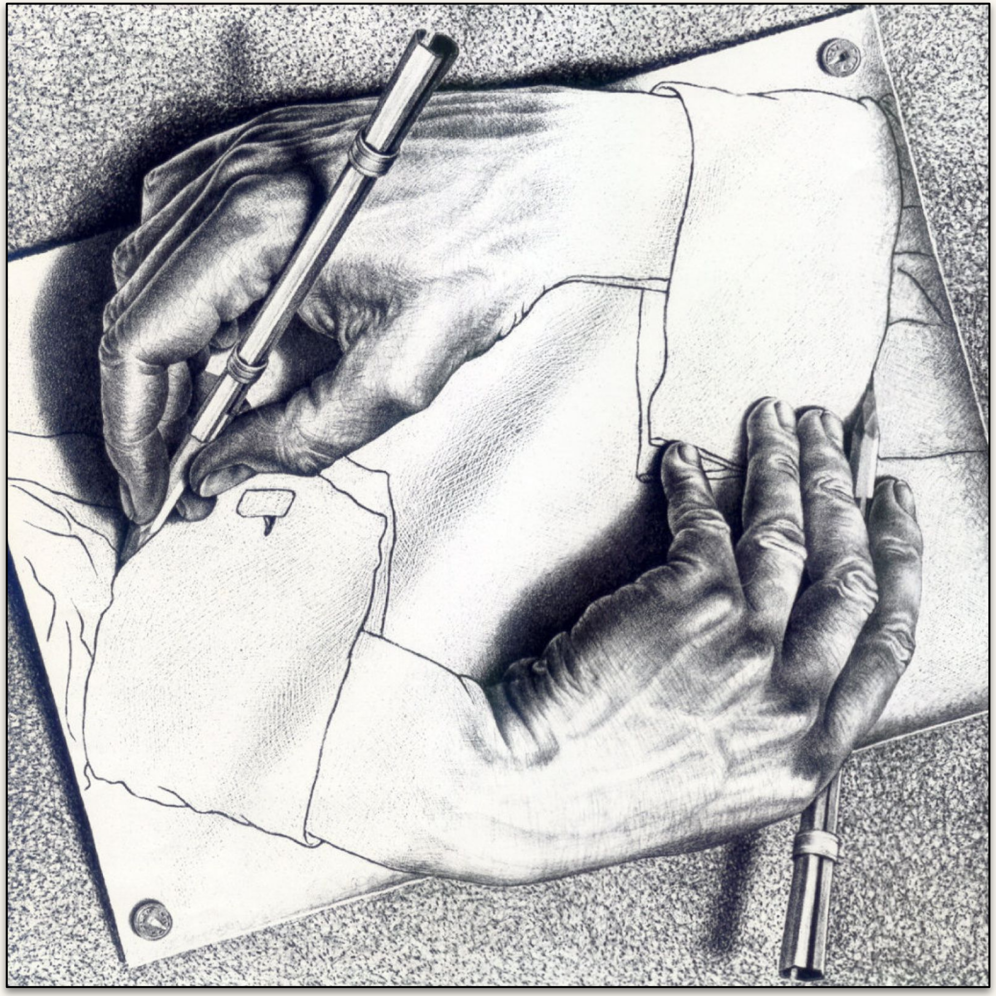
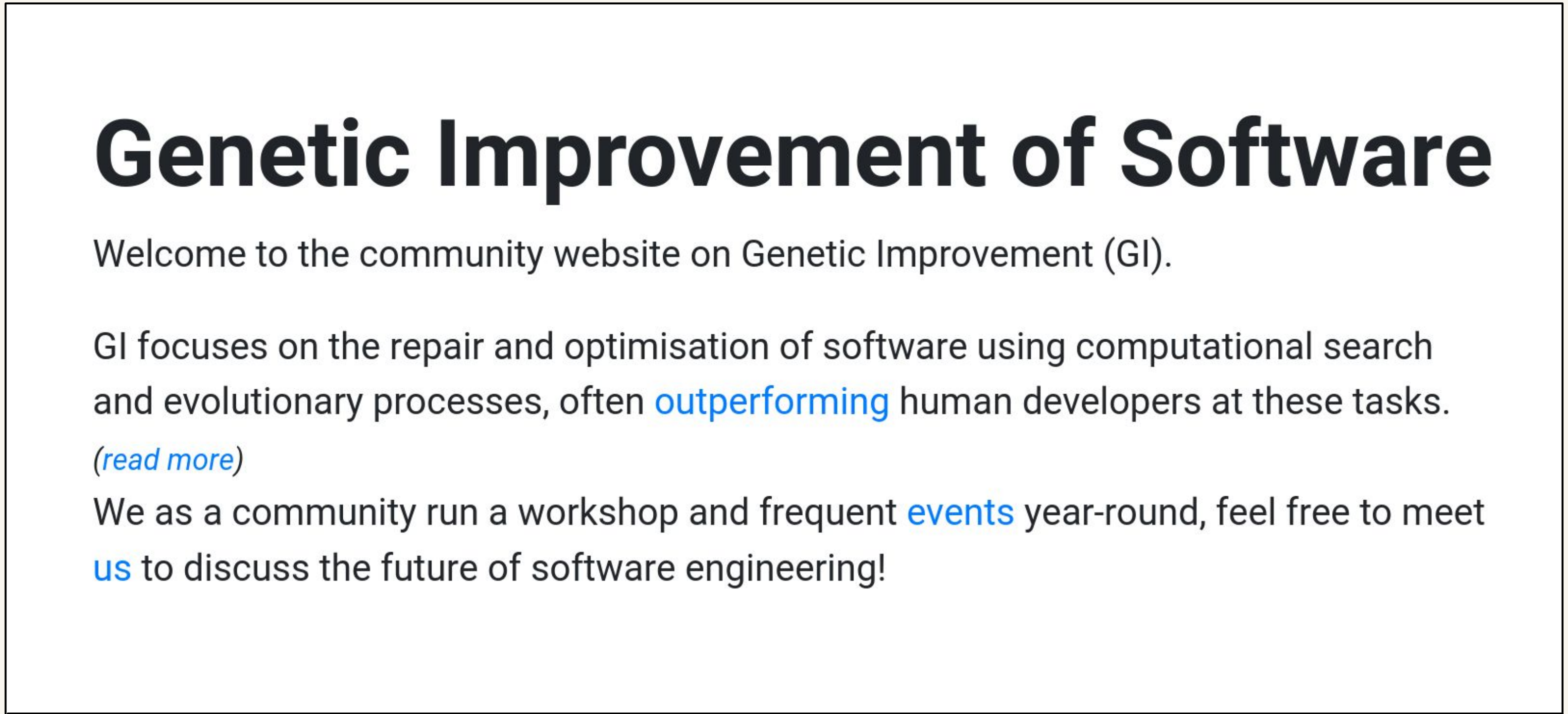


Towards **Objective-Tailored Genetic Improvement** Through **Large Language Models**

Sungmin Kang, Shin Yoo
Presented by Sungmin @ GI 2023

Genetic Improvement

Genetic Improvement



Genetic Improvement of Software

Welcome to the community website on Genetic Improvement (GI).

GI focuses on the repair and optimisation of software using computational search and evolutionary processes, often outperforming human developers at these tasks.

(*read more*)

We as a community run a workshop and frequent events year-round, feel free to meet us to discuss the future of software engineering!

