

# Scaling Program Synthesis by Exploiting Existing Code

**James Bornholt**

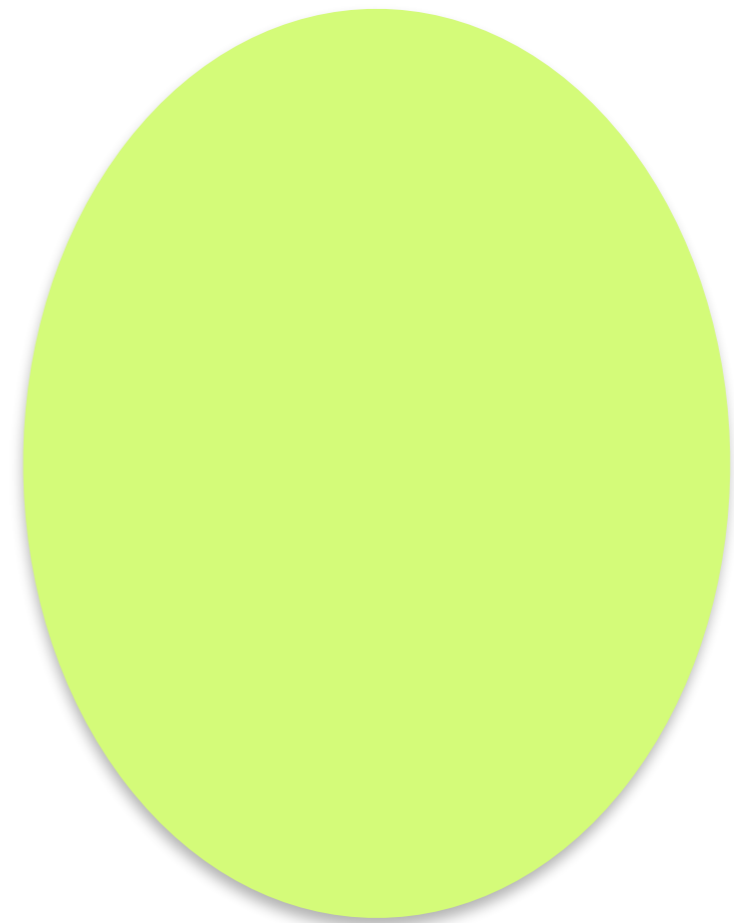
Emina Torlak

University of Washington

# Synthesis: write programs automatically



Syntax

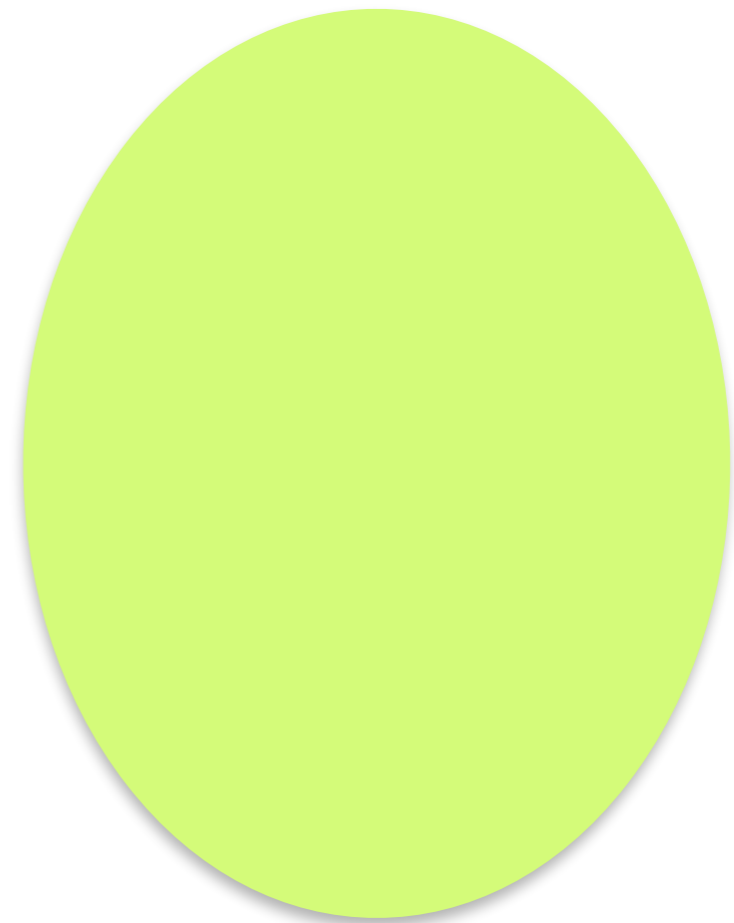


Semantics

# Synthesis: write programs automatically



Syntax

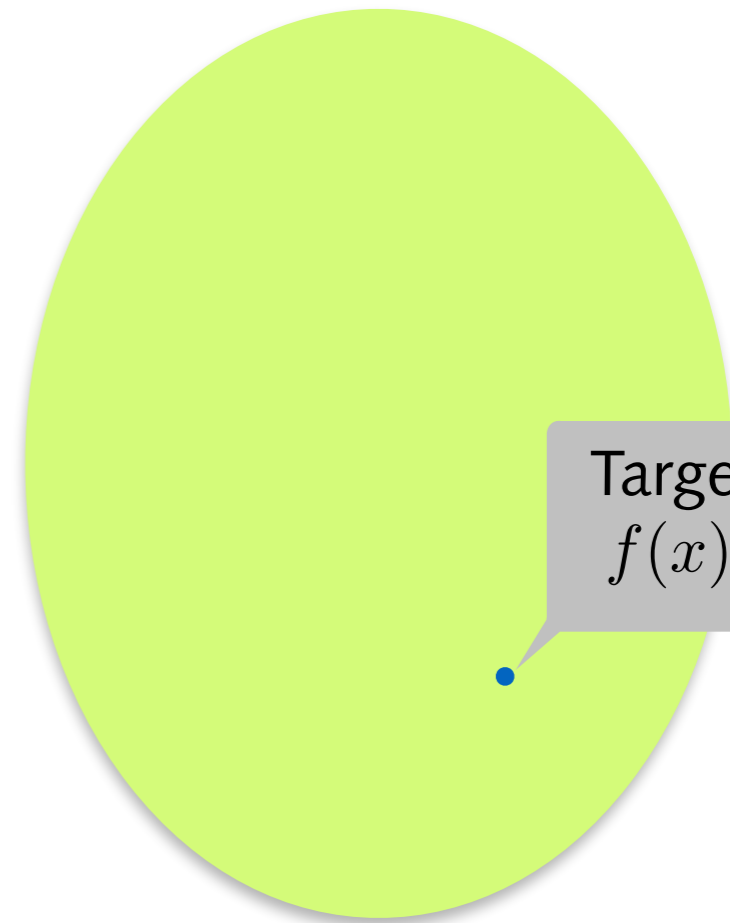


Semantics

# Synthesis: write programs automatically



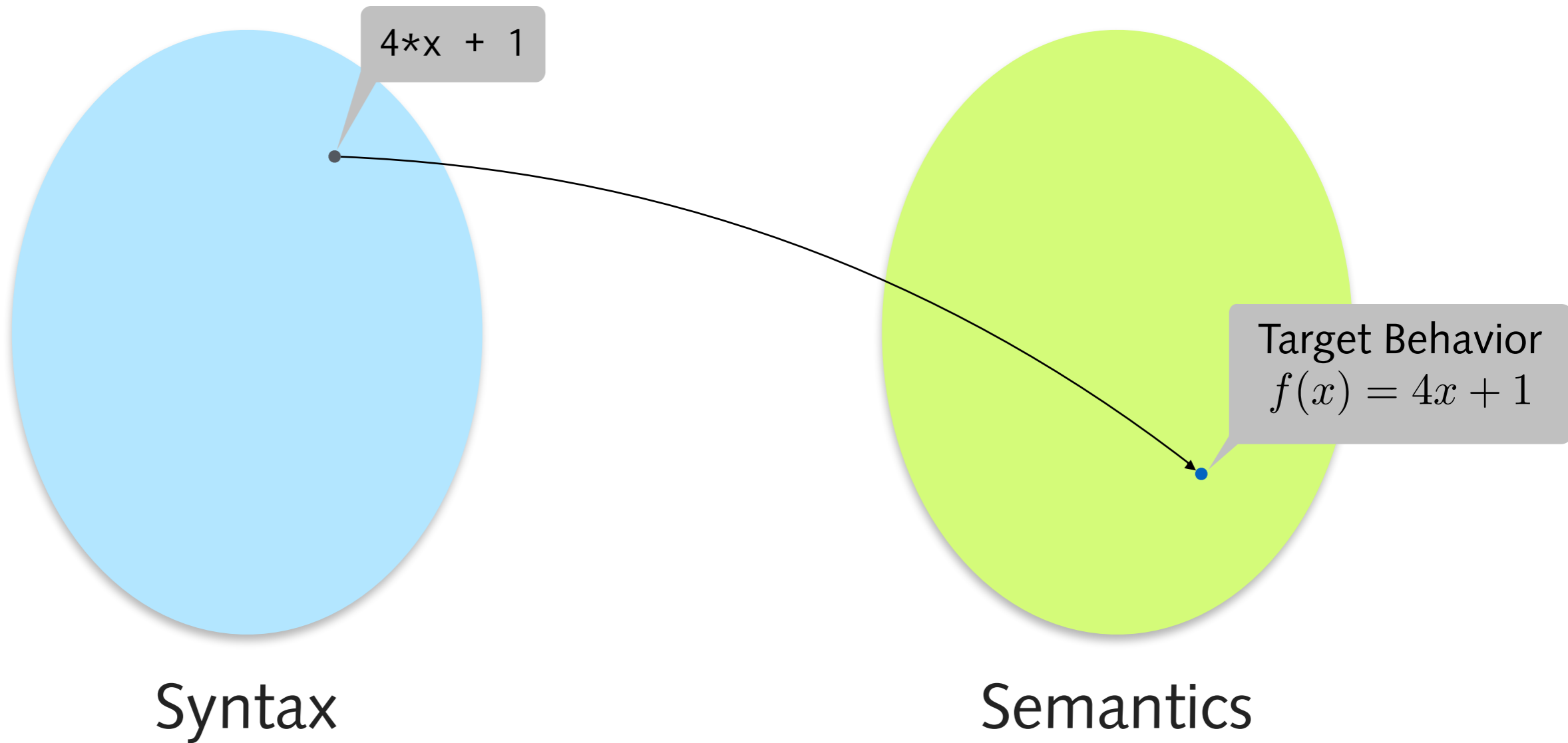
Syntax



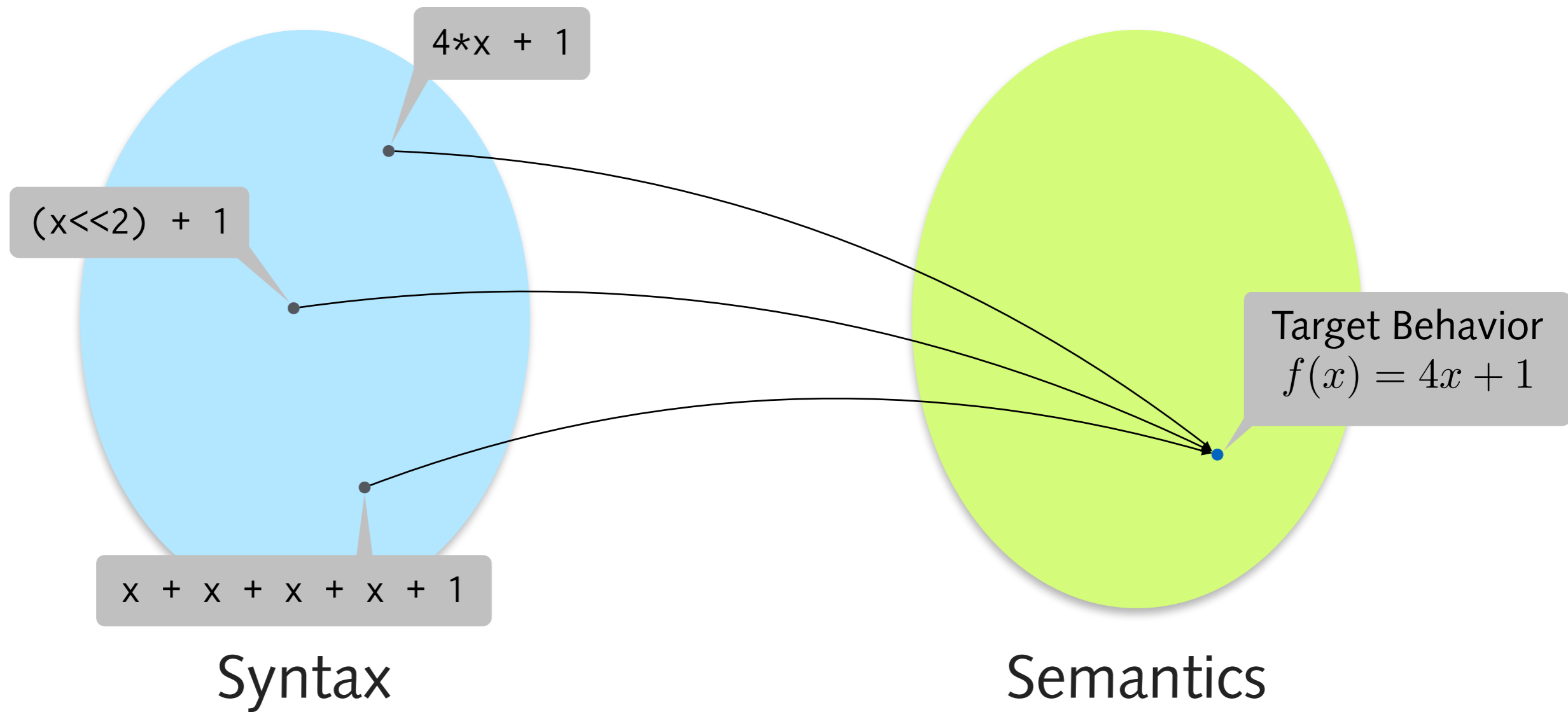
Target Behavior  
 $f(x) = 4x + 1$

Semantics

# Synthesis: write programs automatically



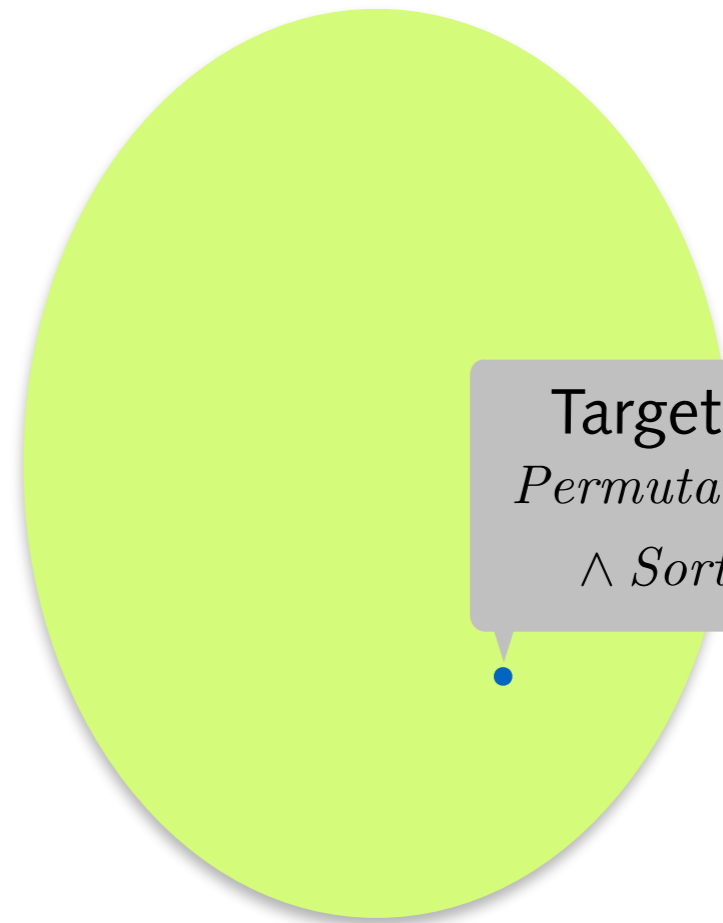
# Synthesis: write programs automatically



