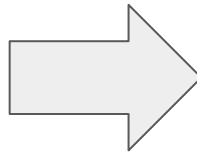
Why IR?

Language **agnostic**

(e.g. C, C++, OpenCL, Swift,
Haskell, Java for LLVM)

We want to improve
compiler decisions, so
use a **compiler's eye** view.

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



```
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  
}
```

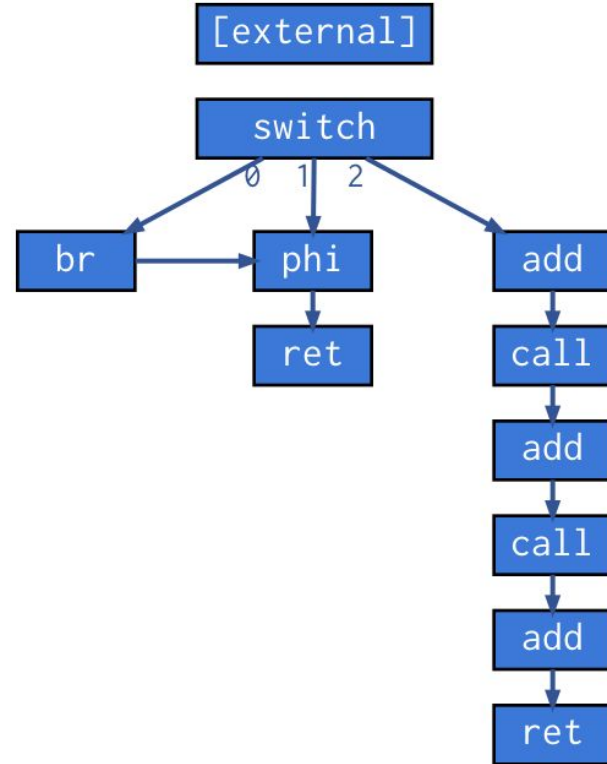
Building ProGraML: Control-flow

Full-flow-graph: represent **each instruction** as a vertex.

Vertex label is the **instruction name**.

Edges are **control-flow**.

Edge position attribute for **branching control-flow**.

