Reliable and Interpretable Artificial Intelligence

Martin Vechev
ETH Zurich

Fall 2017

Tentative Lecture Topics

Block	Lecture Topic	Presenter
	Introduction	Martin
Synthesis	CEGIS, Sketch	Martin
	Programming by Example, Exact Learning with guarantees	Dana
	Learning from Big Code	Veselin
	Neural Turing Machines, Neural RAM Sketching with Neural machines	Matthew
Neural Network Robustness	Foundations of CNNs, finding adversarial examples	Martin/Petar
	Symbolic analysis, proving robustness	Dana/Petar
	Bayesian Synthesis	Swarat
Probabilistic Programming	Discrete semantics + example applications	Martin
	Exact inference for mixed distributions	Timon
	Deep Probabilistic Programming, Measure Theory	Benjamin

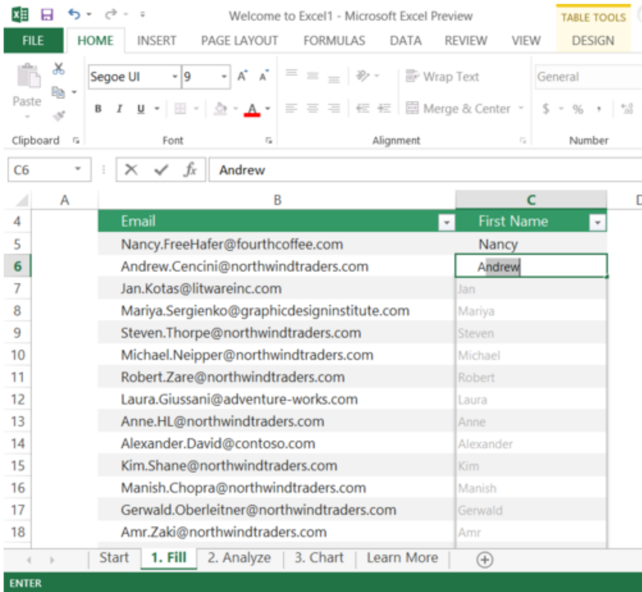
Modern Program Synthesis

Program Synthesis is one of the most exciting recent developments in artificial intelligence. It focuses on one of computer science's most exciting visions: learning a function from specifications (e.g., examples)

The main focus of today's program synthesis vs. that of decades ago is the incorporation of powerful SAT/SMT solvers, multiple ways to convey intent (e.g., labeled data) and bias the learning (e.g., partial programs) that involve the user in the loop.

Program synthesis tends to be quite interdisciplinary: it often combines techniques from several fields: programming languages, machine learning, natural language processing and others.

Flash Fill (Excel feature) demo
(programming by example aka PBE)



Input	Output
(425)-706-7709	425-706-7709
510.220.5586	510-220-5586
1 425 235 7654	425-235-7654
425 745-8139	425-745-8139

Wired Magazine: Excel is now a lot easier for people who aren't spreadsheet- and chart-making pros. The application's new Flash Fill feature recognizes patterns, and will offer auto-complete options for your data. For example, if you have a column of first names and a column of last names, and want to create a new column of initials, you'll only need to type in the first few boxes before Excel recognizes what you're doing and lets you press Enter to complete the rest of the column.

PBE is applicable in various domains: data science, AI powered data wrangling, etc.

"Automating string processing in spreadsheets using input-output examples"; POPL 2011; Sumit Gulwani

Build your own PBE Synthesizer

(e.g., AI data wrangler)

Microsoft's PROSE framework:
https://microsoft.github.io/prose/

Refined intent

Intended program $P \in \mathcal{L}$

Intent spec φ: input-output examples

Ranked set of valid programs $\mathcal{S}$

Best program

Translator

Program Synthesis Framework

User (Debugging)

Interactive questions

Program ranking function $h(P)$

Domain-specific language $\mathcal{L}$

Disambiguation

Test inputs

Deployable code in Python/R/C#/…

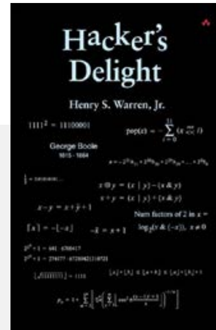
Diving deep into what's new with Azure Machine Learning

Posted on September 25, 2017

https://azure.microsoft.com/en-us/blog/diving-deep-into-what-s-new-with-azure-machine-learning/

SKETCH: isolate rightmost 0

(constraint-based synthesis)




```
bit[W] isolate0 (bit[W] x) { // W: word size
  bit[W] ret=0;
  for (int i = 0; i < W; i++)
    if (!x[i]) { ret[i] = 1; break; }
  return ret;
}
```

Naïve implementation (spec)

```
bit[W] isolate0Sketched(bit[W] x) implements isolate0 {
  return ~(x + ??) &  (x  + ??);
}
```

Sketch – provides intent, restricts hypothesis space

```
bit[W] isolate0Fast (bit[W] x) implements isolate0 {
  return ~x & (x+1);
}
```

Discovered hypothesis that obeys spec

Solved with Counter-example guided inductive synthesis (CEGIS)

CEGIS + Sketch

CEGIS, a general method/concept for synthesis

Counter-Example Guided Inductive Synthesis

We will look into Sketch: a system which uses CEGIS

- This will help to concretely motivate CEGIS
- Sketch system is actively used and has pioneered much of the research in the area
- Download & Try Sketch:
  - https://bitbucket.org/gatoatigrado/sketch-frontend/wiki/Home
  - https://bitbucket.org/gatoatigrado/sketch-backend/commits/all

Dimensions in Program Synthesis

On Board

What is sketching?