# Synthesis: write programs automatically



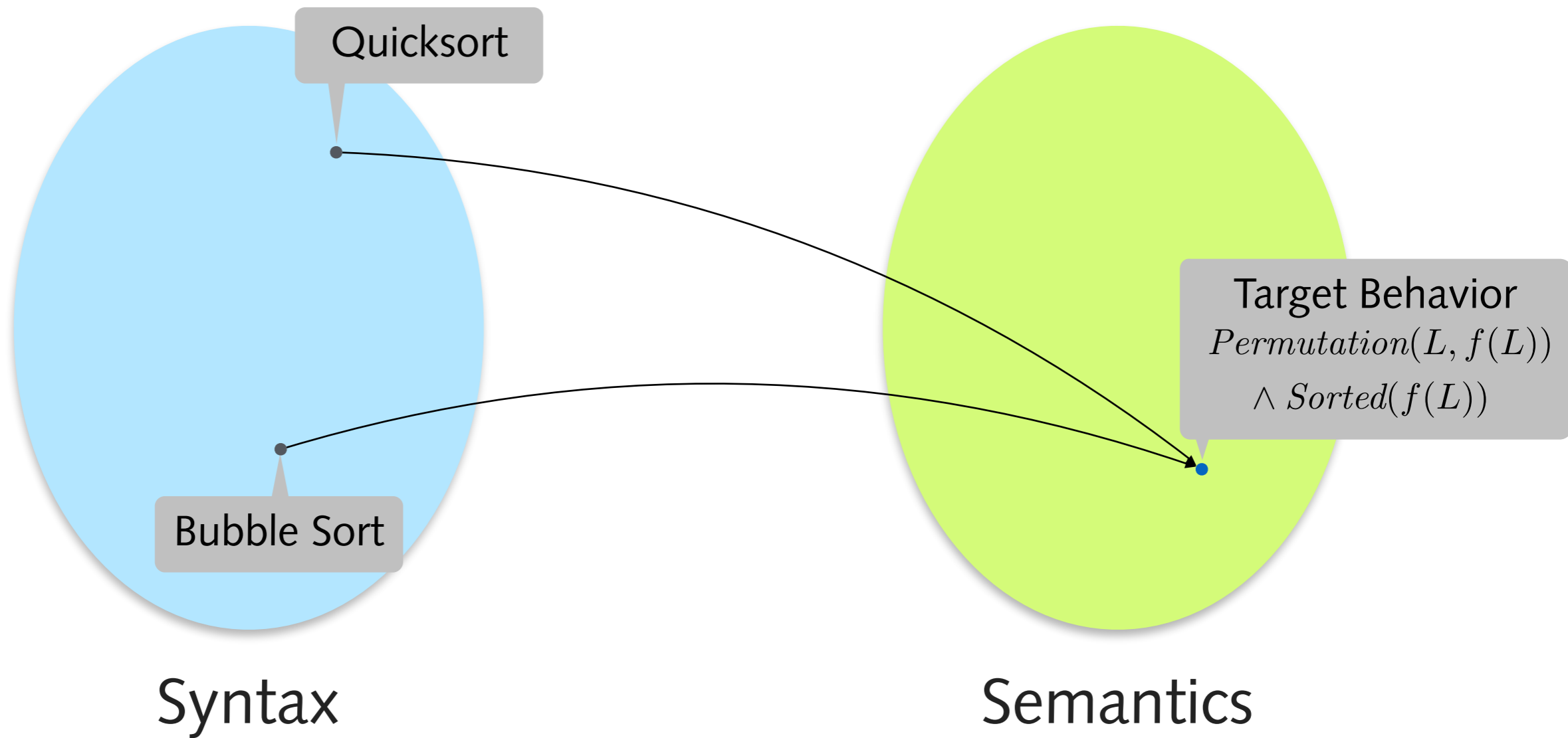
Syntax



Semantics

Target Behavior  
 $Permutation(L, f(L))$   
 $\wedge Sorted(f(L))$

# Synthesis: write programs automatically





# The success of synthesis

End-user programming  
by example [FlashFill, POPL'11]

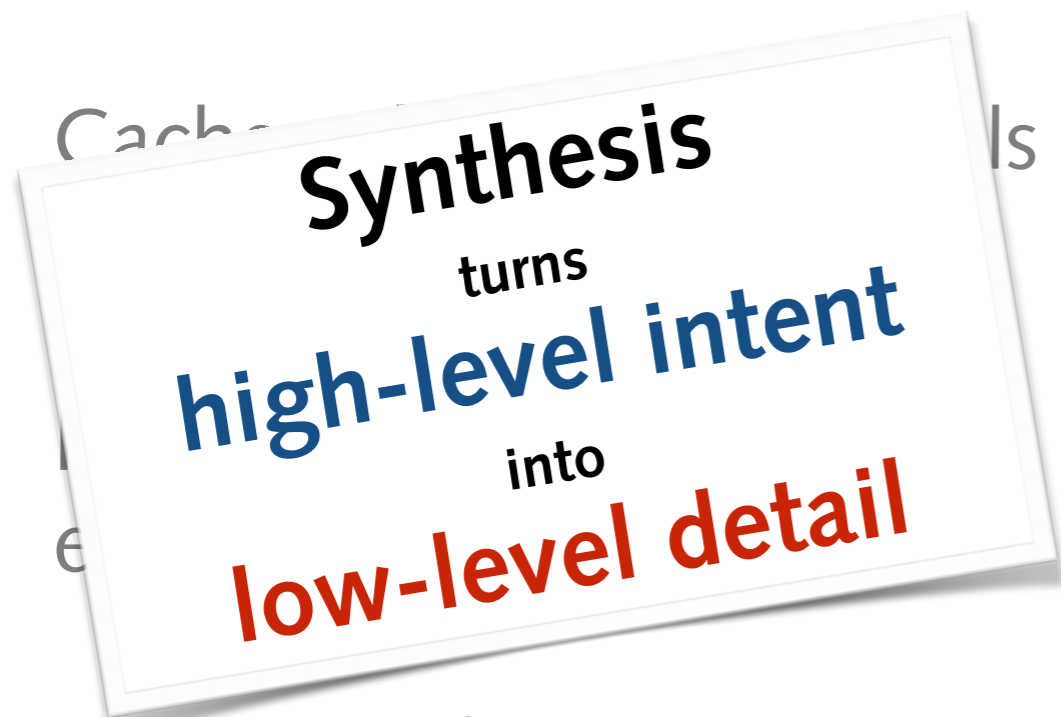
Cache coherence protocols  
[Transit, PLDI'13]

Parallel browser layout  
engines [PPoPP'13]

Compilers for new spatial  
architectures [Chlorophyll, PLDI'14]

# The success of synthesis

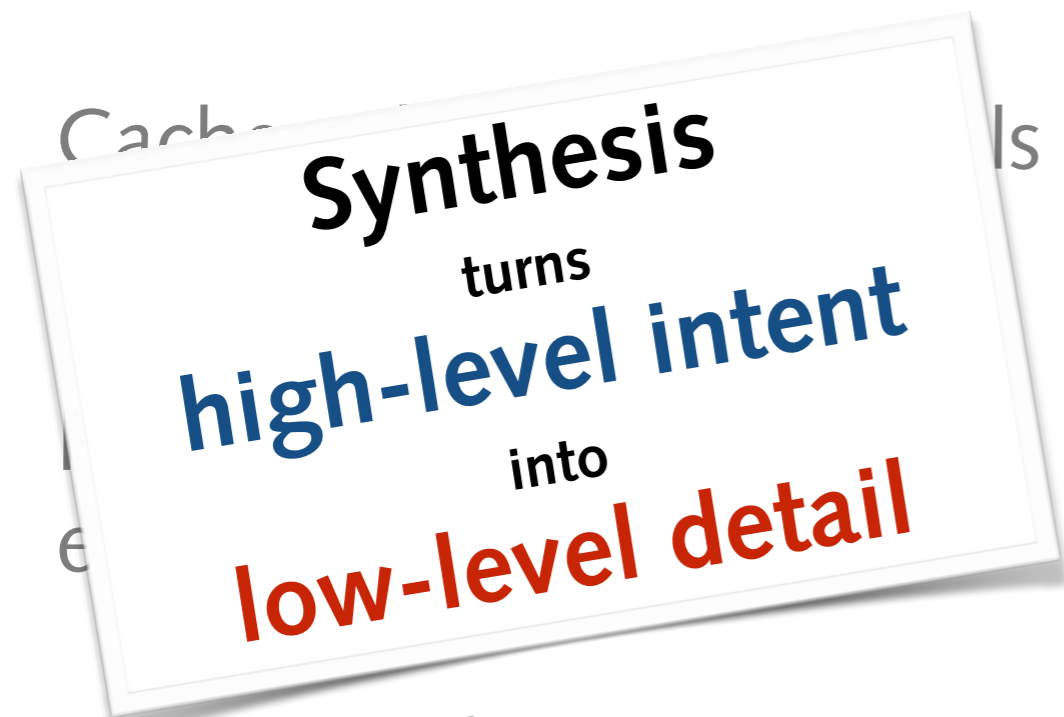
End-user programming  
by example [FlashFill, POPL'11]



Compilers for new spatial  
architectures [Chlorophyll, PLDI'14]

# The success of synthesis

End-user programming  
by example [FlashFill, POPL'11]



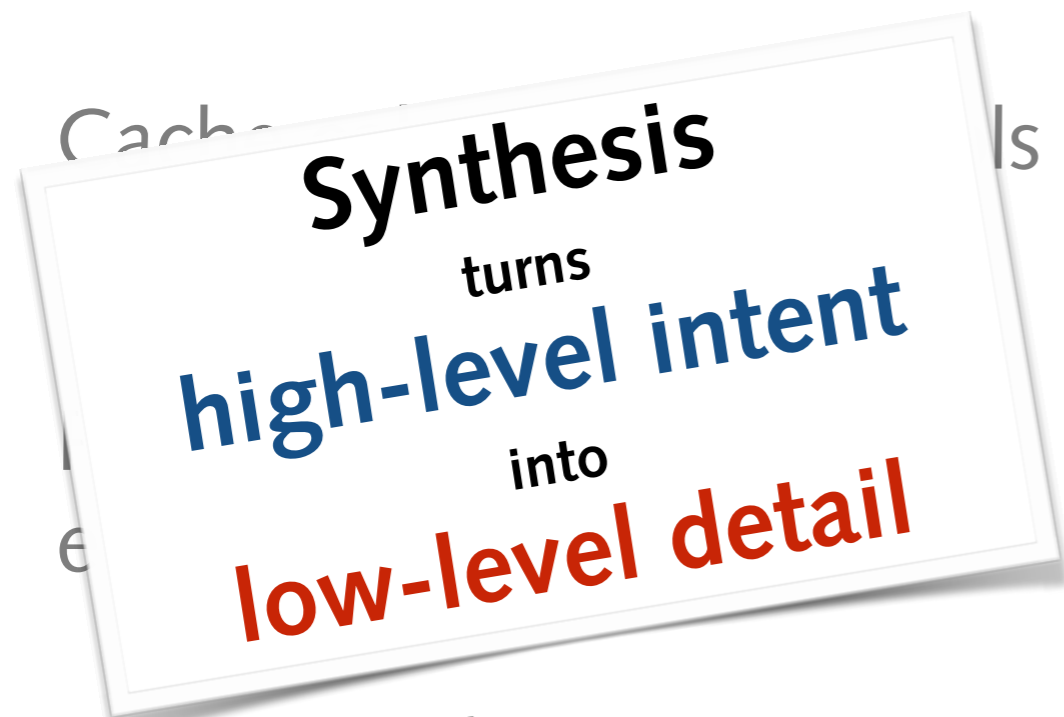
Compilers for new spatial  
architectures [Chlorophyll, PLDI'14]

## Approximate Computing

quality bounds  
into  
approximate programs

# The success of synthesis

End-user programming  
by example [FlashFill, POPL'11]



Compilers for new spatial  
architectures [Chlorophyll, PLDI'14]

## Approximate Computing

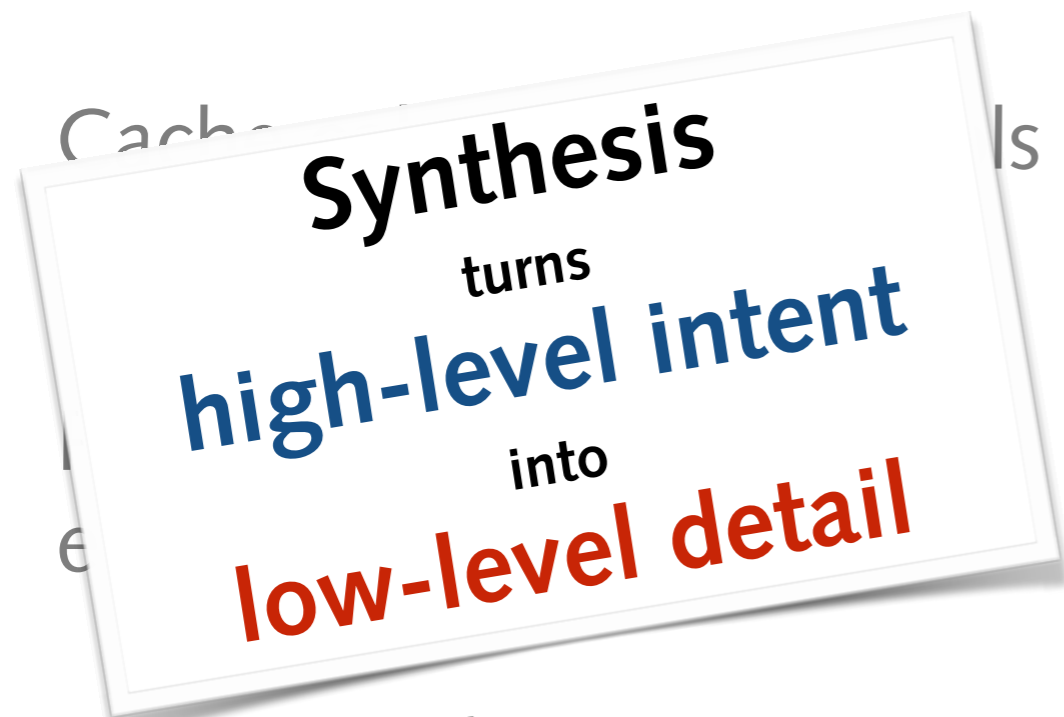
quality bounds  
into  
approximate programs