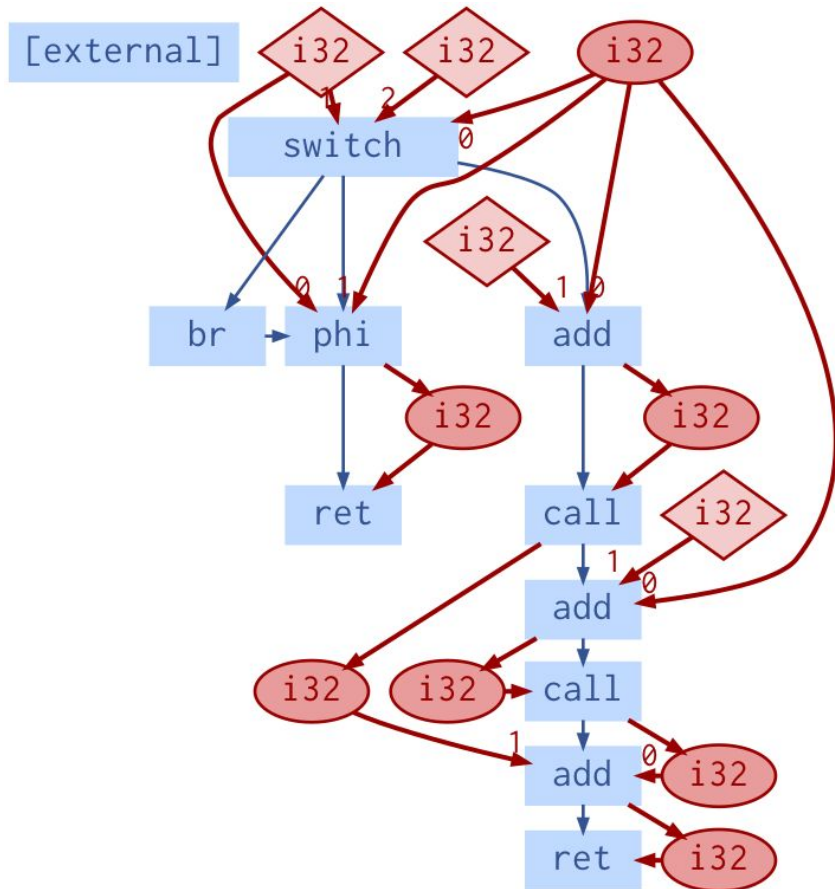


Building ProGraML: Data-flow

Add graph vertices for **constants** (diamonds) and **variables** (oblongs).

Edges are **data-flow**.

Edge position attribute for **operand order**.

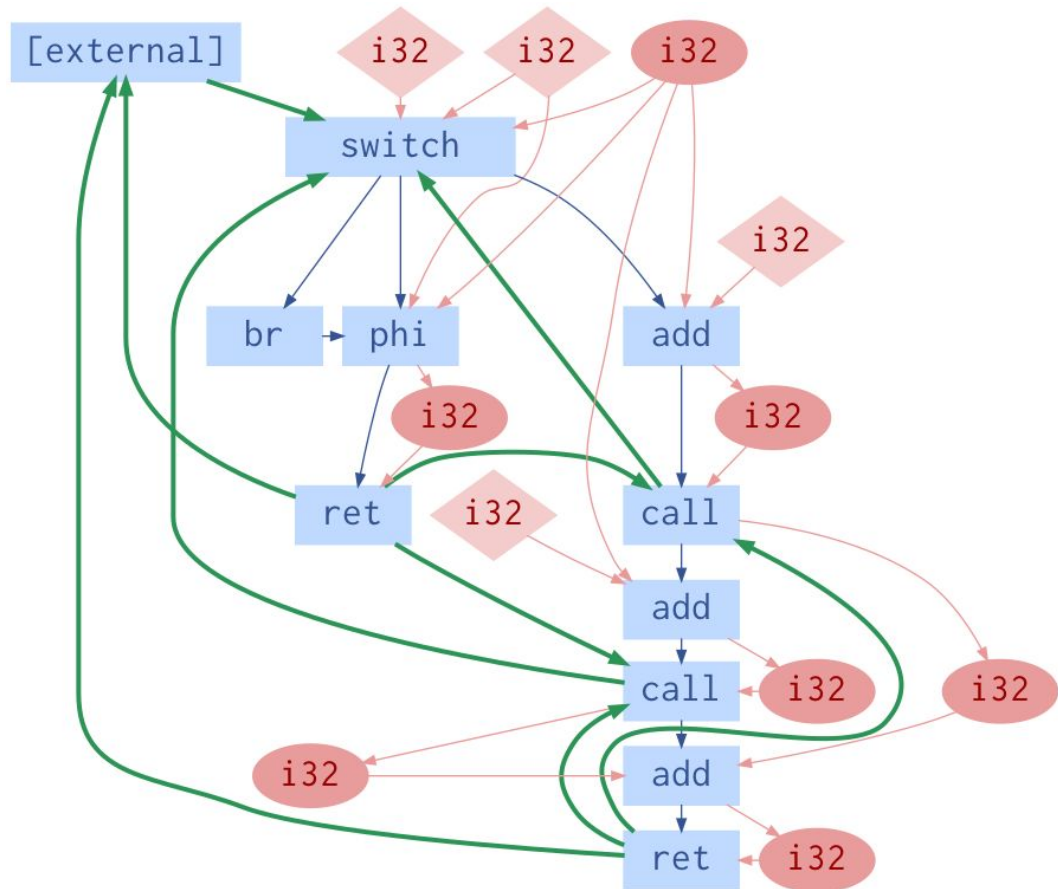


Building ProGraML: Call-flow

Edges are **call-flow**.

Inbound edge to
function entry instruction.

Outbound edge from
(all) **function exit** instruction(s).

