Machine Programming & Data-Driven Dependable and Secure Software Systems

Justin Gottschlich
Principal AI Scientist & Director/Founder of Machine Programming Research (Intel Labs)
Adjunct Assistant Professor (University of Pennsylvania)
Steering Committee Chair, ACM Machine Programming Symposium (MAPS)

Contributors:

Machine Programming Research (MPR), Intel Labs
Legal Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Results have been estimated or simulated.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and noninfringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
Overview

• Machine Programming Research @ Intel
• Discussion of The Three Pillars of MP
  • Separation of Intention is Critical
• The Bifurcated Space of MP
  • Stochastic and Deterministic
• Machine Programming Emphasis @ Intel
  • ControlFlag: a Self-Supervised Systems for MP
  • MISIM: a Code Semantics Similarity System
Overview

• Machine Programming Research @ Intel
• Discussion of The Three Pillars of MP
  • Separation of Intention is Critical
• The Bifurcated Space of MP
  • Stochastic and Deterministic
• Machine Programming Emphasis @ Intel
  • ControlFlag: a Self-Supervised Systems for MP
  • MISIM: a Code Semantics Similarity System
Definition: Machine Programming (MP) is the automation of software and hardware development

Machine Programming Research (MPR)

A New Pioneering Research Initiative at intel labs
Intel Labs’ MPR Goals

*Machine Programming (MP) is the automation of software and hardware development*

**Time:**
Reduce development time of all aspects of software development

*Measured as 1000x+ improvement over human work performed today

**Quality:**
Better software than the best human programmers*

*Measured as superhuman correctness, performance, security, etc.
Intel Labs’ MPR Goals

Machine Programming (MP) is the automation of software and hardware development

**Quality:**
Better software than the best human programmers*
*Measured as superhuman correctness, performance, security, etc.

**Time:**
Reduce development time of all aspects of software development*
*Measured as 1000x+ improvement over human work performed today

**Concrete Data Point:**
“Automatically Translating Image Processing Libraries to Halide” (Ahmad et al., 2019)*
*Funded by Intel’s CAPA Research Center

![Graph showing x86 Performance with speedup on a logarithmic scale.](image-url)
Overview

- Machine Programming Research @ Intel
- Discussion of The Three Pillars of MP
  - Separation of Intention is Critical
- The Bifurcated Space of MP
  - Stochastic and Deterministic
- Machine Programming Emphasis @ Intel
  - ControlFlag: a Self-Supervised Systems for MP
  - MISIM: a Code Semantics Similarity System
The Three Pillars of Machine Programming

Machine Programming (MP) is the automation of software and hardware development

- **Intention**: Discover the intent of a programmer; lift meaning from software
- **Invention**: Create new algorithms and data structures; compositional novelty
- **Adaptation**: Evolve in a changing hardware/software world
The Three Pillars of Machine Programming

Machine Programming (MP) is the automation of software and hardware development

- **Intention**: Discover the intent of a programmer; lift meaning from software
- **Invention**: Create new algorithms and data structures; compositional novelty
- **Adaptation**: Evolve in a changing hardware/software world

Data is a principal driver for all MP systems

The Three Pillars of Machine Programming

- Justin Gottschlich
  Intel Labs, USA
  justin.gottschlich@intel.com
- Armando Solar-Lezama
  MIT, USA
  asolar@csail.mit.edu
- Michael Carbin
  MIT, USA
  mcarbin@csail.mit.edu
- Martin Rinard
  MIT, USA
  rinard@csail.mit.edu
- Saman Amarasinghe
  MIT, USA
  saman@csail.mit.edu
- Joshua B. Tenenbaum
  MIT, USA
  jbt@mit.edu
- Nesime Tatbul
  Intel Labs and MIT, USA
  tatbul@csail.mit.edu
- Regina Barzilay
  MIT, USA
  regina@csail.mit.edu
- Tim Mattson
  Intel Labs, USA
  timothyg.mattson@intel.com
Separation of Intention is Critical