## Hardware Synthesis

programs  
into  
hardware designs

# The success of synthesis

End-user programming  
by example [FlashFill, POPL'11]



Compilers for new spatial  
architectures [Chlorophyll, PLDI'14]

## Approximate Computing

quality bounds  
into  
approximate programs

## Hardware Synthesis

programs  
into  
hardware designs

## Black Box Systems

observed behaviors  
into  
specifications

## **Approximate Computing**

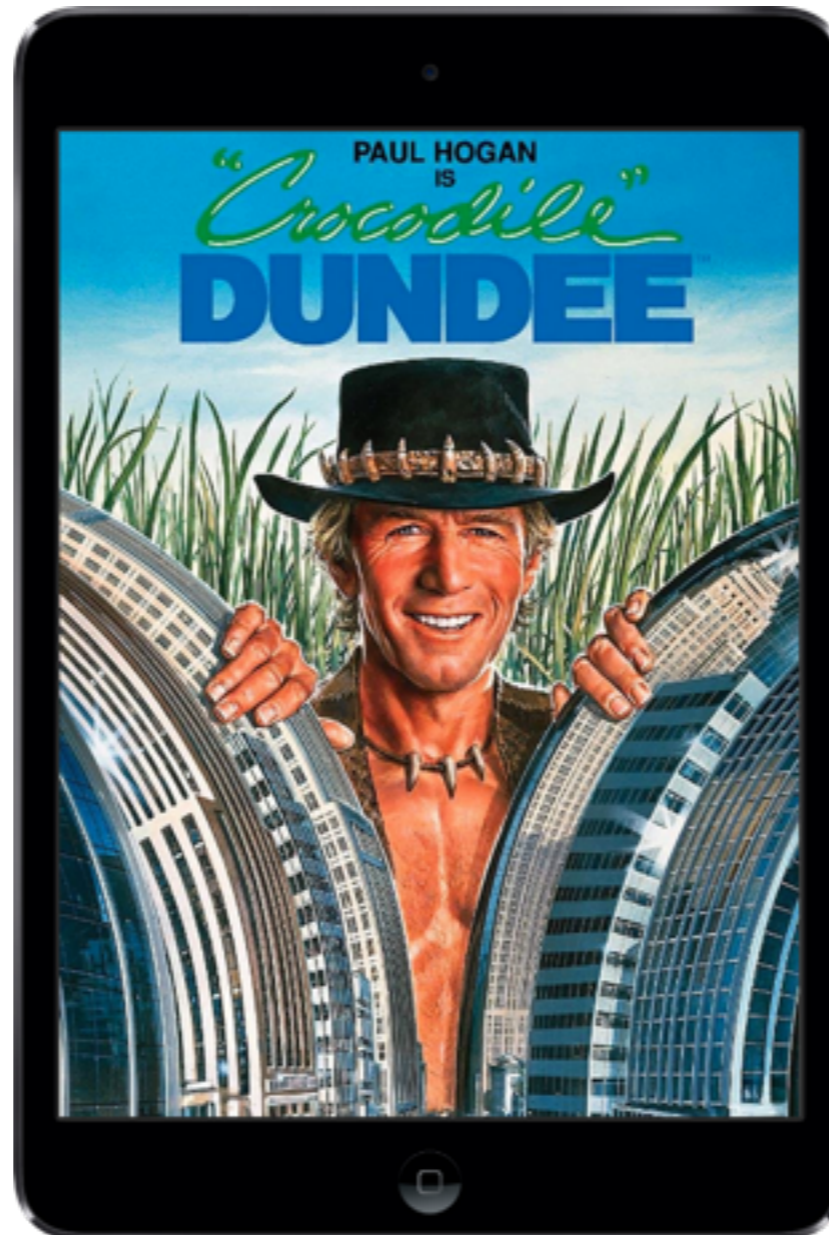
quality bounds  
into  
approximate programs

**Not all applications require  
perfect accuracy**

# Approximate Computing

quality bounds  
into  
approximate programs

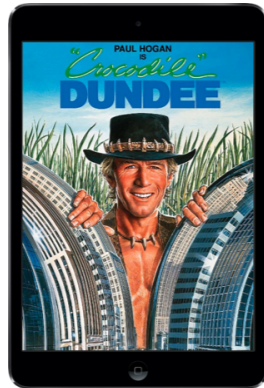
# Not all applications require perfect accuracy



# Approximate Computing

quality bounds  
into  
approximate programs

## Not all applications require perfect accuracy



Video and image quality



**Approximate  
Computing**  
quality bounds  
into  
approximate programs

**Not all applications require  
perfect accuracy**



Video and image quality



Sensors and simulation



Machine learning

# Approximate Computing

quality bounds  
into  
approximate programs

# Not all applications require perfect accuracy

Precise  
Implementation

# Approximate Computing

quality bounds  
into  
approximate programs

# Not all applications require perfect accuracy

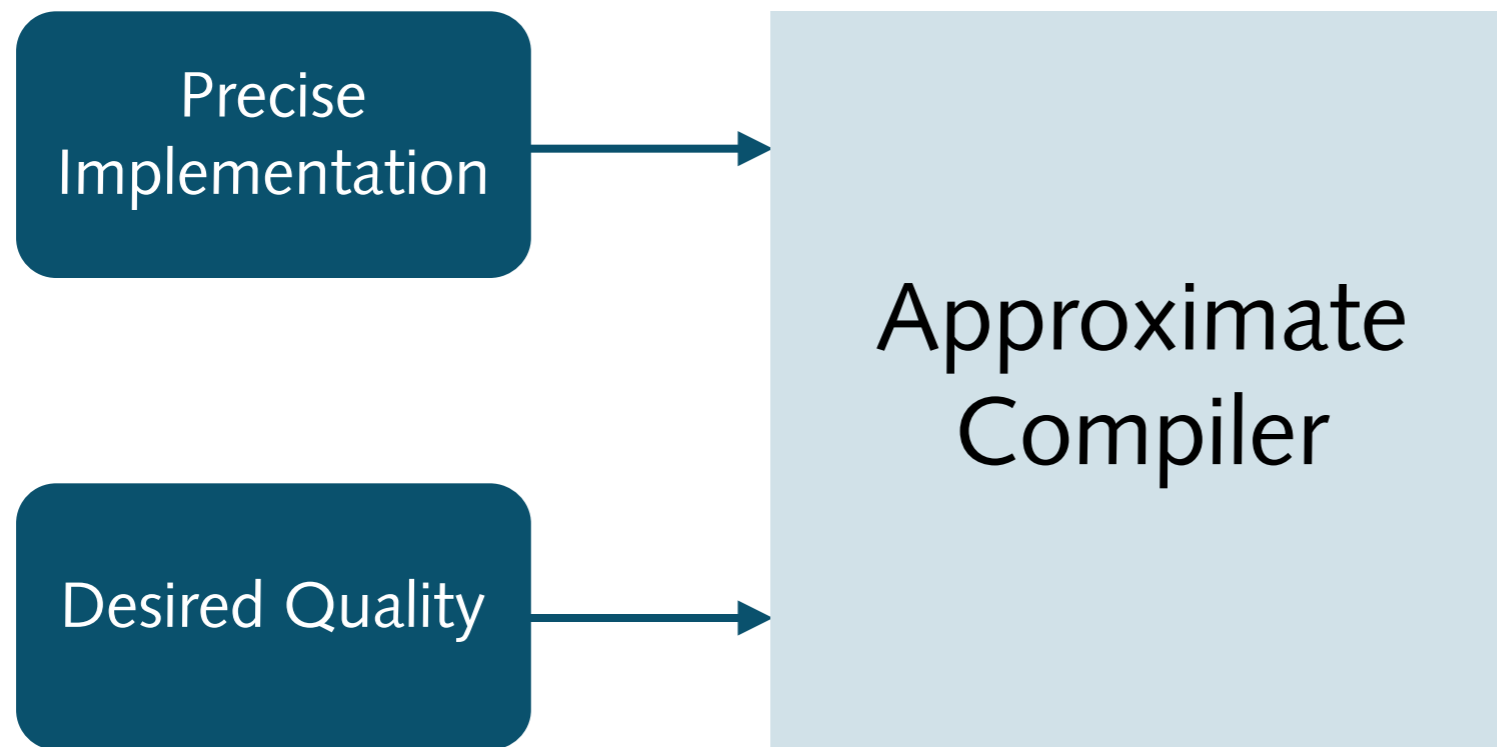
Precise  
Implementation

Desired Quality

# Approximate Computing

quality bounds  
into  
approximate programs

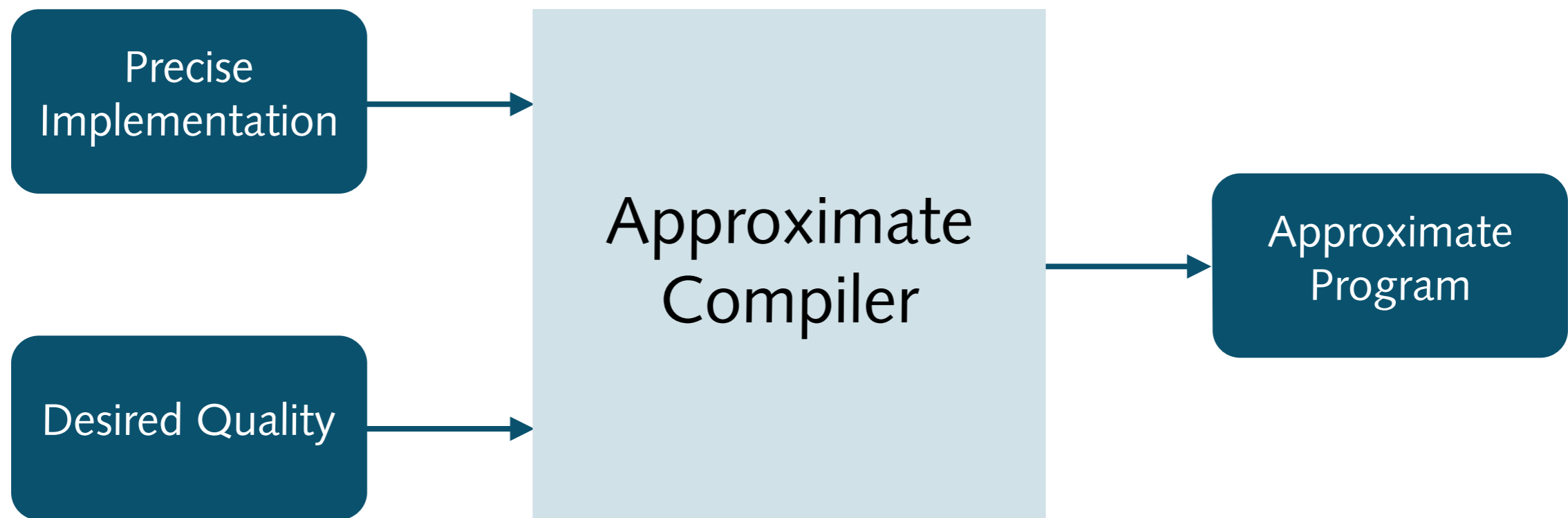
## Not all applications require perfect accuracy



# Approximate Computing

quality bounds  
into  
approximate programs

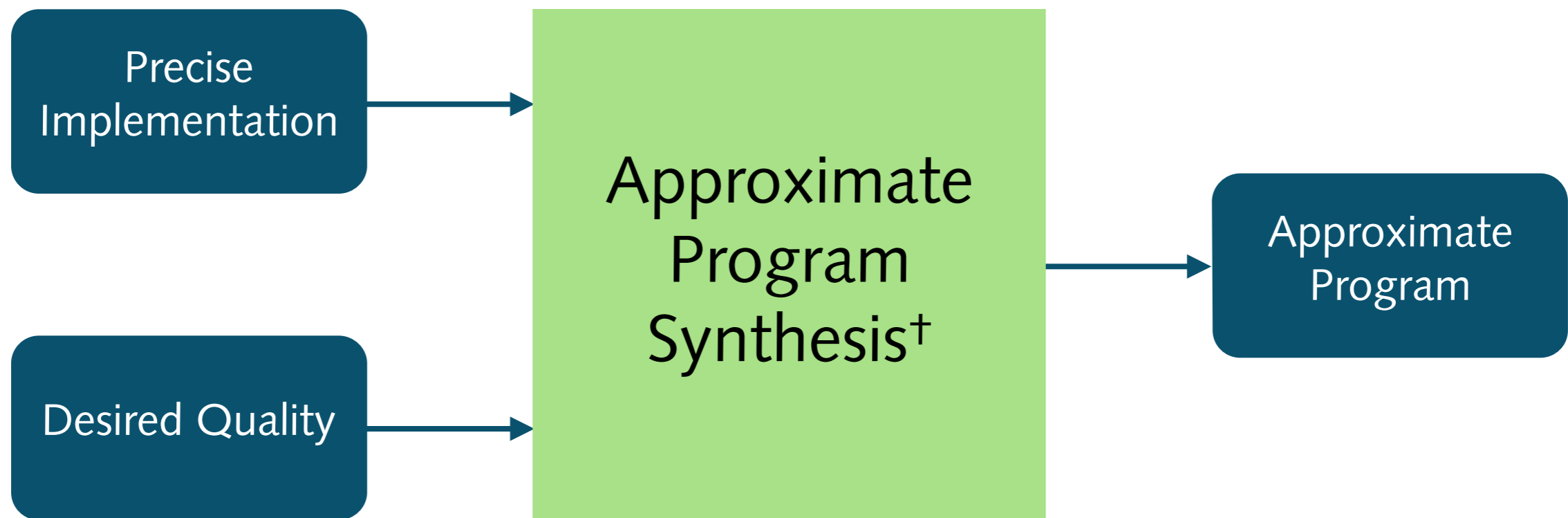
## Not all applications require perfect accuracy



# Approximate Computing

quality bounds  
into  
approximate programs

## Not all applications require perfect accuracy



<sup>†</sup> Bornholt, Torlak, Ceze, Grossman. *Approximate Program Synthesis*. At WAX'15, collocated with PLDI'15.