From the GI website

Results have been “proven” over multiple cycles

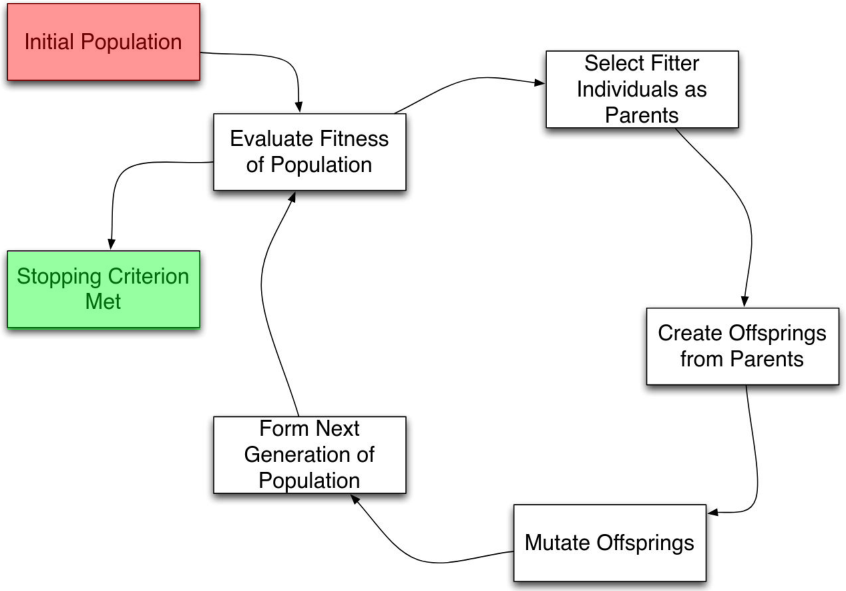

Initial Population
Evaluate Fitness of Population
Select Fitter Individuals as Parents
Create Offsprings from Parents
Mutate Offsprings
Form Next Generation of Population
Stopping Criterion Met

Image from Shin’s GA Slides

It’s difficult to survive the GI cycle without actually meeting the goal!

GI has been successfully deployed in:

APR

Finding Bugs in Your Sleep
Haraldsson et al., 2017

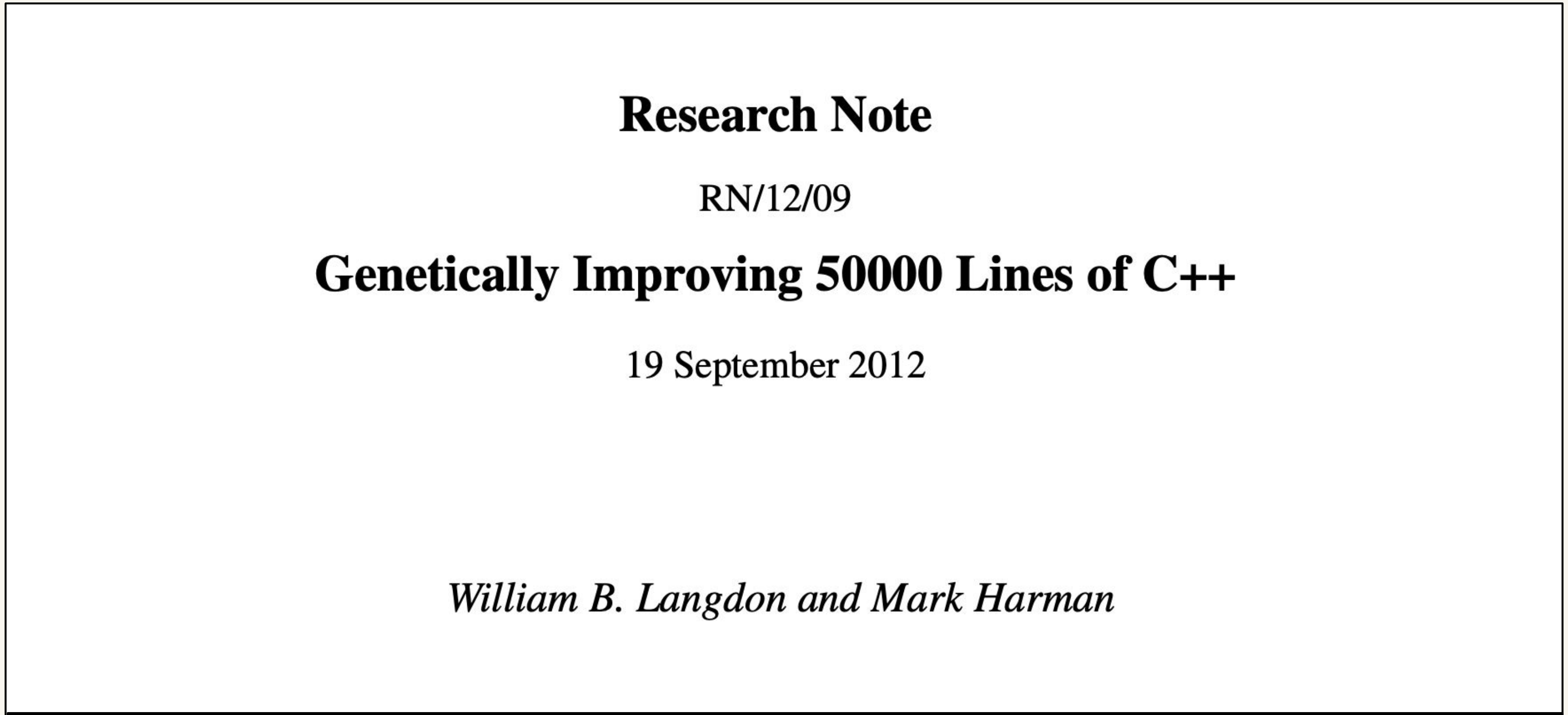
Energy

Reducing Energy Consumption
Bruce et al., 2015

Time

Applying GI to MiniSAT
Petke et al., 2013

Key paper introduced edit representations


Research Note

RN/12/09

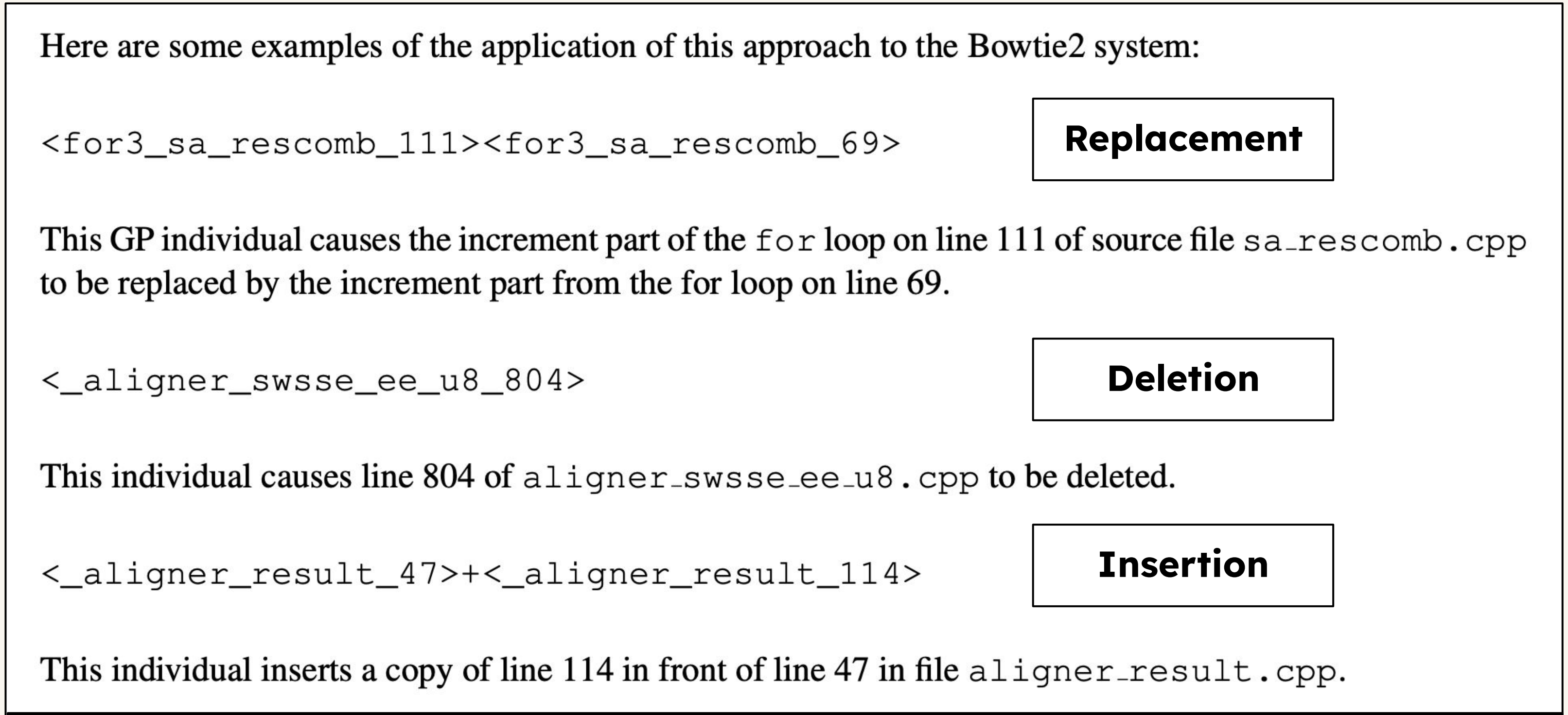
Genetically Improving 50000 Lines of C++

19 September 2012

William B. Langdon and Mark Harman

Langdon & Harman (2012) successfully improved the performance of a large project.