- Requires user only supply **core idea** (improving productivity)
- Enables machine to explore a **wider range** of possible solutions (improving MP-generated solutions)
- Enables automatic SW **adaptation & evolution**

We anticipate this separation will give rise to:

- **Intentional** Programming Languages
  
  Example: Halide/Verified Lifting (Adobe Photoshop)
Separation of Intention is Critical

- Leverages Separation of Intention from Invention & Adaptation
- "Automatically Translating Image Processing Libraries to Halide" (Ahmad et al., 2019)

programmer is forced to stay on this side of the line
Overview

• Machine Programming Research @ Intel
• Discussion of The Three Pillars of MP
  • Separation of Intention is Critical
• The Bifurcated Space of MP
  • Stochastic and Deterministic
• Machine Programming Emphasis @ Intel
  • ControlFlag: a Self-Supervised Systems for MP
  • MISIM: a Code Semantics Similarity System
The Bifurcated Space of Machine Programming

**Techniques used in MP systems**

- **Stochastic**
  - Machine Learning
    - (Neural networks, reinforcement learning, genetic algorithms, Bayesian networks, etc.)

- **Deterministic**
  - Formal Methods
    - (Formal verifiers, spatial and temporal logics, formal program synthesizers, etc.)

**Components used in MP systems**

- **Software**
  - Programming languages, algorithms, data structures, etc.

- **Hardware**
  - Compute, communications, and memory architectures, etc.
The Bifurcated Space of Machine Programming

Techniques used in MP systems

Stochastic

Machine Learning
(Neural networks, reinforcement learning, genetic algorithms, Bayesian networks, etc.)

Deterministic

Formal Methods
(Formal verifiers, spatial and temporal logics, formal program synthesizers, etc.)

Software
Programming languages, algorithms, data structures, etc.

Hardware
Compute, communications, and memory architectures, etc.

Stochastic MP systems tend to improve w/ more iid data

Progressively more approximate
Progressively more precise
The Bifurcated Space of Machine Programming

**Techniques used in MP systems**

**Stochastic**
- **Machine Learning**
  - (Neural networks, reinforcement learning, genetic algorithms, Bayesian networks, etc.)
  - Halide uses stochastic techniques for optimization

**Deterministic**
- **Formal Methods**
  - (Formal verifiers, spatial and temporal logics, formal program synthesizers, etc.)
  - Verified lifting uses formal methods (CEGIS) for semantics verification

**Components used in MP systems**

- Programming languages, algorithms, data structures, etc.
- Compute, communications, and memory architectures, etc.
Concretizing The Two Sides of MP with Neuro-Symbolism

Stochastic (Neuro)

- Machine learning
- ML Model

Deterministic (Symbolic)

- Formal Synthesizer
- Program

Programming Language Constructs

- for (...)
- if (...)
- swap(...)  peek(...)
Overview

- Machine Programming Research @ Intel
- Discussion of The Three Pillars of MP
  - Separation of Intention is Critical
- The Bifurcated Space of MP
  - Stochastic and Deterministic
- Machine Programming Emphasis @ Intel
  - ControlFlag: a Self-Supervised Systems for MP
  - MISIM: a Code Semantics Similarity System
Numerous MP Efforts @ Intel

Debugging / Profiling / Productivity

- ControlFlag, MISIM, & AutoPerf

Automated Performance Extraction

- Inteon’s Tiger Shark (Intel venture)
- MP-based general-purpose compiler (e.g., ML-learned code optimizations)

And Many More ...
Numerous MP Efforts @ Intel

Debugging / Profiling / Productivity

• ControlFlag, MISIM, & AutoPerf

Automated Performance Extraction

• Inteon’s Tiger Shark (Intel venture)
• MP-based general-purpose compiler (e.g., ML-learned code optimizations)

And Many More ...