# Hardware Synthesis

programs  
into  
hardware designs

# Synthesizing circuits from high-level languages

# Hardware Synthesis

programs  
into  
hardware designs

## Synthesizing circuits from high-level languages

```
module example(input a, b, c, output y);  
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

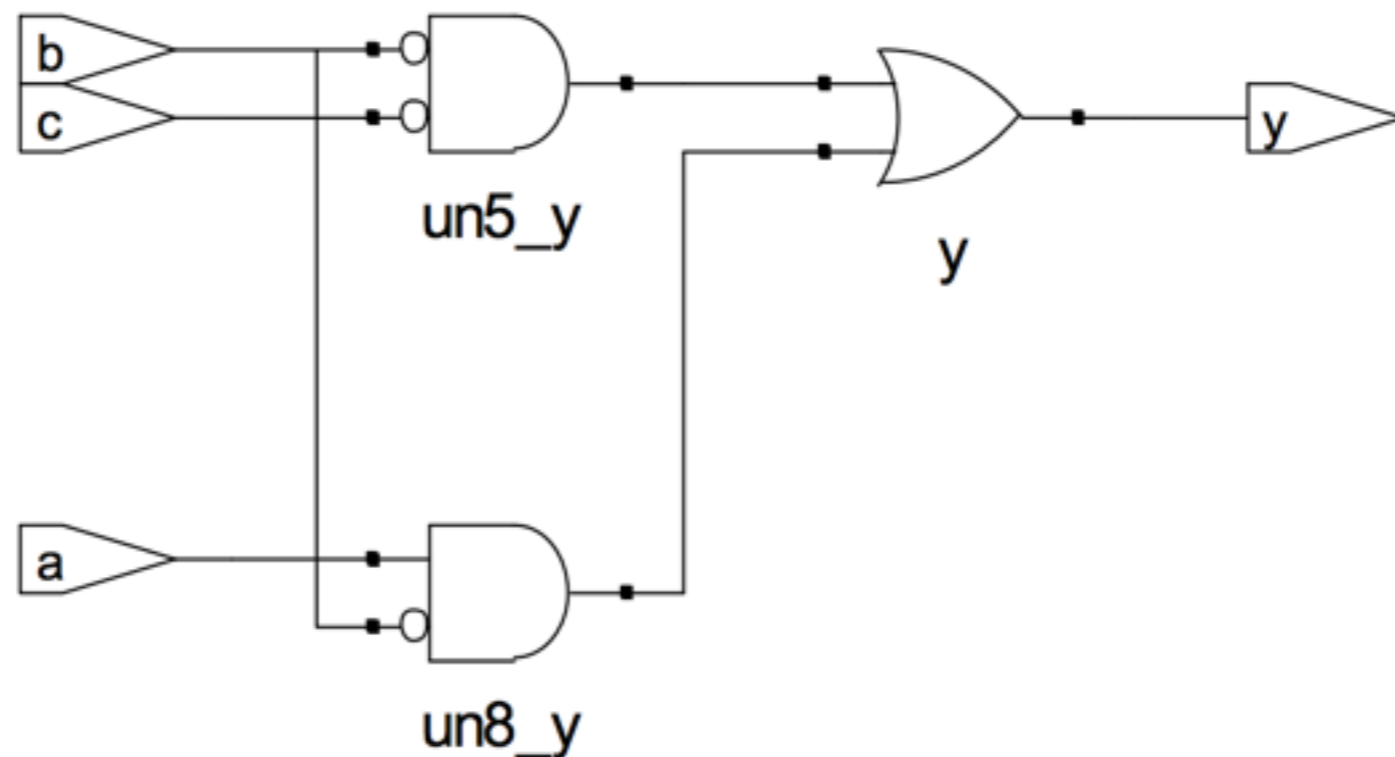


# Hardware Synthesis

programs  
into  
hardware designs

## Synthesizing circuits from high-level languages

```
module example(input a, b, c, output y);  
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

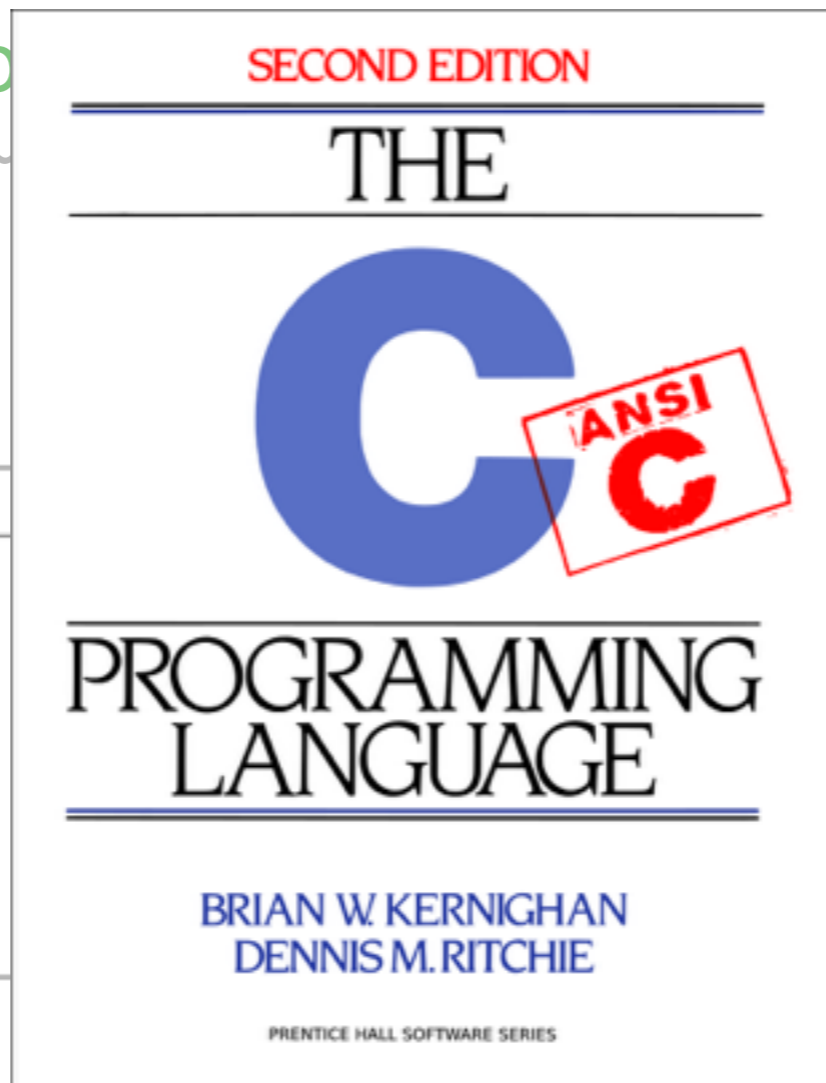


# Hardware Synthesis

programs  
into  
hardware designs

# Synthesizing circuits from high-level languages

```
module example(inp  
  assign y = ~a & ~  
endmodule
```



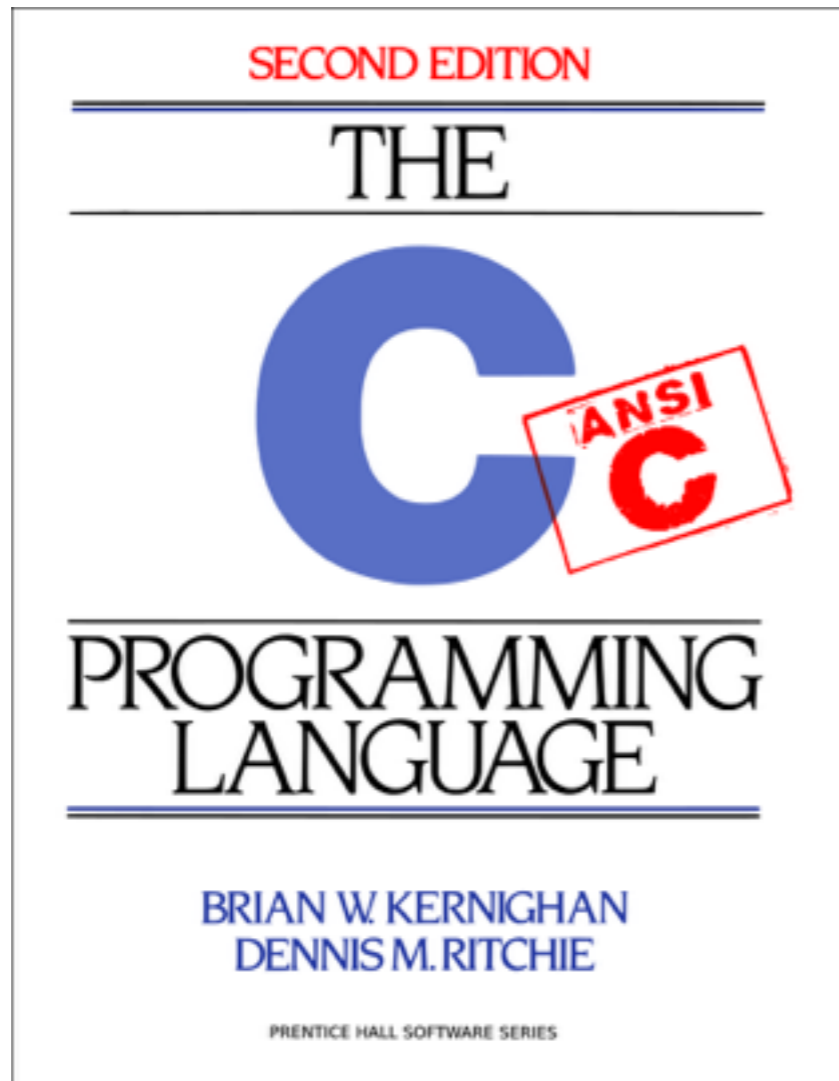
```
t y);  
~c | a & ~b & c;
```



# Hardware Synthesis

programs  
into  
hardware designs

## Synthesizing circuits from high-level languages

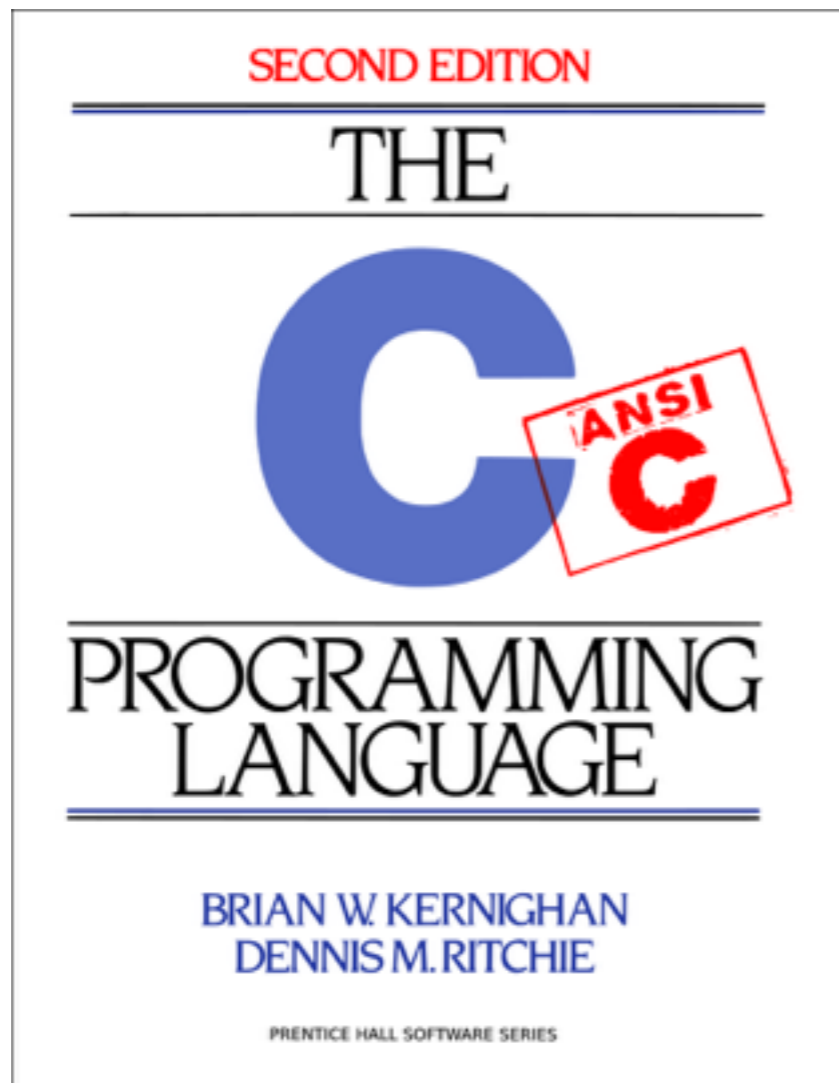


## High-Level Synthesis (HLS)

# Hardware Synthesis

programs  
into  
hardware designs

# Synthesizing circuits from high-level languages



*Place and route*

*Timing closure*

## High-Level Synthesis (HLS)

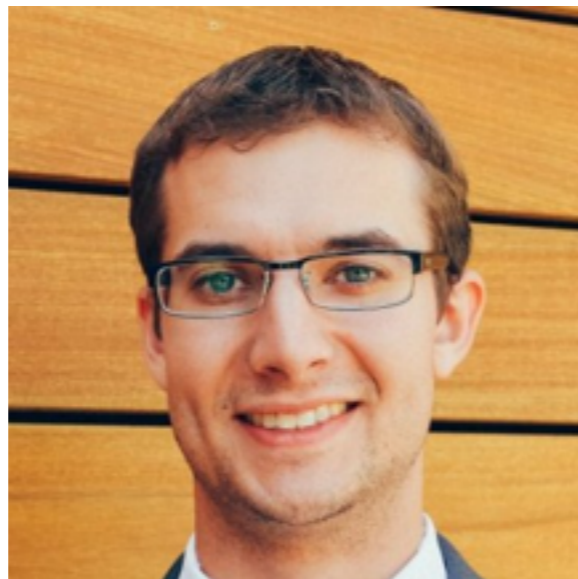
*Crosstalk and feedback*

*Quantum!*

# Hardware Synthesis

programs  
into  
hardware designs

## Synthesizing circuits from high-level languages

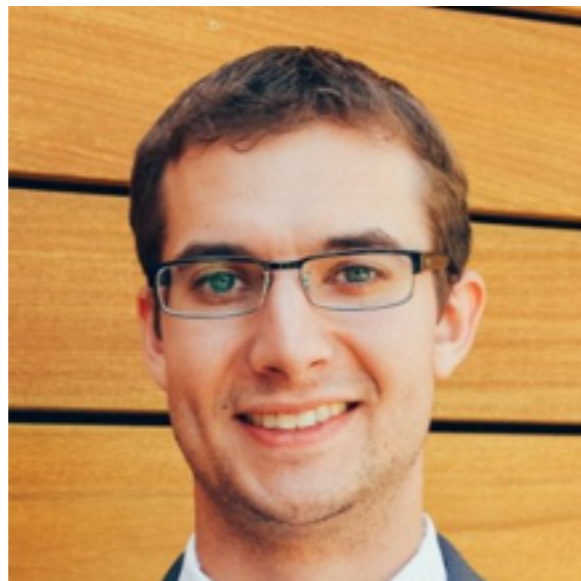


Mark Wyse  
UW grad student and  
HLS extraordinaire

# Hardware Synthesis

programs  
into  
hardware designs

## Synthesizing circuits from high-level languages



Mark Wyse

UW grad student and  
HLS extraordinaire

```
float dist(float a[3], float b[3]) {  
    float r = 0;  
    r += (a[0] - b[0]) * (a[0] - b[0]);  
    r += (a[1] - b[1]) * (a[1] - b[1]);  
    r += (a[2] - b[2]) * (a[2] - b[2]);  
    return sqrt(r);  
}
```