- A concept, to express intent
  - allows users to express insight by defining the hypothesis space and bias the search
  - machine helps with low-level reasoning

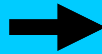
- Sketch is also a program synthesis system
  - specific instance of the concept
  - we will illustrate how CEGIS arises in Sketch

So lets start with the Sketch system and reach CEGIS…

The sketching experience

sketch

The sketching experience



sketch

implementation
(completed sketch)

Part I: Expressing Intent

SKETCH: Expressing Intent

- Sketches are functions with holes
  - write what you know, use holes for the rest
  - express intent in this way

- 2 ways to express intent and control hypothesis space
  - Specifications: how does SKETCH know what function you actually want? Note that specs are just convenient representation of sets of examples

  - Holes: to be instantiated by the learner

Specifications

Constrain function behavior

- assertions:

  assert x > y;

- concrete examples

  x = 5 or x = 6…

- function equivalence:

  blockedMatMul(Mat a, Mat b) implements matMul

Holes

Holes are placeholders for the synthesizer

synthesizer replaces holes with concrete code fragments

fragment must come from a set defined by the user

Holes define the hypothesis space, they allow you to bias the learning towards a function you can interpret more easily

Integer hole

Define sets of integer constants

Example: Hello World of Sketching

spec:

```
int foo (int x)
{
    return x + x;
}
```

sketch:

```
int bar (int x) implements foo
{
    return x * ??;
}
```

Integer Holes via Sets of Expressions

- Example: Find least significant zero bit
  - 0010 0101 → 0000 0010

```
int W = 32;

bit[W] isolate0 (bit[W] x) {      // W: word size
        bit[W] ret = 0;
        for (int i = 0; i < W; i++)
                if (!x[i]) { ret[i] = 1; return ret;  }
}
```

- Trick:
  - Adding 1 to a string of ones turns the next 0 to a 1
  - i.e. 000111 + 1 = 001000

!(x + ??) & (x + ??) → !(x + 0) & (x + 1)

Integer Holes via Sets of Expressions

- Example: Find least significant zero bit
  - 0010 0101 → 0000 0010

```
int W = 32;

bit[W] isolate0 (bit[W] x) {      // W: word size
        bit[W] ret = 0;
        for (int i = 0; i < W; i++)
                if (!x[i]) { ret[i] = 1; return ret;  }
}

bit[W] isolateSk (bit[W] x) implements isolate0 {

        return !(x + ??) & (x + ??) ;
}
```

Integer Holes via Sets of Expressions

- Expressions with `??` == sets of expressions
  - linear expressions `x*?? + y*??`
  - polynomials `x*x*?? + x*?? + ??`
  - sets of variables `?? ? x : y`
- Semantically powerful but syntactically clunky
  - Regular Expressions are a more convenient way of defining sets

Regular Expression Generators

- Construct: {| RegExp |}

- RegExp supports choice '|' and optional '?'
  - can be used arbitrarily within an expression
    - to select operands `{|  (x | y | z) + 1 |}`
    - to select operators `{|  x (+ | -) y |}`
    - to select fields `{| n(.prev | .next)? |}`
    - to select arguments `{| foo( x | y, z) |}`

- Set must respect the type system
  - all expressions in the set must type-check
  - all must be of the same type

Least Significant Zero revisited

- How did I know the solution would take the following form?

```
!(x + ??) & (x + ??)
```

- What if all you know is that the solution involves `x`, `+`, `&` and `!` ?

```
bit[W] tmp=0;
{| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
{| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
return tmp;
```

This is now a set of statements
(and a really big one too!)

Sets of statements

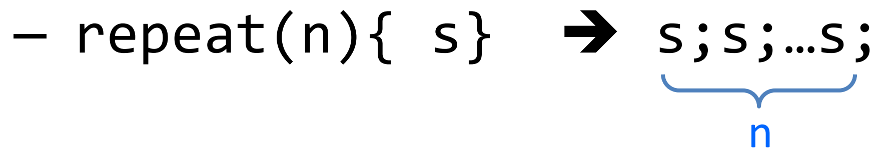
- Statements with holes = sets of statements

- Higher level constructs for statements too
  - repeat

```
bit[W] tmp=0;
repeat(3){
   {| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
}
return tmp;
```

repeat

Avoid copying and pasting



– each of the n copies may resolve to a distinct stmt
– n can be a hole too.

```
bit[W] tmp=0;
repeat(??){
   {| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
}
return tmp;
```

Keep in mind:

– the synthesizer won't try to minimize n
– use the option: --unrollamnt to set the maximum value of n

Part II: Learning Functions

Step 1: Turn holes into special inputs

Step 1: Turn holes into special inputs

- The ?? Operator is modeled as a special input

Step 1: Turn holes into special inputs

- The ?? Operator is modeled as a special input

```
bit[W] isolSk(bit[W] x)
{
    return ~(x + ??) & (x + ??);
}
```

Step 1: Turn holes into special inputs

- The ?? Operator is modeled as a special input

```
bit[W] isolSk(bit[W] x)
{
   return ~(x + ??) & (x + ??);
}
```

```
bit[W] isolSk(bit[W] x, bit[W] c1, c2)
{
   return ~(x + c1) & (x + c2);
}
```

Step 1: Turn holes into special inputs

- The ?? Operator is modeled as a special input
  - we call them control inputs

```
bit[W] isolSk(bit[W] x)
{
   return ~(x + ??) & (x + ??);
}
```

```
bit[W] isolSk(bit[W] x, bit[W] c1, c2)
{
   return ~(x + c1) & (x + c2);
}
```

Step 1: Turn holes into special inputs

- The ?? Operator is modeled as a special input
  - we call them control inputs

```
bit[W] isolSk(bit[W] x)
{
   return ~(x + ??) & (x + ??);
}
```

```
bit[W] isolSk(bit[W] x, bit[W] c1, c2)
{
   return ~(x + c1) & (x + c2);
}
```

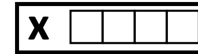
Step 2: Constraining the set of controls