*Beats SOTA by ~2x with 400k labeled data samples
**Beats SOTA by ~5x with 1M labeled data samples (independently confirmed by IBM/MIT)


**"CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks" by Puri et al. (https://arxiv.org/abs/2105.12655)
Numerous MP Efforts @ Intel

Debugging / Profiling / Productivity

• ControlFlag, MISIM, & AutoPerf

Automated Performance Extraction

• Inteon’s Tiger Shark (Intel venture)
• MP-based general-purpose compiler (e.g., ML-learned code optimizations)

And Many More ...

What can we build without labeled data?
Productivity – Debugging

RESULTS: The global cost of software development is US$1.25 trillion

Software development cost structure (US$ billion)

- Total: $1,250
- Overhead: $625
- Programming Wages (admin): $313
- Programming Wages (development): $312
- Debugging: $78
  - 25% - Fixing Bugs
  - 25% - Making Code Work
  - 20% - Designing Code
  - 30% - Writing Code
- Productive: $62
  - 20% - Designing Code


University of Cambridge
Productivity – Debugging

50% of the cost of software development is debugging.

University of Cambridge

Machine Programming Research (MPR), Intel Labs
Debugging: Finding Code Anomalies

What is a code anomaly?

• A piece of code that is irregular

Why care about code anomalies?

• Anomalous code can lead to defects, technical debt, delayed software development (hard to understand code), loss of customer trust
Anomaly found in CURL (~30-year-old software)

CURL developers **rewrite** flagged piece of code found with ControlFlag

Re: Potential confusion in http_proxy.c and a recommendation

> We believe that using “if (s->keepon > 1)” would eliminate this confusion and capture the intended semantics precisely.

I think you've pointed out code that could be written clearer, yes. But I think an even better improvement to this logic would be to use an enum or defined values that include all three used values as state names.

What do you think about my proposal over at: [https://github.com/curl/curl/pull/6193](https://github.com/curl/curl/pull/6193)

On Mon, 9 Nov 2020, Hasabnis, Niranjan via curl-library <curl-library_at_cool.haxx.se> wrote:

https://curl.se/mail/lib-2020-11/0028.html
Anomaly found in CURL (~30-year-old software)

CURL developers rewrite flagged piece of code found with ControlFlag

Re: Potential confusion in http_proxy.c and a recommendation

• Contemporary messages sorted: [by date] [by thread] [by subject] [by author] [by messages with attachments]
From: Daniel Stenberg via curl-library <curl-library_at_cool.haxx.se>
Date: Mon, 9 Nov 2020 23:51:20 +0100 (CET)

On Mon, 9 Nov 2020, Hasabnis, Niranjan via curl-library wrote:

> We believe that using “if (s->keepon > 1)” would eliminate this confusion
> and capture the intended semantics precisely.

I think you’ve pointed out code that could be written clearer, yes. But I
think an even better improvement to this logic would be to use an enum or
defined values that include all three used values as state names.

What do you think about my proposal over at:
https://github.com/curl/curl/pull/6193

https://curl.se/mail/lib-2020-11/0028.html
Limitations in Existing Code Anomaly Detectors

Tools & techniques to identify software defects
- Testing (unit tests, QA, etc.)
- Static analysis
  - Compilers, linters

Limitations
- Continuous manual effort to maintain and update (i.e., adding new rules as things evolve)
- Manual efforts can be error-prone
## ControlFlag

**A Self-Supervised Anomalous Code Detection System**

**Technical Lead:**
Dr. Niranjan Hasabnis  
Intel Labs

### Step 1: Pattern mining

<table>
<thead>
<tr>
<th>Semi-trust (humans must decide this)</th>
<th>Learn idiosyncratic patterns in code</th>
<th>Self-supervision; no labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1.0 Source Code Repositories</td>
<td>Step 1.1 Mine patterns in control structures</td>
<td>Step 1.3 Self-supervised clustering using decision tree</td>
</tr>
<tr>
<td><strong>Codebase</strong></td>
<td><strong>Patterns</strong></td>
<td><strong>Syntax Trees for Patterns</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Source code parser</strong></td>
<td><strong>Self-supervision clustering using decision tree</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Syntax Trees for Patterns</strong></td>
<td><strong>Syntax Trees for Patterns</strong></td>
</tr>
</tbody>
</table>