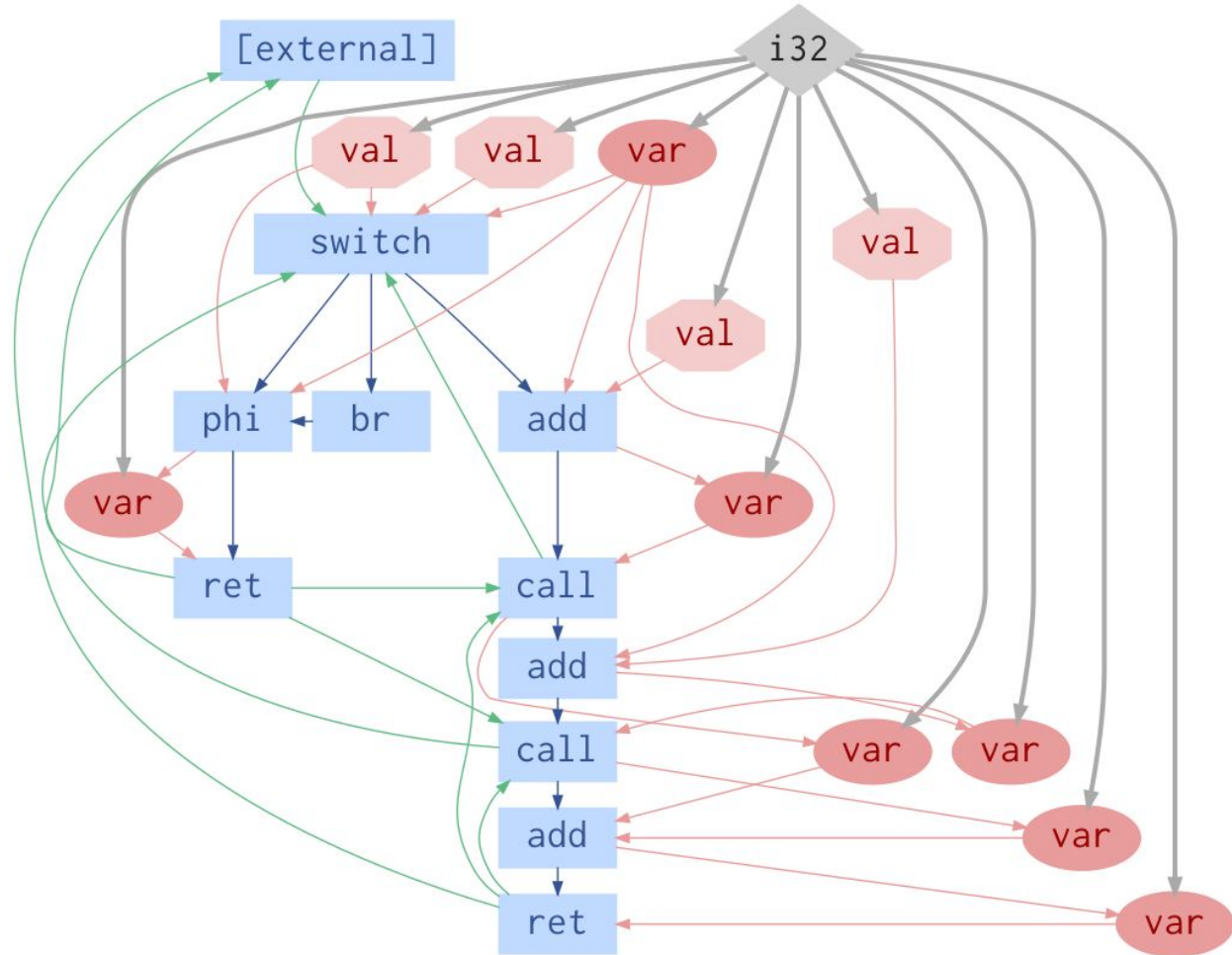
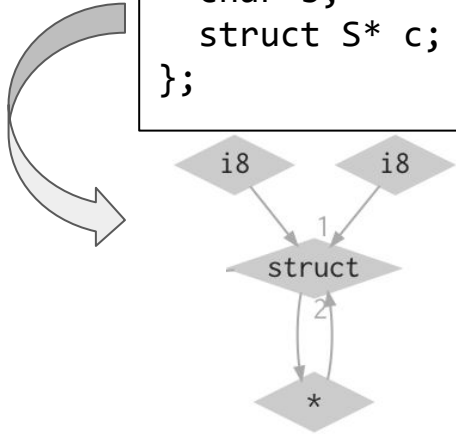