- Correct control
  - causes the spec & sketch to match for all inputs
  - causes all assertions to be satisfied for all inputs
- Constraints are collected into a predicate

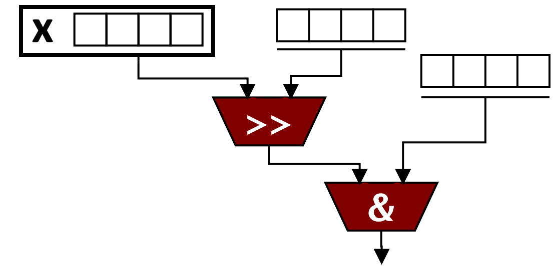
$$Q(in, c)$$

```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
            x = (x & ??)
            + ((x >> ??) & ??);
    }
    return x;
}
```

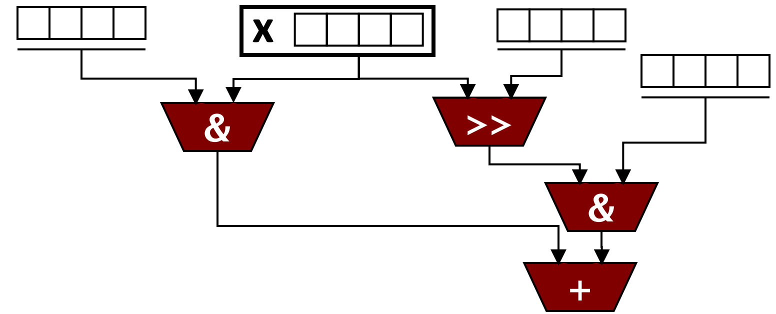
```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
        x = (x & ??)
        + ((x >> ??) & ??);
    }
    return x;
}
```



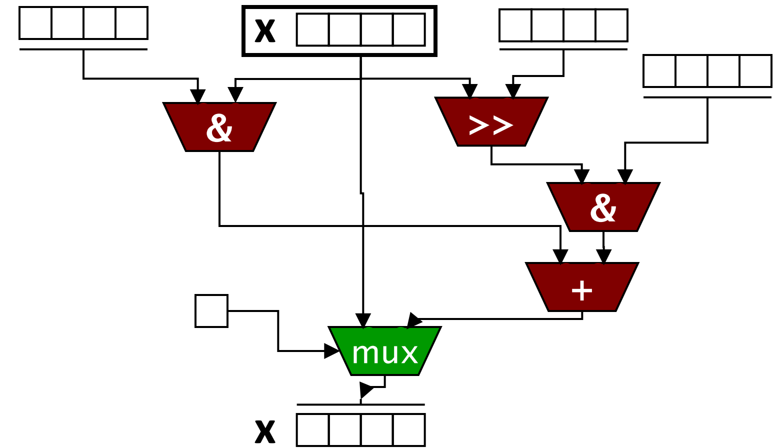
```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
            x = (x & ??)
⇒           + ((x >> ??) & ??);
    }
    return x;
}
```



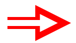
```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
⇒           x = (x & ??)
            + ((x >> ??) & ??);
    }
    return x;
}
```

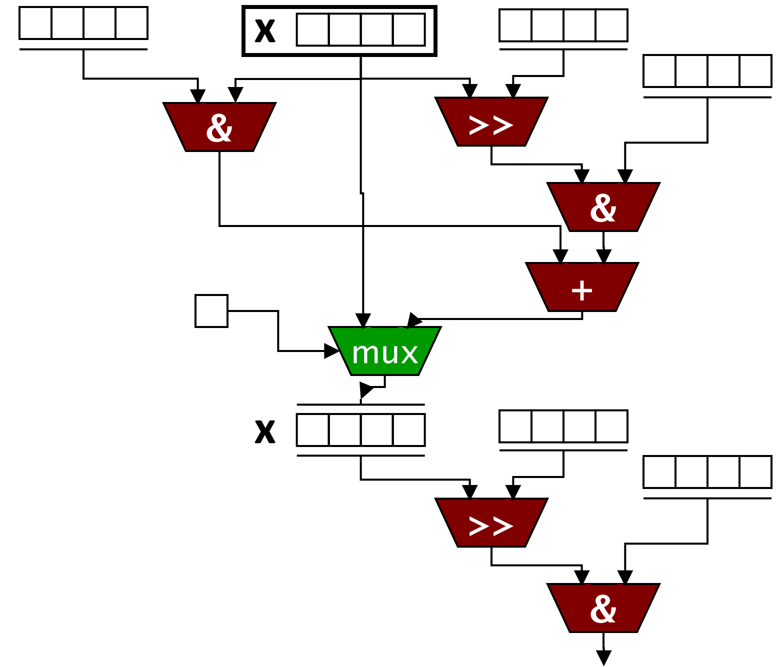


```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
            x = (x & ??)
            + ((x >> ??) & ??);
⇒   }
    return x;
}
```

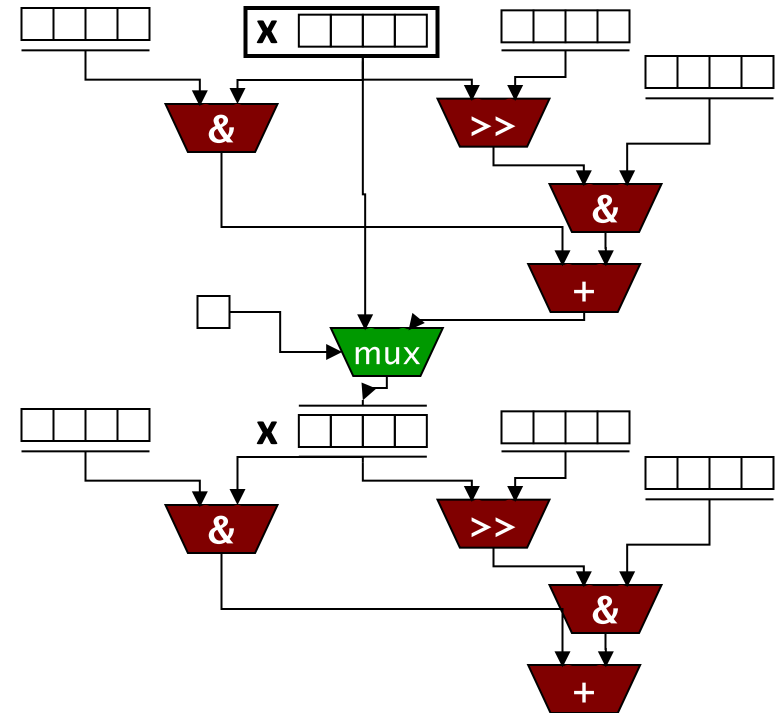


```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
        x = (x & ??)
        + ((x >> ??) & ??);
    }
    return x;
}
```