### Step 2: Scanning for erroneous patterns

<table>
<thead>
<tr>
<th>Step 2.0 Target Code Repositories</th>
<th>Step 2.1 Mine patterns in control structures</th>
<th>Step 2.2 Build representation for patterns</th>
<th>Step 2.3: Find “nearest” patterns in decision tree</th>
<th>Step 2.4: Is pattern an anomaly?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Codebase</strong></td>
<td><strong>Patterns</strong></td>
<td><strong>Syntax Trees</strong></td>
<td><strong>Nearest patterns in training dataset</strong></td>
<td><strong>Is pattern an anomaly?</strong></td>
</tr>
<tr>
<td><strong>Source code parser</strong></td>
<td><strong>Syntax Trees</strong></td>
<td><strong>Patterns</strong></td>
<td><strong>Syntax Trees</strong></td>
<td><strong>Is pattern an anomaly?</strong></td>
</tr>
</tbody>
</table>

"ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures" by Hasabnis & Gottschlich, MAPS '21

Machine Programming Research (MPR), Intel Labs
ControlFlag In The News

Intel wants to make it easier than ever to spot coding errors

So Hayley Sharma 2 days ago

Detect problems in code using Intel's new open source tool

Phoronix

Intel Makes ControlFlag Open-Source For Helping To Detect Bugs In Code

Intel has open-sourced ControlFlag tool which is a machine learning tool that spots bugs in code.

The Machine

Intel open-sources AI-powered tool to spot bugs in code

The Machine

Developers: Intel’s automated debugging tool ControlFlag is now open source

ControlFlag is a machine learning tool to help developers identify bugs in code.

TechRadar

Intel’s ControlFlag debugging tool uses artificial intelligence to clean up code

Blaze Trends

In 2020, a study showed that 37% of bugs are covered by 17% of the code. With ControlFlag, Intel aims to reduce the number of bugs by 10% through code optimization.
ControlFlag

A Self-Supervised Anomalous Code Detection System

Technical Lead:
Dr. Niranjan Hasabnis
Intel Labs

“ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures” by Hasabnis & Gottschlich, MAPS ‘21
Anomalies in Production-Quality, Open-Source Software

Evaluation: Setup

**Training** repository selection
- 6000 GitHub repos for C language having more than 100 stars
- 2.57M programs
- 1.1B Lines of code
- 38M patterns

**Test** repositories
- openssl, curl, ffmpeg
- git, vlc, lcx, lz4, reactos

<table>
<thead>
<tr>
<th>Repo</th>
<th>GitHub stars</th>
<th>Found Anomalies</th>
<th>Scanned Expressions</th>
<th>Types of anomalies found</th>
</tr>
</thead>
<tbody>
<tr>
<td>IoLanguage/io</td>
<td>2.3K</td>
<td>5</td>
<td>1635</td>
<td>Confusing expressions; missing parenthesis</td>
</tr>
<tr>
<td>Git/git</td>
<td>38.9K</td>
<td>6</td>
<td>6341</td>
<td>Confusing expression; character comparison using greater than or less than</td>
</tr>
<tr>
<td>Rubinius/rubinius</td>
<td>3K</td>
<td>2</td>
<td>10135</td>
<td>Character comparison using greater than or less than; missing parenthesis</td>
</tr>
<tr>
<td>FreeRADIUS/freeradius-server</td>
<td>1.5K</td>
<td>3</td>
<td>20621</td>
<td>Character comparison using greater than or less than</td>
</tr>
<tr>
<td>Davidfstr/rdiscount</td>
<td>755</td>
<td>4</td>
<td>472</td>
<td>Character comparison using greater than or less than; missing parenthesis</td>
</tr>
<tr>
<td>Libharu/libharu</td>
<td>1.2K</td>
<td>1</td>
<td>2785</td>
<td>Character comparison using greater than or less than</td>
</tr>
<tr>
<td>Macournoyer/tinyrb</td>
<td>454</td>
<td>3</td>
<td>4369</td>
<td>Character comparison using greater than or less than</td>
</tr>
<tr>
<td>Rhomobile/rhodes</td>
<td>1K</td>
<td>14</td>
<td>76128</td>
<td>Confusing expressions; missing parenthesis; character comparison using greater or less than</td>
</tr>
</tbody>
</table>

"ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures" by Hasabnis & Gottschlich, MAPS '21
Anomaly Found in Proprietary & Deployed Software

An Example of ControlFlag’s Finding

Three defects:
1. Duplicate expression in lines 11 and 12
2. Possible out-of-bounds memory access (memory error) in line 14
3. Information leak, security vulnerability in line 11

“ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures” by Hasabnis & Gottschlich, MAPS ‘21

Machine Programming Research (MPR), Intel Labs
RESULTS: Summary of 1st Proprietary Repo Analysis

Identified 104 potential defects

- 812 scanned files (.C and .H)
- 353K scanned lines of code
- 4600 scanned expressions

3 hours total analysis time (approx.)

- 56 Intel CPU cores

<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anomalies that are critical bugs</td>
<td>2</td>
<td>Type error; memory error; security vulnerability</td>
</tr>
<tr>
<td>Anomalies that can lead to unwanted side-effects</td>
<td>39</td>
<td>Missing NULL check; possible divide by 0; missing return value check</td>
</tr>
<tr>
<td>Anomalies that point to confusing programming style</td>
<td>4</td>
<td>Double parenthesis around expressions, when not required</td>
</tr>
<tr>
<td>Anomalies that point to improvements in programming styles</td>
<td>59</td>
<td>Not using named constants; constant on right hand of equality;</td>
</tr>
<tr>
<td>Total unique anomalies reported</td>
<td>104</td>
<td>Not including false positives</td>
</tr>
</tbody>
</table>

“ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures” by Hasabnis & Gottschlich, MAPS ‘21
RESULTS: Summary of 2nd Proprietary Repo Analysis

Identified 191 potential defects

- **19K** scanned files (.C and .H)
- **10.9M** scanned lines of code
- **18.7K** scanned expressions

8 hours total analysis time (approx.)

- 12 Intel CPU cores

<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugs found (confirmed by group)</td>
<td>5</td>
<td>Bitwise operation instead of Boolean logic operation</td>
</tr>
<tr>
<td>Confusing programming styles that could lead to bugs</td>
<td>22</td>
<td>Overly complex code E.g., ((xxxx[pstate].yyy &amp; 0x1) &gt;&gt; 0)</td>
</tr>
<tr>
<td>Syntactic improvements to code according to standard style guides</td>
<td>164</td>
<td>Stylistic deviations from standards</td>
</tr>
<tr>
<td><strong>Total unique anomalies reported</strong></td>
<td>191</td>
<td>Not including false positives</td>
</tr>
</tbody>
</table>

“ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures” by Hasabnis & Gottschlich, MAPS ‘21
RESULTS: Summary of 2nd Proprietary Repo Analysis

Identified 191 potential defects
- 19K scanned files (.C and .H)
- 10.9M scanned lines of code
- 18.7K scanned expressions

8 hours total analysis time (approx.)
- 12 Intel CPU cores

<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugs found (confirmed by group)</td>
<td>5</td>
<td>Bitwise operation instead of Boolean logic operation</td>
</tr>
<tr>
<td>Confusing programming styles that could lead to bugs</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Overly complex code</td>
<td></td>
<td>E.g., ((xxxx[pstate].yyy &amp; 0x1) &gt;&gt; 0)</td>
</tr>
<tr>
<td>Syntactic improvements to code according to standard style guides</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>Stylistic deviations from standards</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total unique anomalies reported: 191
Not including false positives

Working on a larger scan of ~65M lines of code, which identified 25,000 anomalies.