Building ProGraML: Types

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

Types are **composable**.
Edge position per field.

```
struct S {  
  char a;  
  char b;  
  struct S* c;  
};
```



Learning with ProGraML: Node Embeddings

Use vertex 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	Test coverage
inst2vec [12]	8,565	34.0%
CDFG [14]	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: GGNNs

Message Passing

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

6 typed weight matrices for
{forwards, backwards} {control, data, call}
edge types

Position gating to differentiate
control branches and operand order

Readout Head


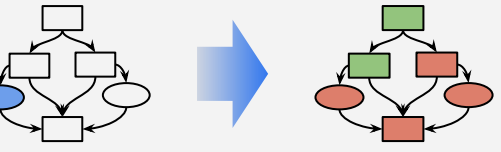


$$R(h_v^T, h_v^0) = \sigma(f(h_v^T, h_v^0)) \cdot g(h_v^T)$$

per-vertex prediction after T
message-passing steps

Deep Data Flow



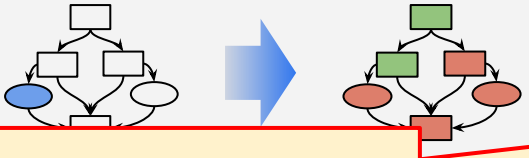


Dataset: 450k LLVM-IRs covering 5 programming languages

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

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 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

inst2vec/CDFG are instruction-level representations, can't reason about variables

Caveat: limited problem size

Data flow analyses iterate until a fixed point is reached.

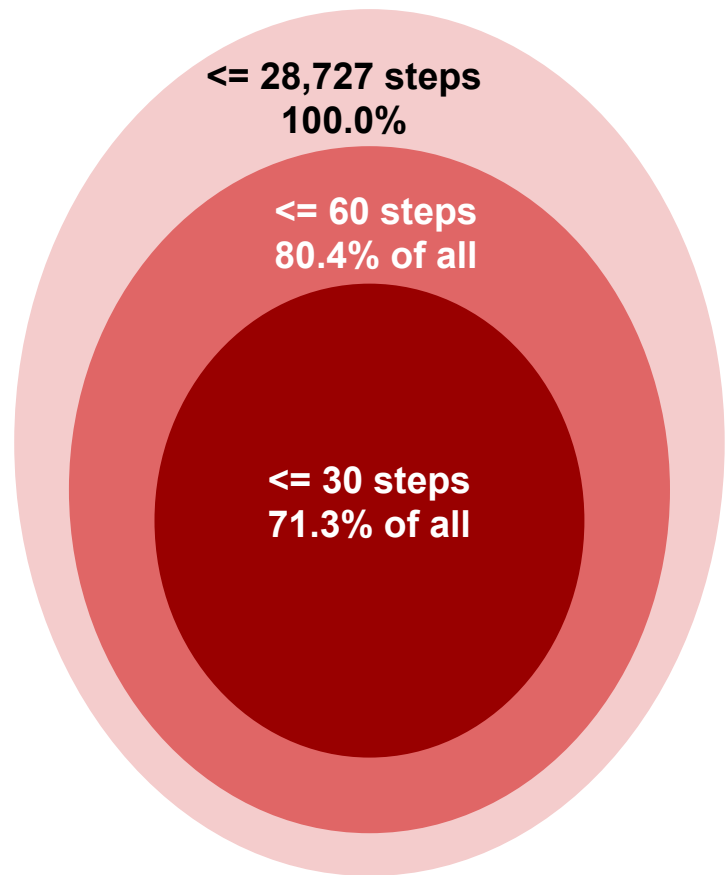
GGNNs iterate for a fixed number of timesteps T .