## **Black Box Systems**

observed behaviors  
into  
specifications

# **Defining system behavior by observation**

# Black Box Systems

observed behaviors  
into  
specifications

## Defining system behavior by observation





# Black Box Systems

observed behaviors  
into  
specifications

# Defining system behavior by observation

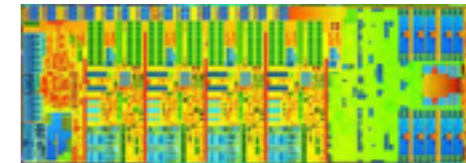


4008

# Black Box Systems

observed behaviors  
into  
specifications

## Defining system behavior by observation



# Black Box Systems

observed behaviors  
into  
specifications

## Defining system behavior by observation

Flash Fill your data

Start typing and let Excel finish your work for you

Email	First Name
Nancy.FreeHafer@fourthcoffee.com	Nancy
Andrew.Cencini@northwindtraders.com	Andrew
Jan.Kotas@itwareinc.com	Jan
Mariya.Sergienko@graphicdesigninstitute.com	Mariya
Steven.Thorpe@northwindtraders.com	Steven
Michael.Neipper@northwindtraders.com	Michael
Robert.Zare@northwindtraders.com	Robert
Laura.Giussani@adventure-works.com	Laura
Anne.HL@northwindtraders.com	Anne
Alexander.David@contoso.com	Alexander
Kim.Shane@northwindtraders.com	Kim
Manish.Chopra@northwindtraders.com	Manish
Gerwald.Oberleitner@northwindtraders.com	Gerwald
Amr.Zaki@northwindtraders.com	Amr
Yvonne.McKay@northwindtraders.com	Yvonne
Amanda.Pinto@northwindtraders.com	Amanda

Start | 1. Fill | 2. Analyze | 3. Chart | Learn More

ENTER

Programming  
by example

# Black Box Systems

observed behaviors  
into  
specifications

# Defining system behavior by observation

Flash Fill your data

Start typing and let Excel finish your work for you

Email	First Name
Nancy.FreeHafer@fourthcoffee.com	Nancy
Andrew.Cencini@northwindtraders.com	Andrew
Jan.Kotas@itwareinc.com	Jan
Mariya.Sergienko@graphicdesigninstitute.com	Mariya
Steven.Thorpe@northwindtraders.com	Steven
Michael.Neipper@northwindtraders.com	Michael
Robert.Zare@northwindtraders.com	Robert
Laura.Giussani@adventure-works.com	Laura
Anne.HL@northwindtraders.com	Anne
Alexander.David@contoso.com	Alexander
Kim.Shane@northwindtraders.com	Kim
Manish.Chopra@northwindtraders.com	Manish
Gerwald.Oberleitner@northwindtraders.com	Gerwald
Amr.Zaki@northwindtraders.com	Amr
Yvonne.McKay@northwindtraders.com	Yvonne
Amanda.Pinto@northwindtraders.com	Amanda

Start 1. Fill 2. Analyze 3. Chart Learn More

ENTER

Programming  
by example

## Automated Synthesis of Symbolic Instruction Encodings from I/O Samples

Patrice Godefroid  
Microsoft Research  
gg@microsoft.com

Ankur Taly\*  
Stanford University  
taly@stanford.edu

### Abstract

Symbolic execution is a key component of precise binary program analysis tools. We discuss how to automatically bootstrap the construction of a symbolic execution engine for a processor instruction set such as x86, x64 or ARM. We show how to automatically synthesize symbolic representations of individual processor instructions from input/output examples and express them as bit-vector constraints. We present and compare various synthesis algorithms and instruction sampling strategies. We introduce a new synthesis algorithm based on smart sampling which we show is one to two orders of magnitude faster than previous synthesis algorithms in our context. With this new algorithm, we can automatically synthesize bit-vector circuits for over 500 x86 instructions (8/16/32-bit, outputs, EFLAGS) using only 6 synthesis templates and in less than two hours using the Z3 SMT solver on a regular machine. During this work, we also discovered several inconsistencies across x86 processors, errors in the x86 tool spec, and several bugs in previous manually-written x86 instruction handlers.

**Categories and Subject Descriptors:** D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.4 [Software Engineering]: SoftwareProgram Verification; D.2.5 [Software Engineering]: Testing and Debugging; F.3.1 [Logic and Meaning of Programs]: Specifying and Verifying and Reasoning about Programs  
**General Terms:** Languages, Verification

**Keywords:** Program Synthesis, Symbolic Execution, x86

### 1. Introduction

Symbolic execution is a key component of precise binary program analysis tools, for test generation [6, 17], program verification [2, 20], malware analysis [1, 23], and other applications. Symbolic execution engines are traditionally written by hand [3, 2, 4, 17, 20, 23]: the effort of executing each individual instruction is described by a symbolic constraint, called symbolic instruction encoding, written manually and derived by reading the processor instruction manual. Unfortunately, the semantics of the instruction set of general-purpose processors such as x86, x64 or ARM

is very complex. For instance, x86 includes hundreds of instructions whose semantics is described in more than 2,000 pages divided in 3 volumes. This complexity makes the manual development of symbolic-execution engines tedious (many instructions, error-prone (many corner cases), partial (not all instructions are usually covered) and imprecise (approximations are often used). Moreover, the official reference manual is often under-specified and may itself contain errors.

In this work, we explore a radically different approach to developing symbolic-execution engines: what if most symbolic instruction encodings could be synthesized automatically?

To do this, we study how to adapt and extend recent advances in automatic program synthesis. Given a functional specification and a set of building blocks, called components, possibly combined together as described in a solution template or program sketch (i.e., a program with holes), automatic program synthesis consists of searching the space of all possible template completions for a fully-defined program (i.e., with no holes left) that satisfies the specification. In our context, we do not have access to a full functional specification of individual processor instructions — such a specification is precisely what we want to infer. But we have access to a cheap and fast specification oracle: we can execute instructions on a processor and observe their input/output behaviors.

Program synthesis from I/O samples has been recently investigated in [13]. There, an I/O oracle-guided synthesis algorithm is presented for loop-free programs. This algorithm consists of computing a set of I/O samples, then synthesizing a candidate program that satisfies those samples (to check whether such a program exists), then computing a distinguishing input that distinguishes this candidate program  $P'$  from some other non-equivalent candidate program  $P''$  (to check whether  $P'$  is unique), and if such an input exists, then query the I/O oracle with this distinguishing input to eliminate either  $P'$  or  $P''$  as a possible solution. This counter-example-guided iterative synthesis process is repeated until one unique solution remains, or no solution exists if the synthesis template is too constrained. The algorithm assumes the existence of a verification oracle which can determine whether a solution is “correct”.

In our context, we do not have access to such a verification oracle. For instructions with small I/O signatures, such as 8-bit instructions, exhaustive testing can provide a verification oracle, but exhaustive testing does not scale to 16-bit or 32-bit instructions. Another practical barrier is that the counter-example-guided iterative synthesis algorithm of [13] can be very expensive when many iterations are needed, as we will show with results of experiments in Section 5.

To improve on this, we propose a new synthesis algorithm based on smart sampling which we show is one to two orders of magnitude faster than previous synthesis algorithms in our context. Given a specific template, the main idea is to generate upfront a set of distinguishing inputs which uniquely determine each possible so-

\*The work of this author was done mostly while visiting Microsoft.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. For copy alternatives, to reprint in servers or to redistribute to lists, requires prior specific permission outside a fee.  
PLDI'12, June 11–16, 2012, Beijing, China.  
Copyright © 2012 ACM 978-1-4503-1912-6... 30:000

## **Approximate Computing**

quality bounds  
into  
approximate programs

## **Hardware Synthesis**

programs  
into  
hardware designs

## **Black Box Systems**

observed behaviors  
into  
specifications

# **Machine Learning and Synthesis**



# Learning programs from examples

## Black Box Systems

observed behaviors  
into  
specifications

$$\exists P. \bigwedge_{x_i \in X} \varphi(x_i, P(x_i))$$

# Learning programs from examples

## Black Box Systems

observed behaviors  
into  
specifications

$$\exists P. \bigwedge_{x_i \in X} \varphi(x_i, P(x_i))$$

Flash Fill your data  
Start typing and let Excel finish your work for you

Email	First Name
Nancy.FreeHafer@fourthcoffee.com	Nancy
Andrew.Cencini@northwindtraders.com	Andrew
Jan.Kotas@itwareinc.com	Jan
Mariya.Sergienko@graphicdesigninstitute.com	Mariya
Steven.Thorpe@northwindtraders.com	Steven
Michael.Neipper@northwindtraders.com	Michael
Robert.Zare@northwindtraders.com	Robert
Laura.Giussani@adventure-works.com	Laura
Anne.HL@northwindtraders.com	Anne
Alexander.David@contoso.com	Alexander
Kim.Shane@northwindtraders.com	Kim
Manish.Chopra@northwindtraders.com	Manish
Gerwald.Oberleitner@northwindtraders.com	Gerwald
Amr.Zaki@northwindtraders.com	Amr
Yvonne.McKay@northwindtraders.com	Yvonne
Amanda.Pinto@northwindtraders.com	Amanda

ENTER

FlashFill



# Learning programs from examples

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24

## Flash Fill your data

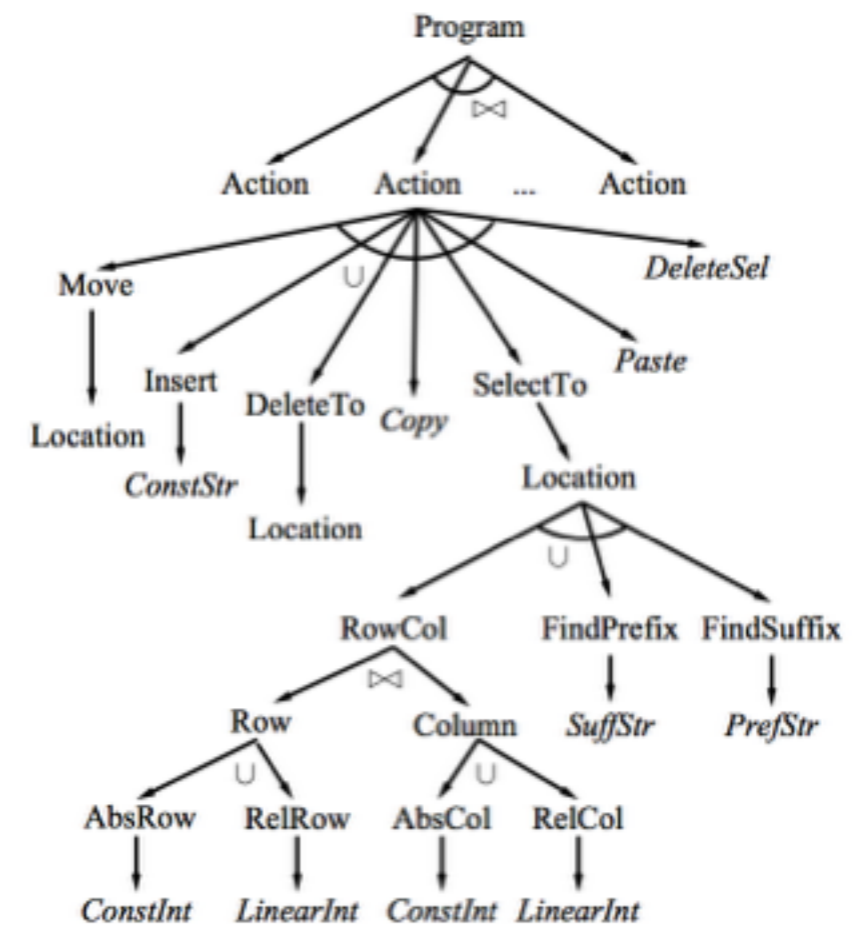
Start typing and let Excel finish your work for you

Email	First Name
Nancy.FreeHafer@fourthcoffee.com	Nancy
Andrew.Cencini@northwindtraders.com	Andrew
Jan.Kotas@litwareinc.com	Jan
Mariya.Sergienko@graphicdesigninstitute.com	Mariya
Steven.Thorpe@northwindtraders.com	Steven
Michael.Neipper@northwindtraders.com	Michael
Robert.Zare@northwindtraders.com	Robert
Laura.Giussani@adventure-works.com	Laura
Anne.HL@northwindtraders.com	Anne
Alexander.David@contoso.com	Alexander
Kim.Shane@northwindtraders.com	Kim
Manish.Chopra@northwindtraders.com	Manish
Gerwald.Oberleitner@northwindtraders.com	Gerwald
Amr.Zaki@northwindtraders.com	Amr
Yvonne.McKay@northwindtraders.com	Yvonne
Amanda.Pinto@northwindtraders.com	Amanda