<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of files (.C and .H)</td>
<td>126,896</td>
</tr>
<tr>
<td>Number of expressions</td>
<td>1,374,028</td>
</tr>
<tr>
<td>Number of lines of code</td>
<td>64,690,054</td>
</tr>
</tbody>
</table>

Intel’s partner is working to integrate ControlFlag as a permanent component of their continuous integration process.

“ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures” by Hasabnis & Gottschlich, MAPS ‘21
CODE SEMANTICS
What are code semantics?
The meaning behind the syntax.

Why should we care?
Many reasons: code comprehension and reasoning (Microsoft/GitHub Co-Pilot), bug detection, etc.
What are code semantics?

The meaning behind the syntax.

Formally, at the highest level

For some set of inputs, $I$
And two programs $P_i$ and $P_j$
If programs, $P_i$ and $P_j$ are executed using inputs $I$ and produce an identical set of outputs $O$
We say they are semantically equivalent

Why should we care?

Many reasons: code comprehension and reasoning (Microsoft/GitHub Co-Pilot), bug detection, etc.
These code snippets are semantically equivalent (according to our prior definition)

Program A

```cpp
int a;
// algorithm
while (!cin.eof()) {
    while (!cin.eof() && !isdigit(cin.peek()))
        cin.get(); // ignore
    // print out result
    if (cin >> a)
        cout << a << endl;
}
```

Program B

```cpp
char *p, *head, c;
p = (char *) malloc(sizeof(char) * 30);
head = p; scanf("%c", p);
while (*p != '\n') { p++; *p = getchar();}
*p = '\0'; p = head;
for (; *p != '\0'; p++) {
    if (*p <= '9' && *p >= '0') {printf("%c", *p);}
    else if (*(p+1) < 58 && *(p+1) > 47) {putchar('
');}
}
```
These code snippets are semantically equivalent (according to our prior definition)

My Opinion:
The **Most** Important Critical Open Problem for MP is Code Semantics Similarity

(this is a strong claim, I generally don’t make such claims unless I feel strongly about something)
CODE SEMANTICS: PROGRAM-DERIVED SEMANTICS GRAPH (PSG)

PSG is a graphical, hierarchical representation of code semantics.

Figure 1: PSG Abstraction Level Spectrum for Semantic Abstraction Levels (SeAL) and Syntactic Abstraction Levels (SyAL), distinguished by color-coding.
PSG OF EXPONENTIATION (POWER) IMPLEMENTED RECURSIVELY

Software Language Comprehension using a Program-Derived Semantic Graph

Preprint, April, 2020

Implementation 1

0 signed int recursive_power(signed int x, unsigned int y)
1 {
2     if (y == 0)
3         return 1;
4     else if (y % 2 == 0)
5         return recursive_power(x, y / 2) * recursive_power(x, y / 2);
6     else
7         return x * recursive_power(x, y / 2) * recursive_power(x, y / 2);
8 }

Implementation 2

0 signed int iterative_power(signed int x, unsigned int y)
1 {
2 }
3     return val;
4 }

Figure 5: PSG of Recursive Power Function. The shaded region denotes overlap in the nodes of the PSG for the iterative power function shown in Figure 6. These total 17 of the 24 total nodes, a 70.83% overlap.

PSG = PROGRAM-DERIVED SEMANTICS GRAPH

Machine Programming Research (MPR), Intel Labs
PSG OF EXPONENTIATION (POWER) IMPLEMENTED RECURSIVELY & ITERATIVELY

Implementation 1

```c
0    signed int recursive_power(signed int x, unsigned int y)
1     { 
2         if (y == 0)
3             return 1;
4         else if (y % 2 == 0)
5             return recursive_power(x, y / 2) * recursive_power(x, y / 2);
6         else
7             return x * recursive_power(x, y / 2) * recursive_power(x, y / 2);
8     }
```

Implementation 2

```c
0    signed int iterative_power(signed int x, unsigned int y)
1     { 
2         signed int val = 1;
3         while (y > 0) {
4             val *= x;
5             y -= 1;
6         }
7         return val;
8     }
```
PSG OF EXPONENTIATION (POWER) IMPLEMENTED RECURSIVELY & ITERATIVELY