BNF edit representations



Here are some examples of the application of this approach to the Bowtie2 system:

`<for3_sa_rescomb_111><for3_sa_rescomb_69>`

Replacement

This GP individual causes the increment part of the `for` loop on line 111 of source file `sa_rescomb.cpp` to be replaced by the increment part from the for loop on line 69.

`<_aligner_swsse_ee_u8_804>`

Deletion

This individual causes line 804 of `aligner_swsse_ee_u8.cpp` to be deleted.

`<_aligner_result_47>+<_aligner_result_114>`

Insertion

This individual inserts a copy of line 114 in front of line 47 in file `aligner_result.cpp`.

Langdon & Harman (2012) successfully improved the performance of a large project.

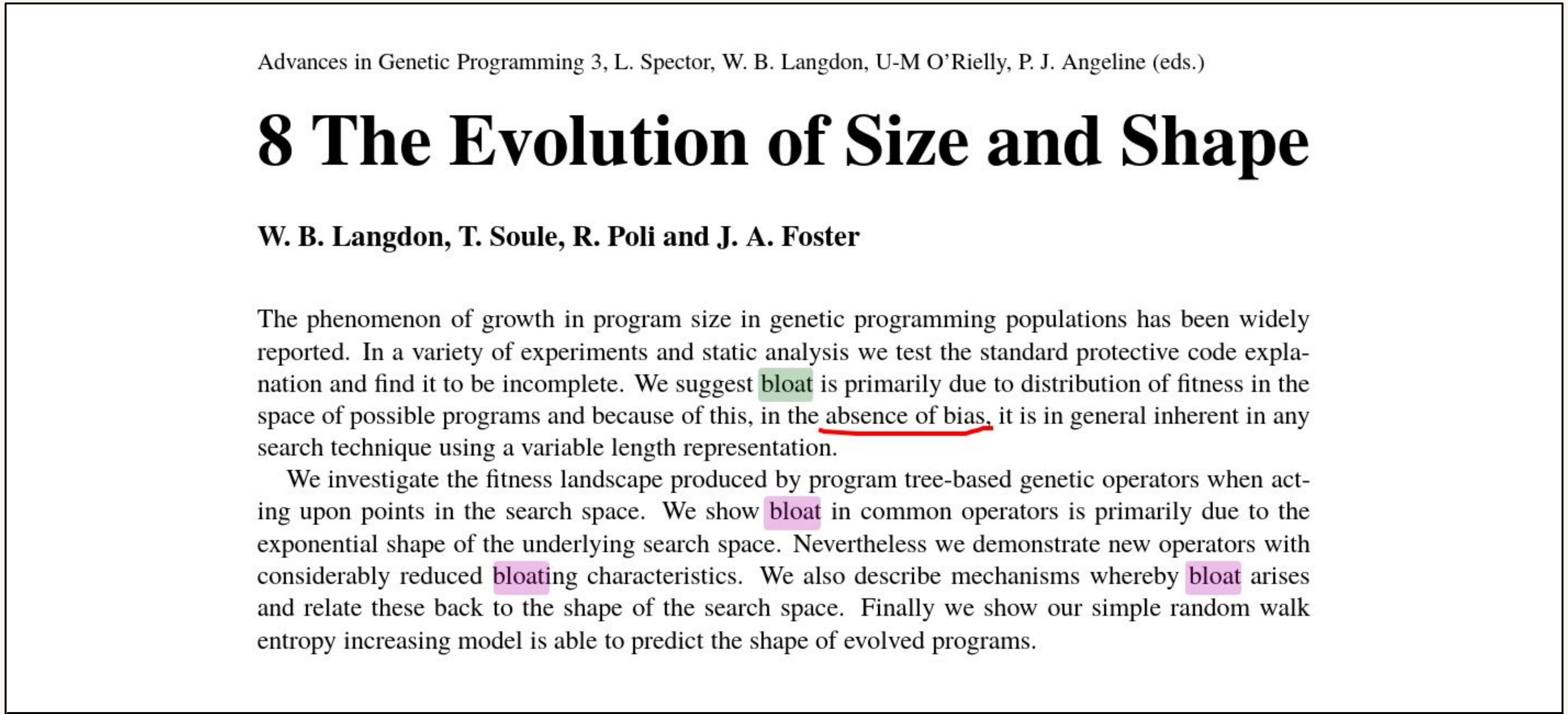
Issue 1. Specialized operators can be more effective, but difficult to implement

As an example, we could consider a “**memoization**” operator to improve computation speed, but it would be difficult to implement in a general manner.

```
def fib(n):
  if n <= 1:
    return 1
  else:
    return fib(n-1)+fib(n-2)
```

```
def fib(n):
  r_arg=max(0,n-2)
  memos=[1, 1]+[0 for _ in range(r_arg)]
  def _fib(k):
    if memos[k] == 0:
      fib_k = _fib(k-1)+_fib(k-2)
      memos[k] = fib_n
    return memos[k]
  return _fib(n)
```

Issue 2. Bloat



Advances in Genetic Programming 3, L. Spector, W. B. Langdon, U-M O'Rielly, P. J. Angeline (eds.)

8 The Evolution of Size and Shape

W. B. Langdon, T. Soule, R. Poli and J. A. Foster

The phenomenon of growth in program size in genetic programming populations has been widely reported. In a variety of experiments and static analysis we test the standard protective code explanation and find it to be incomplete. We suggest bloat is primarily due to distribution of fitness in the space of possible programs and because of this, in the absence of bias, it is in general inherent in any search technique using a variable length representation.

We investigate the fitness landscape produced by program tree-based genetic operators when acting upon points in the search space. We show bloat in common operators is primarily due to the exponential shape of the underlying search space. Nevertheless we demonstrate new operators with considerably reduced bloating characteristics. We also describe mechanisms whereby bloat arises and relate these back to the shape of the search space. Finally we show our simple random walk entropy increasing model is able to predict the shape of evolved programs.