Start 1. Fill 2. Analyze 3. Chart Learn More

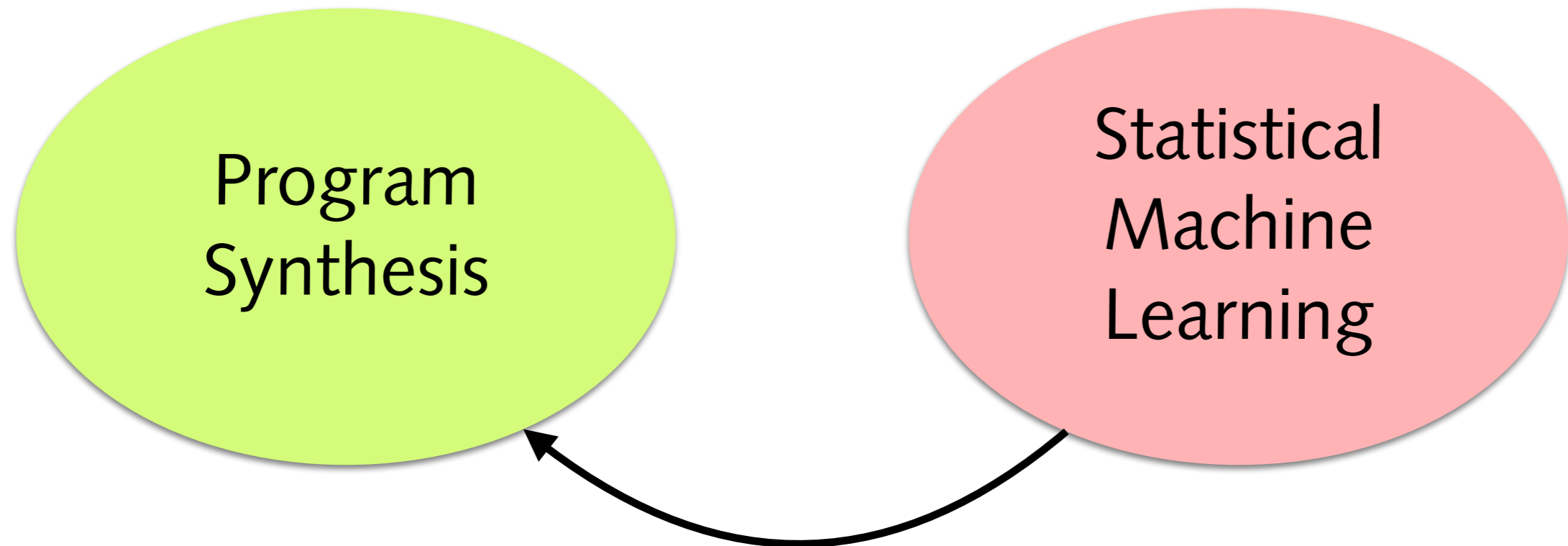
ENTER

FlashFill



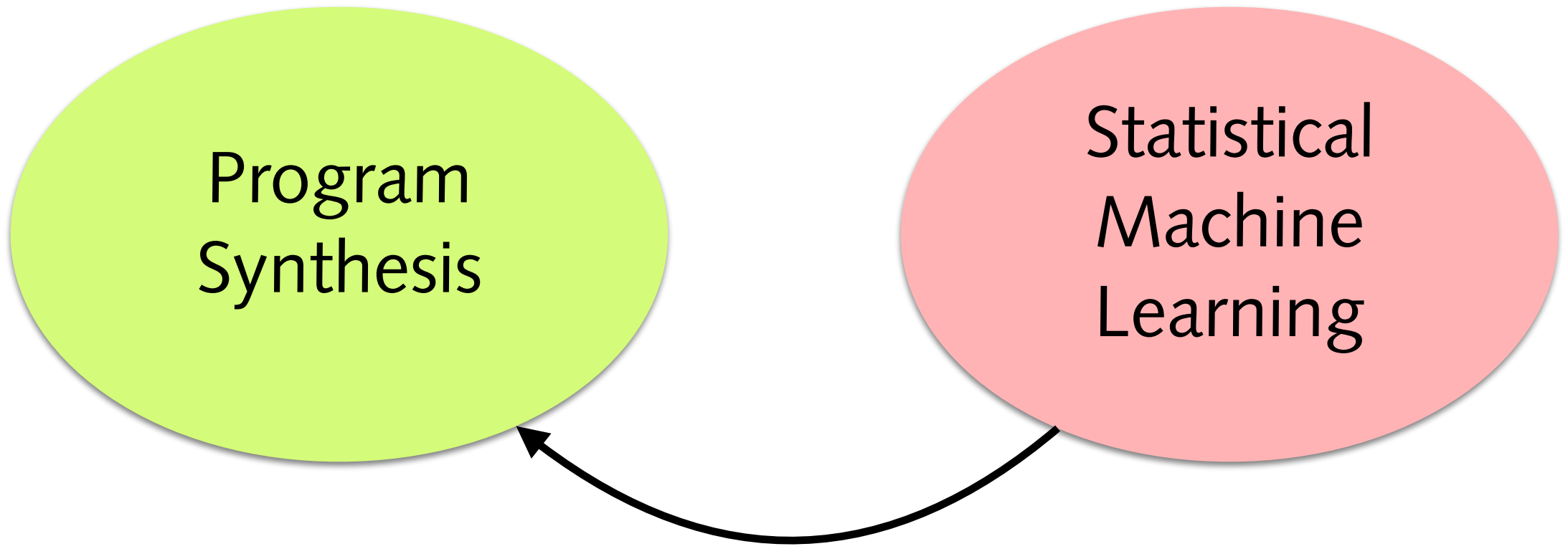
Version Space Algebra

# Machine learning and synthesis



# Machine learning and synthesis

Millions of examples/parameters



Program  
Synthesis

Statistical  
Machine  
Learning

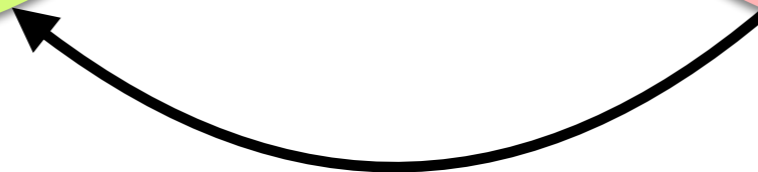
# Machine learning and synthesis

100 instructions

Program  
Synthesis

Millions of examples/parameters

Statistical  
Machine  
Learning



# Machine learning and synthesis

Approximate Computing

Hardware Synthesis

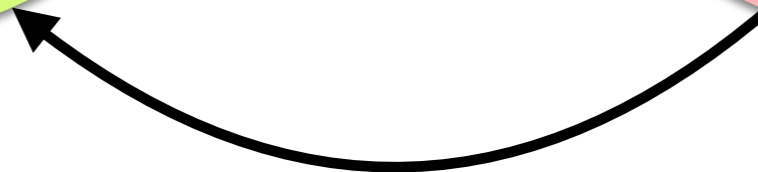
Black Box Systems

100 instructions

Millions of examples/parameters

Program  
Synthesis

Statistical  
Machine  
Learning



# Machine learning and synthesis

Approximate Computing

Hardware Synthesis

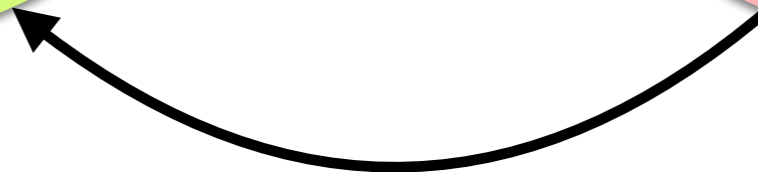
Black Box Systems

100 instructions

Program  
Synthesis

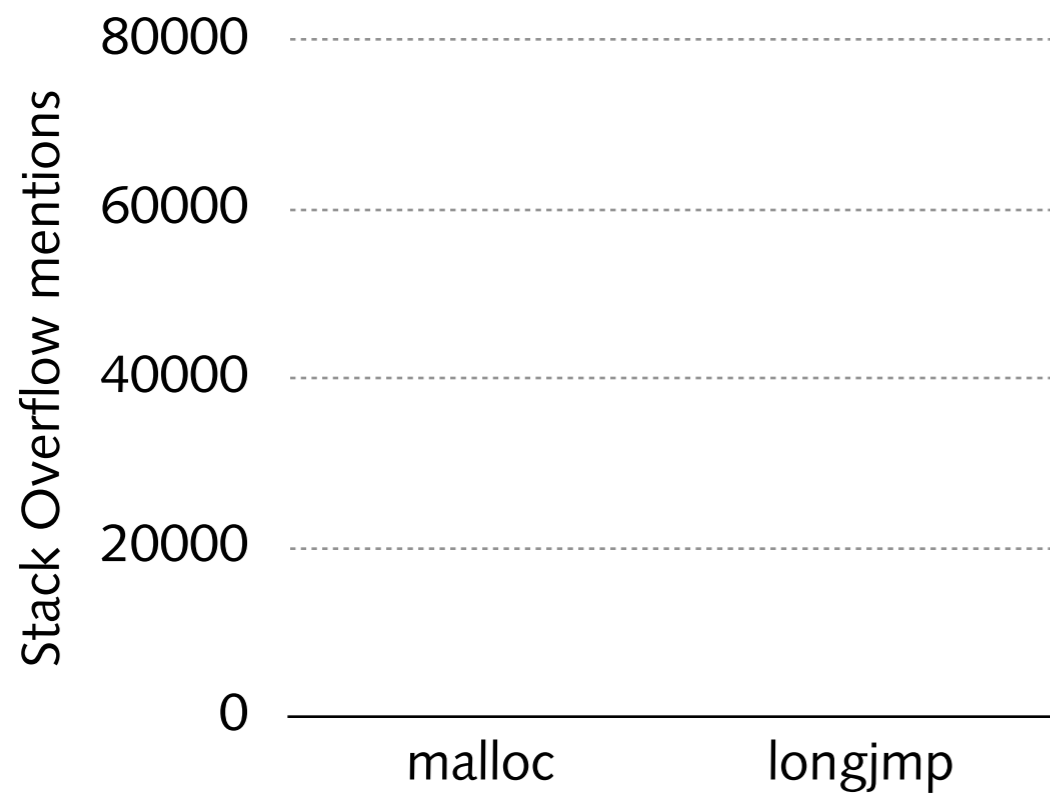
Millions of examples/parameters

Statistical  
Machine  
Learning



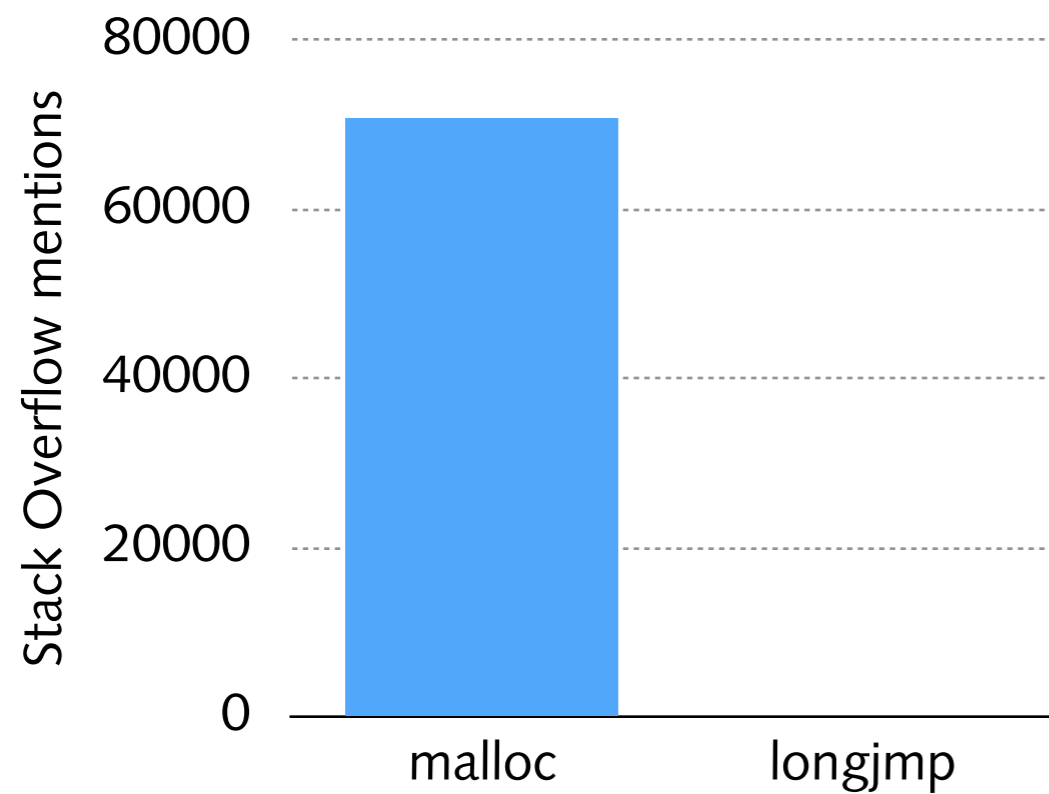
**Programs are not  
uniformly distributed.**