Compared to Aroma’s simplified parse tree (OOPSLA ‘19), PSG has greater graph node matching.

```c
3     return 1;
4     else if (y % 2 == 0)
5         return recursive_power(x, y / 2) *
6     recursive_power(x, y / 2);
7     else
8         return x * recursive_power(x, y / 2) *
9         recursive_power(x, y / 2);
```

**Implementation 2**

```c
0 signed int iterative_power(signed int x, unsigned int y)
1 {
2     signed int val = 1;
3     while (y > 0) {
4         val *= x;
5         y -= 1;
6     }
7     return val;
8 }
```
PSG OF EXPONENTIATION (POWER) IMPLEMENTED RECURSIVELY & ITERATIVELY

Some sub-semantic properties
- Both implement exponentiation (only integers)
- Both are correct
- One is recursive
- One is iterative
- One has multiple branches
- One has one branch path

Each sub-semantic may be useful
- Can influence code comprehension, call stacks, speculative execution (branch prediction), etc.

Compared to Aroma’s simplified parse tree (OOPSLA ’19), PSG has greater graph node matching.

```c
return 1;
else if (y % 2 == 0)
    return recursive_power(x, y / 2) * recursive_power(x, y / 2);
else
    return x * recursive_power(x, y / 2) * recursive_power(x, y / 2);
```

Implementation 2
```c
signed int iterative_power(signed int x, unsigned int y)
{
    signed int val = 1;
    while (y > 0) {
        val *= x;
        y -= 1;
    }
    return val;
}
```
MISIM (MACHINE INFERRED CODE SIMILARITY)

Code semantics similarity system using:

- **Determinism:**
  - new code representation (context-aware semantics structure (CASS))

- **Stochasticism:**
  - learned neural scoring algorithm
Machine Inferred Code Similarity (MISIM)

MISIM has two core novelties: one is deterministic, one is stochastic

[Deterministic] Novel code representation: context-aware semantics structure (CASS)

[Stochastic] Novel learned neural scoring algorithm

Figure 2. Overview of the MISIM System.
MISIM’S ACCURACY

- Compared to SOTA: code2vec, code2seq, NCC, and Aroma.
- Tested on ~19M LOC, 350,000 full C/C++ programs, 400 unique classes.
IBM/MIT's Project CodeNet analysis (2021)


- The C++1000 dataset consists of 1000 classes with 500k programs
- The C++1400 dataset consists of 1400 classes with 420k programs
- MISIM performed 4.4-5.0x better than Aroma for Project CodeNet across ~1M programs
- We are using MISIM (and similar systems) in-house for an upcoming new MP system

Table 3: Similarity MAP@R score from CodeNet (credit: [Puri et al., 2021]).

<table>
<thead>
<tr>
<th></th>
<th>C++1000</th>
<th>C++1400</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma</td>
<td>0.17</td>
<td>0.15</td>
</tr>
<tr>
<td>MISIM</td>
<td>0.75</td>
<td>0.75</td>
</tr>
</tbody>
</table>
Conclusion

• Machine Programming Research charter
• Discussion of The Three Pillars of MP
  • Separation of intention, lifting code semantics
  • Intentional programming languages
• The Bifurcated Space in MP
  • Stochastic and Deterministic
• ControlFlag: A Self-Supervised Systems for MP
• MISIM: A Code Semantics Similarity System
Future and Open Invitation for Collaboration

Future directions
• Growing MP investment across all of Intel
• MPR is hiring PhD+ researchers; please reach out to me

Industrial and academic collaborations
• Teaching MP fundamentals at Berkeley and MIT, Fall 2021
• New Intel/NSF Machine Programming Research Center
• MAPS ’22: Program Chair Prof. Dr. Charles Sutton (Google AI)

Stay current with MP and our open-sourcing
• Intel’s Website, LinkedIn, Twitter, and YouTube MP Channel
• ControlFlag’s open-source link:
  • https://github.com/IntelLabs/control-flag