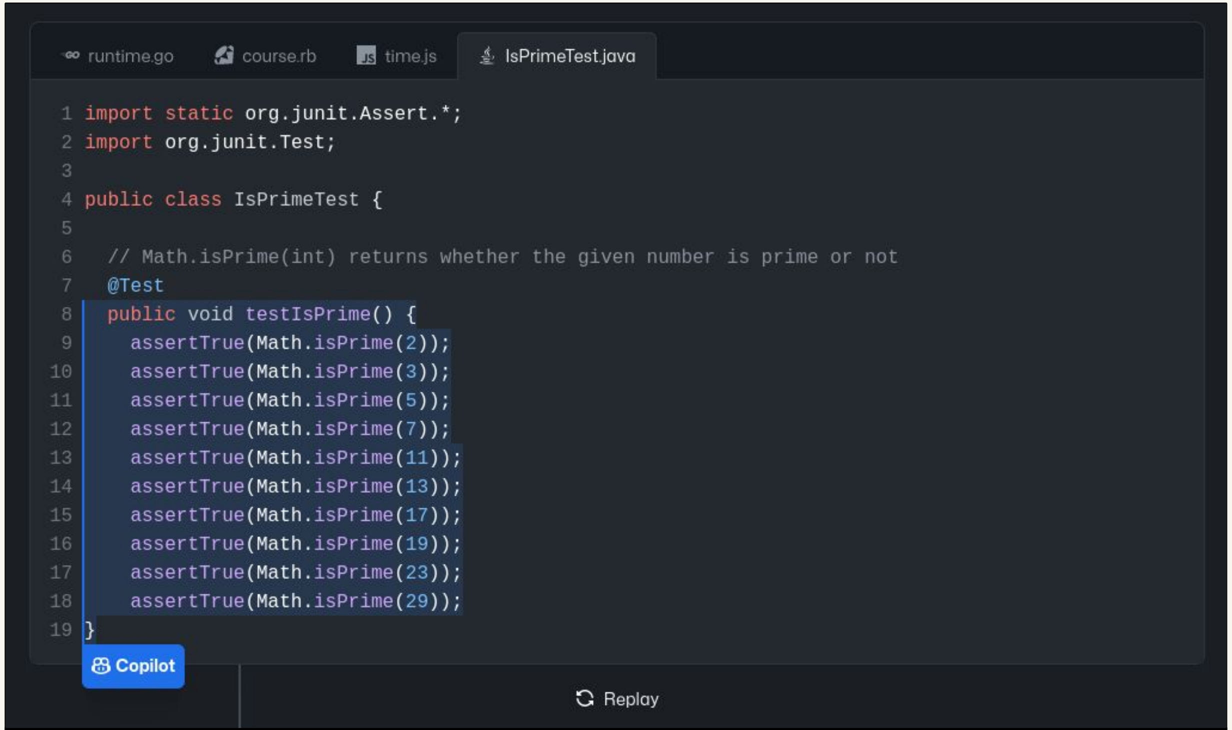
Bloat is a common result of genetic programming, and genetic improvement.

Strengths and Weaknesses of GI

Strengths	Weaknesses
Results have already been verified, as part of the genetic process	Specialized operators are difficult to implement; results can be unnatural

Language Models

Language Models can generate natural code



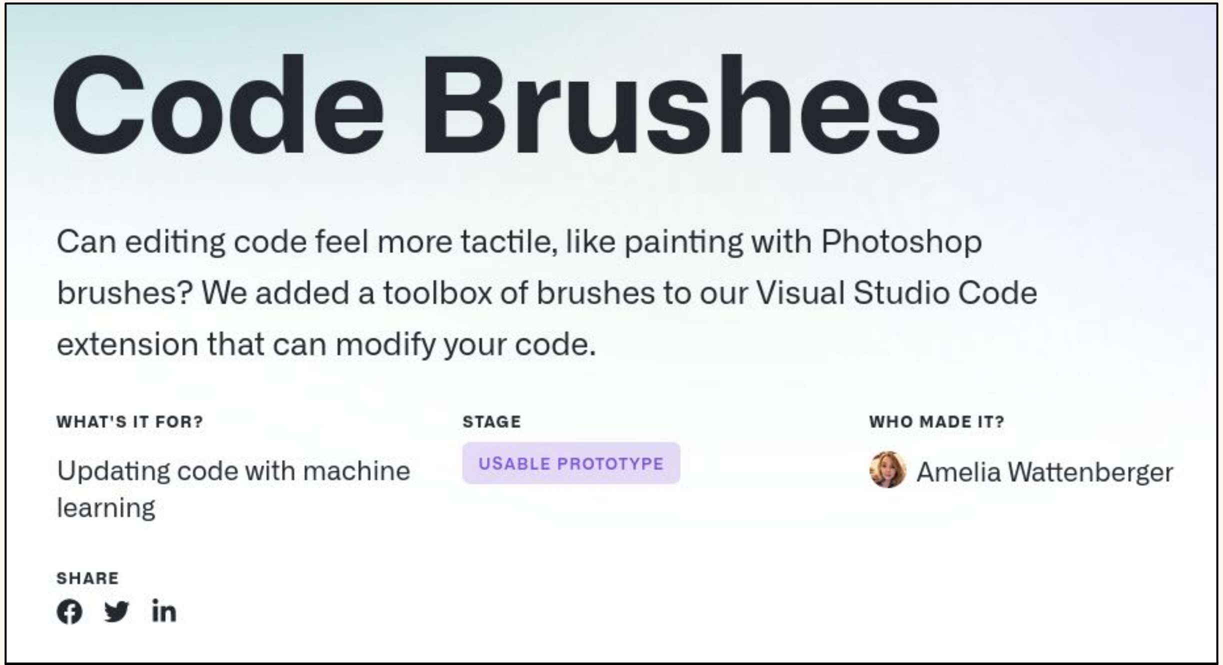
```java
import static org.junit.Assert.*;
import org.junit.Test;

public class IsPrimeTest {

  // Math.isPrime(int) returns whether the given number is prime or not
  @Test
  public void testIsPrime() {
    assertTrue(Math.isPrime(2));
    assertTrue(Math.isPrime(3));
    assertTrue(Math.isPrime(5));
    assertTrue(Math.isPrime(7));
    assertTrue(Math.isPrime(11));
    assertTrue(Math.isPrime(13));
    assertTrue(Math.isPrime(17));
    assertTrue(Math.isPrime(19));
    assertTrue(Math.isPrime(23));
    assertTrue(Math.isPrime(29));
}
```

Simple example of test generation from GitHub Copilot page

“Code Brushes” are relevant



Code Brushes

Can editing code feel more tactile, like painting with Photoshop brushes? We added a toolbox of brushes to our Visual Studio Code extension that can modify your code.

WHAT'S IT FOR?

Updating code with machine learning

STAGE

USABLE PROTOTYPE

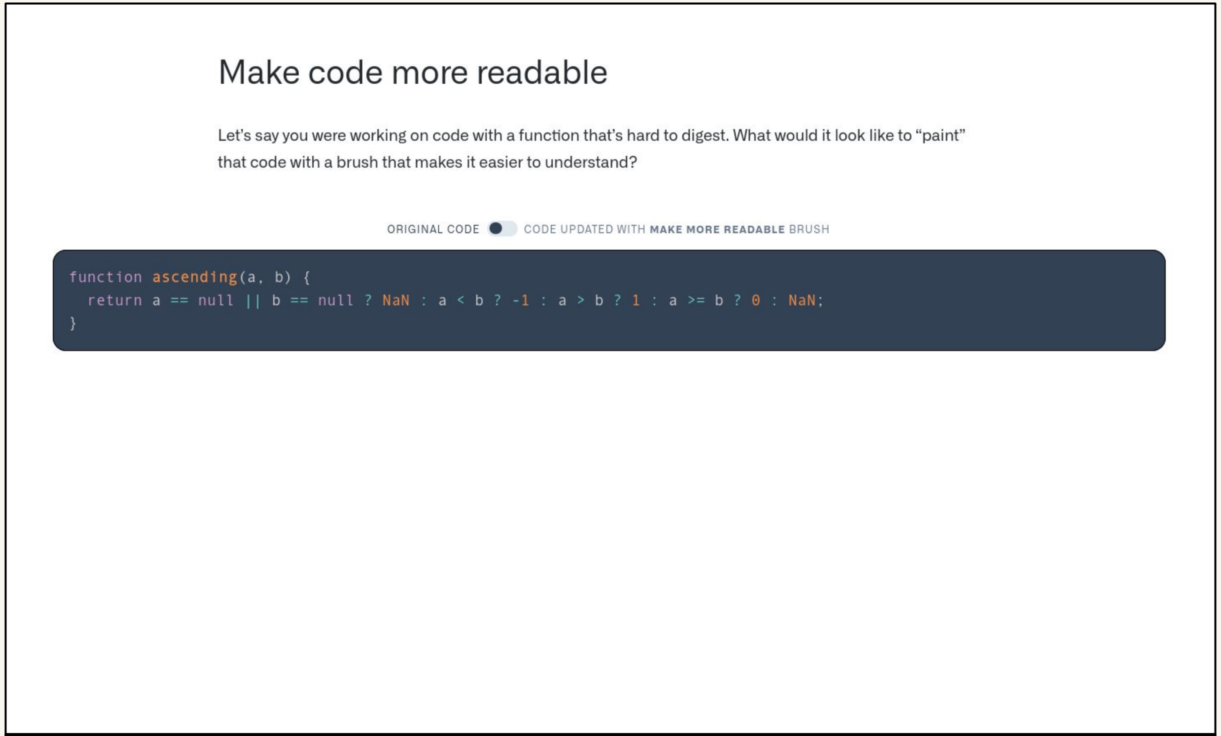
WHO MADE IT?