For each example in the train/test sets, we count the number of steps required for an iterative analysis to solve.

We then filter the train/test set to include only examples which the iterative analysis required $\leq T$ steps to solve.

Previous slide was $T=30$, excluding 28.7% of examples.

Next slide shows performance models, trained on $T=30$, with different inference steps ($T=60$, $T=200$).



Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

F1 scores

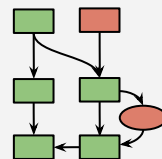
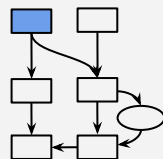
30
timesteps

60 timesteps

200
timesteps

Reachability

Trivial forwards control-flow
E.g. dead code elimination



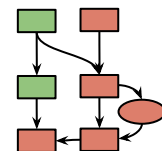
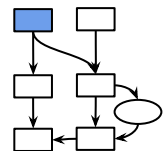
0.998

0.997

0.943

Dominance

Forwards control-flow
E.g. global code motion



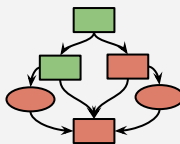
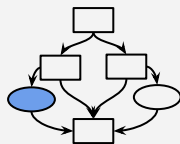
1.000

0.991

0.123

Data Dependencies

Forwards data-flow
E.g. instruction selection



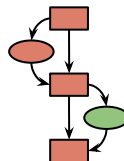
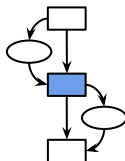
0.997

0.993

0.965

Live-out Variables

Backwards control- and data-flow
E.g. register allocation



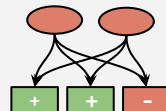
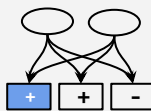
0.937