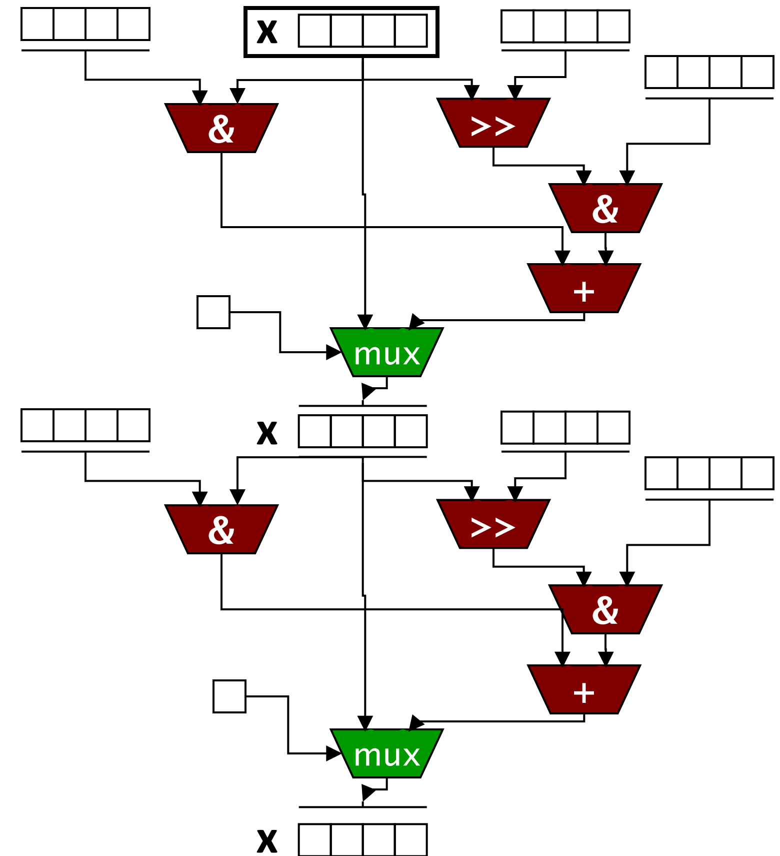




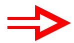
```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
⇒           x = (x & ??)
            + ((x >> ??) & ??);
    }
    return x;
}
```

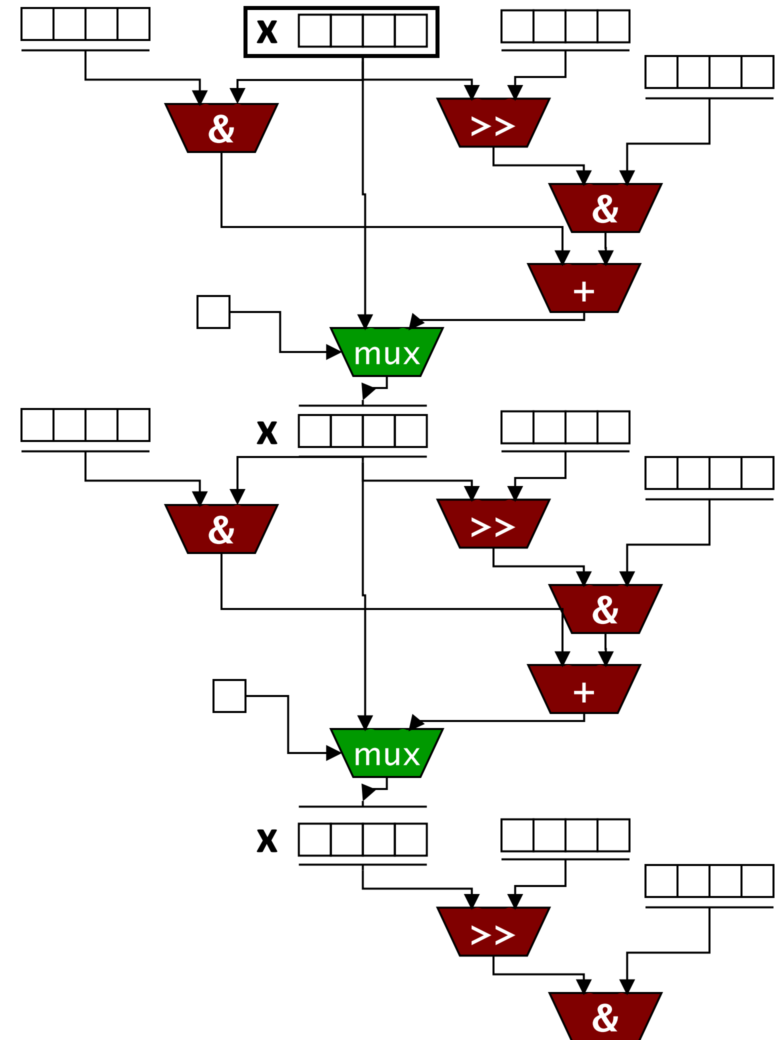


```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
            x = (x & ??)
            + ((x >> ??) & ??);
    }
    return x;
}
```