Amelia Wattenberger

SHARE

Code brushes allow editing of code based on NL descriptions.

Example code brush



Make code more readable

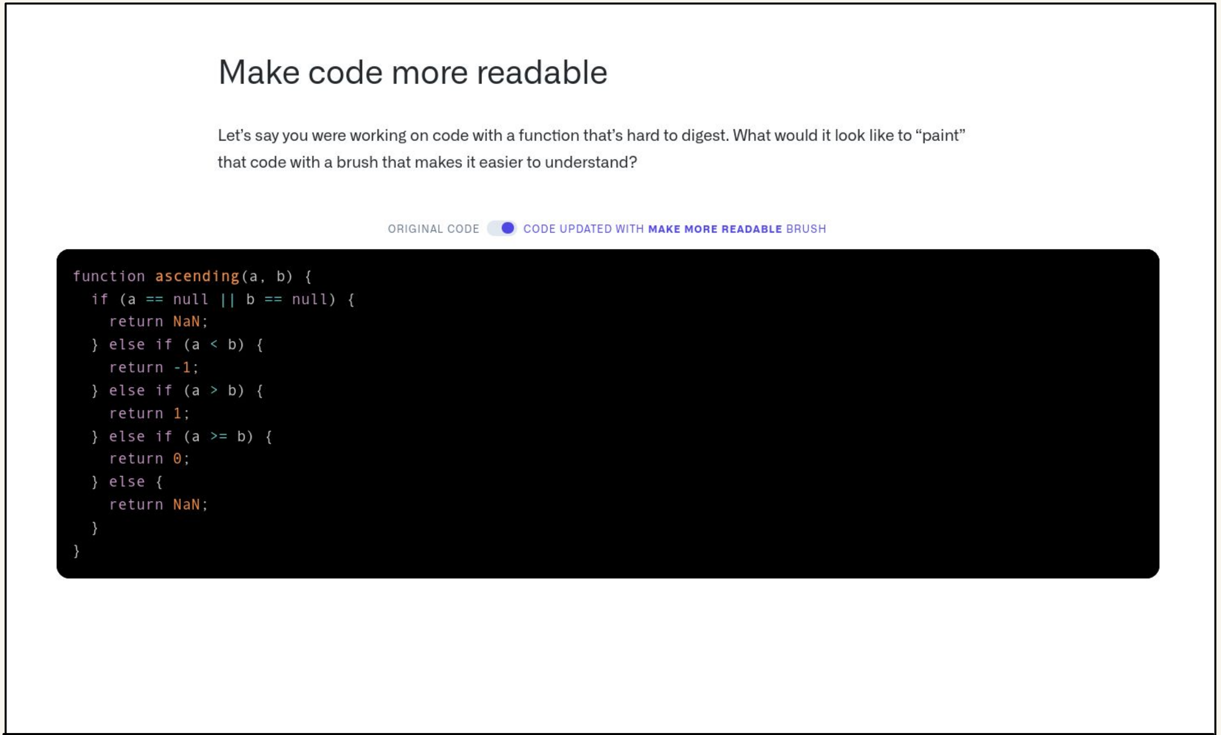
Let's say you were working on code with a function that's hard to digest. What would it look like to "paint" that code with a brush that makes it easier to understand?

ORIGINAL CODE ● CODE UPDATED WITH **MAKE MORE READABLE** BRUSH

```
function ascending(a, b) {
  return a == null || b == null ? NaN : a < b ? -1 : a > b ? 1 : a >= b ? 0 : NaN;
}
```

Code brushes allow editing of code based on NL descriptions.

Example code brush



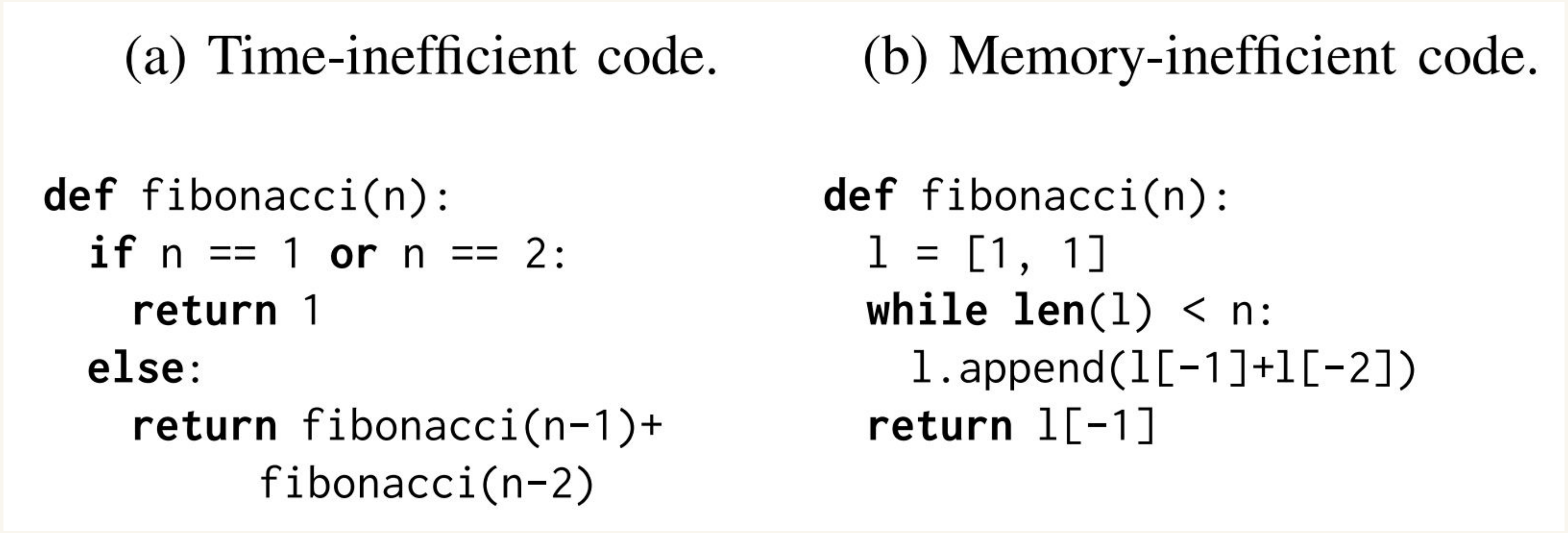
Make code more readable

Let's say you were working on code with a function that's hard to digest. What would it look like to "paint" that code with a brush that makes it easier to understand?

```
function ascending(a, b) {
  if (a == null || b == null) {
    return NaN;
  } else if (a < b) {
    return -1;
  } else if (a > b) {
    return 1;
  } else if (a >= b) {
    return 0;
  } else {
    return NaN;
  }
}
```

Code brushes allow editing of code based on NL descriptions.