0.939

0.625

Global Common Subexpressions

Instruction/operand sensitive
E.g. GCS Elimination



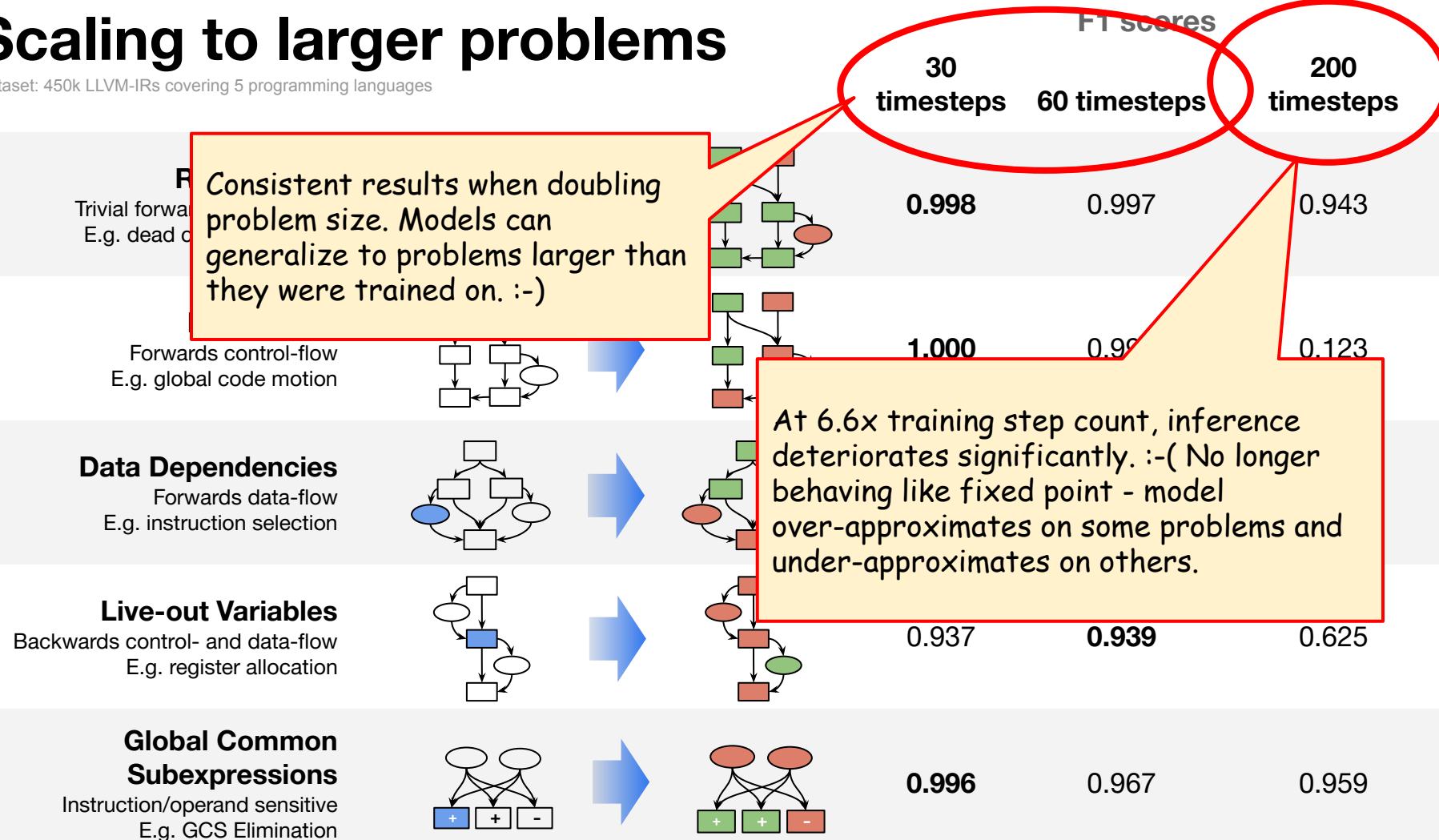
0.996

0.967

0.959

Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

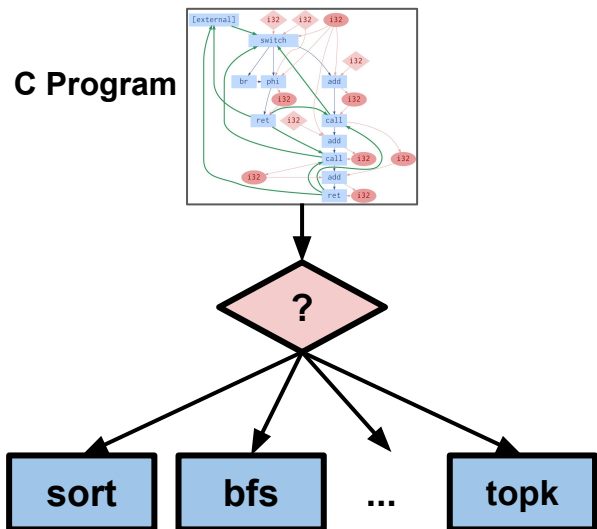


Consistent results when doubling problem size. Models can generalize to problems larger than they were trained on. :-)

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.)

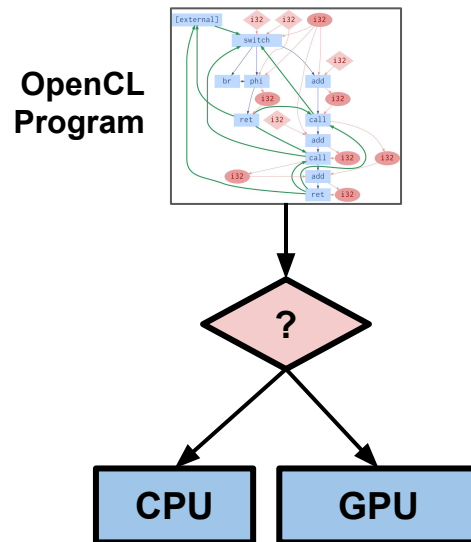
Downstream tasks

1. Algorithm Classification



1.35× improvement over state-of-art

2. Heterogeneous Device Mapping



1.20× improvement over state-of-art

Further Reading

arXiv:2003.10536v1 [cs.LG] 23 Mar 2020

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

Chris Cummins¹
¹School of Informatics
University of Edinburgh
c.cummins@ed.ac.uk

Zacharias V. Zachariadis²
²Department of Computer Science
ETH Zurich
zachari@inf.ethz.ch