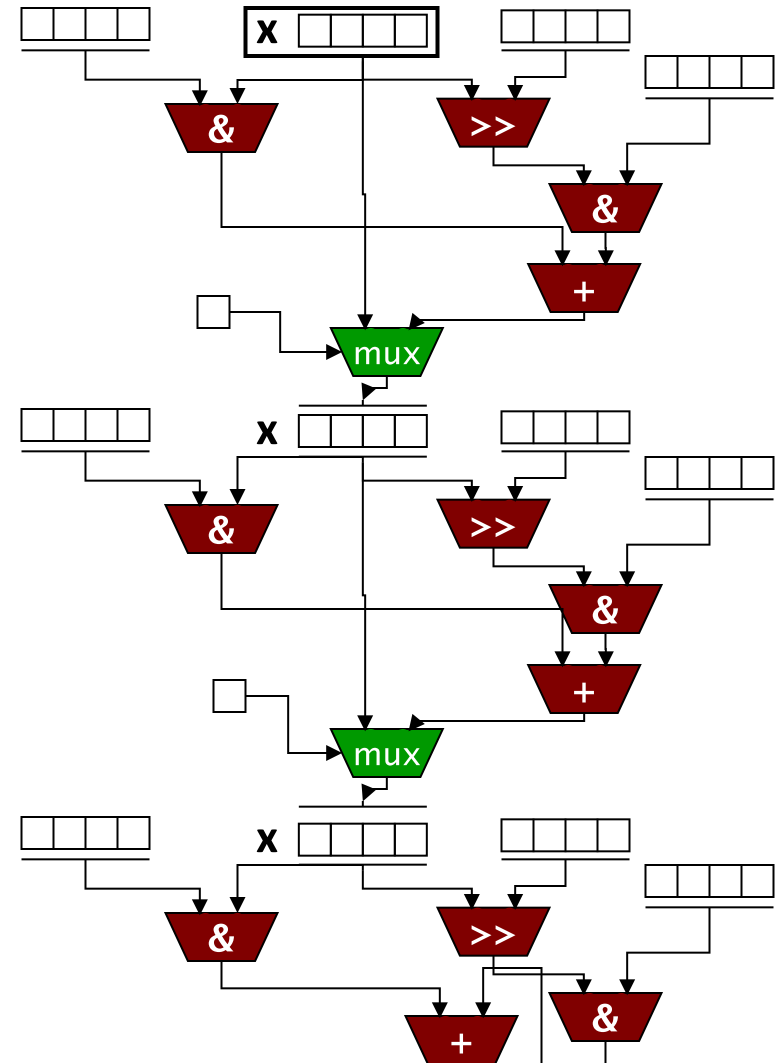


```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
            x = (x & ??)
            + ((x >> ??) & ??);
    }
    return x;
}
```

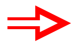


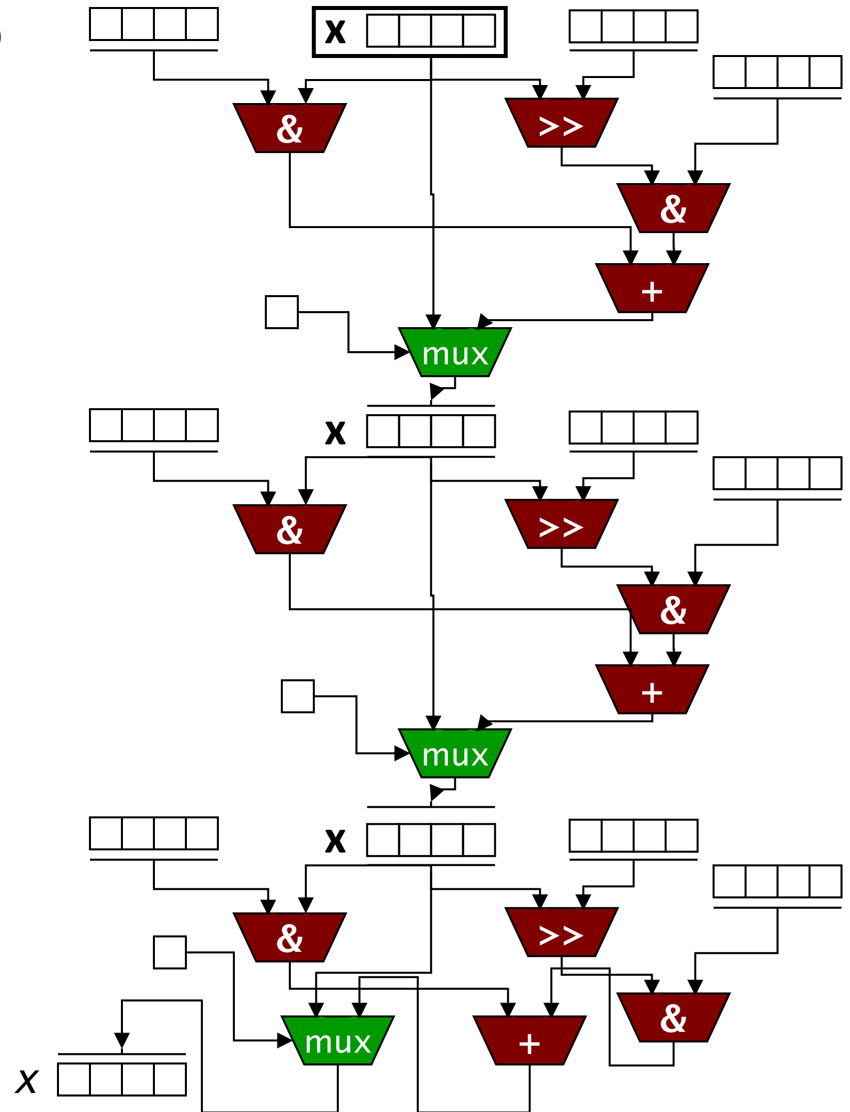


```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
⇒           x = (x & ??)
            + ((x >> ??) & ??);
    }
    return x;
}
```