# Programs are not uniformly distributed.

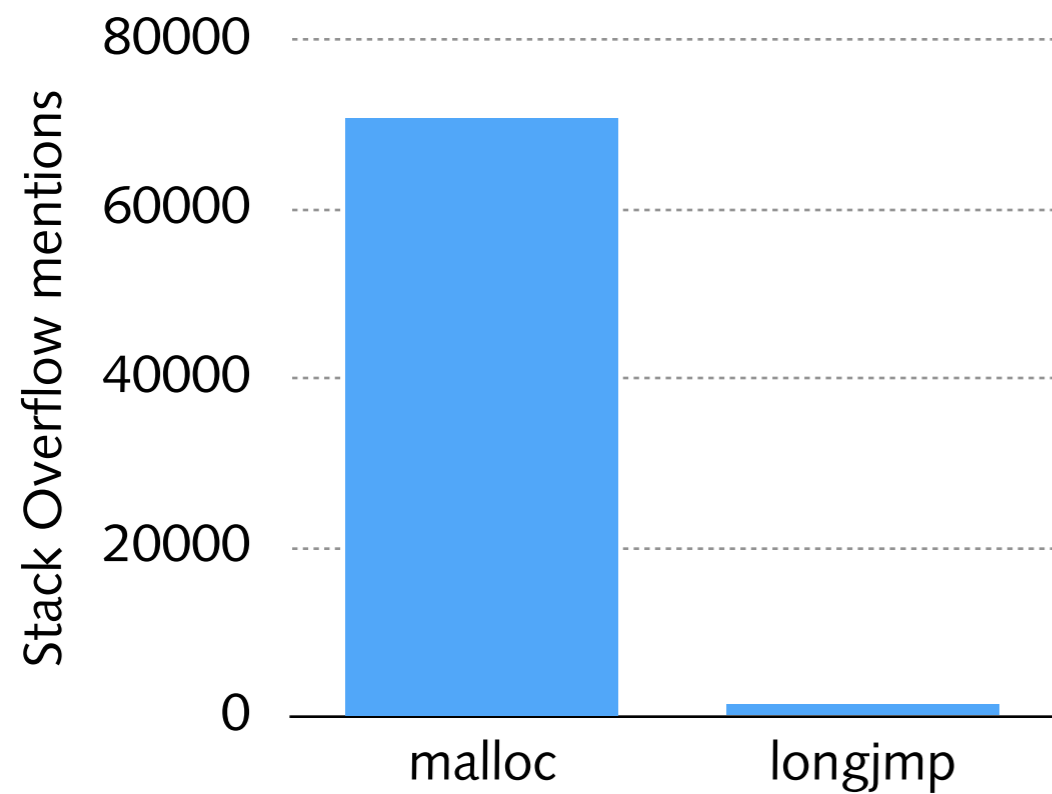




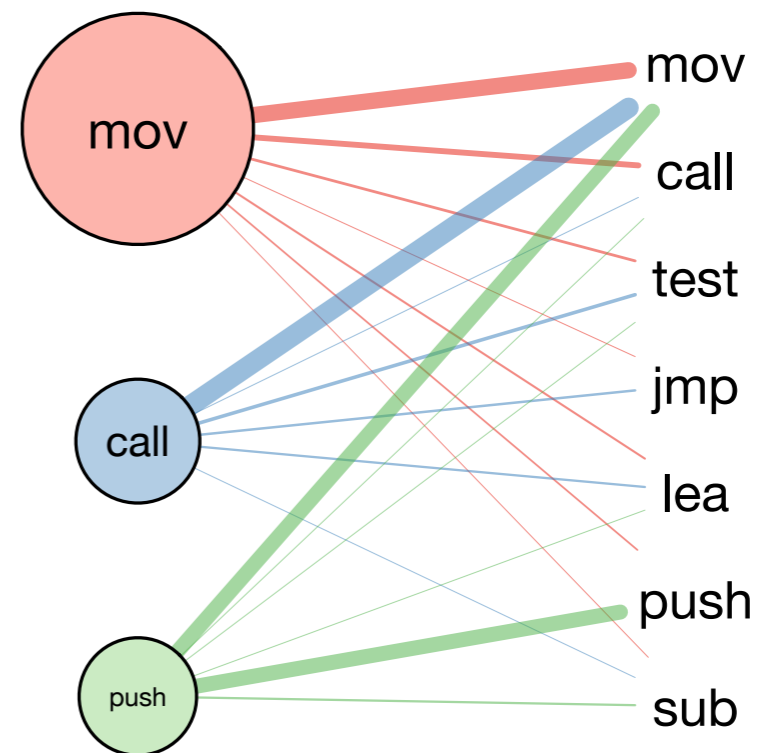
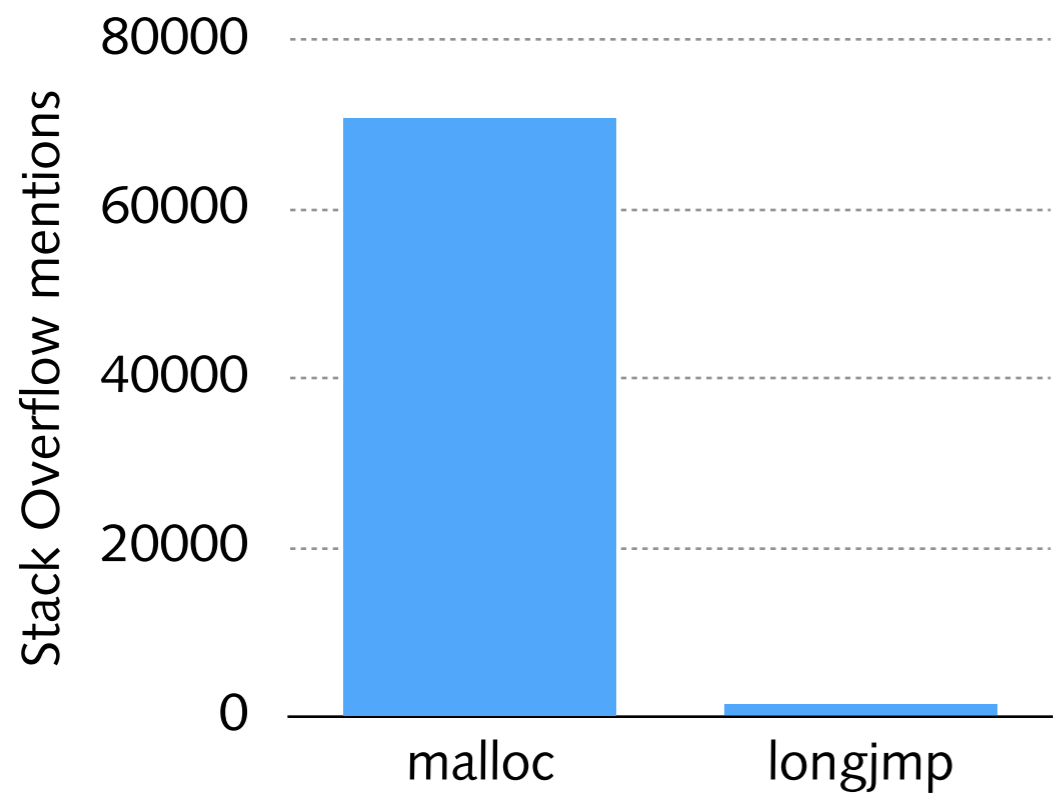
# Programs are not uniformly distributed.



# Programs are not uniformly distributed.



# Programs are not uniformly distributed.



# Component-Based Synthesis

# Synthesis of Loop-Free Programs

# Synthesis of Loop-Free Programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$

# Synthesis of Loop-Free Programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$

and

xor

add

>> 1

# Synthesis of Loop-Free Programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$

and

xor

add

>> 1

x

y



# Synthesis of Loop-Free Programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$

and

xor

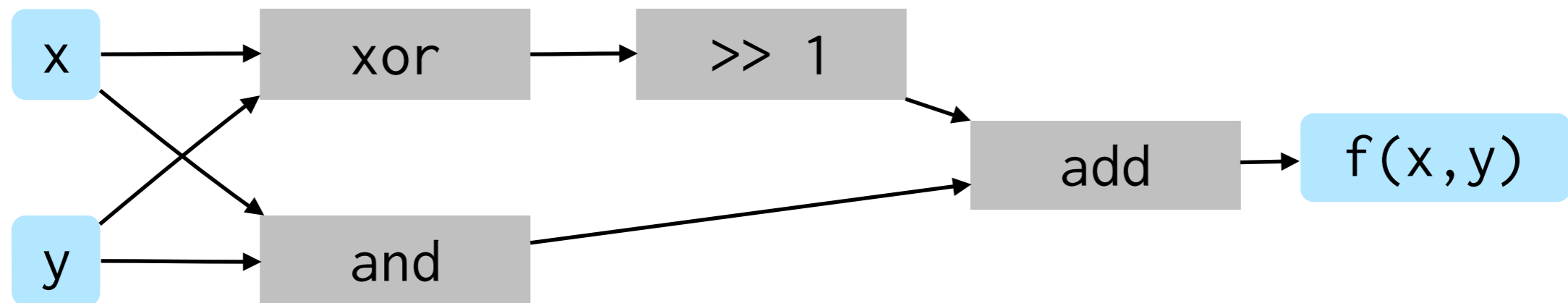
add

>> 1



# Synthesis of Loop-Free Programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$



# Synthesis of Loop-Free Programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$

and

xor

add

>> 1

# Synthesis of Loop-Free Programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$

and

xor

add

>> 1

sub

mul

div

udiv

rem

urem

or

nand

not

neg

<< 1

>>> 1

<< 2

eq

le

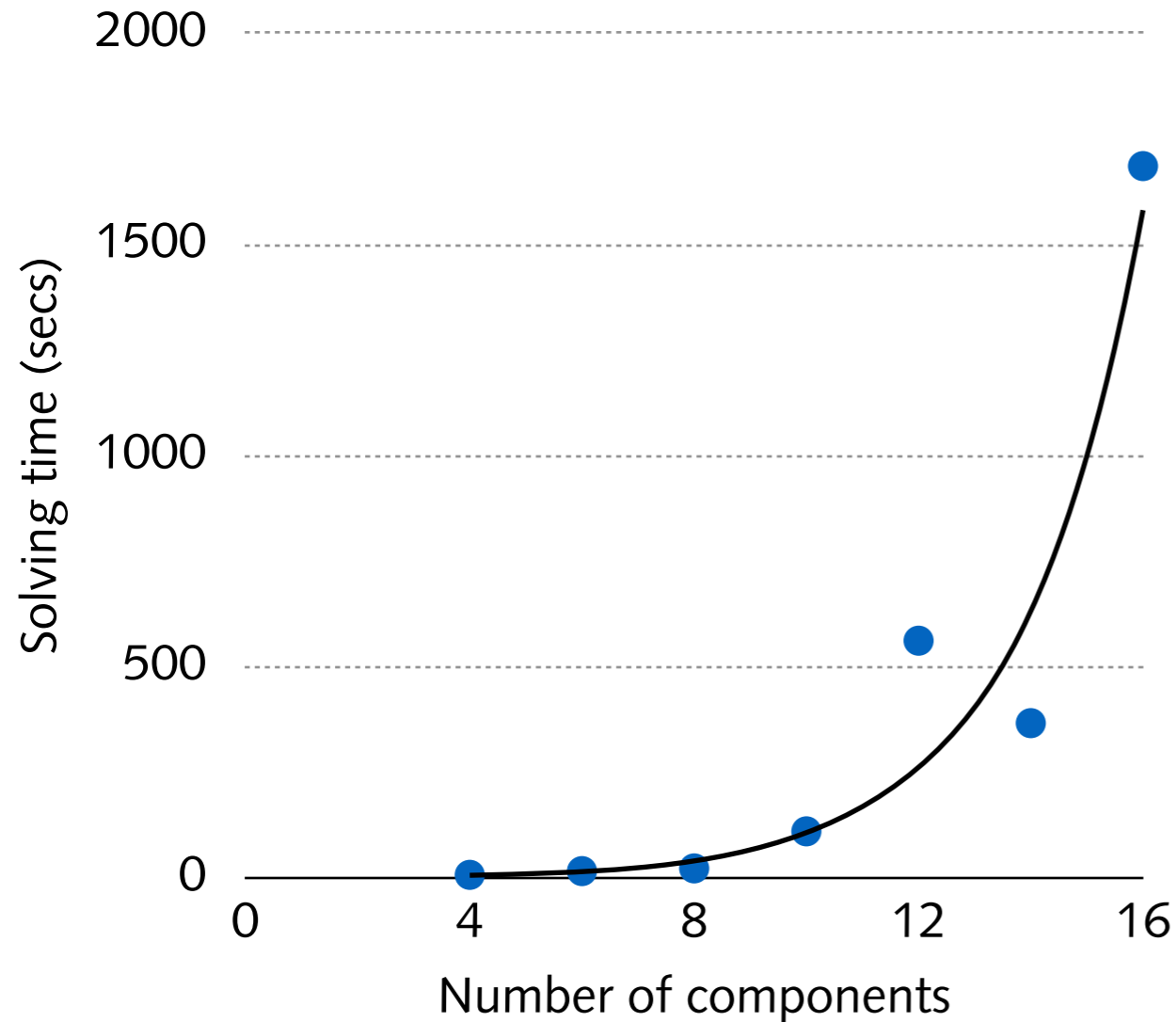
ge

lt

gt

# Synthesis of Loop-Free Programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$



and	xor
add	>> 1
sub	mul
div	udiv
rem	urem
or	nand
not	neg
<< 1	>>> 1
<< 2	eq
le	ge
lt	gt

# Higher-level components

$$f(L) = \min\{l_i \mid l_i \in L\}$$

and

xor

add

>> 1

sub

mul

div

udiv

rem

urem

or

nand

not

neg

<< 1

>>> 1

<< 2

eq

le

ge

lt

gt

# Higher-level components

$$f(L) = \min\{l_i \mid l_i \in L\}$$

# Higher-level components

$$f(L) = \min\{l_i \mid l_i \in L\}$$

quicksort

$\emptyset$

[ ]



# Higher-level components

$$f(L) = \min\{l_i \mid l_i \in L\}$$

$$f(L) = \text{quicksort}(L)[0]$$

quicksort

0

[ ]

# Higher-level components

$$f(L) = \min\{l_i \mid l_i \in L\}$$

quicksort

$\emptyset$

$$f(L) = \text{quicksort}(L)[0]$$

[ ]

**Mine existing code bases for  
common idioms**

# Higher-level components

$$f(L) = \min\{l_i \mid l_i \in L\}$$

$$f(L) = \text{quicksort}(L)[0]$$

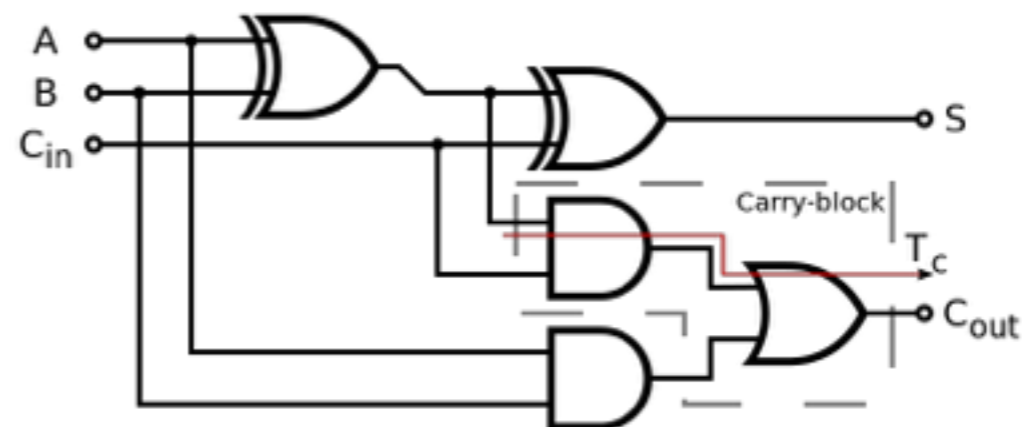
quicksort

$\emptyset$

[ ]