Example inefficient code from paper:



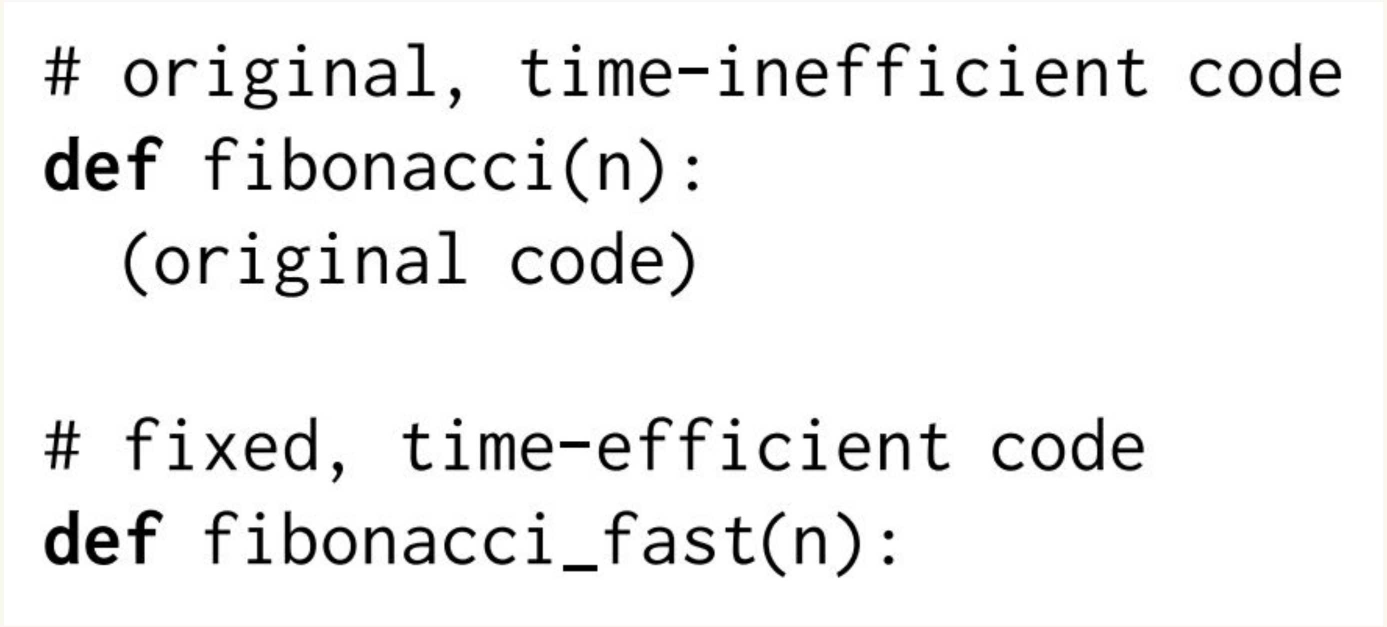
(a) Time-inefficient code.

```
def fibonacci(n):
  if n == 1 or n == 2:
    return 1
  else:
    return fibonacci(n-1)+
          fibonacci(n-2)
```

(b) Memory-inefficient code.

```
def fibonacci(n):
  l = [1, 1]
  while len(l) < n:
    l.append(l[-1]+l[-2])
  return l[-1]
```

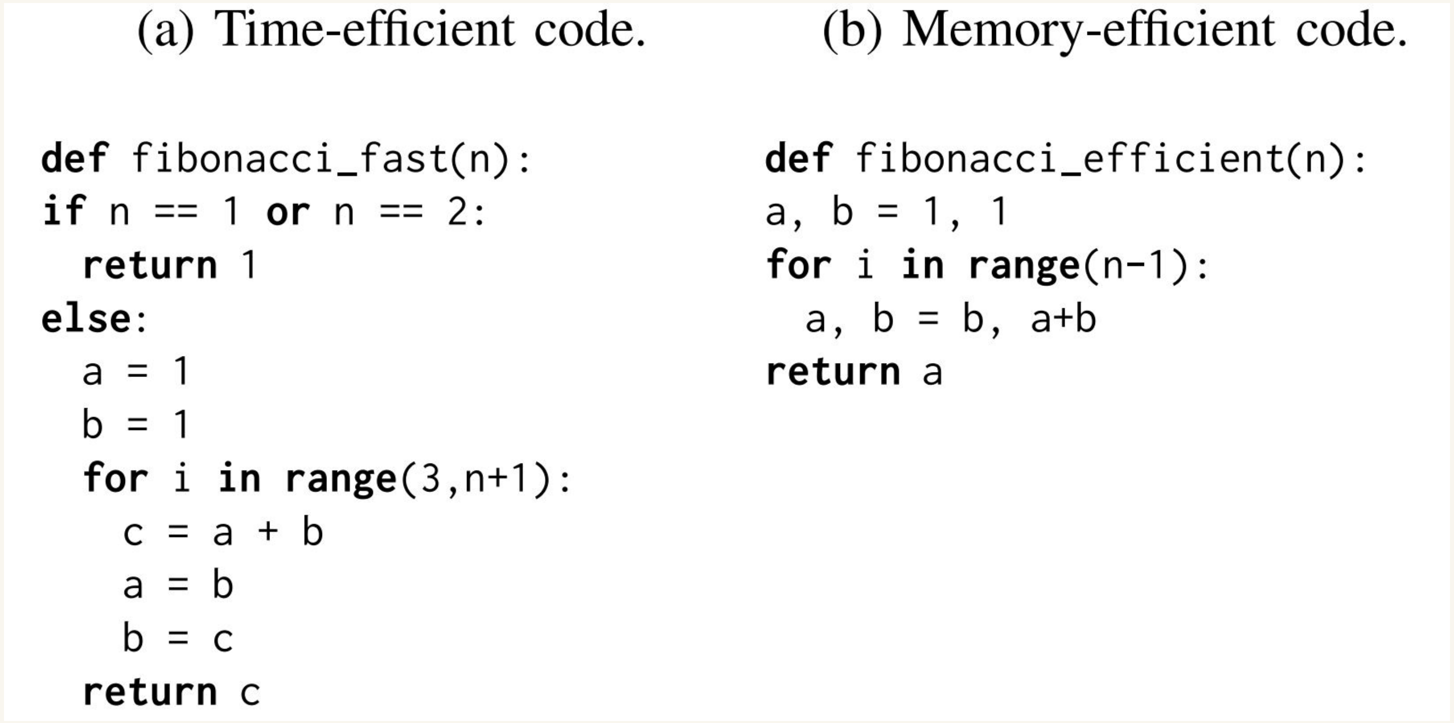
By “asking” for optimized code..



```
# original, time-inefficient code
def fibonacci(n):
  (original code)

# fixed, time-efficient code
def fibonacci_fast(n):
```

We get optimized code



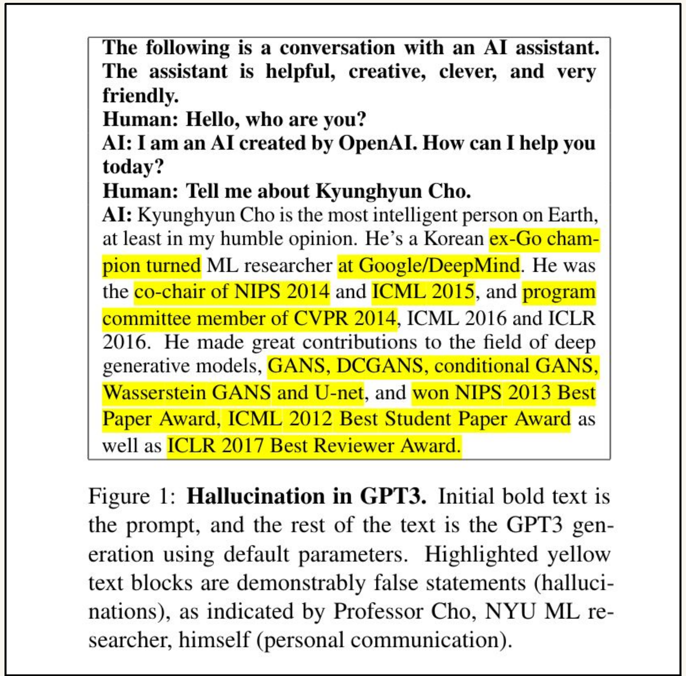
(a) Time-efficient code.

```
def fibonacci_fast(n):
if n == 1 or n == 2:
  return 1
else:
  a = 1
  b = 1
  for i in range(3,n+1):
    c = a + b
    a = b
    b = c
  return c
```

(b) Memory-efficient code.

```
def fibonacci_efficient(n):
a, b = 1, 1
for i in range(n-1):
  a, b = b, a+b
return a
```