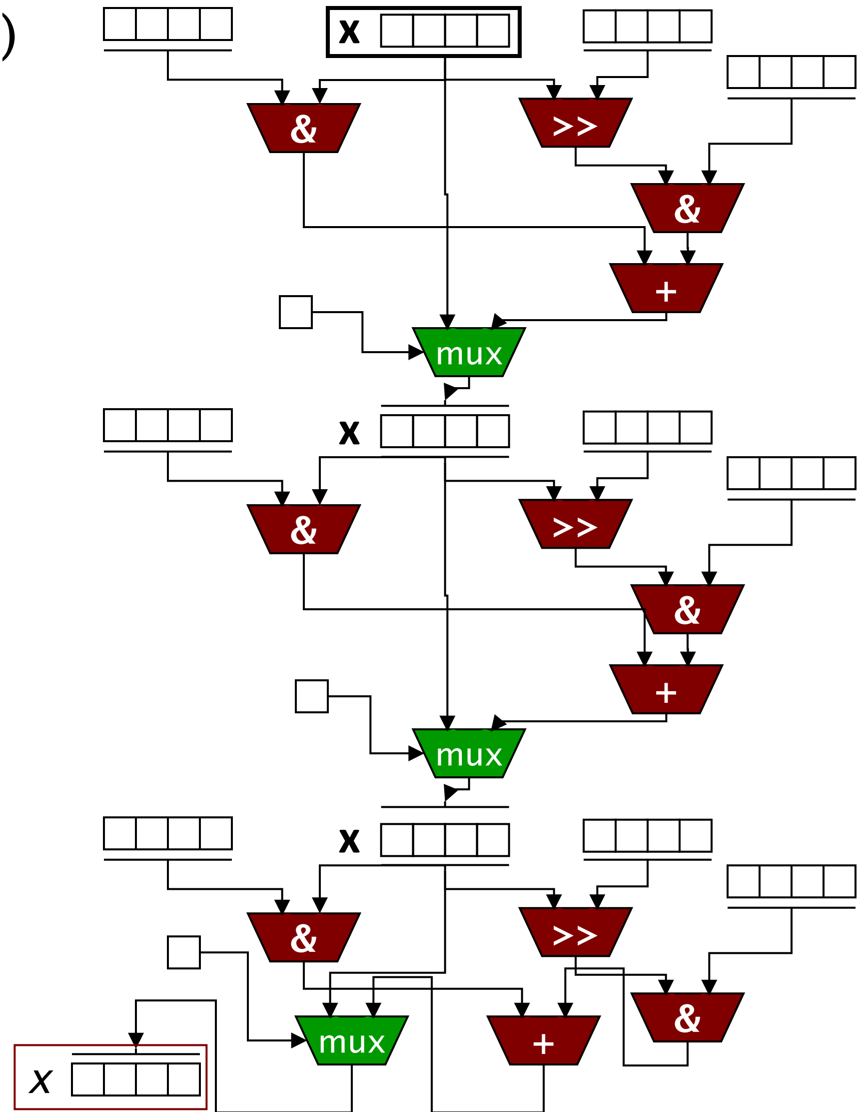


```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
        x = (x & ??)
        + ((x >> ??) & ??);
    }
    return x;
}
```

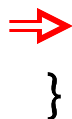




```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
            x = (x & ??)
            + ((x >> ??) & ??);
    }
⇒   return x;
}
```



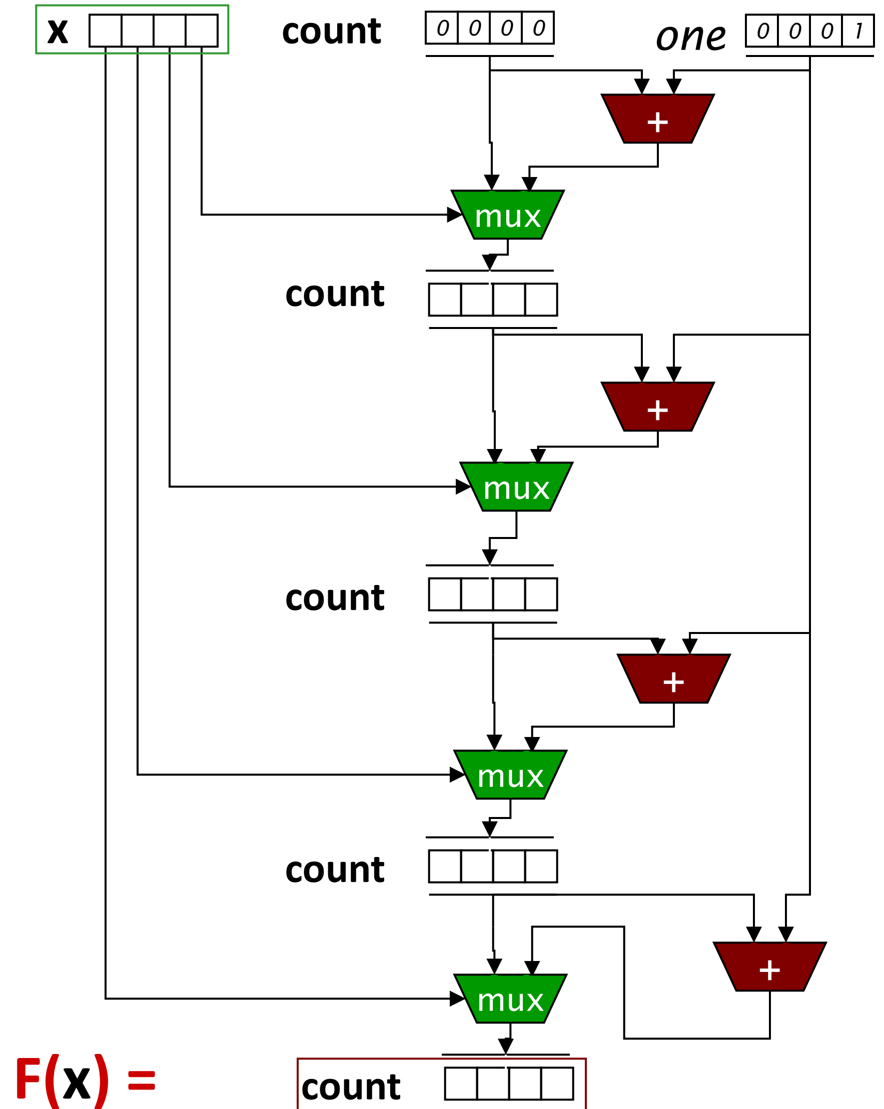
```
int popCountSketched (bit[W] x)
    implements popCount {
    repeat (??) {
            x = (x & ??)
            + ((x >> ??) & ??);
    }
    return x;
}
```