**Mine existing code bases for  
common idioms**

**Hardware  
Synthesis**  
programs  
into  
hardware designs



# Producing candidate programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$

and

xor

add

>> 1



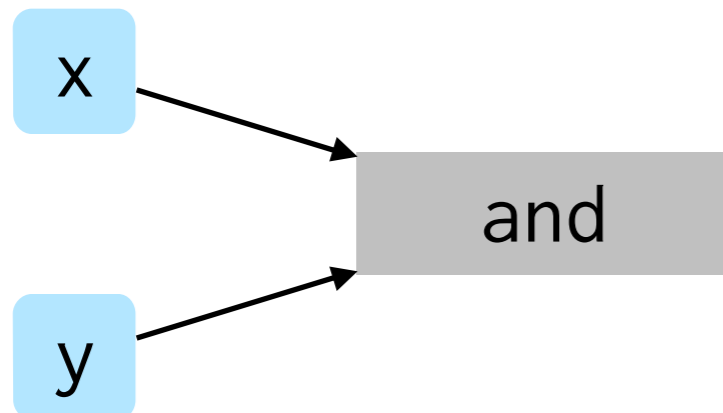
# Producing candidate programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$

add

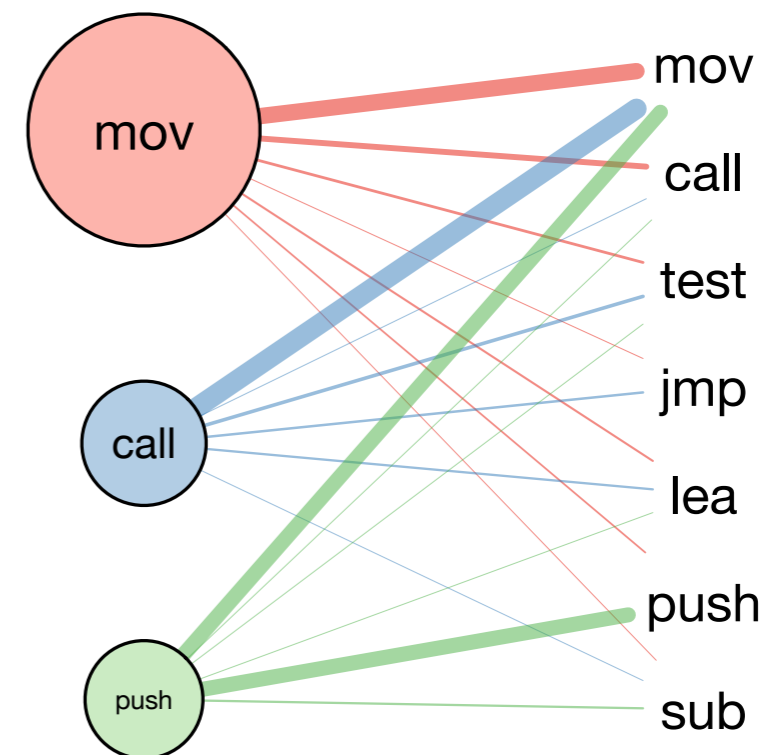
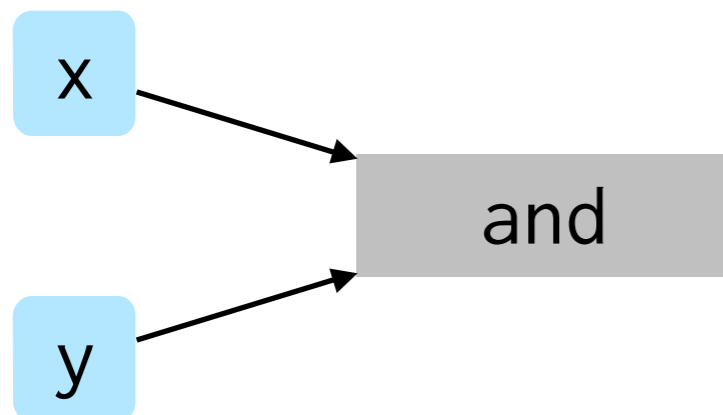
xor

>> 1



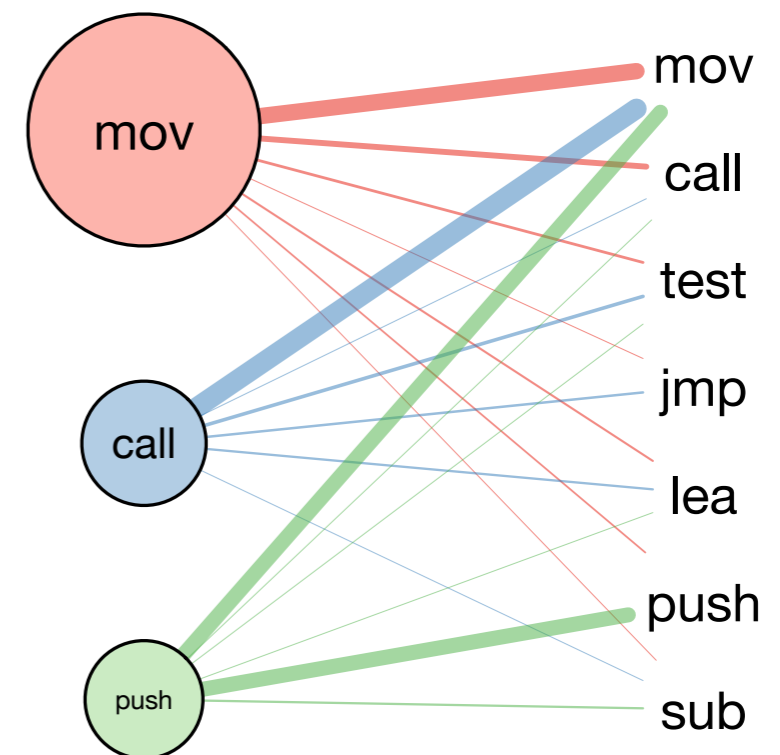
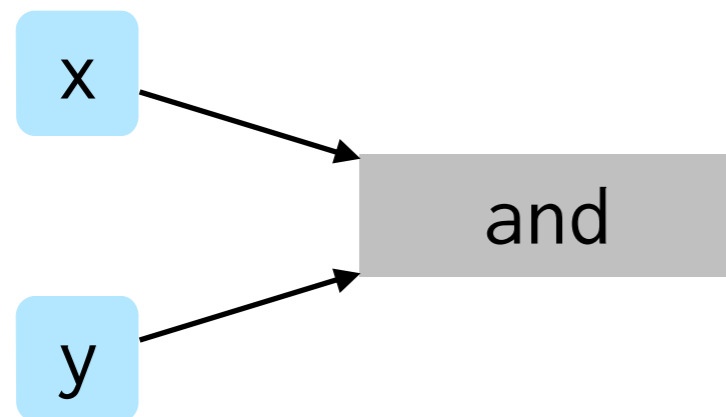
# Producing candidate programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$



# Producing candidate programs

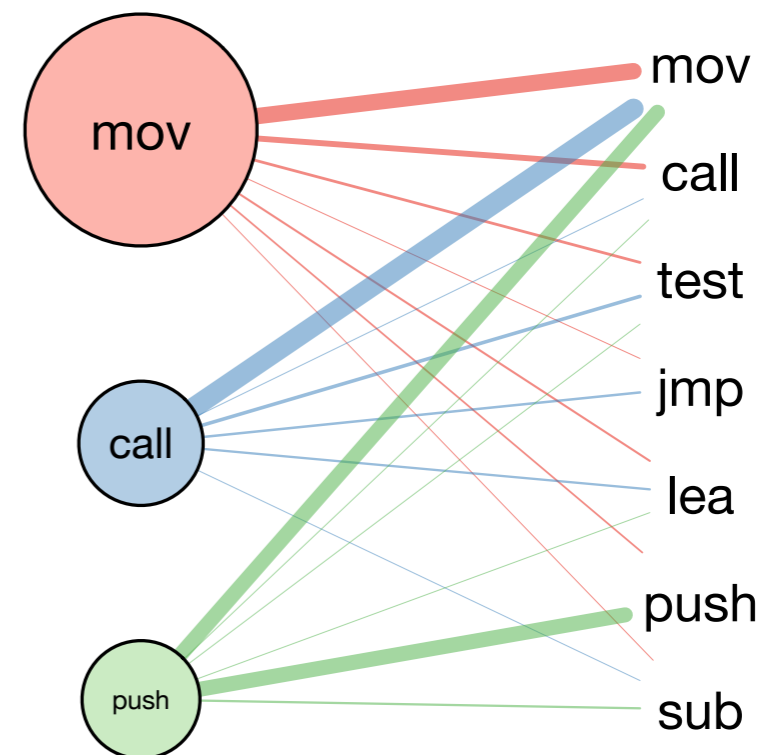
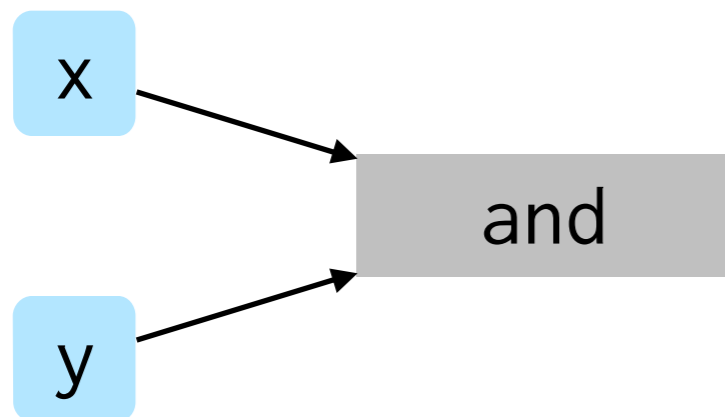
$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$



Learn search heuristics from existing code

# Producing candidate programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$



Learn search heuristics from existing code

Programming by example

[FlashFill, POPL'11]

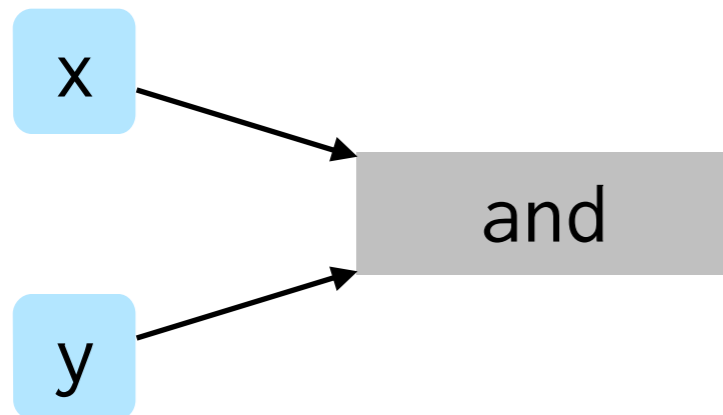
Autocomplete with synthesis

[CodeHint, ICSE'14]



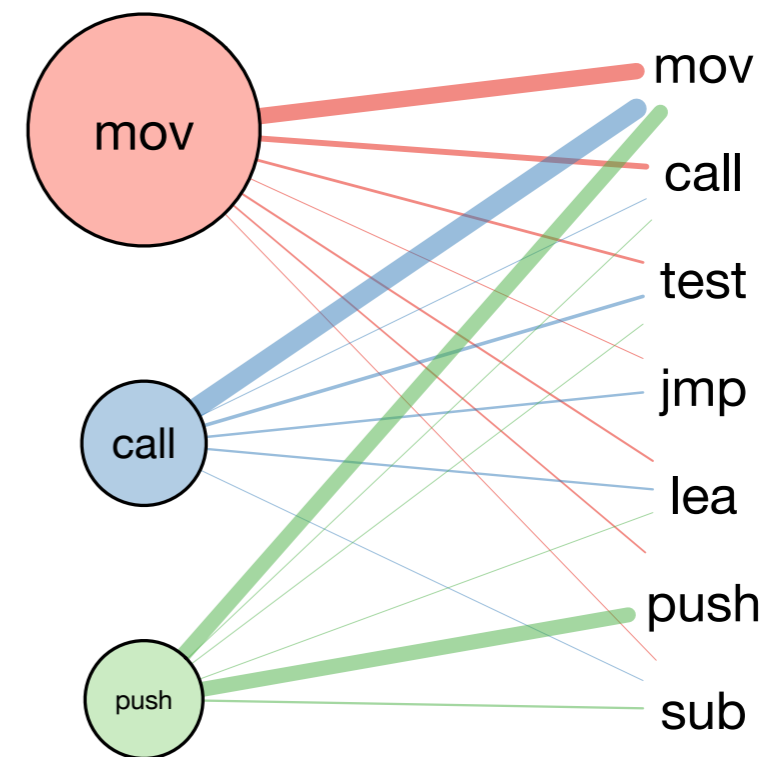
# Producing candidate programs

$$f(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$



Mine existing code bases for common idioms

Learn search heuristics from existing code



Programming by example

[FlashFill, POPL'11]

Autocomplete with synthesis

[CodeHint, ICSE'14]

# Producing candidate programs

Mine existing code bases  
for common idioms

Learn search heuristics from  
existing code

## Black Box Systems

observed behaviors  
into  
specifications

Selecting among many  
candidate solutions

# Sketch-Based Synthesis

**Sketches make synthesis more tractable**

# Sketches make synthesis more tractable

```
bit[W] popCount(bit[W] x) {  
    x = (x & 0x5555) + ((x >> 1) & 0x5555);  
    x = (x & 0x3333) + ((x >> 2) & 0x3333);  
    x = (x & 0x0077) + ((x >> 8) & 0x0077);  
    x = (x & 0x000F) + ((x >> 4) & 0x000F);  
    return x;  
}
```

# Sketches make synthesis more tractable

```
bit[W] popCount(bit[W] x) {  
    x = (x & 0x5555) + ((x >> 1) & 0x5555);  
    x = (x & 0x3333) + ((x >> 2) & 0x3333);  
    x = (x & 0x0077) + ((x >> 8) & 0x0077);  
    x = (x & 0x000F) + ((x >> 4) & 0x000F);  
    return x;  
}
```

```
bit[W] popSketched(bit[W] x) {  
    loop (??) {  
        x = (x & ??) + ((x >> ??) & ??);  
    }  
    return x;  
}
```

# Sketches make synthesis more tractable

```
bit[W] popSketched(bit[W] x) {  
    loop (??) {  
        x = (x & ??) + ((x >> ??) & ??);  
    }  
    return x;  
}
```

# Sketches make synthesis more tractable

```
bit[W] popSketched(bit[W] x) {  
    loop (??) {  
        x = (x & ??) + ((x >> ??) & ??);  
    }  
    return x;  
}
```

```
int[] map(int[] xs) {  
    int[] ys = {};  
    for (int i=0; i<xs.length; i++) {  
        ys.append(?(xs[i]));  
    }  
    return ys;  
}
```

```
int[] foldl(int[] xs) {  
    int acc = ??;  
    for (int i=0; i<xs.length; i++) {  
        acc = ??(acc, xs[i]);  
    }  
    return ys;  
}
```

```
int[] filter(int[] xs) {  
    int[] ys = {};  
    for (int i=0; i<xs.length; i++) {  
        if (?(xs[i])) ys.append(xs[i]);  
    }  
}
```



# Sketches make synthesis more tractable

**Identify common sketches from  
existing code**

# Sketches make synthesis more tractable

Mine existing code bases  
for common idioms

Identify common sketches from  
existing code

# Sketches make synthesis more tractable

Mine existing code bases  
for common idioms

Identify common sketches from  
existing code

**Approximate  
Computing**

quality bounds  
into  
approximate programs

# Sketches make synthesis more tractable

Mine existing code bases  
for common idioms

Identify common sketches from  
existing code

**Approximate  
Computing**  
quality bounds  
into  
approximate programs

Precise  
Implementation

Approximate  
Program  
Synthesis



Approximate Computing

Hardware Synthesis

Black Box Systems

Program  
Synthesis

Statistical  
Machine  
Learning

Approximate Computing

Hardware Synthesis

Black Box Systems

Program  
Synthesis

Statistical  
Machine  
Learning

Approximate Computing

Hardware Synthesis

Black Box Systems

Program  
Synthesis

Statistical  
Machine  
Learning

**Thanks!**