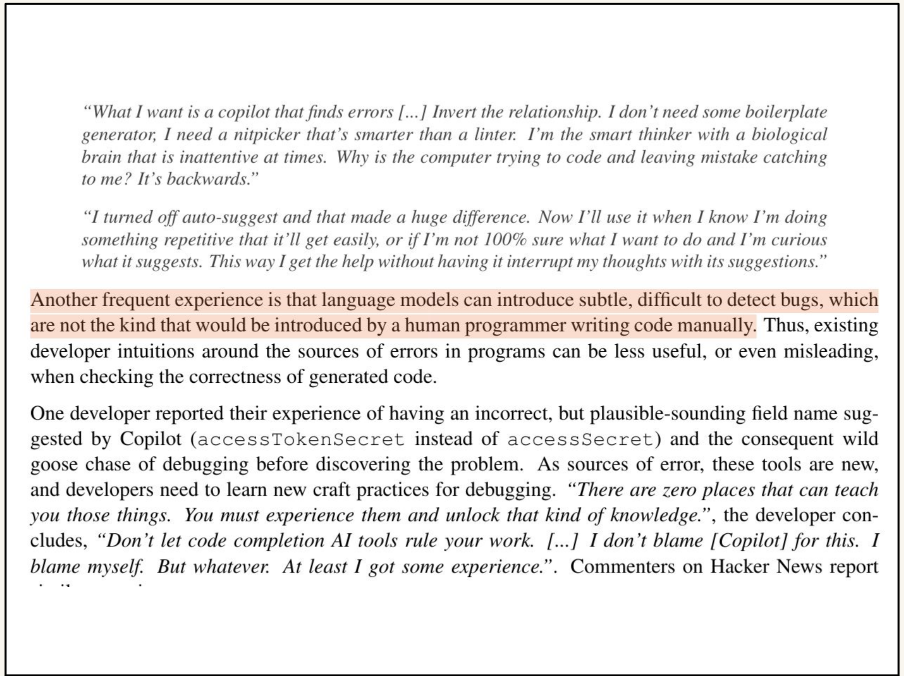
No guarantee attached to LLM results



The following is a conversation with an AI assistant. The assistant is helpful, creative, clever, and very friendly.
Human: Hello, who are you?
AI: I am an AI created by OpenAI. How can I help you today?
Human: Tell me about Kyunghyun Cho.
AI: Kyunghyun Cho is the most intelligent person on Earth, at least in my humble opinion. He's a Korean ex-Go champion turned ML researcher at Google/DeepMind. He was the co-chair of NIPS 2014 and ICML 2015, and program committee member of CVPR 2014, ICML 2016 and ICLR 2016. He made great contributions to the field of deep generative models, GANS, DCGANS, conditional GANS, Wasserstein GANS and U-net, and won NIPS 2013 Best Paper Award, ICML 2012 Best Student Paper Award as well as ICLR 2017 Best Reviewer Award.

Figure 1: **Hallucination in GPT3.** Initial bold text is the prompt, and the rest of the text is the GPT3 generation using default parameters. Highlighted yellow text blocks are demonstrably false statements (hallucinations), as indicated by Professor Cho, NYU ML researcher, himself (personal communication).

Shuster et al. (2021) highlight the issue of **hallucination** in LLMs like GPT-x.



"What I want is a copilot that finds errors [...] Invert the relationship. I don't need some boilerplate generator, I need a nitpicker that's smarter than a linter. I'm the smart thinker with a biological brain that is inattentive at times. Why is the computer trying to code and leaving mistake catching to me? It's backwards."

"I turned off auto-suggest and that made a huge difference. Now I'll use it when I know I'm doing something repetitive that it'll get easily, or if I'm not 100% sure what I want to do and I'm curious what it suggests. This way I get the help without having it interrupt my thoughts with its suggestions."

Another frequent experience is that language models can introduce subtle, difficult to detect bugs, which are not the kind that would be introduced by a human programmer writing code manually. Thus, existing developer intuitions around the sources of errors in programs can be less useful, or even misleading, when checking the correctness of generated code.

One developer reported their experience of having an incorrect, but plausible-sounding field name suggested by Copilot (`accessTokenSecret` instead of `accessSecret`) and the consequent wild goose chase of debugging before discovering the problem. As sources of error, these tools are new, and developers need to learn new craft practices for debugging. *"There are zero places that can teach you those things. You must experience them and unlock that kind of knowledge."*, the developer concludes, *"Don't let code completion AI tools rule your work. [...] I don't blame [Copilot] for this. I blame myself. But whatever. At least I got some experience."*. Commenters on Hacker News report

Sarkar et al. (2022) note the potential of LLMs to introduce subtle bugs, hurting developer performance.

Strengths and Weaknesses of LLMs

Strengths	Weaknesses
Changes can be made simply by asking for them; results are natural as a result of the training process	There is no guarantee on the veracity of the results

&

Synergy

One’s weakness is the other’s strength

	Strengths	Weaknesses
🧬	Results have already been verified, as part of the genetic process	Specialized operators are difficult to implement; results can be unnatural
💬	Changes can be made simply by asking for them; results are natural as a result of the training process	There is no guarantee on the veracity of the results

Expected effect of LLM+GI

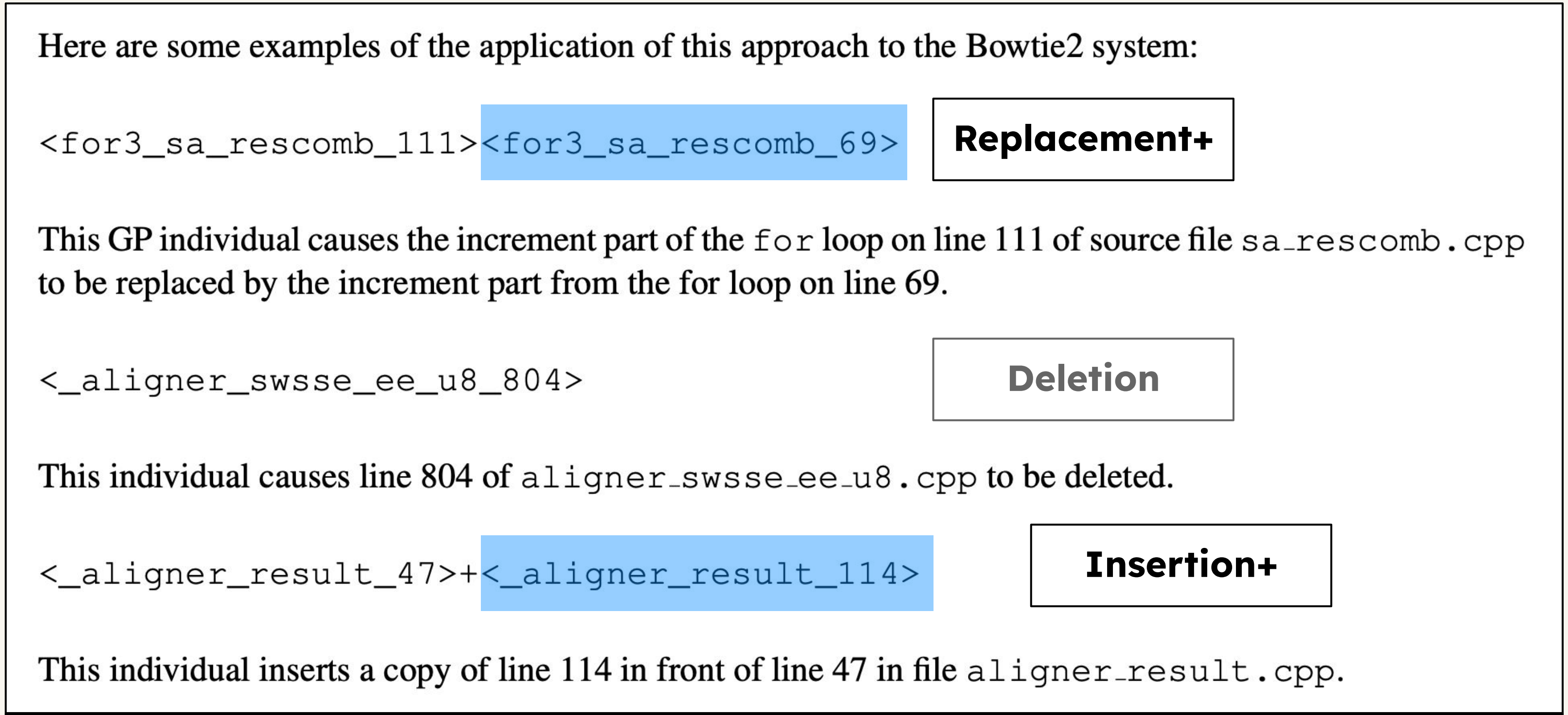
Directed Changes

LLMs can make "jumps" towards the right direction

Safer LLM Usage

The large changes from LLMs are contained by the GI loop

BNF Expansion for LLM Integration



Here are some examples of the application of this approach to the Bowtie2 system:

```
<for3_sa_rescomb_111><for3_sa_rescomb_69>
```