S(x,c) =

Ex : Population count (W=4)

```
int popCount (bit[W] x)
{
    int count = 0;
    for (int i = 0; i < W; i++) {
        if (x[i]) count++;
    }
    return count;
}
```

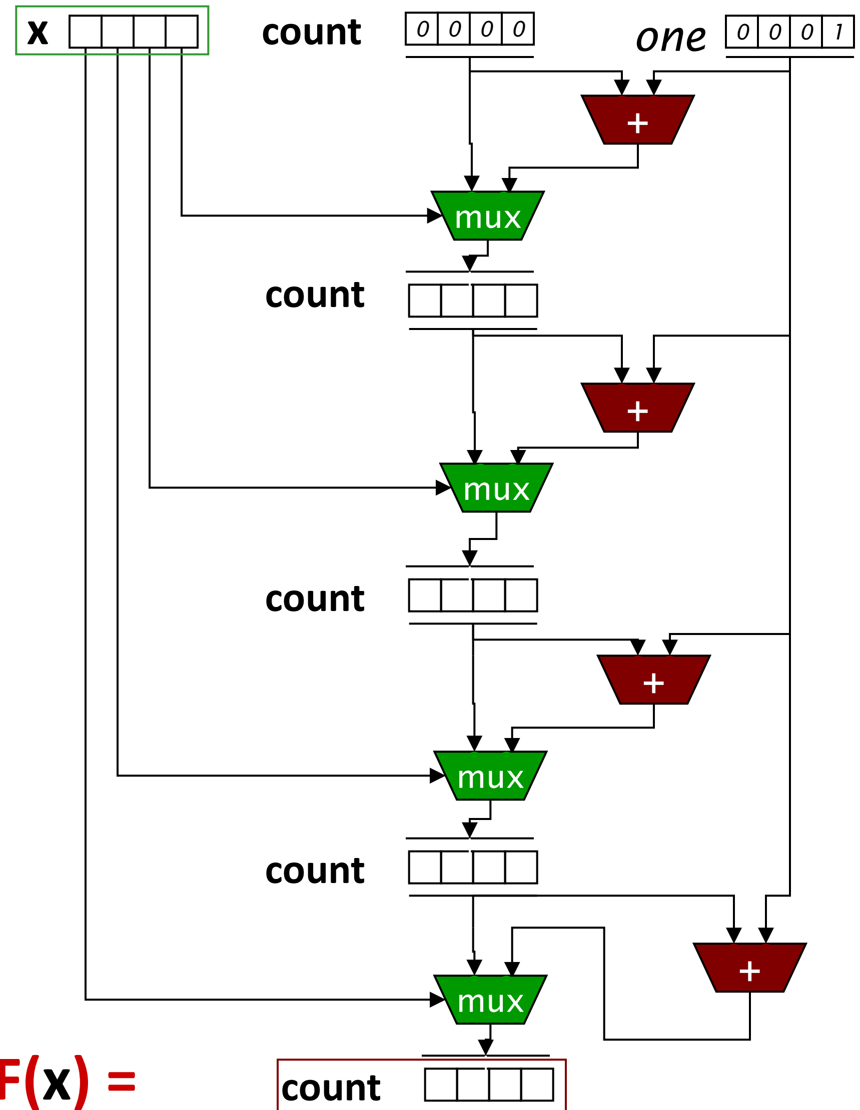


Ex : Population count (W=4)

```
int popCount (bit[W] x)
{
    int count = 0;
    for (int i = 0; i < W; i++) {
        if (x[i]) count++;
    }
    return count;
}
```

Q(in, c) ≜ Q(x, c)

Q(x, c) ≜ S(x,c) == F(x)



A Sketch as a constraint system

Learning reduces to constraint satisfaction

$$\exists c\ \forall x.Q(x,c)$$

A Sketch as a constraint system

Learning reduces to constraint satisfaction

$$\exists c\ \forall x.Q(x,c)$$

Constraints are too hard for standard techniques

Universal quantification over inputs

- Too many inputs
- Too many constraints
- Too many holes

We have reached CEGIS…

CEGIS

Key insight to remember:

CEGIS aims to find the right small set of examples S so that if we learn a function F satisfying S, then F is likely to be correct on all (potentially infinite set of) examples.