Tal Ben-Nun²
²Department of Computer Science
ETH Zurich
talb@inf.ethz.ch

Thomas Heider¹
¹Department of Computer Science
ETH Zurich
thor@inf.ethz.ch

Hugh Leather¹
¹School of Informatics
University of Edinburgh
h.leather@ed.ac.uk

March 25, 2020

ABSTRACT

The increasing complexity of computer systems places a tremendous burden on optimizing compilers, requiring ever more advanced and aggressive optimization. Machine learning offers significant benefits for assessing optimization benefits for diverse reasons: a) to capture and distill state-of-the-art research, analysis and the performance of an optimal heuristic. b) To give us a principled approach to improve on a two way trade: c) representations that accurately capture the semantics of programs, and d) avoid overfitting to a specific architecture, to reason about this representation.

We introduce ProGraML, —Program Graphs for Machine Learning— a novel graph-based program representation using a low-level, language-agnostic, and portable format, and machine learning models capable of performing context-aware program tasks over these graphs. The ProGraML representation is a directed, annotated multigraph that captures control, data, and call relations, and maintains instructions and operand type and redunant. Machine Learning Neural Networks process information through the annotated representation, enabling diverse program, pos-instances, and pre-compiled classification tasks. ProGraML, as a context-independent representation will support currently 50 LLVM real XLA IRs.

ProGraML provides a program purpose program representation that maps transformable models to perform the output of program analysis that are fundamental to optimization. To this end, we analyze the performance of our approach on a set of real-world compiler analysis tasks: control flow, reaching definitions, flows, data operations, variable liveness, and common subexpression elimination. On a benchmark dataset of 290k LLVM IRs, this covering six source programming languages, ProGraML achieves an average F1 of \approx score, significantly outperforming the state-of-the-art approaches. We also apply our approach to two high-level tasks — heterogeneous device mapping and program classification — setting the state-of-the-art performance in both.

1 Introduction

The hardware of computing ecosystems is becoming increasingly complex: multi-core and many-core processors, heterogeneous systems, distributed and cloud platforms. Notably, comparing performance and energy benefits from separate flows is beyond the capabilities of most programmers. In such an environment, high quality optimization heuristics are not just desirable, they are required. Despite this, good optimization heuristics are hard to come by.

Workshop on Embedded Systems

Preprint

<https://arxiv.org/abs/2003.10536>

ProGraML [Download](#)

2020.05.13

Graph [Download](#) [Shareable URL](#)

Entire Program

```
node {
  text: "croots="
}
node {
  text: "switch"
  features {
    feature {
      key: "full_text"
      value {
        bytes_list {
          bytes: "switch 132 00, label 03 [r"
        }
      }
    }
  }
}
node {
  text: "br"
  block: 1
  features {
    feature {
      key: "full_text"
      value {
        bytes_list {
          bytes: "br label 00"
        }
      }
    }
  }
}
node {
  text: "add"
  block: 2
  features {
    feature {
      key: "full_text"
      value {
        bytes_list {
          bytes: "add + add new 132 00, -1"
        }
      }
    }
  }
}
node {
  text: "call"
}
```

In-browser demo

https://chriscummins.cc/s/program_explorer

Source code + datasets

<https://github.com/ChrisCummins/ProGraML>

Apache 2.0

Conclusions

Reasoning about programs requires the right combination of representation + model.

ProGraML: combines control-, data-, call-, and type-graphs to model programs at IR level.

When processed with GGNNs, significantly outperforms prior approaches.

Interesting challenges

1. Processing **arbitrary sized** graphs.

Idea: Structure the MPNN like an iterative DF solver, self-terminating.

2. Handling **unbounded vocabularies**, e.g. compound types or MLIR dialects.

Idea: decompose types into tree structure in graph.

3. Representing **literal values**.

Requires new vocabulary encoding.