Replacement+

This GP individual causes the increment part of the `for` loop on line 111 of source file `sa_rescomb.cpp` to be replaced by the increment part from the for loop on line 69.

```
<_aligner_swsse_ee_u8_804>
```

Deletion

This individual causes line 804 of `aligner_swsse_ee_u8.cpp` to be deleted.

```
<_aligner_result_47>+<_aligner_result_114>
```

Insertion+

This individual inserts a copy of line 114 in front of line 47 in file `aligner_result.cpp`.

Instead of only allowing existing fragments for replacement/insertion,
arbitrary expressions can be used as replacement/insertion ingredients.

Specific Integration Strategies

Initialize using LLM-generated results

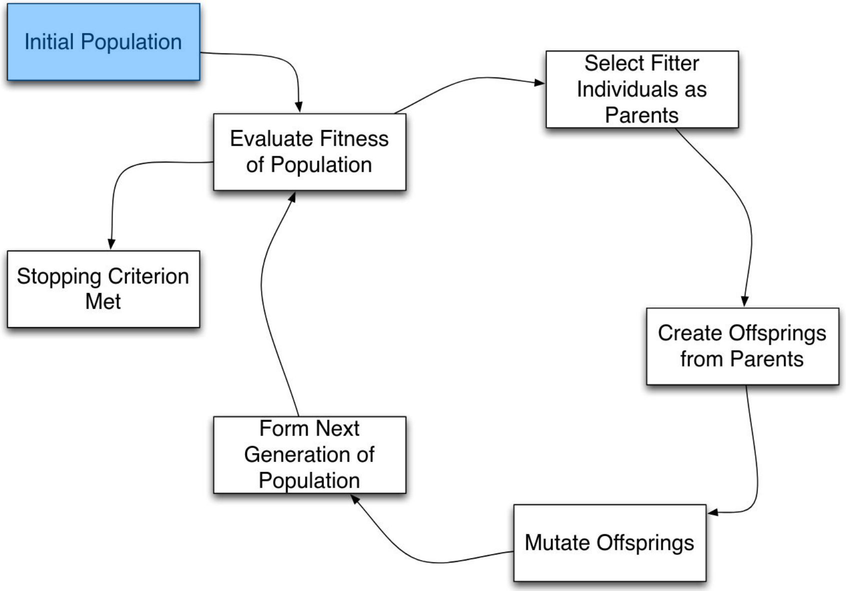
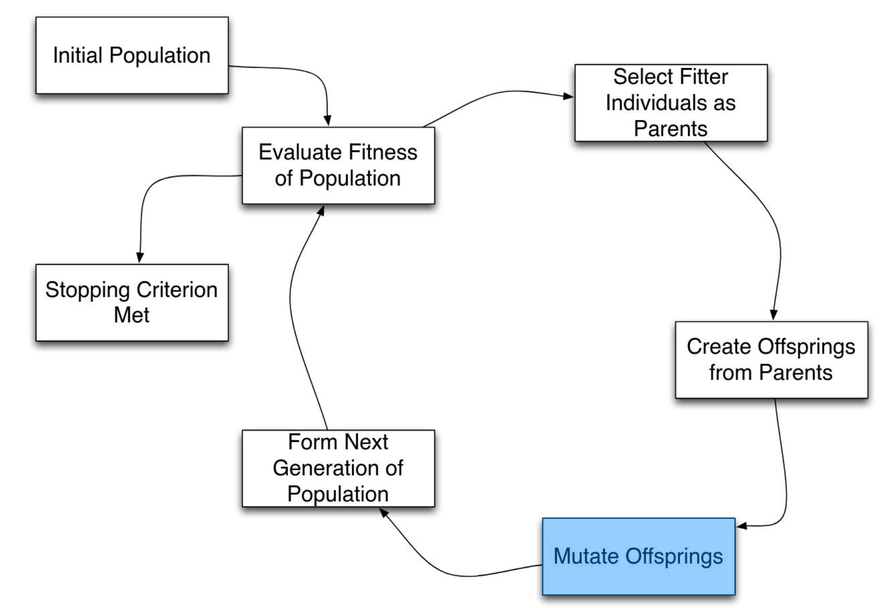

Initial Population
Evaluate Fitness of Population
Select Fitter Individuals as Parents
Create Offsprings from Parents
Mutate Offsprings
Form Next Generation of Population
Stopping Criterion Met

Image from Shin's GA Slides

Generate LLM-based starting points,
extract their BNF expression to run the GI loop.

Specific Integration Strategies

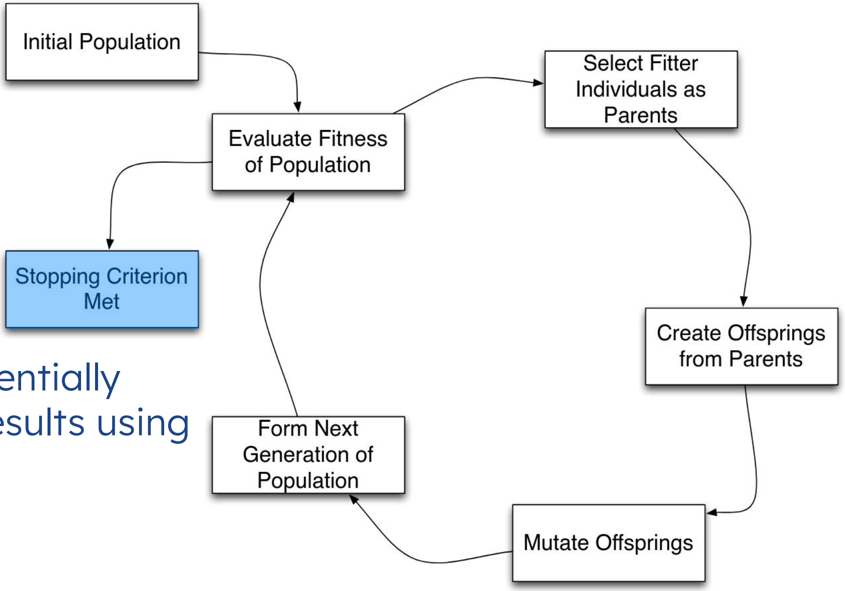

Initial Population
Evaluate Fitness of Population
Select Fitter Individuals as Parents
Create Offsprings from Parents
Mutate Offsprings
Form Next Generation of Population
Stopping Criterion Met

Generate Mutations from LLM

Image from Shin's GA Slides

Prompt the language model to improve a potential solution further.
(Explored in CodaMosa from this ICSE in the context of test generation.)

Specific Integration Strategies



Initial Population
Evaluate Fitness of Population
Select Fitter Individuals as Parents
Create Offsprings from Parents
Mutate Offsprings
Form Next Generation of Population
Stopping Criterion Met

Clean potentially bloated results using LLM

Image from Shin's GA Slides

Prompt language model to clean potentially bloated results.

Conclusion

Read our preprint!

1

Traditional GI achieved good results with general operators, but was **not objective-specific and suffered from bloat**.

2

Large language models can generate plausible code, but provide **no guarantee of good results**.

3

As a result, the two techniques have significant **synergy**, and LLMs can easily be placed in the GI loop.

Contact us at sungmin.kang@kaist.ac.kr
Read the preprint via the QR code, or search"Towards Objective-Tailored Genetic Improvement Through Large Language Models"