Lets use instantiate CEGIS idea to Sketch

A small set of inputs can fully constrain the solution

- focus on corner cases

$\exists c \; \forall x$ in $E.Q(x,c)$ where $E = \{x_1, x_2, \ldots, x_k\}$

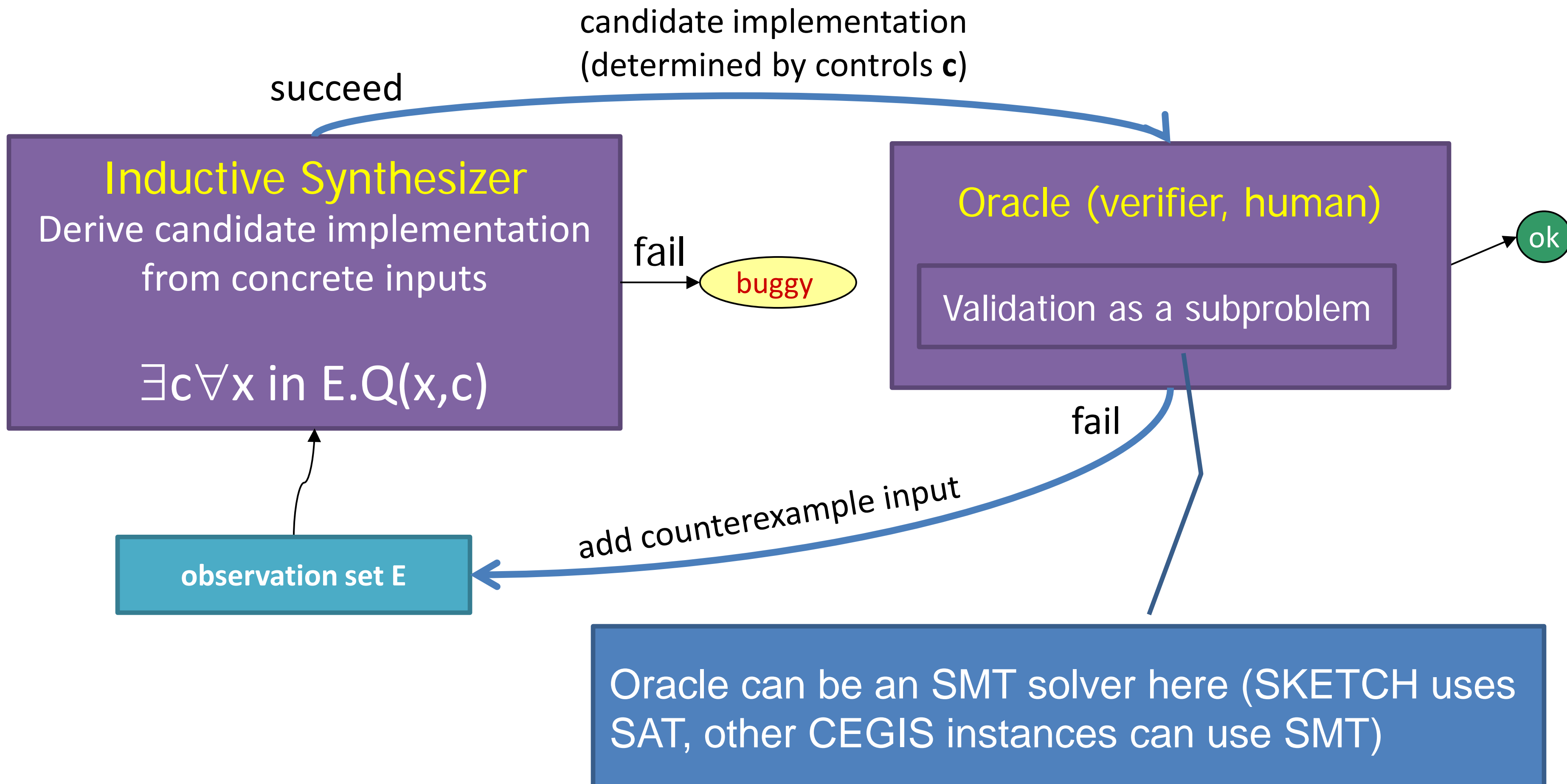
This is an inductive synthesis problem !

- but how do we find the set E?
- and how do we solve the inductive synthesis problem?

Counterexample Guided Inductive Synthesis

Idea:

couple inductive synthesizer with an oracle (e.g., verifier, human)



Solving SKETCHes via CEGIS

Deriving a candidate from a set of observations

$$\exists c\ \forall x \text{ in } E.Q(x,c) \quad \text{where } E = \{x_1, x_2, \ldots, x_k\}$$

Encode c as a bit-vector

– natural encoding given the integer holes

Encode $Q(x_i, c)$ as boolean constraints on the bit-vector

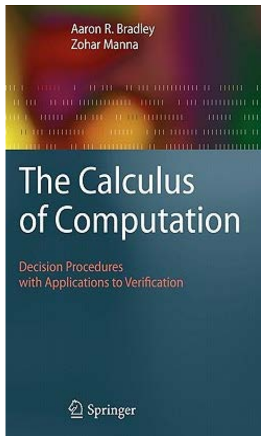
Solve constraints using SAT/SMT solver

– with lots of preprocessing in between

SMT solvers

SMT solvers are specialized theorem provers for specific logical fragments, that decide validity of formulas, e.g. theory of integers:

$$\forall x,y.\ x > 0 \wedge (x = (y + y) \vee x = (y + y) + 1) \rightarrow (x - y) > 0$$



Zohar Manna

- logical fragments: integers, reals, arrays, etc.
- want to ``compile'' many problems into SMT formulas (neural networks, synthesis, etc.) and then use SMT solvers to solve them
- SMT solvers: MSR's Z3, SRI's Yices, etc

We will see examples of using SMT later with neural nets as well. For the course we treat SMT as a black box. Internals of SMT are beyond the scope of the course.

Summary of Lecture

- Landscape of modern synthesis
- How CEGIS works
- MIT's Sketch and its use of CEGIS

Next lecture: PBE, PBE with CEGIS, and complexity bounds on interaction